Adatbázis rendszerek I.

Dr. Kovács László
egyetemi docens

Miskolci Egyetem
Általános Informatikai Tanszék

Tartalomjegyzék

Előszó

Ez itt az előszó helye ...

1. fejezet

Az adatbáziskezelés alapfogalmai

1.1. Információ és adat fogalmai

A számítástechnika fejlődésének egyik fontos jellemzője, hogy egyre több felhasználó egyre több, számítógépen tárolt adatot használ fel. A növekvő információmennyiség mind szélesebb körben válik elérhetővé. Az információ-hozzáférés így jelentkező demokratizálódásának hatása teljes joggal mérhető a könyvnyomtatás jelentőségéhez, ezért szokás ezt a jelenséget *elektronikus Gutenberg forradalomnak* is nevezni. Az elektronikus Gutenberg forradalom legfontosabb jellemzője, hogy

- egyre nagyobb információmennyiség
- egyre szélesebb tömegek számára
- egyre demokratikusabban válik elérhetővé
- a számítógép használatával.

Az elkészített és alkalmazott számítógépi programrendszereknek növekvő adatmennyiséggel kell megbirkózniuk. A hétköznapjainkban is egyre gyakrabban találkozhatunk a számítógépes *információs rendszerek* alkalmazásával, melynek feladata a működés során megjelenő információk számítógéppel történő hatékony feldolgozása, karbantartása a hatékony működés biztosítása céljából. Számítógépes információs rendszer fut az üzemekben, gyárakban a termelés irányítására, a pénzügyi, személyzeti, raktári, anyaggazdálkodási feladatok elvégzésére.

Néhány hasonló alkalmazási terület az élet szinte minden területéről megemlíthető:

- kereskedelem: raktári készlet és megrendelések nyilvántartása,
- kultúra, oktatás: könyvtári információs rendszerek, hallgatói adminisztráció
- közigazgatás: adónyilvántartások
- közlekedés: helyjegy foglalási rendszerek
- egészségügy: beteg nyilvántartás
- tudomány: szakadatbázisok

– posta: ügyfelek, számlák nyilvántartása
– vállalat: termelés irányítási rendszerek
– mérnöki munka: tervezői rendszerek.

A példákat még hosszan lehetne sorolni. A felsorolásban gyakran találkozhattunk két, egymással igen rokon értelmű szóval e rendszerek jellemzésében, az információ és az adat fogalmaival. Az információt és adatot gyakran használják azonos értelemben, pedig e két fogalom között létezik jelentésbeli különbség, melyre oda kell figyelni, és célszerű tudatosítani magunkban e fogalmak pontos jelentését.

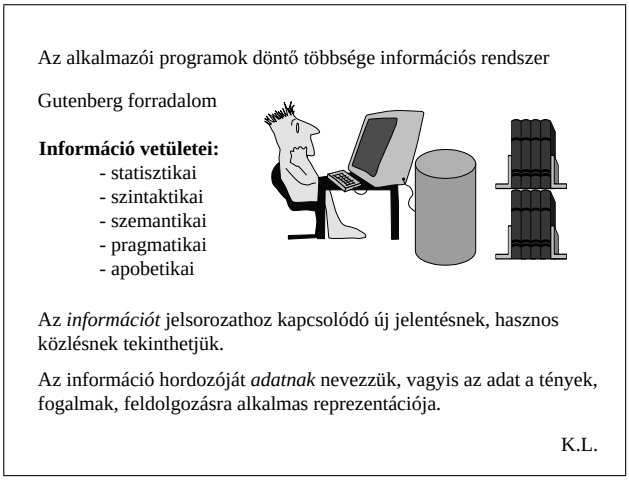
Az információt jelsorozathoz kapcsolódó új jelentésnek, hasznos közlésnek tekinthetjük.

Az információ fogalma alapfogalomnak tekinthető, értelmezésére is igen sokféle megközelítés létezik, melyek közül az egyik az általunk előbb megadott értelmezés. Az információ egyik fontos eleme az újdonság. Információ lehet például egy újsághír a minket érintő törvényváltozásokról. Nagyon sokféle módon, sokféle információhoz jutunk nap mint nap. Az információhoz szorosan kapcsolódik a

– hordozó közeg, jel (hanghullám, fény, . . .)
– jelentés és a
– feldolgozó (például én magam, aki értelmezem a megkapott jelet).

Az információ tehát mindig szubjektív fogalom, függ a feldolgozótól. Az információt igen sok oldalról lehet vizsgálni és elemezni. Az információnak vannak

– statisztikai
– szintaktikai
– pragmatikai



1.1. ábra. Információ és adat fogalomköre

– apobetikai oldalai.

1. A *statisztikai* oldal az információt hordozó jelek előfordulási gyakoriságait vizsgálja. E terület elméletének kidolgozása Shannon nevéhez kötődik. Statisztikai értelmében egy jelsorozat információtartalma a jelsorozat előfordulási valószínűségének reciprokával arányos. Erre egy szemléletes példa, hogy az nem hír, ha egy kutya megharapja a postást, de az már hírnek számít, ha egy postás harapja meg a kutyát.

2. A *szintaktikai* oldal a jelsorozatok formális azonosságait vizsgálja, mely alapján a 'Jani szereti a sört' mondat hasonló felépítésű a 'Jani szereti Marit' mondattal. Az adatok viszont a feldolgozás során nyernek értelmet, és ekkor már a két mondat egészen más hatást válthat ki.

3. Az adatok ezen rejtett tartalma a *szemantikai* oldal, mely a jelsorozat mögött húzódó jelentést, lényeget hangsúlyozza.

4. A *pragmatika* a jelentés gyakorlati hasznosságát emeli ki. Egy elméletileg hasznos információ nem alkalmazható, ha felhasználásához nagyon hosszú idő szükséges, vagy ha az elérhető haszon elhanyagolhatóan kicsi.

5. Míg a pragmatikai oldal a konkrét gyakorlati tevékenységet öleli fel, addig az *apobetikai* oldal az információ mögött rejlő szándékot tartalmazza. Például egy 'Csukd be az ajtót!' jelsorozat pragmatikai oldala, hogy el kell menni az ajtóhoz és be kell csukni, hogy zárva legyen az ajtó. A jelsorozathoz tartozó apobetikai oldal pedig arra vonatkozik, hogy miért is kell az ajtót becsukni, például azért, hogy meg ne fázzunk vagy azért, hogy ne lássanak be a szobába. Majd a későbbi modellezési fázisokban közvetlenebbül is tapasztalhatjuk a különböző oldalak fontosságát.

Az információ, mint szubjektív fogalom, szorosan kötődik hordozó közegéhez, mely már objektív jelenségnek tekinthető.

> *Az információ hordozóját adatnak nevezzük, vagyis az adat a tények, fogalmak, feldolgozásra alkalmas reprezentációja. Az adat objektív, feldolgozótól független.*

A témánkból eredően, mi az adat fogalmát egy kicsit szűkebben, a számítógépen tárolt adatokra értelmezzük. Az adat számunkra a számítógépben tárolt jelsorozatot jelenti, melyből a feldolgozás során nyerhetünk információt. Az adat a számítógépben viszont még információ nélküli jelsorozatként tárolódik.

1.2. Az adattárolás különböző formái

Az információ feldolgozására készített számítógépes programoknál az adatok különböző strukturáltságban kerülnek letárolásra. Az adatok lehetnek lazább szerkezetben, vagy szigorúbb, finomabb struktúrában letárolva.

Az adattárolás módjának megfelelően beszélhetünk:

– szövegszerű rendszerekről

– adatszerű, finoman strukturált rendszerekről és
– szemi-strukturált rendszerekről.

1. A *szövegszerű* tárolásnál a dokumentumok, könyvek, cikkek alkotják a legkisebb elérési egységet, és a dokumentum rögzített belső adatstruktúra nélkül, ömlesztve tartalmazza az információt.

2. Az *adatszerű* tárolásnál az információk megadott struktúra szerint sokkal kisebb adatelemekre szétbontva kerülnek elhelyezésre, minden adatelemhez a struktúrában jelentést és formátumot is hozzárendelve. Ekkor rákérdezhetünk például egy ember nevére, lakcímére, azaz minden egyedi tulajdonságára is.

3. A *szemi-strukturált* rendszerek az előbb említett típusok között foglalnak helyet. Az ilyen jellegű dokumentumokban rendszerint létezik egy lazább, viszonylag nagyobb terjedelmű struktúra, melyen belül azonban az adatok lazább formában, szövegszerűen is elhelyezkedhetnek.

A szövegszerű rendszerekben az információk az ember által könnyen értelmezhető formában kerülnek letárolásra, rendszerint úgy, hogy a dokumentum önmagában is elég az információ kinyeréséhez. Vegyük például az alábbi levél részletet:

Helló Péter!
Képzeld Zolinak van egy nagyon jó CD-je,
egy Kraftwerk lemez, azt hiszem Autobahn
a címe. Talán 2500-ért vette a múltkor. . . .

A megadott szöveg alapján tudjuk, hogy Zolinak van egy Kraftwerk CD-je, melynek címe Autobahn és ára 2500 Ft. A szöveg egyik jellemzője, hogy pontosan meg kell adnia az információ leírását. Ha például több embernek is felsoroljuk a CD gyűjteményeit a levélben, akkor valószínűleg többször is leírjuk a 'van', 'lemez', 'egy' szavakat is, melyek a szöveg önálló megértéséhez kellenek, ezáltal mintegy bőbeszédűvé téve a leírást. A fenti tárolás a számítógépes feldolgozás során tehát viszonylag nagyobb helyigénnyel jelentkezik, és másrészt igen körülményes a szövegben tárolt információk automatikus, program által történő kigyűjtése.

A számítógépes feldolgozásra jobban illeszkedik a strukturált leírási mód, amikor az adatok egy megadott, merev struktúrába rendezetten foglalnak helyet. A fenti CD lista például egy táblázatban jelenik meg, melyben minden sor egy CD-t ír le, és minden oszlopnak megadott jelentése van:

kod	eloado	cim	ar	tulaj
1	Kraftwerk	Autobahn	2500	Zoli
4	Groove	Lifeforce	3100	Laci
3	Enya	Orinoco	2700	Ani
6	Hobo	Vadászat	3600	Zoli

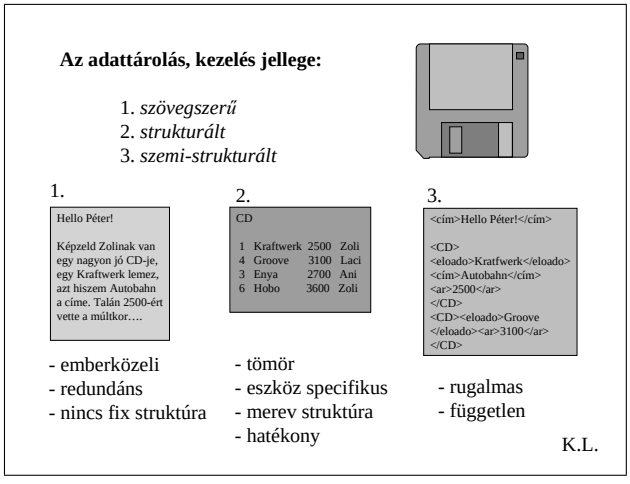
A fenti strukturált megadási mód előnye, hogy igen tömör és a feldolgozása is könnyen algoritmizálható. Sajnos ez a fajta tömör, táblázatos felírás sem alkalmazható mindig, ugyanis a táblázat azonos jellegű, ismétlődő adatsorok megadására

alkalmas, viszont sokszor egy leíró dokumentumon belül több különböző jellegű adatsor található. Emellett eltérő lehet az egyes sorok felépítése is a táblázaton belül, ezért szükségessé válik az adatelem jelentésének is a megadása a dokumentumon belül. Így egy olyan leírási mód jön létre, melyben csak a lényeges adatelemek szerepelnek, de az adatértékek mellett ott szerepelnek a jelentést leíró részek is, így lehetővé téve egy rugalmas szerkezetet. A fenti információk szemi-strukturált módon való leírását mutatja be a következő példa. Az adatértékek jelentését az azokat közrefogó <> jelek adják meg. Itt <...> a kezdő tag és </...> a vég tag.

<cim>Helló Péter!</cim>
<CD>
<eloado>Kratfwerk</eloado>
<cim>Autobahn</cim>
<ar>2500</ar>
</CD>
<CD>
<eloado>Groove</eloado>
<ar>3100</ar>
</CD>

Mint látható az egyes CD-k eltérő tulajdonság szerkezettel rendelkezhetnek. Az ilyen jellegű dokumentumok ugyan az ember által kissé nehezen átláthatóak, viszont a számítógépes feldolgozásuk nem okoz különösebb nehézséget. Természetesen a feldolgozási idő és hatékonyság rosszabb lesz, mint a szigorúan strukturált adattárołási mód esetében.

Ezen tárgy keretében a feldolgozandó témáink az adatszerű, strukturált adattároláshoz fognak kapcsolódni.



1.2. ábra. Adattárolási formák

1.3. Információs rendszerek adatkezelési követelményei

A bevezetőben is említett információs rendszerek egyik fontos jellemzője, hogy nagy adathalmazt kezelnek, és az adatok között is bonyolultabb kapcsolatok állhatnak fenn, és ezen adatokat, kapcsolatokat hosszabb időszakon át is meg kell őrizni. Ma már sok olyan információs rendszer létezik a világon, amelyek több terrabyte nagyságrendű adatot tartalmaznak. Természetesen az információs rendszerek döntő többsége ettől jóval kisebb méretű. A Gartner Group elemzését követve 1997-ben a 400 Gbyte-tól nagyobb méretű adatrendszereket tekintik nagyon nagy adatrendszereknek, melyeket a *VLDB* (Very Large Data Bases) rövidítéssel is szokták jelölni.

Egy érdekes példa a kezelt adatok mennyiségére, hogy a világ talán legnagyobb táblázata a kínai telefontársaság információs rendszerében található, mintegy 1 Tbyte mérettel. Ha a fenti adatmennyiséget könyvben tárolnánk le, akkor mintegy 100000 vastag lexikon méretű könyvre lenne szükség. Ha ezen könyveket egymás mellé szeretnénk tenni, akkor kb. 5 km hosszú polcra lenne szükségünk.

A nagy és bonyolult adatrendszer mellett egy sor más egyedi sajátossággal, követelménnyel is rendelkeznek ezen információs rendszerek, melyeket a program fejlesztői kénytelenek számításba venni, hogy egy jól működő, az igényeket kielégítő termékkel álljanak elő.

A következőkben összefoglaljuk mindazon követelményeket, melyeket egy, a vállalat működése szempontjából fontos információs rendszernek feltétlenül teljesítenie kell.

Nagymennyiségű adatok hatékony kezelése.
A felhasználónak elfogadható időn belül választ kell kapnia a feltett kérdéseire, a kiadott utasítások végrehajtásának nem szabad szokatlanul sokáig tartania. Egy bank-automatánál például nem tartanánk elfogadhatónak, ha percekig kellene várni az azonosításunkra, még akkor sem, ha tudjuk, hogy nagyon sok ügyfele van a banknak. A hatékonyság tehát egyrészt egyfajta *időbeli hatékonyságot* jelent. Másrészt az adatok helyszükségletének sem szabad feleslegesen megnőnie; úgymond kerülni kell a felesleges *redundanciát*, azaz egyazon adatelem többszöri, felesleges megismétlését. Mint majd később látható, a redundancia teljes megszűnése nem mindig kívánatos más, a helytakarékossággal szemben álló egyéb szempontok miatt (ilyen szempont lehet az időbeli hatékonyság időbeli hatékonyság, vagy az adatbiztonság adatbiztonság). Így egy *szabályozott redundancia* kialakítása a fejlesztési cél. A fejlesztőknek meg kell találniuk a hatékonyság különböző szempontjainak a helyes súlyozását, és optimalizálniuk kell az eredő hatékonyságot, hiszen ezek a szempontok gyakran egymással ellentétes lépéseket követelnek, egymás rovására teljesíthetők.

Konkurens hozzáférés támogatása.
A nyilvántartó és információs rendszerek zömében nem egyszemélyi rendszerek. Természetes használati módjuk, hogy egyidejűleg több felhasználó is használja, dolgozik vele. Ez a lehetőség viszont különös elővigyázatosságot igényel, hiszen a párhuzamos változtatások, műveletek, ha nincsenek összehangolva, akkor torz

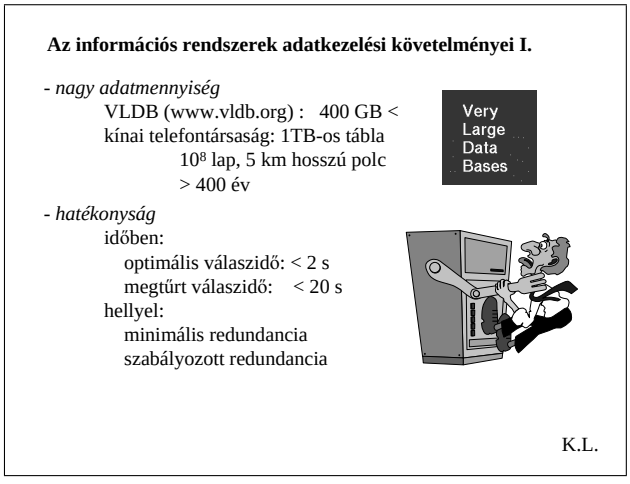
eredményeket szülhetnek, egymás hatásait kiolthatják vagy elferdíthetik. Ennek egyik szemléletes példája, az ún. '*lost update*' jelensége (1.4. ábra), vagyis amikor az egyik alkalmazásban végrehajtott adatérték-módosítás hatását törli egy másik alkalmazásban elvégzett módosítás. Ekkor úgymond elveszik, megsemmisül az első módosítás. Ez igen kényes hatással is járhat, ha mindez mondjuk egy bérügyi nyilvántartó rendszerben zajlik le, és az első program a rendszeres fizetést, a második program meg a rendkívüli jutalmakat teszi fel a számlára. Senki sem örülne az elveszett fizetésnek vagy jutalomnak. Az osztott erőforráshasználat problémája egyébként nemcsak itt jelentkezik, hasonló nehézséggel találkozhattunk az operációs rendszerek területén is. A probléma megoldásához nyilván kell tartani az elvégzett műveleteket, és gondoskodni kell a műveletek szabályozott sorrendben történő végrehajtásáról is.

A példában a B alkalmazás viszi fel a normál fizetés értékeket, és A a jutalmakat. Mindkét program a számla módosításához előbb kiolvassa a számla aktuális állását, majd a memóriában megnöveli a kiolvasott értéket, és végül vissza írja a módosított értéket a lemezre. Ha egy megadott számlát, melynek aktuális állása 2 egység, az A program 5 egységgel, míg a B program 3 egységgel növel meg, akkor a két program egymás utáni lefutása után 10 egységre változik a számlaállás.

Tegyük fel azonban, hogy a két program párhuzamosan fut, méghozzá oly módon, hogy az egyes műveletek végrehajtási sorrendje a következő:

$$r_A \rightarrow r_B \rightarrow u_A \rightarrow u_B \rightarrow w_A \rightarrow w_B.$$

Ekkor mindkét program a 2 értéket olvassa be a memóriába. Ezért az A program 7, a B program 5 értéket fog visszaírni a lemezre, ezáltal az utolsó írást követően csak 5 egység marad a számlán, és az A program által írt módosítást a B felülírja, kitörli. Így az A általi módosítás elveszik. Ezt a jelenséget természetesen, amennyire lehet, meg kell akadályozni.



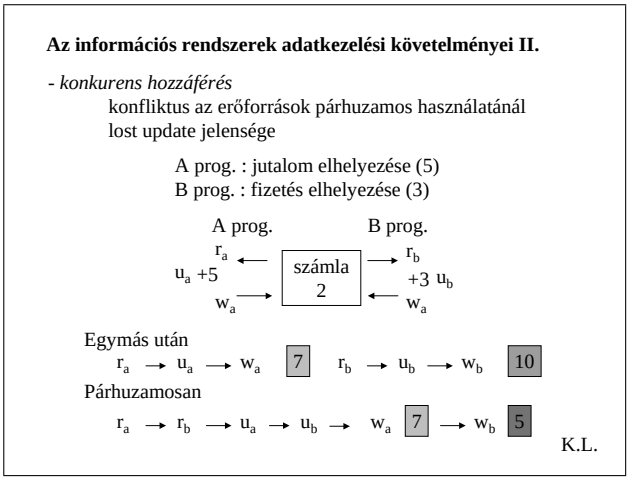
1.3. ábra. Információs rendszerek adatkezelési követelményei I.

Integritásőrzés.

A modellezett, programba leképzett valóság mindig rendelkezik belső törvényszerűségekkel. Egy ilyen szabály lehet például az, hogy minden embernek van születési ideje, vagy az, hogy az ember életkora nem lehet negatív értékű. A letárolt adatok helyessége, integritása alatt azt értjük, hogy az adatok minden megadott belső szabálynak megfelelnek. Az előbbi példát véve ismét; az nem fordulhat elő egy védett adatrendszerben, hogy egy életkor -13 év legyen. Természetesen a szabályok a példában megadottaknál sokkal bonyolultabbak is lehetnek, amikor az adatok közötti kapcsolatokat is vizsgálni kell. Erre vehetjük azt az esetet, amikor egy kereskedő csak akkor szállíthat, ha a kért áruból legalább egy megadott mennyiség a raktáron marad és a megrendelő minden előző számláját már kiegyenlítette. Az integritás megőrzéséhez a rendszernek nyilván kell tartania a szabályokat valamilyen hatékonyan kezelhető formában, majd minden művelet alkalmával ellenőriznie kell, hogy a kapott adathalmaz megfelel-e a letárolt szabályoknak. Mindezt olyan hatékonysággal, hogy a végrehajtási idő még az elviselhetőség határán belül maradjon. Mint majd a későbbiekben látni fogjuk, a szabályok lehetnek statikus jellegűek, amikor a megkötés az adatrendszerben tárolt adatértékekre vonatkozik, és lehetnek dinamikus jellegűek, amikor az elvégezhető műveletekre és azok sorrendiségére feltétel lett kijelölve.

Védelem.

A számítógépes információs vagy irányítási rendszereket a felhasználó a manuális tevékenységek kiváltására használja, hiszen ha mindent duplázva, kézzel és számítógéppel is végig kell vinni, akkor a munka hatékonysága nemhogy növekedne, inkább csökken. Ez azt is jelenti, hogy a felhasználó rábízza magát, azaz összes adatát a számítógépes rendszerre, vagyis a legtöbb adat csak ott tárolódik. A tárolt *adatok elvesztése* szinte pótolhatatlan veszteséget okozhat. Ezért a rendszernek fel kell készülnie, amennyire csak lehet a veszélyekre, az adathordozó meg-



1.4. ábra. Lost update jelensége

sérülésére vagy az operációs rendszer, a program összeomlására. Az adatvesztés elkerülésének fontosságát, az elveszett adatok visszaszerzésére irányuló igényt jól mutatja a Kürt cég sikertörténete is, mely többek között sérült adathordozók helyreállítására specializálódott. Az adatok sérülés elleni védelmére az adatokat lemásolják, és az elvégzett műveleteket naplózzák.

Az adatsérülés mellett a felhasználóra leselkedő másik veszélytípus az adatok illetéktelen személyekhez történő kerülése. Bizonyára akadnak olyanok, akik nagyon kíváncsiak lennének egy gyár termelési adataira, a technológiára, vagy éppen mások adataira, esetleg egy légvédelmi rendszer részleteire. A rendszernek tehát, ismét hasonlóan az operációs rendszerekhez, szabályoznia és ellenőriznie kell az *adat-hozzáféréseket*, különbséget kell tennie az egyes felhasználók között az elvégezhető műveletek tekintetében. Ehhez viszont nyilván kell tartani a jogosult felhasználókat, azok jogait és minden műveleti igény kiadásakor ellenőrizni kell, hogy elvégezhető-e a kért művelet. Ennek egyik ismert megoldása a hozzáférés monitoring rendszer, mely ellenőrzi és naplózza is az erőforrás-hozzáféréseket. A hozzáférés-védelem egyik elterjedt módszere a titkosítás. A kódolandó adatokat áttranszformálják kódolt adatokra, ahol a kódolás folyamata rendszerint egy vagy több paramétertől, kulcs értéktől is függ.

Hatékony programfejlesztés.

A rendszer kifejlesztési idejének lerövidítésére több oldalról is jelentkező nyomás hat. Egyrészt a szoftverpiacon folyó versenyben a rövidebb határidő előnyhöz juttathatja a versengőket, hiszen a felhasználó minél előbb szeretné kihasználni a rendszer által nyújtott előnyöket. Másrészről a gyorsaság bizonyos értelemben alapkövetelmény is, hiszen a rendszer mindig a valóság egy modelljének felel meg, és a modellezett valóság elég gyakran változik, például megváltoznak a szabályozók, a törvények. Egyik felhasználó sem kíván olyan rendszert megrendelni, mely használhatatlan lesz mire elkészül. A modellezett valóság változásai azonban

1.5. ábra. Információs rendszerek adatkezelési követelményei II.

előbb vagy utóbb mindenképpen kisebb vagy nagyobb mértékben bekövetkeznek. Viszont ezek a rendszerek már elég drágák ahhoz, hogy minden kisebb változtatás után a felhasználó egy újabb rendszert rendeljen meg. A rendszernek tehát elég rugalmasnak kell lennie, hogy a változtatások elvégezhetők legyenek. A *rugalmasság* és a *gyorsaság* igényei hatékony fejlesztőeszközök használatát teszik szükségessé a programfejlesztés során. A futtató környezetek sokfélesége felveti a szabványosság kérdését is. A szabványos eszközök használata ugyanis megkönnyíti az alkalmazások új platformra történő átültetését, és a fejlesztők egymás közötti kommunikációját, továbbá a fejlesztő rendszer elsajátítását is hatékonyabbá teszi.

Az előzőekben felsorolt szempontok és problémák tükrében talán már érzékelhető, hogy nem kis feladat például egy jó nyilvántartó rendszert kifejleszteni. Ha a rendszer adatkezelő részét emeljük csak ki, akkor is hatalmas munka lenne az igényeket kielégítő alkalmazás kifejlesztése az eddigiekben megismert programfejlesztő rendszerekkel, mint például a C programozási nyelvvel. Vegyük egy kicsit részletesebben, miként is járhatnánk el az adatrendszer C nyelvvel történő kifejlesztésénél, ezzel egy rövid áttekintést adva az adatkezelés legfontosabb alapfogalmairól is.

1.4. Áttekintés az adattárolás struktúrájáról

Az alapvető adattárolási mechanizmusok áttekintése során számos olyan fogalmat frissítünk fel, melyekre a későbbiekben, az adatbázisok tárgyalása során is szükség lesz majd. A felelevenítés mellett ezen áttekintés arra is rá kíván világítani, hogy milyen összetett és aprólékos feladat az adatok hatékony tárolási mechanizmusának megvalósítása.

Az adatkezelő rendszereknél a *permanens* adatok állnak az adatkezelés középpontjában, tehát azok az adatok, melyekre hosszú ideig, az alkalmazásból történő kilépés után is szükség van. A permanens adatok tárolására a háttértárolók szolgálnak, ahol az adatok állományokba szervezetten helyezkednek el. A nagy adatmennyiségből következően minden pillanatban az adatoknak csak egy töredéke fér el fizikailag a központi memóriában. Ha az állományon (fájl) belüli tárolás kérdését vizsgáljuk, akkor meg kell ismerni a lehetséges fájlszervezési módszereket és azok hatékonyságait az olyan alapvető műveleteknél, mint az

- adatelemek megkeresése, lekérdezése
- adatelemek bővítése, módosítása, törlése
- segédinformációk tárolása.

A leggyakoribb művelet a felsoroltak közül a *lekérdezés (query)*. Természetesen lehetnek olyan alkalmazások, ahol ez nem teljesül, mert például csak archiválni kell, és csak nagy ritkán van szükség visszakeresésre. Nézzük tehát milyen fizikai tárolási struktúrát válasszunk, ha a hatékony lekérdezésre koncentrálunk.

A hatékonyságnövelési lehetőségek keresését valójában már az adathordozó szintjén el kell kezdeni, hiszen a mágneslemezt véve, mint leggyakoribb adathordozót, az adatelem beolvasása a címének ismeretében három fő lépésből áll, melyekhez különböző időszükséglet rendelhető:

– *fejmozgatás*: a fejet a lemez megfelelő sávjára, cilinderére kell mozgatni;
– *fejkiválasztás*: lemezcsomag esetén a megfelelő lemez kijelölése;
– *forgatás*: az adott sávon belül a megfelelő szektor, blokk mozgatása a fej alá.

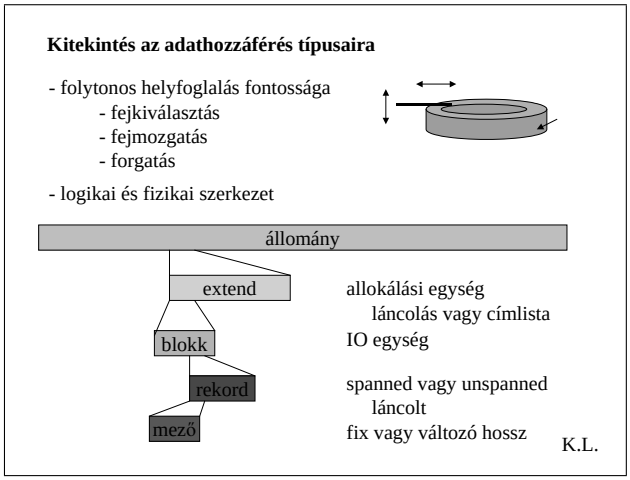
Ebből a három elemből a *fejmozgatás* igényli a legtöbb időt, átlagosan a teljes idő mintegy 80 százalékát. A fejmozgatáshoz szükséges idő azonban lényegesen függ attól, hogy hol helyezkedik el az olvasandó sáv, és mi volt az előzőleg felkeresett sáv. Minden sávváltás időszükséglettel jár, mégpedig minél nagyobb volt a távolság, annál több idő szükséges. Ebből számunkra két fontos megállapítás is levonható:

– az egymás után, együtt olvasott adatokat célszerű ugyanazon vagy szomszédos sávokra elhelyezni;
– egy adatelemnek, más programok véletlenszerű sávpozícióit feltételezve, az optimális elhelyezkedése a középső sávokban található.

Ha van befolyásunk az adatelemek elhelyezésére, akkor a fentiek figyelembe vételével kell dönteni.

A következő lépcsőfok a megfelelő *állomány szervezés* kiválasztása. Az állományokon belül az adatok *blokkokban* tárolódnak, ahol egy blokk egy adatátviteli egységnek fogható fel, azaz egy írás vagy olvasás művelete minimum egy blokk adatmennyiséget mozgat az adathordozó és a központi egység között. Az állományhoz tartozó és egymás után következő blokkok nyilvántartására is több különböző módszer létezik:

– *Blokkok láncolása*, azaz minden blokkban van egy mutató a következő blokkra. Az első blokk címének ismeretében sorban felkereshető az összes többi blokk.



1.6. ábra. Az adathozzáférés típusai

– *Blokk címlista*, azaz minden fájlhoz létezik egy lista, amely a hozzá tartozó blokkok címeit tartalmazza. Ez a lista lehet egyszerű, összefüggő, de lehet bonyolultabb felépítésű is.

A blokkok a fájlok fizikai elhelyezkedését mutatják, de emellett a fájl belső logikai struktúrával is rendelkezik. A logikai fájlszerkezet alapvetően lehet

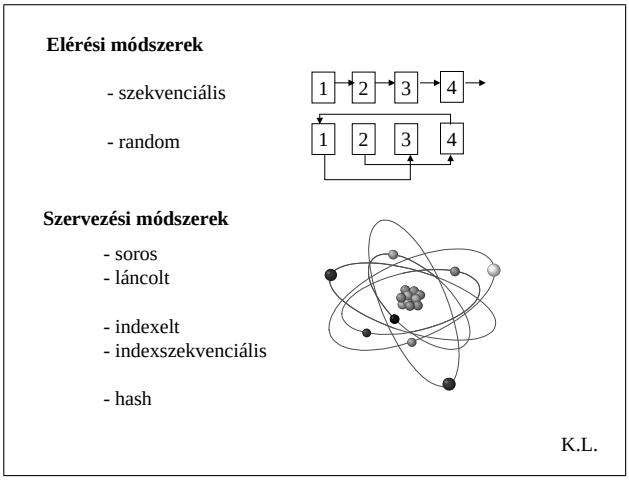
– stream jellegű vagy
– rekord jellegű.

A *stream* fájltípusnál a fájl belső struktúra nélküli byte vagy karakter sorozatból áll. A karaktersorozatot a felhasználó program értelmezi saját igénye szerint. Ugyanazt a fájlt a különböző programok másképp értelmezhetik. A rekord jellegű szerkezetnél feltesszük, hogy a fájl felbontható logikai részelemekre, úgynevezett rekordokra. A *rekord* logikailag összetartozó adatelemek, mezők összessége. Egy rekord jelentheti például egy alkalmazott összes adatát, és egy *mező* egy adatelem, mint például a név, beosztás vagy a fizetés. A rekord és a blokk viszonya alapján beszélhetünk

– spanned rekordokról, amikor egy rekord több blokkra is kiterjedhet, és
– unspanned rekordokról, amikor egy rekord csak egy blokkhoz tartozhat.

A rekord jellegű állományokban a rekordok lehetnek *fix hosszúak*, vagy *változó hosszúságúak*. Változó hosszúságú rekordok esetén a rekordok elhelyezkedését jelezheti

– rekordvég karakter,
– mutató a következő rekord elejére,
– blokklista a rekordokra hivatkozó mutatókkal.



1.7. ábra. Fájl-elérési és -szervezési módszerek

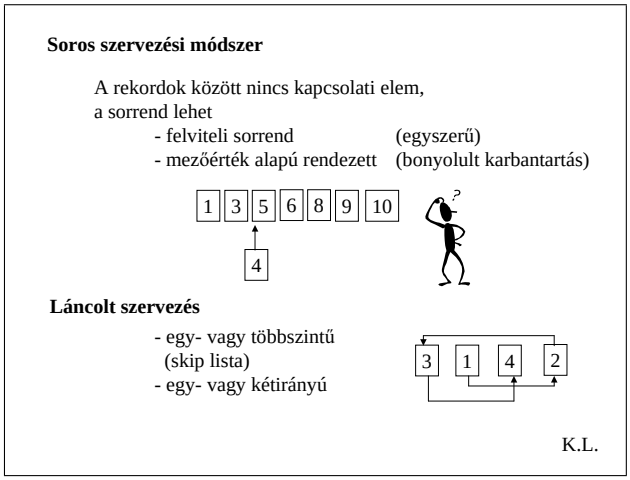
Ha az adatelemek gyors elérését célozzuk meg, akkor a belső struktúra nélküli stream fájlkezelés nem megfelelő, hiszen ebben az esetben csak a fájl soros átolvasásával találhatjuk meg a keresett elemet, ami azt is jelenti, hogy átlagosan a fájl felét át kell olvasni egy elem eléréséhez. A rekordjellegű megközelítés esetében a soros átolvasás mellett már más rekord elérési módszerek is alkalmazhatók. Az alkalmazható elérési módok:

- soros elérés,
- szekvenciális elérés,
- indexelt elérés,
- random, hashing elérés.

Az egyes elérési, szervezési módok közötti különbség megértéséhez szükség van egy újabb fogalom, a *rekordkulcs* megismerésére. A rekord mint már említettük, egy egyed több tulajdonságát tartalmazhatja. Ezen tulajdonságok között vannak olyanok, melyek több egyednél is ugyanazt azt értéket vehetik fel. A dolgozók adatait leíró rekordban például a születési hely, vagy a fizetés több dolgozónál, azaz több rekordban is lehet ugyanaz az érték. Vannak azonban olyan mezők, azaz az egyed olyan tulajdonságai, melyeknek egyedieknek kell lenniük. Ilyen tulajdonság lehet például a dolgozó törzsszáma, személyi száma. Az ilyen tulajdonságot, vagy tulajdonságcsoportot, mely egyedisége révén alkalmas az egyed, azaz a rekord egyértelmű azonosítására, *rekordkulcsnak* vagy röviden kulcsnak nevezzük.

Az adatelemek keresésénél kiemelt fontosságúak lesznek azok az esetek, amikor a keresés a kulcsra vonatkozik, mivel ez a rekordok egy természetes keresési módját jelenti. Látható, hogy ekkor nem a pozíció alapján keresünk, mint ahogy az a tömbök esetében megszokott volt, hanem érték alapján.

Az egyes fájlszervezési módok az alábbiakban foglalhatók össze.



1.8. ábra. Fájlszervezési módok I.

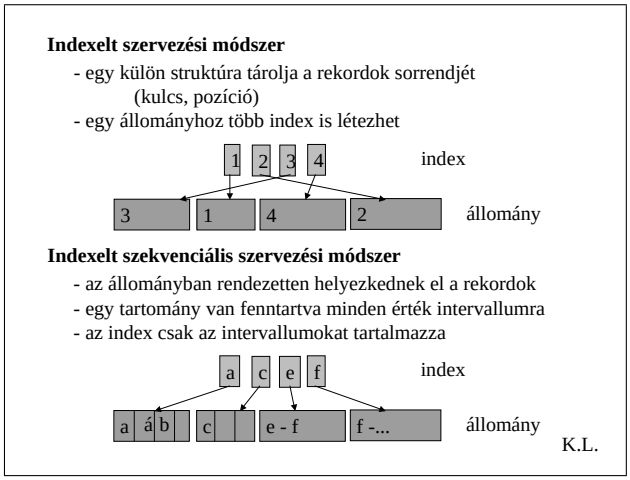
Soros elérés.
A rekordok a fájlban tetszőleges sorrendben, például a felvitel sorrendjében helyezkednek el, azaz nincs kapcsolat a rekord kulcsértéke és a rekord fájlon belüli pozíciója között. Mivel ekkor egy adott kulcsértékű rekord bárhol elhelyezkedhet a fájlban, a kereséskor a fájl minden rekordját át kell nézni egymás után a fájl elejétől kezdve, amíg meg nem találjuk a keresett rekordot. Ez átlagosan a fájlban tárolt rekordok felének átnézését igényli, ezért az egyedi keresések szempontjából nem a legjobb módszer. Természetesen ha a keresésnél szükség van az összes rekordra, akkor ez a fájlszervezés lesz a leghatékonyabb.

Szekvenciális, kulcs szerint rendezett elérés.
A rekordok a fájlban a kulcsértékeik alapján sorba rendezve helyezkednek el, pontosabban a rekordok a fájlban a kulcsértékeik növekvő vagy csökkenő sorrendjében érhetők el. A szekvencia előnye, hogy nem szükséges a teljes fájlt végignézni adott kulcsértékű rekord keresésekor, mivel a sorrendbe rendezés miatt egy adott kulcstól jobbra csak tőle nagyobb (növekvő rendezést feltételezve) kulcsértékű elemek helyezkedhetnek el. Ez a sorrendbe rendezés megvalósítható fizikai szekvenciával vagy logikai, láncolt szekvenciával. A fizikai szekvenciánál a rekordok fizikai helye megfelel a sorrendben elfoglalt helyének. Ezáltal gyors lesz az egymást követő rekordok elérése, de egy új rekord beszúrása esetén át kell rendezni a rekordokat a fájlon belül; egyes rekordokat át kell vinni más blokkokba, hogy helyet biztosítsunk a beszúrandó rekordnak. Gyakran változó fájl esetén tehát nem javasolt ez a módszer. A logikai szekvencia esetén a rekordok a bevitelük sorrendjében helyezkednek el fizikailag a fájlban, és a sorrend szerinti rendezettséget mutatók segítségével valósítják meg, azaz minden rekord tartalmaz egy mutatót a sorrendben őt követő rekordra. Így beszúráskor csak a mutatókat kell átrendezni, a rekordok fizikai pozíciója ugyanaz marad. Mivel mind a két esetben továbbra is a fájl elejéről kiindulva, az egymást követő elemek ellenőrzésével lehet keresni, ez a módszer sem igazán hatékony számunkra.

Indexelt struktúra.
A keresés meggyorsítható lenne, ha nem kellene minden, a keresett rekordot megelőző rekordot fizikailag is átolvasni. Valójában a keresett rekordot megelőző rekordokból csak azok kulcsértékei fontosak számunkra, a rekord többi mezője lényegtelen. Mivel rendszerint a kulcsmező nagysága csak egy töredéke a teljes rekord méretének, ezért ezen redukált adatokat sokkal gyorsabban át lehetne olvasni, sőt egy minőségi javulást hozna, ha az átolvasandó sor elhelyezhető lenne a memóriában, ugyanis ekkor az egymás utáni elembeolvasások helyett egy sokkal gyorsabb keresési módszer, a bináris keresés is alkalmazható lenne. Mindennek megvalósítása az *index szerkezet*, mely egy külön listában tartalmazza a rekordok kulcsait és az elérésükhöz szükséges mutatókat. A legegyszerűbb index-szerkezet az indexlista, mely az összes rekord kulcsát tartalmazza egy listában, ahol a kulcsértékek rendezetten helyezkednek el. Mivel nagy fájlméreteknél az indexlista is olyan hosszú lehet, hogy már nem fér egyszerre a memóriába, a lista kezelhetővé tételére új megoldásokat kerestek. Ennek egyik módszere az *indexszekvenciális* fájlszerkezet, az ISAM szerkezet, melyben a rekordok fizikailag is rendezetten helyezkednek el a fájlban, mint a szekvenciális állományoknál, így az indexlistának

nem szükséges minden elemet tartalmaznia, csak bizonyos jelző rekordokat, mondjuk minden k.-at, és a rekord keresését a fájlban a hozzá legközelebb eső, tőle kisebb kulcsértékű jelzőrekordtól kell csak kezdeni. Az állományban történő szekvenciális keresés sem tarthat sokáig, hiszen maximum k rekordot kell egymásután átnézni. Az ISAM fejlettebb módozatainál külön van cilinder (sáv) és blokk-index, mely a fejmozgás csökkentése érdekében ugyanazon cilinderen helyezkedik el mint a blokk, a rekord. Ez a megoldás igazán akkor hatékony, ha a cilinder index a központi memóriában helyezkedhet el.

A másik lehetséges, elterjedt módszer a többszintű, *hierarchikus indexstruktúra* bevezetése, melyben a felül elhelyezkedő listából nem közvetlenül a rekordokra, hanem újabb indexlistákra történik hivatkozás. A hierarchikus indexlisták között kiemelkedő szerepet játszik a *B-fa*. A B-fa előnye, hogy minden, közvetlenül a rekordokra mutató listája a fa (a levél listák), azaz a hierarchia azonos szintjén helyezkedik el. Ez azt is jelenti, hogy minden rekordot közel azonos idő alatt érhetünk el, ezért az indexszerkezet kiegyensúlyozottnak mondható. Az indexlisták kihasználtsága is jónak mondható, hiszen minden lista, az elsőt kivéve, kapacitásának legalább a felét kihasználja. A keresés elve megfelel a keresőfák megszokott algoritmusának. A fa új elemek beszúrásakor sajátosan, lentről felfelé bővül. Előbb megkeresik azt a levél listát, ahol lenni kellene a bejegyzésnek. Ha van még itt hely, beszúrják ide a bejegyzést a sorrend szerinti helyre. Ha nincs hely, akkor megkeresik a helyét, majd a középső elemet kiemelik ebből a csomópontból, feltolják a szülő listába, majd létrehoznak egy újabb csomópontot, melybe a túlcsordult csomópontból a kiemelt elemtől nagyobb kulcsbejegyzések kerülnek át, és a túlcsordult csomópontban csak a kiemelt elemtől kisebb bejegyzések maradnak meg. Mivel a szülő lista is túlcsordulhat ez a művelet egy rekurzív folyamatot eredményez. A listaméretet úgy választják meg, hogy az még beleférjen a memóriába. Mivel így rendszerint sok bejegyzés van egy listában, a fa mélysége, a



1.9. ábra. Fájlszervezési módok II.

szintek száma, alacsonyan tartható, ezért néhány blokk olvasása elegendő a rekord megtalálásához. A szintek száma egyébként csak logaritmikusan nő a rekordszám függvényében.

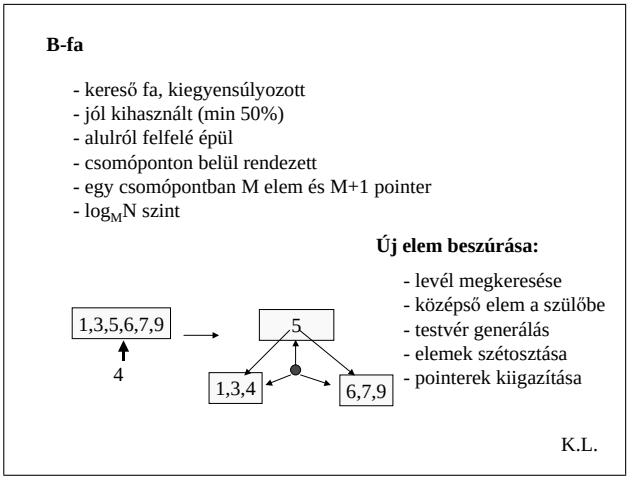
Ez az indexelési módszer tekinthető az egyik leghatékonyabb módszernek. Az indexelés feladatát még annyi bonyolíthatja, hogy esetleg több mező szerint kívánunk indexelni, vagy éppen nem kulcs mező szerint kívánjuk az indexelést elvégezni a lekérdezések jellegét figyelembe véve. Ha egy mezőnél egy érték több rekordban is előfordul, akkor nem szokás minden ilyen értéket külön felvenni az indexlistába, hanem csak egyet, az első előfordulást, és a többi rekord ebből a rekordból kiinduló láncolt listán keresztül érhető el.

Ha több mező szerint kívánunk indexelni, és az egyes indexeket valamilyen módon szeretnénk összevonni, illeszteni, akkor kapjuk a többdimenziós indexszerkezeteket. Ennek egyik lehetséges megvalósítása a *többdimenziós indexfa*, melyben az egyes szintek ciklikusan az egyes mezőkhöz vannak rendelve. A keresés elve pedig megegyezik az egydimenziós keresőfa elvével, azzal a különbséggel, hogy minden szinten más-más mező szerint történik az elágazás.

Hashing.

Az indexszerkezetek révén nagyon gyorsan meghatározható a rekordok pozíciója, csupán az indexlistákat kell átolvasni. Még jobb lenne, ha sikerülne még ezt a munkát is megspórolni. Erre irányulnak a hashing elérési módszerek, amikor a rekord pozícióját közvetlenül a rekord kulcsértékéből határozzák meg, tehát csak egy blokkolvasásra van szükség a rekord eléréséhez. Sajnos azonban nem mindig igaz, hogy egy blokkolvasás elegendő, ez csak ideális esetben valósul meg.

A hash elérési módszer alapelve, hogy a kulcs értékéből valamilyen egyszerűbb eljárással, egy *h() hash függvényt* alkalmazva meghatároznak egy pozíciót. Numerikus kulcsok esetén a $h(x) = x \ mod \ n$ egy szokásos hash függvény, ahol x a kulcs érték és n a hash tábla rekeszeinek a darabszáma. A $h(x)$ megadja, hogy



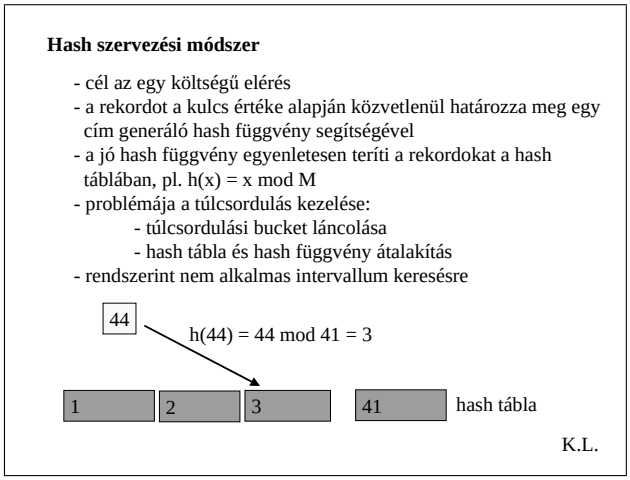
1.10. ábra. B-fa

mely rekeszbe tegyük le az x kulcsú elemet. Mivel a hatékony kezelhetőség végett a lehetséges pozíciók darabszáma lényegesen kisebb a lehetséges kulcs értékek darabszámánál, így szükségszerűen több kulcsérték is ugyanazon címre fog leképződni. Egy címhez rendelt tárterületet szokás *bucket*-nak is nevezni, ami lemez esetében rendszerint egy blokknak felel meg. Ha több rekord kerül egy címre, mint amennyi egy bucket-ban elfér, akkor lép fel a *túlcsordulás* jelensége, amikor is egy újabb blokkot, területet kell a címhez hozzákötni. Tehát egy címhez több különálló terület is tartozhat, melyeket láncolással kötnek össze.

Láncolás esetén a rekord kereséséhez több blokkot is át kell nézni, amely lényegesen csökkenti a hash elérési módszer hatékonyságát. A túlcsordulás mellett a hash módszer másik hátránya, hogy csak nagyon körülményesen lehet vele megvalósítani a rekordok kulcs szerint rendezett listájának előállítását, hiszen a hash módszer az egymást követő rekordokat tetszőlegesen szétszórhatja a címtartományon a kiválasztott hash függvénytől függően. A jó hash függvény a túlcsordulást a rekordok egyenletes elosztásával tudja kivédeni. Mivel a rekordokhoz rendelt címek eloszlása nagyban függ a kulcsértékek eloszlásától, a túlcsordulás soha sem védhető ki teljesen.

A fájlszerkezetek rövid áttekintése is világosan érzékelteti, hogy mennyi mindent kell figyelembe venni és milyen összetett lehet az optimális szerkezet megválasztása. Pedig ez a rész csak egy kis töredéke az optimális adatkezelő program tervezése során felmerülő problémáknak, hiszen nem említettük a többi komoly kérdést, mint például a konkurens hozzáférés, a védelem, vagy az integritásőrzés.

Talán érzékelhető, hogy milyen reménytelen vállalkozásnak tűnik, megfelelő fejlesztőkapacitás és idő hiányában, a minden igényt kielégítő információs, adatfeldolgozó rendszerek nulláról induló, hagyományos, például C nyelven történő kifejlesztése. Ez olyan mennyiségű munka, amit csak a legnagyobb szoftverfejlesztő cégek tudnak elvégezni. Mi marad hát a többieknek?



1.11. ábra. A hash fájlszervezési mód

1.5. Adatbázisrendszerek, adatbázis és adatbáziskezelő fogalmai

A világon nagyon sok és egymástól nagyon különbözően működő információs rendszer létezik. A különbözőség ellenére azonban az is észrevehető, hogy mindegyikben az adatok kezelése szinte ugyanolyan módon, funkciókkal zajlik. A szokásos funkciók közé tartozik az adatok felvitele, törlése, módosítása és lekérdezése. Az adatok formálisan azonos kezelése tette azt lehetővé, hogy a nagyobb cégek előállítsanak olyan keretrendszereket, amelyek beépíthetők a legkülönbözőbb információs rendszerekbe az adatok karbantartására. A nagytömegű adatok feltételeknek eleget tevő kezelését biztosító rendszereket *adatbáziskezelő* rendszereknek nevezik. Maga az adatbáziskezelés fogalma sem egy egzaktul definiált fogalom, így a mostani értelmezést egy bevezető értelmezésnek tekinthetjük, melyet a későbbiekben még pontosítani fogunk.

Természetesen az adatbáziskezelő rendszerek sem hirtelen, minden előzmény nélkül jelentek meg a piacon, a köztudatban. A számítógépeket megjelenésük után a tárak kis kapacitása miatt elsősorban numerikus számítások elvégzésére használták. Később a technológia fejlődésével mind nagyobb mennyiségű információ tárolására váltak alkalmassá, és megjelentek a kimondottan nagy mennyiségű adatok hatékony kezelésére készült rendszerek is. Az első szekvenciális fájlok még az 1940-es évek végén jelentek meg. Az első nem szekvenciális hozzáférést biztosító fájlrendszert 1959-ben fejlesztették ki az IBM-nél. Az 1960-as években egy sor új, harmadik generációsnak nevezett programozási nyelv jelent meg, mint a Fortran, Basic, PLI, melyek között volt egy, amely kimondottan adatkezelés orientált céllal jött létre, a Cobol. Egyes statisztikák szerint még pár évvel ezelőtt is az alkalmazások többsége ezen a nyelven készült, megelőzve a C, C++ nyelvet is, melyeket inkább rendszerfejlesztésre használnak. Nem sokkal ezután megjelentek az első adatbáziskezelő rendszerek is. Az 1961-es évben dolgozták ki a *hálós adatmodell* alapjait, majd nem sokkal rá megjelent a *hierarchikus adatmodell* is. Az első hálózatos, konkurens hozzáférést biztosító adatbank 1965-ben jelent meg az IBM-nél, és a SABRE nevet kapta. Az adatbáziskezelő rendszerek maguk is jelentős fejlődésen mentek keresztül azóta; jelentősen megváltozott a használati módjuk, az általuk támogatott adatmodell jellege. Az induló időszak hierarchikus, majd hálós adatmodelljei után az 1970-es években indult el hódító útjára a ma legelterjedtebb adatbáziskezelő típus, a *relációs adatbáziskezelés*. Az adatbázisokkal kapcsolatos elméleti kutatások is megszaporodtak, az 1970-es években indultak be a VLDB és a SIGMOD konferenciák. Az 1980-as években a relációs adatbáziskezelők SQL kezelő felülete is szabvánnyá vált, és megjelentek a relációs adatbázist kezelő alkalmazások hatékony fejlesztését szolgáló negyedik generációs, *4GL rendszerek* is. Évtizedünkben az adatbáziskezelés területén is tért hódítanak az új elvek, mint az objektum orientáltság vagy a logikai programozás, és a hálózatok elterjedésével az osztott adatbáziskezelők szerepe is egyre nő. Emellett napjainkban egyre nagyobb szerepet kapnak az ismertetett adatszerű információkezeléstől eltérő felépítésű és funkciójú, szövegszerű kezelést megvalósító információs rendszerek is, melyek tágabb értelemben kapcsolhatók az adatbáziskezelés területéhez.

E rövid kis történelmi áttekintés után nézzük meg most már pontosabban, mit értünk adatbáziskezelés alatt, és mik az ide csatlakozó legfontosabb fogalmak.

Első alapvető fogalmunk az *adatbázis* fogalma. A fogalom definícióját az adatbázisokhoz rendelhető legfontosabb tulajdonságok megadásával írhatjuk le. Ha az irodalomban utánanézünk, hamar rájövünk azonban, hogy nincs egy egységesen elfogadott definíció az adatbázis fogalmára, úgymond mindenki szabadon értelmezheti, hogy mit érez fontosnak kiemelni az adatbázis fogalmából. Ezek a definíciók szerencsére, néhány kivételtől eltekintve, nem mondanak ellent egymásnak, inkább más-más aspektust hangsúlyoznak. A sokszínűség bemutatására előbb következzék néhány válogatás a lehetséges, megadott definíciókból, majd megadjuk a saját definíciónkat is, ezzel is bővítve a rendelkezésre álló választékot.

Elsőként egy olyan példa következzék, melyet nem ajánlok elfogadásra, annak túl általános volta miatt:

> *Az adatbank rekordok összessége. ('Eine Datenbank ist eine Sammlung von Datensätzen', részlet egy Works leírásból.)*

Ez a definíció nem tesz különbséget a normál fájl és egy adatbázis között, pedig nem minden fájl tekinthető adatbázisnak.

Az Oxford értelmező szótár megfogalmazása sem igazán elfogadható számunkra:

> *Adatbázis: Általában és szigorúan véve olyan adatállomány (data file), amely egy adatbáziskezelő rendszerrel hozható létre és érhető el.*

Ez lényegében áttolja a definíciót az adatbáziskezelőre, másrészt ebből úgy tűnhet, hogy minden fájl külön adatbázis, pedig mint látjuk ez nem igaz, több fájl együtt fog sok esetben egy adatbázist jelenteni.

A következőkben már tankönyvekből vett definíciókat olvashatunk.

> *Az adatbázis összetartozó és kapcsolódó adatok rendszere. (Elmasri – Navathe)*

A fenti definíció túl általánosan fogalmaz, így átöleli az adatkezelés szinte teljes területét, ezért önmagában nem fogadható el. A szerzők maguk is pontosítják az értelmezést a definíció után egy tulajdonságlistával. A definíció viszont nagyon helyesen rámutat arra, hogy az adatbázis magja az adatok és a közöttük fennálló kapcsolatok együttes tárolása.

> *Adatbázisokon voltaképpen adatoknak kapcsolataikkal együtt való ábrázolását, tárolását értjük. (Horváth Katalin – Dr. Szelezsán János)*

Itt már láthatóan nem a formai megjelenést emelték ki, hanem helyesen a belső tartalmi vonatkozás kerül előtérbe. Számunkra leglényegesebb mondanivalója ennek a definíciónak, hogy a valóság modellezésénél nem elegendő pusztán csak az egyedeket letárolni, hanem az egyedek között fennálló kapcsolatok nyilvántartása is fontos. Mit érne egy olyan rendőrségi nyilvántartás, melyben mind az autók, mind az állampolgárok adatai benne vannak, de a rendszer nem tárolná, hogy melyik autó kinek a tulajdona. A kapcsolatok a valóságmodell szerves részei, és az adatbázisnak ezen kapcsolatokat is tárolnia kell.

Még egy kicsit többet mond a következő definíció:

> *Az adatbázis véges számú egyed-előfordulásnak, azok egyenként is véges számú tulajdonságértékének és kapcsolat-előfordulásainak az adatmodell szerint szervezett együttese. (Dr. Halassy Béla)*

E megfogalmazás lényeges és új eleme, hogy az adatok azért nem tetszőleges formátumban tárolódnak, hanem minden adatbázisnak van egy belső logikai struktúrája, melybe be kell illeszkednie minden tárolt adatnak. Ilyen struktúrára, vagy adatmodellre több példát is láttunk a történelmi áttekintésben, ahol például a relációs adatmodell is szerepelt. Mint majd később látni fogjuk, ez elsősorban nem fizikai struktúrát jelent, hanem logikait.

A következő definíció a neves amerikai szakértőtől származik:

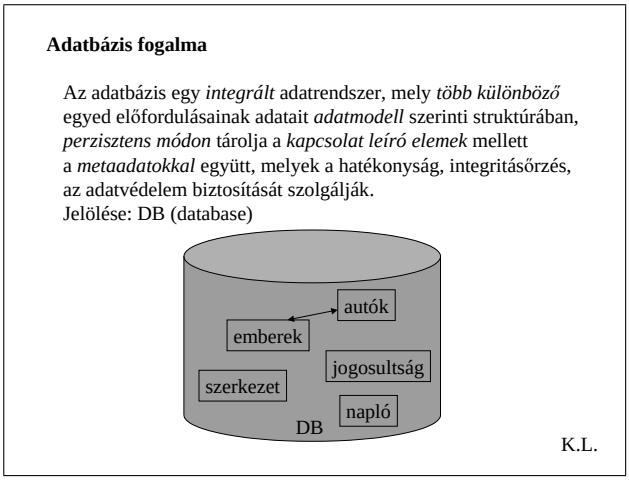
> *Az adatbázis a felhasználók által rugalmasan kezelhető adatok rendszere. (C.J. Date)*

A definíció lényeges mondanivalója, hogy az adatbázisban tárolt adatokat többen, viszonylag rugalmas keretek között, tehát nem csak megszabott módon, használhatják.

Végül az utolsó példa is hasonló elemeket emel ki:

> *Az adatbázis összetartozó adatok azon rendszere, mely megosztott több felhasználó között, és az elérést egy központi vezérlő program szabályozza, és a felhasználónak nem kell ismernie az adatok fizikai tárolási mechanizmusát. (J. G. Hughes)*

Itt is történik utalás a logikai modell és a fizikai tárolás különbségére, valamint a konkurens hozzáférésre.



1.12. ábra. Adatbázis fogalma

Az előző példák és a bevezetőben említettek alapján felállíthatunk egy összesítő definíciót az adatbázis fogalmára. Mindenekelőtt nézzük meg, mit kell tárolni az adatbázisban. Nyilvánvaló, hogy a modellezett valóságban szereplő egyedeknek és kapcsolataiknak szerepelniük kell. A felhasználó ezekkel az adatokkal fog dolgozni, ezen adatok kezelésére készülnek a különböző felhasználói programok. Ezeket az adatokat szokás tényleges, elsődleges adatoknak is nevezni. Az adatkezeléssel szemben felállított követelmények kielégítéséhez ezen adatok önmagukban nem elegendőek, gondoljunk csak arra, hogy a hatékony adatkeresés indexstruktúrát vagy hash szerkezetet igényel, vagy például az adatvédelem biztosításához szükség van a hozzáférési jogok tárolására és az adatmásolatok megőrzésére. Ezek a szerkezetek az elsődleges adatokra vonatkozó információkat tárolnak, ezért nevezik ezen adatokat metaadatoknak, tehát adatokra vonatkozó adatoknak. Következzék tehát a definíció:

> *Adatbázis: egy olyan integrált adatszerkezet, mely több különböző objektum előfordulási adatait adatmodell szerint szervezetten perzisztens módon tárolja olyan segédinformációkkal, ún. metaadatokkal együtt, melyek a hatékonyság, integritásőrzés, adatvédelem biztosítását szolgálják. Az adatbázis szó rövidítésére gyakran használják az angol rövidítését, a DB-t.*

Az adatbázisok elvileg tetszőleges méretűek lehetnek. Az elsődleges adatok száma nullától, az üres adatbázistól, a végtelen értékig terjedhet. Az elméletileg végtelen kapacitást a gyakorlatban a rendelkezésre álló hely, vagy éppen a belső tárolási struktúra korlátozza.

Az adatbázis, mint a fentiekből kitűnik, egy összetett adatstruktúrának tekinthető. Az adatstruktúra viszont az alkalmazások passzív elemeit jelenti, és kell egy algoritmus, egy program, amellyel felhasználhatók ezek az adatok, életre kelthetők az információk. Így az adatbázishoz kapcsolódnia kell egy kezelő programnak, amit *adatbáziskezelőnek* neveznek. Az adatbáziskezelő rendszer értelmezése jóval egységesebb, mint az adatbázis értelmezése volt. Egy általánosan elfogadott definíciónak tekinthető a Codd által megadott értelmezés, mely szerint

> *az adatbáziskezelő rendszer az a program, mely az adatbázishoz történő mindennemű hozzáférés kezelésére szolgál.*

Forsthuber anyagában egy részletesebb felsorolás is található, hogy milyen feladatok elvégzésére szolgál az adatbáziskezelő rendszer, nevezetesen

- adatbázisok létrehozására
- adatbázisok tartalmának definiálására
- adatok letárolására
- adatok lekérdezésére
- adatok védelmére
- adatok titkosítására
- hozzáférési jogok kezelésére
- fizikai adatszerkezet szervezésére.

A felsorolás alapján érzékelhető, hogy a hozzáférés nem egy egyszerű írási vagy olvasási műveletet jelent, hiszen az adatbáziskezelő rendszernek kell gondoskodnia

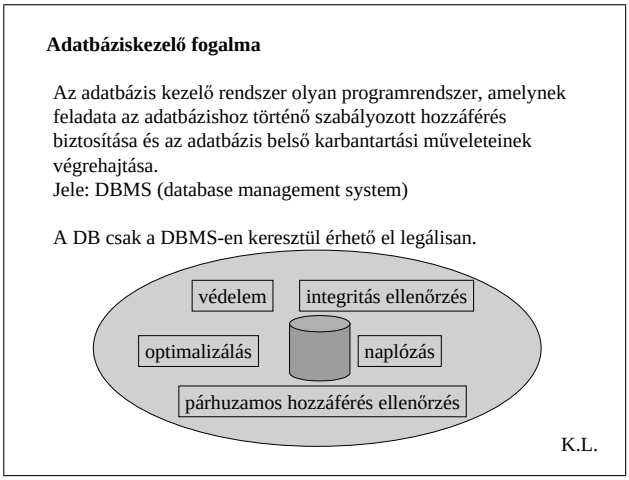
a már korábban említett integritási, hatékonysági és védelmi feltételek megőrzéséről. Az adatbáziskezelő rendszer emiatt egy bonyolult programrendszernek tekinthető, mely sok funkcióját, összetettségét tekintve leginkább az operációs rendszerekhez hasonlítható. Az integritási, hatékonysági és védelmi feltételek ellenőrzését és betartatását az adatbáziskezelő rendszer a háttérben végzi el, mintegy a felhasználó közvetlen utasítása vagy éppen tudta nélkül. Mindez azért történik így, hogy a felhasználó véletlenül vagy szándékosan se tudja elrontani az adatbázist. Az adatbázis helyessége megőrzésének fontossága miatt definíciónkban külön kiemeljük az adatbáziskezelő rendszer ezen tulajdonságát:

> *Adatbáziskezelő rendszer: Az a programrendszer, melynek feladata az adatbázishoz történő hozzáférések biztosítása és az adatbázis belső karbantartási funkcióinak végrehajtása. Az adatbáziskezelő rendszer rövidítése az angol elnevezés alapján: DBMS.*

A DBMS és az operációs rendszer hasonlata annyiban is helytálló, hogy mindkettő egy alsó szoftverréteget valósít meg, amit a felhasználó nem közvetlenül, hanem segédprogramokon keresztül ér el. Az adatbáziskezelés esetében is a felhasználó nem közvetlenül a DBMS-t kezeli, hanem egyéb segéd- és alkalmazói programokat futtat, melyek majd a DBMS-en keresztül érik el az adatbázisban tárolt adatokat. Maguk a DBMS rendszereket forgalmazó cégek is készítenek ilyen segédprogramokat, de egyedi fejlesztéssel is létrehozhatunk adatbázisbeli adatokat kezelő programokat. A DBMS-hez integráltan tartozó segédprogramok angol rövidítése UIT (User Interface Tools).

Ezek alapján egy hatékony adatkezelő rendszernek tartalmaznia kell egy adatbázist, egy adatbáziskezelő rendszert, valamint alkalmazói és segédprogramokat.

> *Az adatbázis, az adatbáziskezelő rendszer, valamint az alkalmazói és segédprogramok együttesét adatbázisrendszernek nevezik, melynek rövidítésére a DBS angol betűszót használják.*



1.13. ábra. Adatbáziskezelő rendszer fogalma

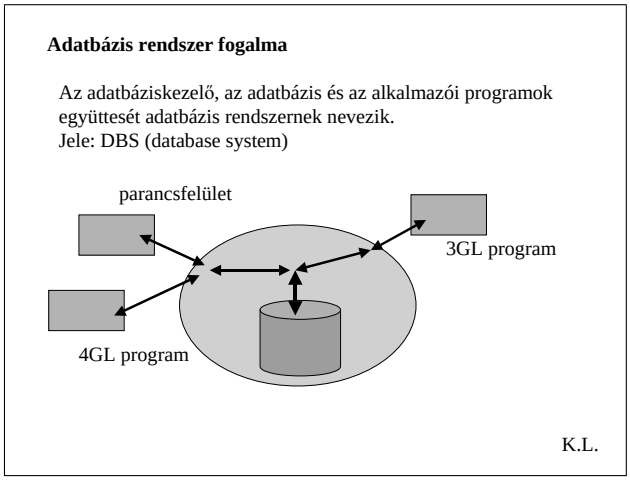
A DBS-en belül az alkalmazói és a segédprogramok állnak legközelebb a felhasználóhoz. A felhasználó ezzel a komponenssel kommunikál. A kiadott utasítások értelmezése után az adatkezelésre vonatkozó részlépéseket a program a DBMS felé továbbítja. Ezekután a DBMS elvégzi a megfelelő adatbázis módosításokat vagy adatbázis olvasási műveleteket, és az eredményt továbbítja az alkalmazói program felé.

A segédprogramok között kiemelt helyen szerepelnek a felhasználók egy szűk, kiemelt csoportjának készült programok. Ez a csoport annyiban játszik kiemelt szerepet, hogy az ő feladatuk az adatbázis menedzselése. Ehhez kiemelt jogosultságokkal rendelkeznek, és csak ezekkel a jogosultságokkal lehet számos adminisztrációs műveletet elvégezni, mint például a fizikai tárolási struktúra szabályozása, jogosultságok kezelése. Az ezen funkciók elvégzésére alkalmas segédprogramok természetesen megfelelő módon, például jelszóval védettek a jogosulatlan hozzáférés ellen. Ezen jogokkal felruházott felhasználókat nevezik *adatbázis adminisztrátoroknak* (rövidítése DBA).

Az adminisztrátorok mellett dolgoznak az *operátorok*, feladatuk a rutinszerű rendszertevékenységek elvégzése, mint például a mentések, rendszerindítások vagy zárások végrehajtása.

A felhasználóknak egy tágabb csoportja az *alkalmazásfejlesztők* köre. A fejlesztőknek a DB oldaláról ismernie kell az adatbázis adatmodelljét, a metaadatok megadásának módját, az alkalmazói programok fejlesztésének lehetőségeit, hogy csak a legfontosabb követelményeket említsük. A fejlesztők egyik csoportja az *adatbázis tervezők,* akik az adatmodell megtervezésével foglalkoznak, míg a másik csoportjának a felhasználói programok megírása, tesztelése a feladata.

A felhasználók legnépesebb csoportja az *alkalmazók* köre, akik az elkészült alkalmazásokat használják, számukra az adatbázis azon adatokat jelenti, amelyekkel az alkalmazások során találkoznak, az adatbázis vagy a DBMS létezéséről, vagy netán belső működéséről semmilyen ismerettel sem kell rendelkezniük.



1.14. ábra. Adatbázis rendszer fogalma

Az adatbázisrendszer mellett az adatkezelés másik fő variánsa a hagyományos fájlkezelő rendszer. Ez a fajta adatkezelési technika is elterjedt, használata bizonyos esetekben előnyösebbnek bizonyul a DBS alkalmazásánál. A rendszerfejlesztő egyik fontos feladata a megfelelő adatkezelési módszer kiválasztása, annak eldöntése, hogy mikor érdemes DBS-t és mikor fájlkezelő rendszert használni. A döntés helyes meghozatalához ismerni kell mindkét rendszer előnyeit és hátrányait is. A DBS-t illetően - részben az előzőekben ismertetett fogalmakon alapulva - az alábbi előnyök emelhetők ki:

Az egyedtulajdonságok, kapcsolatok és metaadatok egységes tárolási rendszere.
Az adatbázis nem egy speciális alkalmazói programhoz készült, hanem tetszőlegesen sokfajta alkalmazói program is futhat rajta, több alkalmazói program adatait is összefogja. Ezért szokás az adatbázisban tárolt adatokat integrált adatoknak is nevezni. A normál fájlkezelésnél ezzel szemben rendszerint minden alkalmazói programhoz el kell készíteni a saját adatrendszerét.

Adatfüggetlenség.
Az adatfüggetlenség kérdése az egyik legfontosabb jellemzője az adatkezelés fejlődésének. Az 1950-es évek elején megírt adatkezelő programok egyik jellemzője volt, hogy a programkód teljes mértékben tükrözte az adatok tárolásának szerkezetét, hiszen a program szinte közvetlenül elérhette az adatokat. Ezt a fejlettségi szintet nevezik a teljes adatfüggőség korának. Ez azt is jelentette, hogy ha megváltozott a fizikai struktúra, akkor át kellett írni a programot is. Mivel a hardver gyors fejlődése miatt erre gyakran szükség volt, a fejlődés következő lépcsőjében beépült egy fájlkezelő rendszer az operációs rendszerbe, és az alkalmazás már egy átdolgozott, áttranszformált képet kapott a fájlszerkezetről. A programban egy logikai adatszerkezetet kell csak definiálni, a fizikai szerkezetre való leképzést a konverter végzi.

A *fizikai adatfüggetlenség* azt jelenti, hogy a fizikai adatszerkezet, az elérési mód megváltoztatható anélkül, hogy a programot is módosítani kellene. A mai hagyományos fájlkezelésnél rekordszintű függetlenség valósul meg, hiszen az állományt sima rekordsorozatnak képzelhetjük el, pedig valójában nem az; ezzel szemben a rekord mezőit úgy kell megadni, ahogy azok fizikailag is megvalósulnak. Az adatfüggetlenség következő, az adatbázisokban megvalósuló szintje a mezőszintű fizikai adatfüggetlenség, hiszen az adatbázisban az egy rekordba tartozó mezők is fizikailag szétszórtan helyezkedhetnek el az adathordozón.

Az adatfüggetlenség másik típusát *logikai adatfüggetlenségnek* nevezik, mely szintén megvalósul az adatbázisoknál. Ez alatt azt értjük, hogy a letárolt logikai adatmodell maga is bővíthető, illetve bizonyos mértékben módosítható anélkül, hogy az alkalmazói programokat módosítani kellene. Ha tehát egy objektumtípushoz újabb tulajdonságot kívánok letárolni, nem kell egyetlen egy meglévő alkalmazói programot sem módosítani.

Nagyobb adatabsztrakció.
Az adatbáziskezelésnél az adatok a felhasználó szemszögéből tekintve adatmodellben tárolódnak, ezért a felhasználónak nem kell törődnie a fizikai tárolás részleteivel, és egy magasabb absztrakciós szinten értelmezheti az adatrendszert. A részletek rejtve maradnak a felhasználó és a programfejlesztő előtt is.

Adatmegosztás, párhuzamos hozzáférés.
A DBMS felkészült az integrált adatokhoz történő osztott hozzáférések kezelésére. A hagyományos fájlkezelő alkalmazásoknál csak igen nagy ráfordítással tudnánk biztosítani a DBMS-ben megvalósított konkurens hozzáférést támogató koncepciót. Az adatmegosztás révén a helyigény is csökkenthető, és így mindenki a legaktuálisabb adatokhoz férhet hozzá.

Ellenőrzött redundancia.
Mivel több alkalmazás is ugyanazt az adatbázist használja, ezért a felhasznált adatok is egy helyen, egy kézben összpontosulnak. A hagyományos fájlkezelésnél ha több alkalmazásnak is szüksége volt egy adatra, akkor az adat több fájlban is letárolásra került. Ez a redundancia számos hátránnyal járt, kezdve a felesleges helyfoglalástól a konzisztencia megőrzésének problémájáig.

Hozzáférési jogosultságellenőrzés, adatvédelem.
A DBMS az operációs rendszerekhez hasonlóan nyilvántartja a jogosult felhasználókat. Ennek során nyilvántartja az azonosító nevet, a jelszót, a tulajdonában lévő adatokat, az engedélyezett műveleteket. Az adatvesztés okozta károk minimalizálására mind statikus, mind dinamikus védelem használható. A statikus védelem egyik eszköze a mentés, a dinamikus védelemhez pedig a naplózás tartozik.

Optimalizált fizikai adatszerkezetek.
A DBMS-ben implementálták mindazokat a hatékony adatstruktúrák kezelésére (mint a hashing, indexelés, stb.) alkalmas algoritmusokat, melyekkel jelentősen javítható a műveletek hatékonysága, gyorsasága. Normál fájlkezeléskor, nagyobb méretű adatszerkezetek esetén a programozónak kellene mindezen hatékonyabb adatstruktúrákat megvalósítania.

Integritási feltételek érvényesítése.
A DBS keretén belül, magában az adatbázisban tárolhatjuk az adatrendszerre vonatkozó megszorításokat integritási szabályok formájában. Az adatbázis módosításakor automatikusan ellenőrzi a DBMS, hogy nem sérült-e meg valamely integritási szabály. Ha megsérülne, akkor nem hajlandó elfogadni a változtatást. Mivel mindezt a DBMS végzi, nem a felhasználói programnak kell törődnie az integritási problémák teljességével. Így az alkalmazó programozó mentesül egy felelősségteljes feladat alól, míg a hagyományos fájlkezelésnél a programozó vállán volt minden felelősség az integritási szabályok betartatását illetően.

Szabványosság, hatékonyság, rugalmasság.
A szabványos adatmodellek és kezelő felületek, interfészek használatával az elkészült rendszer jobban érthetővé válik mások számára is, ezáltal későbbi fej-

lesztések, módosítások is könnyebbé válnak. A fejlesztő a DBMS-en keresztül jobban rákényszerül a szabványos eszközök használatára, mint a több szabadságot nyújtó normál fájlkezelésnél. Emellett az sem elhanyagolható, hogy a DBMS-ekhez számos fejlesztő eszköz áll rendelkezésre, jelentősen megnövelve a fejlesztés hatékonyságát. A szükséges változtatások gyorsabban végrehajthatók, nagyobb rugalmasságot biztosítva a rendszernek.

A felsorolt előnyök ellenére vannak olyan esetek, amikor célszerűbb a normál fájlkezelést választani. Ennek elsősorban az az oka, hogy a biztonságos, hatékony adatbáziskezelés biztosítását meg kell fizetni a felhasználónak, ami többlet időt és többlet költséget is jelent. Ezért nem célszerű adatbázist használni, ha

- az alkalmazás, az adatrendszer viszonylag egyszerű, és nem várhatók változtatások az adatrendszerben a jövőben sem. Ekkor felesleges beruházás a drágább DBMS beszerzése.
- az alkalmazás real-time követelményeket támaszt az adatrendszerrel szemben, azaz nagyon gyors adatkezelésre van szükség. Ez a DBMS összetettsége miatt nem biztosítható.
- egyfelhasználós adatrendszer esetén, amikor konkurens hozzáférés nem fordulhat elő. Ekkor maga az alkalmazás kézben tudja tartani az adatintegritás megőrzését, az adathozzáférés biztosítását. Bár nagyobb időráfordítással, de költségkímélőbben meg lehet oldani a feladatot a normál fájlkezeléssel.

Ha a feltételek mérlegelésével megszületett a döntés, hogy DBS-t alkalmaznak, akkor kezdődhet el a DBS fejlesztésének igazán érdekes és izgalmas folyamata. Ennek során ki kell választani a megfelelő adatmodellt és DBMS-t, el kell készíteni a modellezett valóság megfelelő adatmodelljét, létre kell hozni az adatbázist és a szükséges alkalmazói programokat is. S a szakemberek hosszú együttes munkájának eredményeként megszülethet az igényelt adatbázis rendszer.

1.6. Modellezés szerepe az adatbáziskezelésnél

Az alkotó tevékenység során alapvető szerepet játszanak a különböző modellek a környező világ megértésében, leképzésében és átalakításában. A modellek teszik lehetővé a lényeg kiemelését és szemléltetését. A *modell* fogalma alatt rendszerint két különböző dolgot szokás érteni: egyrészt olyan rendszert, amely a valóság egy vizsgált szeletével struktúrában vagy viselkedésben megegyezik, vagy hasonló jelleget mutat fel, és célja a vizsgálatán keresztül a valóság állapotára, viselkedésére vonatkozó következtetések levonása. Másrészt a modell kifejezéssel jelöljük azon eszközrendszert is, amellyel az előző értelemben vett modell leírható, megadható. Tehát a modell egyrészt egy jelölésrendszert, másrészt egy elkészült leírást is jelenthet. Sokféle és igen változatos modellekkel találkozhattunk már az eddigiekben. Magukat a programokat is egyfajta modellnek tekinthetjük, hiszen a vizsgált valóságot írják le a programozási nyelvek utasításainak, kifejezéseinek a segítségével. Amikor alkalmazást készítünk, több lépcsőn keresztül modellezzük a feladatot. Előbb egy áttekintő leírást készítünk, melyben közérthetően, az emberi

fogalmakhoz közelállóan vázoljuk fel a megoldást. Később ez alapján készítjük el a programozási nyelv segítségével megadott leírást.

Az adatbáziskezelés területén a modellezésnek még nagyobb a szerepe, mint a hagyományos programfejlesztési eszközöknél. Nézzük mi indokolja, miben jelentkezik a modellezés fontossága. A hagyományos alkalmazások egyfajta problématerületre készülnek, behatárolt funkciókkal és adatelemekkel. Egy könyvelési rendszer például nem alkalmazható könyvtár nyilvántartásra. Ezt úgy is kifejezhetjük, hogy az alkalmazásba beleégetődött a problématerület modellje. A modell megváltoztatásához újra kell írni az alkalmazás bizonyos részleteit.

Az adatbáziskezelő rendszerek ezzel szemben nemcsak egy problématerülethez készültek, hanem általános célúak. Az adatbáziskezelő rendszernek a modellezett területtől függetlenül biztosítania kell az adatbázisban tárolt adatok hatékony kezelését. Emiatt az adatbáziskezelőkbe nem lehet egyetlen egy fix modellt beégetni. A DBMS-nek nagyon sokféle különböző modell kezelésére alkalmasnak kell lennie. Ez pedig csak úgy oldható meg, ha a DBMS maga is rendelkezik egy felülettel, amelyen keresztül megadható, hogy milyen legyen az aktuálisan tárolandó adatrendszer struktúrája, modellje. Tehát a DBMS-ekhez mindig csatlakozik egy leíró nyelv, egy modell. A hagyományos alkalmazásokhoz ezzel szemben nem rendelhető ilyen rugalmas modell (modell, mint leíró rendszer). Az informatikában azokat a modelleket, amelyek az adatok struktúrájának leírására szolgálnak, *adatmodelleknek* nevezik.

> *A valóság adatstruktúrájának, integritási szabályainak megadására szolgáló formalizmust, amely az adatrendszeren elvégezhető műveleteket is definiálja, adatmodellnek nevezzük.*

Mivel a DBMS-ekhez rendelt modell is ugyanezen célt szolgálja, ezért a DBMS-hez rendelt modell is adatmodellnek tekinthető. Az adatmodellek leírásával egy későbbi fejezetben fogunk részletesebben foglalkozni. Az adatmodellekről azonban már most is megjegyezhetjük, hogy központi szerepet játszanak az adatbáziskezelésben, hiszen ezen keresztül adhatjuk meg a megvalósítandó rendszer leírását az adatbáziskezelőnek. Ahhoz, hogy használni tudjuk az adatbáziskezelőt, ismernünk kell az adatbáziskezelőhöz tartozó adatmodellt. Mint várható, nemcsak egyféle adatmodell létezik. Az adatbáziskezelés fejlődésével újabb és újabb adatmodellek jöttek létre. A DBMS-ek egyik fő jellemzője, hogy mely adatmodellhez kapcsolódnak. A korábban már említett relációs, hierarchikus vagy hálós előtagok is az alkalmazott modellt azonosítják, azaz például a relációs DBMS a relációs adatmodellen nyugszik.

Mivel az adatmodell egy jelölésrendszeren alapszik, ezért nem igényli feltétlenül egy DBMS létét, azaz vannak olyan adatmodellek is, melyekhez nem létezik DBMS. Ennek ellenére ezen adatmodellek sem tekinthetők felesleges, selejt adatmodelleknek. Egyrészt ezen modellek rendszerint a végső DBMS adatmodell kialakításában segítenek, hiszen emlékszünk rá, hogy maga a programozás is egy többlépcsős modellezési folyamatként értelmezhető; másrészt ezek a modellek visszahatnak a DBMS-ek fejlesztésére is, és egyszer talán rajtuk alapuló DBMS-ek fognak megjelenni a piacon, mint ahogy ez a múltban már párszor megtörtént.

1.7. Áttekintés az adatbázisrendszerek architektúrájáról

Az előzőekben megismert alapfogalmakra építve most egy részletesebb ismertetést adunk az adatbázisrendszerek elvi felépítéséről, majd ezt követően az adatbázisok és az adatbáziskezelő alkalmazások fejlesztéséről esik szó. A bemutatandó fogalmak az adatbáziskezelés általános koncepciójának a megértését, elsajátítását szolgálják, és a gyakorlatban használatos ismeretek, részletek tárgyalására a későbbi fejezetekben kerül majd sor.

Az adatbázisrendszerek belső architektúráját többféle szempont szerint is elemezhetjük. Az előző fejezetben bemutatott struktúra funkcionális elemzésen alapult. A három alapvető komponens,

- az adatbázis,
- az adatbáziskezelő rendszer és
- az alkalmazói vagy segédprogramok

mindegyike különböző feladatot látott el. A különbözőség a komponensek éles fizikai elkülönülésében is megnyilvánul. Az adathordozón más-más helyen, más-más azonosítóval szerepelnek, és kiemelten, külön-külön is mozgathatjuk, vagy módosíthatjuk őket. Az egyes komponensek létrehozásának ideje is elkülönül egymástól, és esetleg más-más helyről is beszerezhetjük őket. Elsőként a DBMS kiválasztása történik meg, majd ennek segítségével lehet létrehozni az adatbázist, majd az erre épülő alkalmazásokat. Személyileg is szétválnak az egyes komponensek kezelésével megbízott posztok; szükség van többek között rendszeradminisztrátorra, operátorra a DBMS kezeléséhez, a programozók és a felhasználók pedig az alkalmazásokhoz kötődnek.

E szembetűnő funkcionális szempontok szerint történő strukturálás mellett más módon is elemezhetjük az adatbázisrendszereket. A legismertebb, sőt szabványként is elfogadott strukturálás, az *ANSI/SPARC architektúra* néven ismert struktúra. Nevét onnan kapta, hogy az ANSI/SPARC Study Group on DataBase Management Systems bizottság dolgozta ki. A bizottságot az 1970-es évek elején hozták létre az ANSI szabványosítási hivatal keretében, hogy meghatározza az adatbáziskezelés azon területeit, melyben lehetséges és célszerű a szabványosítás. A vizsgálat eredményeként megszületett egy általános DBS modell, amelyben kiemelt hangsúlyt kaptak az egyes komponensek közötti interfészek.

Az ANSI/SPARC architektúra az adatbázis leírására három szintet tartalmaz: a *külső* (external), a *koncepcionális* (conceptual) és a *fizikai* (internal) szintet. Az egyes szintek az adatbázisrendszer - mint egység - különböző megvilágításainak, megközelítéseinek felelnek meg, ezért ezeket szokás nézeteknek (view) is nevezni.

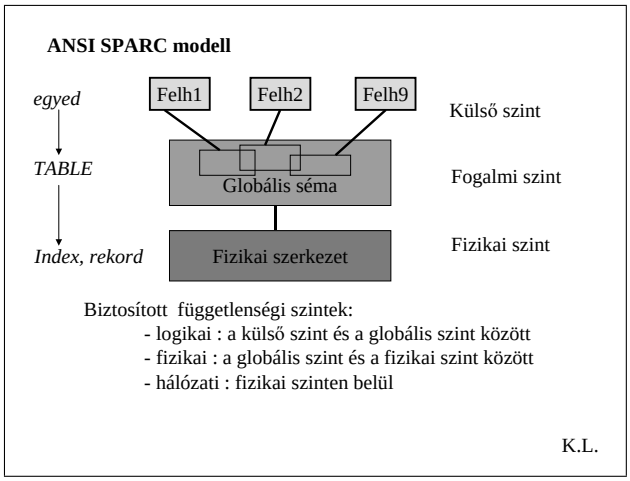
A *külső szint* foglalja magában mindazt, amit egy-egy felhasználó az adatbázisból lát, amit a felhasználó adatbázis alatt gondol, ami számára az adatbázist jelenti. Ez az egyedi látásmódok összessége. Mivel egy adatbázisrendszerhez több alkalmazó, felhasználó is kapcsolódhat, ezért több ilyen nézetet is tartalmazhat. Ezek a nézetek rendszerint különbözőek, hiszen a felhasználók a teljes adatbázis

más-más részletét látják csak. Így például más adatokat kezel, más adatokhoz férhet hozzá a teljes vállalati adminisztrációs rendszeren belül egy pénzügyi, vagy egy személyzeti adminisztrátor. Mivel az adatrendszer más elemeivel ők soha nem kerülnek kapcsolatba, ezért számukra ezek az adatok jelentik a teljes adatbázist; ők úgy látják, hogy az adatbázis csak azokat az adatokat tartalmazza, melyekkel kapcsolatba kerülnek. Az egyes nézetek különbözősége azonban nem zárja ki azt a lehetőséget, hogy az egyes nézeteknek közös elemei is legyenek.

Igaz, hogy sok különböző külső nézet létezik, de ezek mindegyike valójában ugyanannak az adatbázisnak a különböző részleteit tartalmazza. Ezért értelmezhető az a látásmód, mely a teljes adatbázist tartalmazza. Ilyen nézettel kell rendelkeznie például a rendszeradminisztrátornak, vagy az adatbázistervezőnek. Ez a *közösségi nézet* a koncepcionális szinten helyezkedik el. A közösségi nézetből viszont csak egy van, és az összes külső nézet ennek egy-egy szeletét jelenti.

Míg a felhasználó a külső nézetében a modellezett valóság egyedeit, egyedkapcsolatait látja, addig az adatbázistervező a DBMS által támogatott adatmodellben, tehát egy absztraktabb leírásban gondolkodik. Az adatbázis azonban fizikailag is létezik, valamilyen fizikai adatstruktúrában letárolva az adathordozón. Mint már láttuk, a kiválasztott adatstruktúra is lényeges szerepet játszik az adatbázis hatékonyságánál, és az adatbázis adminisztrátor egyik feladata éppen a megfelelő fizikai struktúra kialakítása, hangolása. Tehát létezik egy, a fizikai tárolási szerkezethez közel álló nézet is, melyet *adattárolási nézetnek* neveznek.

Ez a három szint, mint láttuk, az adatbázisképek *absztrakciós szintjében* is különbözik egymástól. Erre az absztrakciós szintkülönbségre a számítástechnika más területeiről is hozhatunk példát: amikor egy közgazdász egy termelésoptimálási programot használ, akkor ott a gépkapacitásokat például egy mátrixban letárolva látja maga előtt. A programozó számára az adatok egy tömbben letárolva jelennek meg. Fizikailag pedig azonos hosszú rekeszek sorozataként tárolódnak az adatok.



1.15. ábra. ANSI/SPARC modell

Azért is használható ez a hasonlat, mert az adattárolási nézet is folytonos tárolást tételez fel, és nem törődik az adatok blokkokba történő tördelésével, a blokkok láncolásával.

Az egyes szinteken a nézetek megadása különböző adatmodellek segítségével történik. Ennek során meg szokták különböztetni magát a nézetet, vagyis azon adatokat amit látok, az adatok tárolási struktúrájától, az adatszerkezettől. Az adatszerkezet leírását *sémának* (schema) nevezik. Egy ház esetén a tervrajz mint szerkezeti leírás, séma szerepelhet, és maga a tervnek megfelelő szerkezettel elkészült ház lehet a nézet. A séma mondja meg például, hogy van egy kút a kert sarkában. A megvalósulás pedig már egy konkrét kutat tartalmaz. A különböző megvalósulásoknál a kút más és más alakban jelenhet meg. Mint ahogy több ház is felépülhet egy tervrajz alapján, ugyanúgy több nézet is létrejöhet egy séma alapján. Az adatbázis tervezésekor elsőként a sémákat kell létrehozni, és ez a váz töltődik fel a használat során adatokkal. A séma alapján felépülő konkrét adatrendszert séma megvalósulásnak, *adatelőfordulásnak*, angolul *instance*-nek nevezik.

A sémák megadására valamilyen sémaleíró nyelvet, modellt lehet használni, és mivel a sémák elsődlegesen adatszerkezet definíciókként értelmezhetők, ezért ezt a nyelvet *adatdefiníciós nyelvnek* is nevezik, és a DDL (Data Description Language) rövidítést használják a jelölésére. A hagyományos programozási nyelvekben is találkozhatunk DDL elemekkel, mint az int, struct, array, stb. kulcsszavak, melyek többé vagy kevésbé helyileg is elkülönülnek a vezérlési nyelvelemektől. Az adatbáziskezelésnél, mint majd látni fogjuk, ez az elkülönülés élesebb a hagyományos nyelveknél megszokottól. Mivel a séma csak váz, melyet majd adatokkal kell feltölteni, szükség van olyan eszközre is, mely lehetővé teszi az adatok kezelését. Az ilyen, adatkezelési utasítások végrehajtására szolgáló kifejezések alkotják az *adatkezelő nyelvet*, melynek rövidítése DML (Data Manipulation Language). A hagyományos programozási nyelveknél a jól megismert read, write stb. utasítások sorolhatók a DML-hez.

A programok készítésénél szokásos vezérlési szerkezetek, mint a ciklusszervezés vagy az elágazások létrehozására szolgáló nyelvi elemek is megtalálhatók a legtöbb DBMS-nél, habár ezek viszonylag lazábban kötődnek a DBMS-hez, annak elsődleges adatorientáltsága miatt. E laza kapcsolatot mutatja, hogy számos DBMS nem is tartalmazott procedurális nyelvet, hanem csak egy adatkezelő, adatdefiníciós résznyelvet, melyet valamely hagyományos programozási nyelvvel együtt lehetett az alkalmazások elkészítésére felhasználni. A hagyományos programozási nyelvet nevezték *gazda* (host) nyelvnek, és ebbe kellett beültetni az adatkezelő és adatdefiníciós utasításokat. A gazda nyelv és a DB kezelő nyelv közötti kapcsolatot lazának nevezik, ha élesen elkülönül egymástól a két nyelv a programon belül, mint azt a következő példa is mutatja:

```
if (a > 8) {
EXEC SQL INSERT INTO A VALUES(3);
b = 3;
}
```

Ebből rögtön látható, hogy van benne, és az is, hogy hol, DML utasítás. A szoros kapcsolat ezzel szemben azt jelenti, hogy az adatbázis DML utasításai közvetlenül nem érzékelhetők, minden adatkezelő utasítás megfelel a gazda nyelvben megadott szintaktikának. A szoros kapcsolatban rejlő nehézségek miatt napjainkban még csak a laza csatolás terjedt el.

Az egyes nézetekben az adatok egyedekhez kötődve jelennek meg, ahol az adatbázis több előfordulását is tartalmazza egy egyedtípusnak. Itt az *egyedtípus* alatt az egyed leírására szolgáló sémát értjük, például egy könyv egyed esetén a könyvtári nyilvántartásban a séma egy címet, ISBN számot, egy leltári számot, kiadót, szerzőt tartalmazó rekordszerkezetként is elképzelhető. A rendszerben minden könyvet azonos szerkezettel, a könyvtípussal írunk le. Az egyes könyvek lesznek a könyv *előfordulások*.

Az egyedelőforduláshoz tartozó adatokat szokás rekordnak is nevezni. Az ANSI/SPARC architektúra különböző szintjein egy egyed rekordjai más-más alakot ölthetnek. A külső szint rekordjai különbözhetnek egymástól az egyes nézetekben, illetve lehetnek ezen rekordoknak közös elemei is. A koncepcionális szinten létezik egy olyan koncepcionális rekord, mely az egyes külső rekordok mezőinek egyesítését tartalmazza. A fizikai szint rekordja szintén különbözhet a logikaitól, hiszen a koncepcionális rekord fizikailag lehet partícionált vagy éppen tartalmazhat duplikált elemeket is.

A rekord mindig egy egyedelőforduláshoz tartozik, miközben az adatbázis több előfordulást is tárolhat. Az adatbázis tehát tartalmazza az egyedeket, mint önálló egységeket, és az egyedösszességet, mint csoportot. Ebből eredően az egyedeket kezelhetjük csoportosan és egyénenként. Ezt a kétfajta megközelítést *halmazorientált* és *rekordorientált* megközelítésnek nevezik. A műveletek is lehetnek ennek megfelelően rekordorientáltak vagy halmazorientáltak. Egyes DBMS-ek különbözhetnek egymástól abban is, hogy mely megközelítést támogatják.

A szintek sémáinak és nézeteinek különbözőségéből következik, hogy az egyes szintek kapcsolódásánál leképezést, illesztést kell végezni. Az egyes szintek között pedig igen intenzív kapcsolat áll fenn. Hiszen amikor egy alkalmazás, egy felhasználó kiad egy utasítást, akkor azt a saját külső sémájában fogalmazza meg. Egyidőben több olyan utasítás keletkezhet, melyek mindegyike más-más külső sémát használ.

A műveletek összehangolására, vezérlésére minden utasítást le kell fordítani a koncepcionális szint sémájára. A fizikai szintű műveletek elvégzéséhez ismerni kell az adatok fizikai sémáját, tehát szükség van a koncepcionális és a fizikai séma közötti leképzésre is. A fizikai művelet elvégzése után az eredmény visszajuttatásához ugyanígy el kell végezni a leképzéseket, csak most fordított irányban. A leképzések - melyek egyértelműen megnövelik a műveletek végrehajtási idejét - legfontosabb célja a már korábban említett függetlenség biztosítása. A felvázolt ANSI/SPARC architektúra is az adatbáziskezelésben megvalósuló adatfüggetlenség megnyilvánulására példa, hiszen a felhasználó, az alkalmazásfejlesztő függetlenítheti magát a többi alkalmazástól, a koncepcionális tervezés pedig függetlenítheti magát a fizikai, belső megvalósulástól.

A leképzések elvégzése az elmondottakból következően csak a DBMS feladata lehet, hiszen az egyes alkalmazói programoknak nem kell ismerniük a teljes adat-

bázist, a koncepcionális sémát. A DBMS központi szerepe indokolja, hogy egy kicsit részletesebben is foglalkozzunk az általa elvégzett tevékenységekkel, a belső struktúrájával.

1.8. A DBMS belső szerkezete

A DBMS, mint már említettük, mindkét másik DBS komponenssel, a DB-vel és az alkalmazói programokkal is kommunikál. Az adatbázishoz, mint a külső adathordozón letárolt fizikai adatszerkezethez történő hozzáféréshez a DBMS is felhasználhatja a hardver fölött elhelyezkedő operációsrendszer IO szolgáltatásait. Így a DBMS tehermentesíthető lesz az alacsony szintű IO műveletek végrehajtása alól, és egyszerűbbé válik a DBMS implementálása is. E feladatátruházásból az is következik, hogy a DBMS valójában nem közvetlenül a DB-vel, hanem az OS-el áll kapcsolatban, valamint az is érzékelhető, hogy a DBMS hatékonyságát az OS-hez történő illesztése is számottevően befolyásolja.

A probléma fontosságát mutatja, hogy számos kutatási program témája a kapcsolódás hatékonyságának növelése, és már több javaslat is született a DBMS-OS kapcsolat javítására. Egy ma még rendszerint meglévő gyenge pont a *kétszintű tárolási architektúra* (two levels storage) használata. Ebben az elnevezésben a kétszintűség arra utal, hogy a DBMS-nek nyilván kell tartania, hogy mely rekordjai, blokkjai találhatók meg a memóriában, és melyek vannak kint a lemezen, tehát különbséget tesz a belső és a külső tárolás között. Erre azért is szükség van mivel a DBMS az OS-től eltérő módon értelmezi az adatállományok belső struktúráját. A DBMS elemi tárolási egysége különbözik például az OS IO elemi egységétől, így a legtöbb DBMS saját fájl és bufferkezeléssel rendelkezik, melyek felhasználják az OS elemi IO szolgáltatásait. Hatékonyabb megoldást jelent ezen a téren az egyszintű tárolás (single level storage) alkalmazása, melynek lényege, hogy a felhasználói processz elől elrejtsék a fizikailag meglévő tárolási struktúra megosztottságát, ami a memória és háttértár különböző kezelési módjából ered. Ebben az esetben a DBMS összes adata egyetlen egy hatalmas virtuális címtartományban helyezkedik el, ezzel egységessé válik az adatelemekre történő hivatkozás, függetlenül attól, hogy fizikailag hol helyezkedik el. Azaz nem a DBMS-nek kell nyilvántartania, hogy a rekord most éppen benn van-e a memóriában, vagy sem. Az IO rendszer mellett más, mind a DBMS-ben, mind az OS-ben előforduló egyéb területek is, mint a védelem, vagy az osztott erőforrás-felhasználás, hasonló lehetőségeket adnak a DBMS további teljesítménynöveléséhez.

A DBMS másik oldala az alkalmazói programok rétege. Természetesen nem minden program tud kommunikálni a DBMS-sel. Az alkalmazói programot fel kell készíteni az adatcserére. A kommunikáció ugyanis felügyelet alatt, megadott szabályok szerint megy végbe. Rendszerint mind a küldő, mind a fogadó oldalon létezik egy, a kommunikációra szolgáló komponens, az *adatkommunikációs* (DC) komponens, melyek megértik egymást, úgymond azonos nyelven beszélnek. Emellett szükség van még olyan komponensre is, amely a DBMS és az alkalmazói program között továbbítja az üzeneteket. Az üzenetküldés lehet egyszerű, ha például azonos processzen belül fut mindkét elem, de lehet összetettebb is, amikor a két adat egy hálózat más-más csomópontján található, tehát hálózati kommunikációs

szoftverre is szükség van. Ezen változó összetételű komponensekből felépülő, az alkalmazás és a DBMS között húzódó szoftver réteget szokás program interfésznek is nevezni.

Abból a tényből, hogy a DBMS és az alkalmazói programok külön processzeket alkothatnak, következik, hogy egy általános DBS egy *kliens-szerver* architektúrájú rendszernek is tekinthető, hiszen van egy kliens oldal, az alkalmazói program, mely adatbáziskezelési igényekkel, utasításokkal lép fel a DBMS-sel szemben, ami az utasításokat, mint szerver végrehajtja.

Igaz, hogy a kliens és a szerver is elhelyezkedhet ugyanazon a gépen, de rendszerint amikor a kliens-szerver architektúrára gondolunk, sokunknak rögtön a hálózat jut eszünkbe, hiszen a klienst és a szervert a hálózat különböző csomópontjaihoz kötve szoktuk használni. Ennek az elrendezésnek is számos előnye van:

- gyorsabb végrehajtás, több processzor dolgozik egyidejűleg,
- a szervert illeszteni lehet az adatbáziskezeléshez,
- a kliens gépet illeszteni lehet a felhasználói igényekhez,
- több kliens gép is csatlakozhat egy szerverhez,
- rugalmas kiépítés, fejlesztési lehetőség.

Nézzük meg milyen globális folyamatok zajlanak le az adatbáziskezelő rendszeren belül. A DBMS sok-sok különböző egységből felépülő nagy gyárhoz hasonlítható, melyben az egységeknek összehangolt munkát kell végezniük, hogy biztosítsák a rendszer hatékony működését. A DBMS esetében a feladat összetettsége és bonyolultsága megköveteli, hogy részfeladatokat jelöljünk ki, és külön modulokat hozzunk létre az elkülöníthető tevékenységekhez. A már megismert funkcionális követelmények jó támpontot adnak a DBMS belső funkcionális felbontásához, csak néhány új fogalom szerepel itt a teljesebb leírás végett.

A DBMS-ek legfontosabb komponenseinek vizsgálatánál a DBMS-t rendszerint két nagy struktúra egységre bontják: egy felhasználóhoz közeli rétegre (*Data System*), és egy a hardverhez kapcsolódó rétegre (*Storage System*). Míg a Data System feladata az adatok adatmodell szerinti kezelése, a Storage System az adatok fizikai tárolási struktúrájával dolgozik. A DBMS-en belül a két réteg között intenzív kommunikáció folyik, mindkettő a felhasználó és az adatbázis közötti adatcsatorna szerves része. A Data System fogadja a felhasználó utasításait, majd értelmezi az utasítások végrehajthatóságát és meghatározza az utasításhoz tartozó fizikai műveletsort. Ez a műveletsor kerül át a Storage System-hez, amely saját IO rendszerében elvégzi a fizikai adatátviteli lépéseket, ügyelve a konkurens hozzáférésből és a védelmi szempontokból adódó feladatokra.

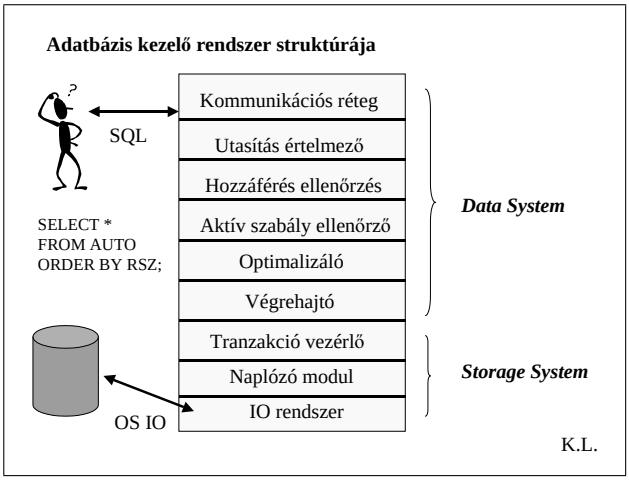
Mindkét réteg tovább bontható funkcionális elemeire. Elsőként vegyük a Data System legfontosabb komponenseit:

- *Adatkommunikációs komponens*, melyet már ismertettünk és melynek feladata az interface biztosítása a segédprogramok, a felhasználók felé.
- *Utasításértelmező.* E komponens feladata egyrészt az utasítások szintaktikai ellenőrzése, másrészt az utasítások tartalmi, végrehajthatósági

vizsgálata. Ehhez az adott adatmodell, adatkezelő nyelv ismerete mellett szükség van a kezelt adatbázis adatbázismodelljének az ismeretére is. Itt most nem az adatok ismeretére gondolunk, hanem az adatstruktúrára, illetve a védelmi és integritási feltételekre. Ezek az adatok pedig a már korábban említett metaadatok közé tartoznak. A DBMS az adatbázishoz tartozó metaadatokat az adatszótárban, Data Dictionary-ban tárolja. Az értelmezés feladatát, az eltérő jelleg miatt gyakran külön választják az adatdefiníciós és adatkezelő értelmezőkre.

– *Optimalizáló.* Mivel a felhasználóktól összetett utasítások is érkezhetnek, és egyidejűleg több utasításcsoport is állhat végrehajtás alatt, nem lényegtelen az elemi utasítások végrehajtási sorrendje. Egy utasítás ugyanis rendszerint több elemi részlépésre bontható fel, több utasítás esetén a generált elemi utasítások végrehajtási sorrendje is tág határok között változhat, tehát egyazon feladatcsoporthoz több elemi utasítássorozat is tartozik, és ezen sorozatok mind helyigényben, mind gyorsaságban különbözhetnek egymástól. Emiatt a DBMS egyik lényeges feladata az optimális elemi utasítássorozat kiválasztása.
– *Végrehajtó.* Az optimális elemi utasítássorozat végrehajtása történik ebben a modulban. Az utasítás-végrehajtás vezérlése mellett e komponens feladata az elemi utasítások végrehajtási kódjainak a tárolása, felhasználása is. Ebben a modulban is történik döntéshozatal, ugyan már sokkal szűkebb hatáskörben, mint az előző komponensnél, ugyanis bizonyos műveleteknél az algoritmusváltozat kiválasztása, csak az előző elemi lépés eredményének ismeretében történik meg. Az elemi utasítások végrehajtása során rekordszintű IO utasítások állnak elő, melyek feldolgozása már a Storage System feladata lesz.

A Storage System legfontosabb komponensei:



1.16. ábra. Az adatbáziskezelő rendszer struktúrája

– *IO rendszer.* Mint már említettük, a DBMS specifikus igényei miatt saját IO rendszert, bufferkezelést, helyfoglalási mechanizmust építenek be a legtöbb fejlettebb DBMS-be. Ez a rutinkönyvtár az OS IO kezelésénél magasabb szintű, a DBMS bufferhez kapcsolódó rutinokat tartalmaz. Ezek a rutinok már közvetlenül hívhatják az OS alacsony szintű IO rutinjait.
– *Konkurens hozzáférés vezérlés.* Majd a későbbiekben látni fogjuk, hogy milyen eszközök állnak rendelkezésre a megosztott erőforrások kezelésére. E modul tartalmazza mindazokat az alacsony szintű adatstruktúrákon értelmezett módszereket, melyek az osztott hozzáférés vezérléséhez szükségesek.
– *Adatvédelmi rendszer.* E modul feladata a különböző adatsérülések, rendszer leállások okozta veszteségek minimalizálása, az adatok megfelelő védelmének biztosítása. Ez a komponens is több olyan kisebb részből áll elő, mint például a rendszeres háttérmentés végzésére szolgáló rutin.

A DBMS belső struktúrájával közvetlenül sem a felhasználó, sem az alkalmazó programozó nem fog találkozni, számukra rejtve maradnak a DBMS összetettségének jelei. Ami egy felhasználót igazában érdekel, az a DBS felhasználói kapcsolattartása, azon segédprogramok rendszere, melyeken keresztül elérhetők az adatbázisban tárolt adatok.

1.9. Az adatbáziskezelő rendszerek osztályozása

A DBMS-ek lentebb megadandó osztályozása nem a belső felépítésen, hanem a DBMS-nek a fejlesztő, a felhasználó felé mutatott képén alapszik, melyhez az alábbi szempontok köthetők:

– adatmodell,
– felhasználók száma,
– DBMS csomópontok száma,
– támogatott hardver és OS típusok.

A kezelő nyelvet, a DBMS viselkedését tekintve a legalapvetőbb kritérium a DBMS-hez tartozó adatmodell, de ennek ismertetése előtt a többi, gyorsabban áttekinthető szempontot nézzük át. A felhasználók száma alapján, hasonlóan az operációs rendszerekhez, megkülönböztetünk egyfelhasználós és többfelhasználós rendszereket. Korábban már említettük, hogy az adatbáziskezelést elsődlegesen többfelhasználós környezetre tervezték, de a hatékonyabb adatkezelés, az egyszerűség, az alacsonyabb költségek miatt, esetleg betanulási céllal sokan vásárolnak egyfelhasználós rendszereket, melyek szinte kivétel nélkül egyfelhasználós operációs rendszereken – MSDOS, Windows – futnak. A csomópontok száma alapján beszélhetünk önálló DBMS-ről, amikor csak egy gépen fut a DBMS, és osztott DBMS-ről, amikor több csomóponton fut egyidejűleg. A támogatott hardver és OS típusa a felhasználó számára a termék kiválasztásakor válik fontossá, hiszen a meglévő hardver és OS feltételek behatárolják a választási lehetőségeket.

Az adatmodell, mint már említettük az adatok logikai tárolási formátumát határozza meg: olyan vázat ad, melybe az adatok majd beletölthetők lesznek. Az adatmodell megadása eszerint egy szerkezetleírást jelent, hasonlóan egy normál program struktúra deklarációjához. Ezt az elképzelést azonban meg kell még toldani annyival, hogy az adatrendszer ismerete nem csak az adatszerkezet ismeretét, hanem az adatok kezelési módjának az ismeretét is magában foglalja. Ezért az adatmodellbe a statikus szerkezet leírás mellett a dinamikus, az adatokon értelmezett műveleteket is beleértik. Az adatmodell megadásánál mind a szerkezet, mind a műveletek megadása logikai szinten, és nem fizikai szinten történik, ezért az adatmodell egy elvontabb absztraktabb, formálisabb leírást jelent. Összegezve tehát az adatmodell olyan matematikai formalizmus, mely az adatok és az adatokon értelmezett műveletek leírására szolgál. Az egyes adatmodellek a kiválasztott formalizmus jellegében különböznek egymástól, a deduktív adatbázisok például logikai formalizmust használnak fel.

Több adatmodell is létezik, de ezekből négy terjedt el igazán a gyakorlati életben: a hierarchikus, a hálós, a relációs és az objektum-orientált adatmodellek. Ezek közül a relációs adatmodell a legnépszerűbb ma, a hálós modell kezd háttérbe szorulni, míg a hierarchikus már inkább a múlté, az objektum-orientált modell pedig csak a jövőben válik igazán piacéretté.

A *hierarchikus adatmodell* az adatokat egy hierarchikus faszerkezetben tárolja. E fa mindegyik csomópontja egy rekordtípusnak felel meg. A hierarchikus modell alapja, hogy a gyakorlati életben a szervezetek vagy éppen a struktúrák nagyon gyakran hierarchikus felépítésűek, gondoljunk csak a vállalati hierarchiára vagy egy gyártmány alkatrészeinek hierarchiájára. Emiatt természetesnek tűnik, hogy a modellezés megkönnyítésére a valóságban leggyakrabban használt, hierarchikus modellt hozzuk létre. Ez a modell a gyakorlati alkalmazások során fejlődött ki, ezért nincs olyan elméleti megalapozottsága mint a későbbi adatmodelleknek. A modellhez kapcsolódó DML nyelvek mind rekordorientált adatmegközelítést alkalmaztak. A bonyolultabb kapcsolatok ábrázolása csak kerülőutakon lehetséges. A modell előnye, hogy a hierarchikus szerkezet egyszerűen leírható, és tárolása a mágnesszalagos tárolási formához is jól illeszkedik.

A *hálós adatmodell* a hierarchikus modell továbbfejlesztése, amely jobban illeszkedik a bonyolultabb kapcsolatok ábrázolásához is. Ebben a modellben az egyedek között tetszőleges kapcsolatrendszer, egy kapcsolatháló alakítható ki. Az adatszerkezet leírása, mivel a háló tetszőleges nagy lehet, nem egy adategységgel, hanem több kisebb, hierarchikus felépítésű adategységgel történik. Ehhez a modellhez is rekordorientált adatmegközelítést alkalmaztak a DML kialakításakor. A hálós modellen alapuló DBMS-ek igen elterjedtek a nagygépes környezetekben, hiszen a hálós modell nagy adatmennyiségek viszonylag gyors feldolgozását teszi lehetővé. A kezelőnyelv bonyolultsága, viszonylag merevebb szerkezete gátolta szélesebb körben történő elterjedését.

A *relációs adatmodell* sokkal rugalmasabb szerkezetet biztosít az elődeihez viszonyítva. Az adatbázis azonos rekordtípusokat tartalmazó táblákból épül fel, ahol minden tábla teljesen egyenértékű, és nincs semmilyen, az adatdefiníciókor véglege-

sen lerögzített kapcsolat, váz, mint ami az előző modelleknél előfordult. A relációs modellben az egyedek közötti kapcsolatok az adatértékeken keresztül valósulnak meg. A relációs modellben a táblákon értelmezett műveletek ugyan halmazorientáltak, de számos olyan implementáció létezik, melyben rekordorientált műveletek használhatók. A modell elterjedése az egyszerűségének és rugalmasságának köszönhető.

Az *objektum-orientált adatmodell* célja az objektumorientáltság szemléletmódjának alkalmazásával minél valósághűbb adatmodellt megalkotása. Az egyedek ugyanis sokkal szemléletesebben írhatók le az objektumokkal, mint a relációs modellben szereplő rekordokkal. Az objektum orientáltság a megvalósult rendszerekben lehet teljes vagy részleges. A részleges OODBMS-ek rendszerint csak strukturálisan objektum-orientáltak, a funkcionális, aktív elemek csak a teljes OODBMS-ekben jelennek meg. Az OODBMS-ek elterjedését az egységes elméleti alapok hiánya és az implementációs nehézségek fékezik.

Az egyes adatmodellek ismertetéséhez, bővebb leírásához szükség van bizonyos adatmodellezési alapfogalmak megismerésére, melyek szorosan kötődnek az adatbázisrendszerek tervezésének módszertanához, ezért a fenti modellek teljesebb bemutatására a későbbi fejezetekben fog sor kerülni.

1.10. Az adatbázisrendszerek tervezési lépései

Az adatbázisrendszerek tervezésének vizsgálatakor abból a tényből kell kiindulni, hogy a DBS is egy számítógépen futó program, egy szoftver termék, ezért az általános szoftverfejlesztési irányelvek itt is érvényesek. A szoftverfejlesztés (SE, Software Engineering) általános metodikája mellett természetesen a DBS-ek specifikumait is figyelembe kell venni. Az SE folyamatának egyik szemléletes megjelenítője az *SE piramis*, ami a következő lépéseket foglalja magába:

- *követelmény analízis*: a vizsgált problématerület elemzése, a megoldandó feladatok, a követelmények meghatározása;
- *rendszerelemzés:* a problématerület modellezése, a belső struktúrák és működés feltárása több különböző szemszögből és különböző részletességgel;
- *rendszertervezés*: az elkészítendő szoftver belső struktúrájának, működésének több különböző szemszögből történő feltárása, különböző részletesség mellett;
- *kódolás*: az elkészült modell leírás átkonvertálása a számítógép által érthető formára, valamely programozási nyelvet felhasználva;
- *tesztelés*: az elkészült kód hibamentességének ellenőrzése;
- *karbantartás*: folyamatos ellenőrzés, módosítások, hibakijavítások sorozata.

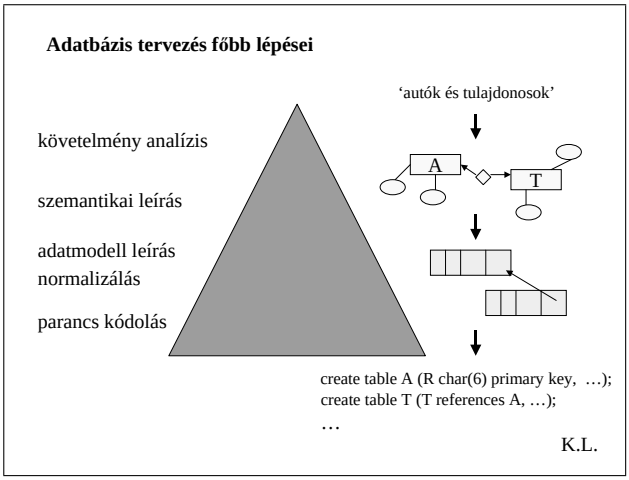
Természetesen itt nem szabad elfelejteni, hogy a tervezés folyamata nem egy egyszerű szekvencia, mivel bizonyos elemek többször is megismétlődnek, így a tervezés ciklikus folyamattá alakul át, és a fenti felsorolás ennek a ciklusnak az

elemeit sorolja fel egy kvázi, logikai sorrendben. Ha az egyes tevékenységek absztrakciós szintjeit vizsgáljuk, akkor látható, a piramisban fentről lefelé haladva egyre csökken a leírásmód elvontsága és egyre nagyobb szerepet kap a konkrétabb megfogalmazás.

A szoftvertermék leírása a fejlesztés során fokozatosan alakul át az absztrakt, esetleg pár mondatos köznapi leírásból a futtatható többezer soros gépi kódsorozatig. Ez az átalakulás a modellek sorozatán keresztül valósul meg. Előbb absztraktabb, később konkrétabb, részletekkel gazdagított modellek jelennek meg. A DBS rendszer sem tér el ettől a fejlesztési metodikától, sajátossága, egyedisége leginkább a felhasznált modellekben rejlik. A DBS rendszerek jellegzetessége, hogy kiemelt helyre, elsődlegesen az adatrendszerre koncentrál. Különösen fontos tevékenység az adatbázis megtervezése, hiszen a DBS központjában a DB áll, és ennek hatékonysága, korrektsége az összes alkalmazás teljesítményére kihat. Az adatbázis tervezése kettős célt követ: egyrészt ki kell elégíteni a felhasználók információigényét, másrészt ügyelni kell az információfeldolgozás hatékonyságára is.

A DB tervezésekor is több modellen keresztül jutunk el a fizikai adatbázishoz, hiszen induláskor rendszerint csak igen informális ismereteink vannak a feladatról, a modellezett világról. Ezzel szemben az elkészített adatbázisrendszer igen szigorú, formális nyelvvel írható le. Több modell létezik, melyek különböző absztrakciós szinteken írják le a megvalósítandó adatbázist. Mivel ezek mindegyike az adatbázis leírására szolgál, mindegyikre használható az adatmodell kifejezés.

Az adatmodelleket alapvetően két nagy csoportba szokás osztályozni. Az egyik adatmodelltípus a *szemantikai adatmodell* (SDM) vagy koncepcionális (conceptual) adatmodell, mely elvontabb szinten, részletek nélkül, emberközelien írja le az adatszerkezetet. Az SDM-ek alkalmasak az adatbázis lényegének a kiemelésére, a szerkezet megértésére. Egy adott adatbázis SDM ugyanaz marad akkor is, ha



1.17. ábra. Az adatbázistervezés lépései

esetleg módosul a kiválasztott DBMS, hiszen a modellezett valóság is változatlan marad, és az SDM valóságközeli modell-leírás.

A másik csoport a konkrétabb, *DBMS közeli adatmodellek* köre, melyekből már megemlítettük a relációs, a hálós, a hierarchikus és az OO modelleket. Ezek a modellek rugalmasságuk ellenére mégiscsak egyfajta korlátozást, keretet szabnak, melybe bele kell gyömöszölni a valóságot. A valóság tökéletes leírásához azonban a meglévőnél sokkal gazdagabb lehetőségekre lenne szükség, mint amit a hagyományos DBMS adatmodellek támogatnak. Mivel ezek a modellek túlságosan távol vannak a modellezett valóság közvetlen leírásától, ezért rendszerint felhasználják a szemantikai adatmodelleket is a tervezésnél, első lépcsőként, és az elkészült SDM-et konvertálják át DBMS adatmodellre.

A modellek szerepének kiemelésével újrafogalmazhatjuk az adatbázis tervezésének főbb lépéseit:

- Igényfelmérés és analízis.
- Koncepcionális adatbázismodell elkészítése.
- DBMS rendszer kiválasztása.
- A fogalmi modell átkonvertálása adatbázis adatmodellre.
- A fizikai adatmodell tervezése.
- Adatbázis implementálása.

Az egyes tervezési lépések módszereinek és eszközeinek részletes tárgyalására egy későbbi fejezetben kerül sor.

Elméleti kérdések

1. Ismertesse az információs rendszer fogalmát, és főbb jellemzőit.
2. Adja meg az információ fogalmát és vetületeit.
3. Hasonlítsa össze az adat és információ fogalmait; az adattárolás főbb formái.
4. Jellemezze a konkurens hozzáférésből eredő nehézségeket.
5. Mire szolgál az integritásőrzés mechanizmusa?
6. Adja meg az állomány szerkezetének építő egységeit; állomány szervezési módszerek.
7. Adja meg az adatbáziskezelő és az adatbázis rendszer fogalmát.
8. Hogyan definiálható az adatbázis fogalma?
9. Ismertesse a B-fa felépítés algoritmusát.
10. Mit jelent a Lost update jelenség fogalma?
11. Ismertesse az adatbázis használatának előnyeit, hátrányait.
12. Adja meg az index struktúra szerepét, a B-fa fogalmát, jellemzését.
13. Az adatbázis ANSI SPARC modellje, és az adatbázis függetlenségi szintjei.
14. Ismertesse az ISAM struktúrát.
15. Hogyan működnek az alap hash algoritmusok?
16. Mit jelent a séma kifejezés?
17. Mik az adatkezelő nyelvek főbb típusai?
18. Milyen főbb komponensekből áll a DBMS?
19. Mire szolgál az utasítás értelmező és az optimalizáló modul?
20. Mik a DBMS Storage System elemei?
21. DBMS osztályozási szempontjai, létező DBMS termékek.
22. Milyen DBMS adatmodell típusok léteznek?
23. Sorolja fel az SE piramis rétegeit.
24. Mit reprezentál az SE piramis alakja?
25. Milyen feladatokra használna DBMS-t?
26. Milyen módon adhatók parancsok a DBMS-nek?
27. Milyen értelemben rejti el a DBMS a DB-t?
28. Hasonlítsa össze a hash és az index alapú elérési módszereket, melyik mikor előnyösebb?
29. Maximum mennyi elem helyezhető el egy N szintű B-fában?
30. Adjon meg egy algoritmust a B-fából való törlésre.
31. Milyen költséggel lehet megkeresni a B-fában egy adott kulcsú rekordot?
32. Mi a különbség a soros és a láncolt elérés között?
33. Mi a hash algoritmus erőssége és gyengesége?
34. Mennyiben hasonlít és tér el egymástól egy táblázat kezelő és egy DBMS?
35. Mit jelent a VLDB és a DBA kifejezés?
36. Mi a leggyakoribb hash függvény és miért?
37. Mi ronthatja le a hash módszerek hatékonyságát?
38. Milyen függetlenségi szinteket lehet értelmezni a DBMS esetében?
39. Mi a különbség a halmaz és rekord orientált megközelítések között?
40. Sorolja fel és jellemezze az adatbázis tervezés főbb lépéseit.
41. Miért szokott prím szám lenni a hash tábla mérete?

Feladatok

1. Építsen fel egy B-fát az alábbi elemekből, melyek beépülési sorrendje adott. A fa fokszáma 4, és a beszúrandó elemek listája: 45,2,34,1,67,21,26,54,12,43,28,32.
2. Építsen fel egy B-fát az alábbi elemekből, melyek beépülési sorrendje adott. A fa fokszáma 5, és az elemek listája:145,22,134,21,267,121,126,54,212,243,128,32.

*3. Építsen fel egy B-fát az alábbi elemekből, melyek beépülési sorrendje adott. A fa fokszáma 4, és a beszúrandó elemek listája: 6, 12, 9, 2, 5, 4, 15, 20, 1, 3, 10, 14, 17, 16, 21, 25, 24.

*4. Építsen fel egy alap hash táblát az alábbi elemekből, melyek beépülési sorrendje adott. A hash függvény: x mod 7, egy bucket kapacitása 3 rekord, és az elemek listája: 45,2,34,1,67,21,26,54,12,43,28,32.

5. Adjon meg olyan bemenő adatsort, mely mellett az x mod 7 hash függvény alkalmazásával túlcsordulás lép fel. A bucket kapacitása 4 rekord.

2. fejezet

Szemantikai adatmodellek

2.1. Adatmodellek

Mint már említettük, az adatbázisok készítésénél és használatánál modellekben kell gondolkodnunk. Az egyes DBMS-ekhez saját modellek tartoznak, amelyekben megvalósul a felhasználó vagy tervező, és az adatbázis kapcsolata. Az adatok tárolásának leírására szolgáló modelleket adatmodelleknek nevezik. A következőkben az adatmodell fogalmát részletesebben fogjuk elemezni.

Nézzük, mit is tudunk már az adatmodellekről. Az előző fejezet alapján megállapíthatjuk, hogy az *adatmodell*

- egy eszközrendszer, amellyel leírható a vizsgált valóság;
- több különböző absztrakciós szinten is létezhet;
- megkülönböztetünk DBMS-hez kötődő és emberközeli adatmodelleket.

Ha tehát mi magunk kívánnánk egy saját adatmodellt létrehozni, aminek egyébként semmi akadálya sincs (az irodalomban is számtalan különböző adatmodell változat lelhető fel), akkor mindenekelőtt egy olyan eszközrendszert, fogalom- és jelölésrendszert kellene létrehoznunk, melyek elemeit a modellezett valóság különböző elemeihez hozzárendelve, leírható vele a problématerület minden lényeges eleme. Az egyes adatmodellek lényegileg az alábbi szempontokban térhetnek el egymástól:

- milyen szinten írják le a valóságot,
- a valóság mely elemeire terjednek ki, és
- milyen jelölésrendszert használnak.

Elméletileg a különböző elemek változtatásával a megoldások elképesztően széles skáláját lehetne létrehozni. Példaként vehetünk egy olyan modellt, melyben a modellezett elemeket lerajzoljuk. Ez a megoldás ugyan mindenki számára könnyen érthető jelölésrendszert ad, viszont nyilvánvalóan láthatók e javaslat hátrányai is: a jelölésrendszer csak szűk problémakörre alkalmazható, a valóság egyes elemei nem fejezhetők ki vele, és nehézséget okozna a számítógépes értelmezés is. Az értelmes

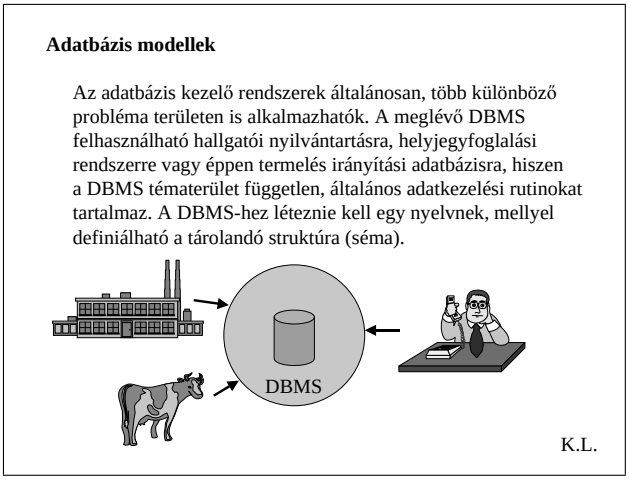
és hasznos adatmodellek köre tehát sokkal szűkebb az elméletileg létrehozható modellek körénél, de még így is számtalan modellváltozat él a gyakorlatban.

A különböző adatmodellek megjelenésével együtt felmerült az igény e fogalom pontosabb jelentésének tisztázására is. Szükség van tehát egy egységes értelmezésre, mely irányt mutat a későbbi adatmodellek kidolgozására, és amely alapján megítélhető az egyes változatok jósága is.

Az adatmodell pontos fogalmának meghatározása az 1980-as évek elején jelent meg az irodalomban. Pontosabban, a nagyobb nyilvánosság számára 1981-re jelent meg Codd cikke, melyben elsőként adja meg az adatmodell egzaktabb értelmezését. Ezen megközelítést aktualizálva, mi a következőkben adjuk meg az adatmodell jelentését.

> *Az adatmodell olyan matematikai formalizmus, mely a valóság adatorientált leírására alkalmas. Az adatmodellnek a valóság teljes értékű megadásához az alábbi három komponenst kell tartalmaznia:*
>
> – *strukturális rész, mely a valóságban megtalálható adattípusok és kapcsolataik leírására szolgál;*
> – *műveleti rész, mely felhasználásával különböző lekérdezési vagy módosítási tevékenységeket végezhetünk;*
> – *integritási rész, mely az adatbázisban megvalósuló adattípusokra, adatértékekre és kapcsolatokra, valamint az elvégezhető műveletekre ad megszorítást.*

E hármas tagolódás az adatmodellek lényeges jellemzője, és a gyakorlati DBMS adatmodellek leírásaiban, utasításaiban is jól megfigyelhető e komponensek jelenléte. E komponensek jelentésének bemutatására vegyünk egy egyszerűsített példát.



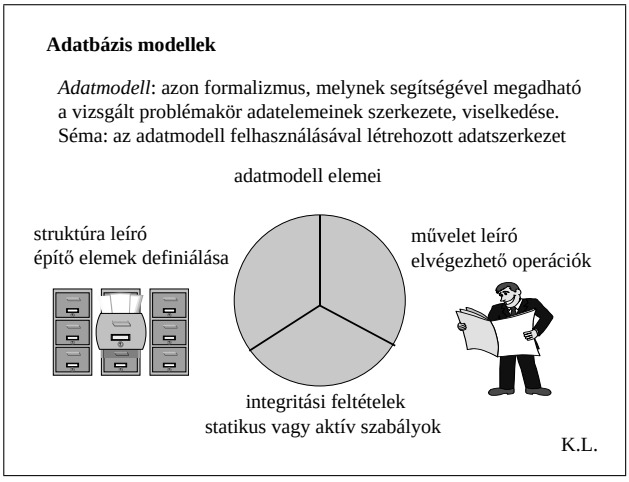
2.1. ábra. Adatbázis modellek

Egy bankkártya kezelő információs rendszert kiválasztva, az adatbázis leírására szolgáló adatmodell esetén, a felhasznált adatmodellnek

- a strukturális részben biztosítania kell a lehetőséget az ügyfelek, a kártyák és a számlák adatainak nyilvántartására, illetve eszközt kell adnia az ügyfél és kártya, valamint az ügyfél és számla összetartozások nyilvántartására is;
- a műveleti részben lehetőséget kell adnia a kártyához tartozó számlák és ügyfelek adatainak lekérdezésére, és a számlákhoz kapcsolódó kifizetések és befizetések teljesítésére;
- az integritási részben eszközt kell adnia arra, hogy az adatbázisba beépíthessünk értékekre vagy műveletekre vonatkozó megszorításokat: mint például azt, hogy csak annyi pénz adható ki, hogy a számlán maradjon még X összeg. Egy másik megszorítás lehet az, hogy egy kártyához csak egy ügyfél tartozhat, és nem létezhet két azonos kódszámú kártya.

A modell egyes komponenseit igen különböző módon és formában lehet megvalósítani. Tekintsük át ezért a következőkben, milyen szempontokat lehet figyelembe venni egy adatmodell értékelésénél, mikor tekinthetünk egy adatmodellt jónak. E szempontokat egy listában foglaljuk össze.

- *Elméletileg megalapozott*: az adatmodellek viselkedésének, kapacitásainak és továbbfejlesztésének megítélésénél számos előnnyel jár, ha a modell biztos matematikai alapokon nyugszik. E téren kiemelhető a relációs adatmodell, melynek egzaktsága kiemelkedik a többi modell közül.
- *Megfelelő absztrakciós szint és egyszerűség*: a modellnek minél szélesebb körben érthetőnek, elfogadottnak kell lennie. Igazodjon az alkalmazási körülményekhez. Az egyszerűség, a közérthetőség mellett a megbízhatóságot is növelje.



2.2. ábra. Adatmodell és séma fogalma

- *Teljesség*: a modellnek lehetőség szerint a valóság minden igényelt elemére ki kell terjednie. Sokszor egyszerűbb elemekkel is lehet összetettebb dolgokat modellezni (például az értékek egyediségének ellenőrzését gyalogmódszerrel, azaz az értékek átolvasásával is el lehet végezni, ha nincs lehetőség ilyen integritási feltétel megadására, azonban ezek sohasem egyenértékűek a testreszabott megoldással).
- *Megvalósíthatóság*: a DBMS adatmodellek fontos kritériuma, hogy a rendelkezésre álló hardver és szoftver technológiák mellett az adatmodellt hatékonyan, elfogadható végrehajtási idő mellett tudja kezelni.

A teljesség kedvéért megemlítjük, hogy az adatmodell megadott értelmezése mellett egyéb megközelítések is léteznek. E megközelítések jellemzője, hogy sokkal tágabban veszik az adatmodellek körét. Egy ilyen értelmezés olvasható például Halassy könyvében, mely szerint az adatmodell az egyed-, tulajdonság- és kapcsolat típusok, ill. az ezekre vonatkozó korlátok szervezett együttese. Mint látható, ez a megközelítés két ponton tér el lényegesen az előző definíciótól:

- nem tesz említést a műveleti részről, csak a strukturális komponens meglétét kívánja meg,
- a strukturális komponens megadásánál konkretizálja a modell leíró elemeit az egyed-, tulajdonság- és kapcsolat típusokra.

Itt nem térünk ki, hogy mit is értünk pontosabban az egyed és a tulajdonság alatt, mert ezeknek a fogalmaknak egy külön alfejezetet fogunk szentelni az ER modellek keretében. A most megadott definícióról mindesetre megállapítható, hogy igen szoros kapcsolatban áll egy létező adatmodellel, az ER modellel. Ugyanis a gyakorlatban adatmodellnek tekintett rendszerek döntő része nem tesz eleget az általunk megadott kritériumoknak. Úgy is mondhatjuk, hogy a mi definíciónk egy szigorú, szűkebb értelemben vett adatmodellt jelöl ki. A lazább, tágabb értelemben vett megközelítés szerint az adatmodelleknek nem szükségszerű része a műveleti és integritási rész. Így a tágabb értelmű adatmodellek körébe felvehető egy sor olyan modell is, melyek a tervezés elősegítése céljából a valóság strukturális vetületének a leképzésére szorítkoznak. A következő fejezetben – a szemantikai adatmodellek között – részletesebben is meg fogunk ismerkedni néhány ilyen szűkebb értelemben vett adatmodellel.

Az adatmodell definícióját azonban nemcsak szűkítési céllal lehet módosítani. Codd értelmezésében az adatmodellhez a fenti három komponens mellett hozzátartozik még egy negyedik, az úgynevezett értelmezési, *interpretációs* rész is, amely megadja, hogy mi a jelentése az egyes modellelemeknek. Eszerint a jelentés nélkül létrehozott modell nem használható pragmatikus célokra. E felfogás jogosságát vitatja Gilula, aki szerint különbséget kell tenni információmodell és adatmodell között. E két fogalom viszonya hasonló az információelméletben szereplő jelentés és jelhordozó fogalmak közötti különbséghez. Az egyikhez tartozik értelmezés, jelentés, míg a másik formális leírást takar.

Gilula értelmezése szerint csak az információs modellhez szükséges jelentést, interpretációt csatolni. Az adatmodell ezzel szemben formális leírást jelent, így nem szükséges hozzá interpretációt is kötni. Ez az érvelés azon a tényen alapszik, hogy egy adatbázis szerkezetét akkor is fel tudom tárni, ha nem ismerem az

egyes szerkezeti elemek jelentését, és akkor is tudunk adatokat lekérdezni és az integritási szabályok által meghatározott kereteken belül felvinni és módosítani, ha nem vagyunk tisztában az adatok, az értékek jelentésével.

2.2. Szemantikai adatmodellek áttekintése

A szemantika szó magyarul egy jelsorozat jelentése. A *szemantikai adatmodell* (SDM) kifejezés nem egy önálló modellt jelöl, hanem egy modellcsaládot, melybe több különálló modell is beletartozik. Ami közös ezekben a modellekben, hogy mindegyik a felhasználóhoz közelálló, jelentésgazdag szemantikai eszközkészlettel, modellel írja le a modellezett valóságot. A *szemantikai adatmodellek* célja az, hogy a valóság leírását a számítógépnél megszokott egyszínű, szintaktikai kezelés, leírás helyett szemantikailag is gazdagabbá tegye. Itt például gondolhatunk arra, hogy milyen hasznos lenne, ha a DBMS tudná, hogy nincs értelme mondjuk az autó súlyát a gyártási évével összehasonlítani, habár szintaktikailag mindkettő azonos, hiszen egy-egy numerikus értékkel adhatók meg. A szemantikai modellek célja olyan leírást nyújtani, amelyben sokkal sokrétűbben adhatók meg a valóságban fennálló viszonyok.

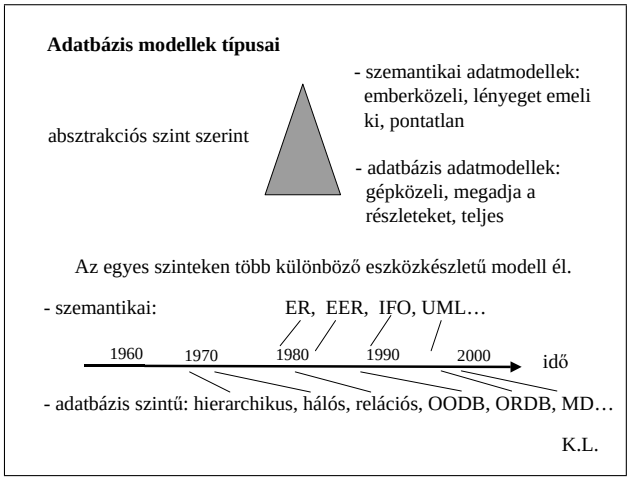
Mint már említettük, az SDM modelleknek ma még elsődlegesen csak a tervezés során van szerepe, mivel még nem készültek SDM modellen alapuló DBMS implementációk. Ha közvetlen megvalósításuk nincs is, de igen erős hatást fejtettek ki az egyes SDM tervezetek az adatbázismodellek kutatásában, melyekből a jövő DBMS modelljei fejlődhetnek ki. Ezt jól példázza az objektum-orientált adatmodellek elterjedése, melyek nem is olyan régen még csak SDM formájában léteztek, mint tervezési segédeszközök, mára azonban már DBMS modellé nőtték ki magukat.

Az SDM modellek igen különböznek eszközkészletükben, ábrázolási módjukban, de sok közös vonás is felfedezhető bennük. Közösek a modellek céljai és a legfontosabb irányelvek, melyek e célok elérését irányozzák elő.

A valós világ tudatos modellezésének kérdése az emberiség igen régi problémája. Ismereteink szerint, már az antik görög kultúrában, főleg Arisztotelész nevéhez kapcsolódóan, megjelent az a ma is élő világszemlélet, mely szerint a világ több egymástól elkülöníthető egységből áll, melyek között különböző strukturális, asszociációs kapcsolatok állhatnak fönn. Magunkon is megfigyelhetjük, hogy amikor egy problémakört modellezünk, leírunk, a valóságot valamiképpen lebontjuk több, egymástól jól elkülöníthető, önálló léttel bíró egységre, melyeket *objektumoknak* is nevezhetünk. A modellezés során önálló fogalmakat használunk az egyes objektumok azonosítására. Az egyes objektumok között különböző *asszociációkat* fedezhetünk fel, melyek utalhatnak tartósabb, mélyebb, strukturális kapcsolatokra, de lehetnek ideiglenes jellegű kapcsolatok is. Egy iskolai osztály modellezésénél önálló egységként megjelenő fogalmak lesznek többek között a diák, a név, az életkor, a lakcím, az iskolai pad, stb. Az egyes objektumok közötti tartósabb kapcsolatokra lehet példa a diák és a név, a diák és a lakcím (feltéve, hogy minden diáknak rendelkezni kell névvel, lakcímmel), míg a diákok közötti baráti viszony egy lazább kapcsolatot reprezentál a modellen belül.

Az SDM modellek célja a bennük megjelenő valóság minél hűbb és teljesebb leírása, mely során a számítástechnikai realizálhatóság hatékonysága csak másodlagos szerepet játszik. Az SDM modellekben megvalósuló legfontosabb törekvések a következő pontokban foglalhatók össze:

- Az SDM-nek lehetőséget kell adnia a modellezett világ *emberközeli, természetes leképzésére.* A modellnek expliciten ki kell tudnia fejezni az adatbázis jelentését. A modell ne tárolási hatékonyság orientált, hanem felhasználó orientált módon írja le a modellezett valóságot, ezáltal hatékonyabb eszközt adva a tervezőnek.
- Az absztrakciós szintek megnövelése. Az adatmodellnek lehetőséget kell adnia a valóságban megjelenő *komplex objektumok rugalmas leképzésére*, mely során a kevésbé fontos részletek fokozatos eltüntetésével (*információ elrejtés*) a kívánt absztrakciós szintre hozhatjuk az objektum leírását.
- A modellben több absztrakciós szint is megvalósítható, melyekből a felhasználó szabadon, rugalmasan választhat a feladat jellegétől függően. A modellnek lehetőséget kell adni a relatív nézetek, szemléletmódok kifejezésére is, ne legyen a tervező keze megkötve a szemantika megjelenítésénél.
- Az objektumok közvetlen, direkt leképzése, az *objektum egysége* álljon a középpontban. Ez a törekvés arra irányul, hogy a modellezett világ bármely, tetszőlegesen komplex objektuma egyedileg, a modell egyetlen elemével leírható legyen. Azaz ne kelljen a valóságban egy egyedként előforduló objektumot mesterségesen szétbontani, mivel a modellben csak így írható le az objektum. Ezáltal a modellnek képesnek kell lennie tetszőleges komplex struktúrák egy elemként történő megjelenítésére is.
- *Teljesség.* A teljesség alatt azt értjük, hogy a modellben minden olyan eszköznek rendelkezésre kell állnia, mellyel bármely modellezett problé-



2.3. ábra. Adatbázis modellek típusai

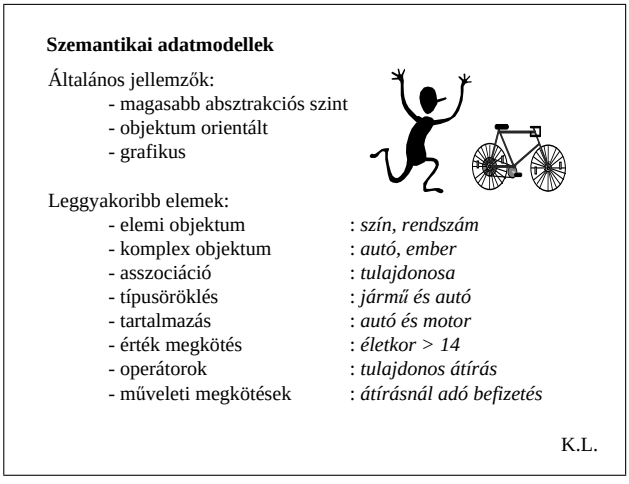
makör valósághűen, szemantikai veszteség nélkül írható le. Ez megköveteli a modell eszköz- és fogalomkészletének jól átgondolt kiválasztását, hogy lefedjen minden lényeges szemantikai elemet. Ez természetesen nem azt jelenti, hogy a valóság minden szemantikai eleméhez léteznie kell egy modellelemnek, hanem megengedhetők olyan specifikumok is, amelyeket a modell több eleméből felépített konstrukciókkal lehet csak leírni.

– *Egzaktság.* A modell eszközkészletének, leíró nyelvének egyértelműnek, pontosnak kell lennie, nem fordulhat elő benne bizonytalanság, többértelműség. Az egzakt nyelvezet biztosítja az elkészített modellek
  – konzisztenciáját (ellentmondás-mentességét),
  – jóságát,
  – egyértelműségét.

Az SDM modellekben a fogalmak két nagy csoportját különböztethetjük meg, hiszen az SDM is, mint minden más adatmodell, tartalmazhat statikus, struktúra leíró elemeket, és a műveletek leírására alkalmas dinamikus elemeket.

A *statikus elemek* leggyakoribb képviselői:

– *Elemi egyedtípusok és típuskonstruktorok.* A típuskonstruktorok segítségével az elemi típusokból tetszőleges struktúrájú új típusok hozhatók létre. Az elemi adattípusok lehetnek standard típusok, mint az egész vagy karakter típus, és lehetnek absztraktabb jellegűek is, mint a hivatkozás (ref vagy pointer) típus. A típuskonstruktorok között kiemelt szerepet játszanak a csoportképzés és az aggregáció operátorai. Az aggregáció különböző típusokat fog össze, azaz az eredményül kapott új típus minden előfordulása a hozzá tartozó típusok egy-egy előfordulását fogja tartalmazni. Az aggregáció jól használható a kapcsolatok tárolására.



2.4. ábra. Szemantikai adatmodellek

A legtöbb modell megengedi az aggregáció ortogonális bővítését, azaz amikor az aggregáció egy másik aggregáció típust is tartalmazhat. A csoportképzés egy csoportot képez a megadott típusú egyedekből, azaz a csoportképzéssel megalkotott egyedtípus minden előfordulása az alaptípus több előfordulásának halmazát tartalmazza. A csoportok és aggregációk egymásba ágyazásával tetszőleges mélységű struktúrák, típusok alakíthatók ki.

– *Tulajdonságok.* A tulajdonság az objektum egy jellemzőjét tartalmazza. A tulajdonság mint az egyed része, felfogható az objektum struktúra tagjaként is, ami egy passzív elem. Ebben a megközelítésben az alapobjektum egy aggregációként értelmezhető. Egy másik megközelítési mód az, hogy azt az állítást, hogy Péter kora 38 év, úgy értelmezzük, hogy a Péter egyedhez hozzárendeljük a kor típus egy egyedét, a 38 évet.
Tehát a tulajdonságot, mint hozzárendelést értelmezzük, mely az egyik egyedtípus egy egyedelőfordulásához hozzárendeli egy másik egyedtípus egy egyedelőfordulását. Ez a leképzés függvényként jelenik meg, így a tulajdonság mint függvény is értelmezhető. Ez a megközelítés egy aktívabb szemléletmódot tükröz.

– *Specializáció* (IS_A) kapcsolat. Az egyedek, objektumok közötti kapcsolatokban kiemelkedő szerepet játszik a specializáció. Ez azt jelenti, hogy az egyik objektum a másik objektumnak egy speciális megvalósulása, tehát hordozza mindazon tulajdonságokat, melyek a szülő objektumot is jellemzik. A hallgató és az oktató is egy speciális megjelenése az ember objektumnak, a hallgatóhoz is mindazon tulajdonságok hozzárendelhetők, ami egy általános emberre jellemző, de itt kötött a foglalkozás tulajdonság értéke, illetve emellett létezhetnek csak a hallgatóra jellemző tulajdonságok is, mint például a tankör tulajdonság. Bizonyos modellek nemcsak az általánosabb típusból történő specializálást engedik meg, hanem a fordított eljárást is, amikor a speciálisabb típusokból képezünk egy általánosabb típust. A specializáció fogalmát eddig elsődlegesen egy típus-tulajdonság öröklési mechanizmusként értelmeztük, de emellett ez a fogalom is értelmezhető másfajta megközelítésben is. Ha arra gondolunk, hogy a specializáció révén a speciális típus bármely előfordulása egyben az általános típus előfordulása is, vagyis amely feltételek az általános típus előfordulásaira teljesülnek, azoknak a speciális típus előfordulásaira is teljesülni kell, akkor a specializáció egyfajta integritási feltételként kezelhető. Mind a specializáció, mind az általánosítás ugyanazon eredményhez vezet, csak a komponensek létrehozásának sorrendje különbözik. Az általánosításnál előbb a speciálisabb típusok jönnek létre, és ezeket követi az általánosított típus. A specializációnál épp fordított a tevékenységi sorrend. A kialakult szerkezet rendszerint bonyolultabb egy normál fánál, hiszen ugyanazon típus több más típus általánosítása vagy specializációja is lehet. Ez a típusmegosztás jelensége. Ha a specializációnál az általános típus minden előfordulása csak maximum egy speciális, leszármazott típushoz tartozik hozzá, akkor *diszjunkt specializálás*ról beszélhetünk.

- *Meta egyedtípusok.* A meta egyedtípusokat az elvontabb fogalmak leírására lehet felhasználni, melyek a létező egyedtípusok alapján definiálhatók. A meta egyedtípus olyan típust jelent, melynek előfordulásai maguk is típusok, és ekkor minden előforduláshoz, vagyis típushoz hozzárendelhető egy vagy több tulajdonságérték, mely érték tehát az egész típusra, az adott típus minden előfordulására azonos értéket jelent.
- *Statikus integritási feltételek.* Az integritási feltételek az egyedek között fennálló kapcsolatokat írják le és szabályozzák. Ebbe a tulajdonságok közötti kapcsolatok is beletartoznak, hiszen a tulajdonság is értelmezhető objektumként, egyedelőfordulásként.

A *dinamikus elemek* leggyakoribb képviselői:

- A modellben értelmezett *műveletek* köre. A műveletek magukban foglalják az egyedtípusokhoz és egyedelőfordulásokhoz tartozó operációkat: az adatkezelés, adatlekérdezés lehetőségeit.
- *Leszármaztatott tulajdonságok.* Olyan tulajdonságok megadására ad ez a fogalom lehetőséget, melyek értéke egy műveletsor eredménye, tehát más, már létező tulajdonságértékektől függ az értéke. A leszármazást az egyes modellek nemcsak a tulajdonságokra, hanem típusokra is értelmezik. Erre jó példa a tinédzserek típusa, amely az emberek típusából származtatható oly módon, hogy a megadott intervallumba eső korral rendelkező ember előfordulások tartoznak a tinédzser típushoz. Ez a fajta leszármazás szoros rokonságban van a típus specializációval.
- *Triggerek.* A trigger egy alapvető fogalom az adatbáziskezelésben. A trigger egy eseménylekezelő mechanizmus, mely két elemből áll:
  - esemény megadása,
  - választevékenység megadása.

  A választevékenység akkor hajtódik végre, amikor az esemény bekövetkezik. Az SDM-ben a triggerek az adatmodellt érintő változásokhoz – például új egyedelőfordulás felvitele vagy tulajdonság módosítása – kötődnek. Hasonlóan a választevékenységek köre is az SDM műveletekhez kapcsolódik.
- *Műveleti integritási feltételek.* Ebbe a fogalomkörbe a modellben értelmezett műveletekre vonatkozó megkötések tartoznak. A megkötések tipikus esetei, amikor a műveleteket leszűkítjük speciális egyedtípusokra.

A fenti felsorolás megadja az SDM modellek legfontosabb komponenseit. Ezt a felsorolást azonban semmiképpen sem szabad szigorú szabályként értelmezni, hiszen nagyon sokféle SDM modell létezik, melyek egymástól mind kifejezőképességben, mind formalizmusban eltérhetnek. Így több olyan modell is létezik, melyek a felsorolt elemeknek csak egy részét, néha csak egy töredékét tartalmazzák. Így a fenti felsorolás inkább összesítésnek tekinthető, mely lefedi az egyes SDM modelleket.

Nyilvánvalóan felmerül a kérdés, hogy ha a fenti komponensek együttese biztosítja a legjobb megoldást, miért léteznek még egyéb, egyszerűbb SDM modellek

is. Erre a kérdésre a válaszunk az, hogy e szempontok fontossága nem egycsapásra vált ismertté a köztudatban, hanem hosszabb folyamat eredményeként alakult ki. Előbb egyszerűbb modellek jöttek létre, melyek folyamatosan bővültek újabb és újabb elemekkel. Másrészt azt is be kell vallani, hogy bár azt mondtuk, hogy az SDM esetén nem elsődleges szempont a DBMS megvalósíthatóság hatékonysága, azonban teljesen nem vethető el ez sem. Így az olyan SDM modellek elterjedése valószínű, melyek jól igazodnak a létező DBMS kezelő nyelvekhez. Így például hiába tudunk az SDM modellben műveleti integritási feltételeket megadni, ha az alkalmazott DBMS ezt a szolgáltatást nem tudja nyújtani. A gyakorlatiasság igénye tehát az egyszerűbb SDM modelleket is életben tartja.

Zárójelben megjegyezzük, hogy ez a gyakorlatiasság hosszabb távon azzal járhat, hogy egy szintaktikailag erősebb DBMS installálása után újra kell tervezni (re-engineering) az addigi alkalmazás modelljét, hogy kihasználhassuk az új rendszer előnyeit.

A következőkben áttekintjük a gyakorlatban legfontosabb SDM modelleket, melyeket majd az adatbázisok tervezése során fogunk felhasználni. Előbb az egyszerűbb, a relációs DBMS-ekhez közel álló ER adatmodellt vesszük, majd a fejlettebb objektum orientált irányba haladunk tovább.

2.3. Az ER adatmodell

A kidolgozott számtalan SDM modell közül elsőként a legegyszerűbb, gyakorlatban igen elterjedt módszert, az ER modellt vesszük át. Az ER vagy E/R modell, vagy rövidítés nélkül az *egyed-kapcsolat* (Entity Relationship) modell az SDM modellek legismertebb képviselője. Az ER modell alapjai 1976-ban jelentek meg, Chen publikációja nyomán. Az ER modell igen szoros kapcsolatban áll a korábban kidolgozott relációs adatmodellel, habár indulásként önálló adatmodellként kívánták bevezetni, ma már számos kiegészítés és javítás után, mint a relációs modellezés bevezetőjeként alkalmazzák. Előnyei közé tartozik az egyszerűség, a szoros kapcsolat és a könnyű konvertálhatóság a relációs modell felé.

Az ER modell, mint azt a neve részben mutatja, három alapelemen nyugszik: az *egyedeken* (entity), az egyedek közötti *kapcsolatokon* (relation) és az egyedek *tulajdonságain* (attributes). Az ER modell kizárólag a valóság strukturális leírására szorítkozik, megengedve bizonyos egyszerűbb integritási feltételeket is. Ezen egyszerűség miatt korábban vita bontakozott ki, hogy mennyiben tekinthető egyáltalán adatmodellnek az ER rendszer. Codd, a relációs modell atyja, így jellemezte a modellt:

> *"Nem nevezném az ER rendszert adatmodellnek, mivel egy adatmodellnek többet kell nyújtania az adatbázis tervezés támogatásánál. Nincs igazi megvalósulása, hiányzik a pontosság és az egyértelműség."*

Codd értelmezésében tehát az ER modell csak az adattervezést támogatja, nem biztosít megfelelő precizitást, ezért nem tekinthető adatmodellnek. Mi azonban már kissé tágabban értelmezzük az adatmodell fogalmát, hasonlóan Date-hez, aki szerint:

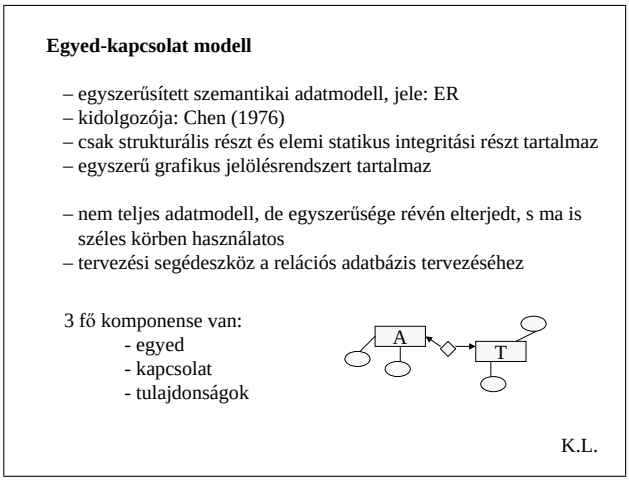
"Az ER modell valóban nevezhető adatmodellnek, amely azonban csak egy kis réteg az alap relációs adatmodell felett."

Mint látni fogjuk, az ER modellel egy egyszerű és elterjedt modellezési segédeszközt fogunk megismerni. Az ER alapfogalmak pontosabb megismeréséhez induljunk ki a modellezett világból. Hogyan írnánk le mondjuk egy egyetem szerkezetét? Azt mondhatnánk, hogy az egyetemen vannak hallgatók, tanárok, tantárgyak, órák. Minden hallgatót meg tudunk különböztetni egymástól, hasonlóan a tanárokhoz, tantárgyakhoz vagy órákhoz. Mindegyik a világ egy önálló darabja, szereplője, korábbi szóhasználattal élve objektuma.

Az ER terminológiában a modellezett világ azon szereplőit, melyek önálló léttel bírnak és melyekről több különböző információt tartunk nyilván, egyedeknek nevezik.

Így egyednek tekinthető például a hallgató, akiről nyilvántartjuk többek között a nevét, a korát, az érdemjegyeit, stb. Az egyedek között vannak hasonló szerkezetűek, és vannak egymástól igen különböző felépítésűek. A hasonló felépítésű egyedek - például hallgatók - alkotnak egy *egyedtípust.* Ilyen egyedtípus a tanárok, a tantárgyak típusa is. Ekkor minden egyedtípus több *egyedelőfordulást* ölel át, ahol egy egyedelőfordulás egy konkrét egyedet, egy konkrét hallgatót, vagy egy konkrét tanárt jelöl. A valóságban jól meg tudjuk különböztetni az egyik hallgatót a másiktól, az egyik tantárgyat a másiktól. Hogyan tesszük ezt? Úgy, hogy az egyik hallgató barna hajú, a másik fekete hajú, vagy az egyik hallgatót Nagy Gabriellának hívják, a másikat pedig Varga Tibornak. Tehát mindegyik hallgató rendelkezik egy sor olyan tulajdonsággal, melyek más-más értékeket vehetnek fel az egyes hallgatóknál.

A tulajdonság az egyedhez kapcsolódó, leíró szerepet betöltő értéket jelölő elem.



2.5. ábra. Az egyed-kapcsolat (ER) modell

Az egyedelőfordulásokat is az egyed tulajdonságai alapján azonosíthatjuk be. Az egyes egyedtípusok pedig elsődlegesen a típushoz tartozó tulajdonságok körében térnek el egymástól. Hiszen más adatokat tartunk nyilván egy tanárról, mint egy tantárgyról. Az egyedtípus lényeges jellemzője tehát a hozzá tartozó tulajdonságok köre. Az egyedelőfordulások és az egyedtípusok azonban nem izolált, elszigetelt szereplői a modellezett világnak, az egyedek kapcsolatban állnak más egyedekkel, így összetettebb struktúrát hozva létre. A példánknál maradva egy hallgató több tantárgyhoz is kötődik, egy tanár is kapcsolódhat több tantárgyhoz.

A kapcsolat az egyedek közötti asszociációs viszonyt ábrázolja.

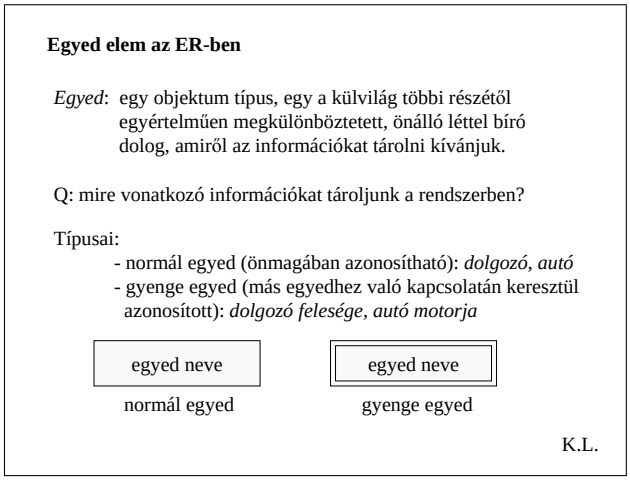
Az egyedek között különböző bonyolultságú kapcsolatok állhatnak fenn, és a modell akkor jó, ha alkalmas a kapcsolatok árnyalt kifejezésére. Az ER modell a valóság hűbb leképzése érdekében az egyszerű egyed és kapcsolat elemek mellett megkülönbözteti mind az egyedeknek, mind a kapcsolatoknak több változatát. E változatok között lényeges jelentésbeli és viselkedési különbségek vannak, és ezenkívül egészen másféle relációs modellbeli elemekre képződnek le.

Az ER modell egyik lényeges tulajdonsága, hogy *grafikus jelölésrendszert* alkalmaz. A grafika, a szöveges leírástól eltérően sokkal kifejezőbb és lényegretörőbb az emberek számára, így kiválóan alkalmas a fontosabb fogalmak és kapcsolatok kiemelésére. A ma használatos ER modell teljes elemkészletének grafikai szimbólumait a következőkben adhatjuk meg:

Egyed: egy a külvilág többi részétől egyértelműen megkülönböztethető dolog, objektum.

Altípusai:

– *Normál egyed*: rendelkezik olyan tulajdonságcsoporttal, mely egyértelműen azonosítja az egyedet. Egy autó például normál egyed, hiszen mind



2.6. ábra. Egyed elem az ER modellben

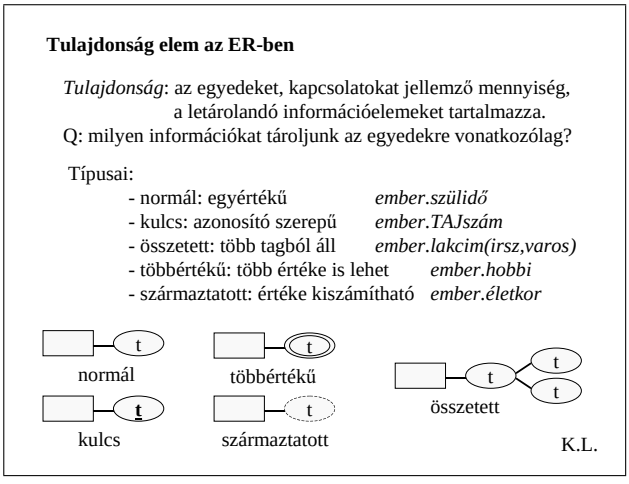
a rendszáma, mind a gyártási száma egyedi azonosítóként szolgálhat. A jele téglalap, melynek belsejében az egyedtípus azonosító neve áll.

– *Gyenge egyed*: nincs azonosító tulajdonságrendszere, így más egyedhez fűződő kapcsolata szükséges az azonosításához. Előfordulhat például, hogy egy személyt nem önmagában azonosítunk, mert nem tudjuk az azonosító adatait, csak a nevét és azt hogy egy másik azonosított személynek valamilyen rokona. Így az illető egyértelmű kijelöléséhez szükség van a másik ismert személy megadására is. A gyenge egyed grafikus jele a dupla kerettel rajzolt téglalap, középen az azonosító névvel.

Tulajdonság: az egyed egy meghatározott jellemzője.

Altípusai:

– *Egyszerű tulajdonság*: egy elemi értékkel leírható tulajdonságot ad meg. A testmagasság például egyszerű tulajdonság, hiszen egy skalár szám elegendő a megadásához. A hobbi ezzel szemben nem egyszerű tulajdonság, hiszen több értéket is felvehet egyidejűleg az ember egyed esetén, hiszen egy embernek több hobbija is lehet. A tulajdonságot ellipszisben adjuk meg, az ellipszis közepébe írva a tulajdonság azonosító nevét. Mivel tulajdonság önmagában nem állhat, ezért mindig meg kell adni, hogy mely egyedhez (vagy kapcsolathoz) kötődik. A kapcsolódást egy vonallal jelöljük, amely a megfelelő tulajdonságot és az egyedet köti össze.
– *Összetett tulajdonság*: olyan tulajdonság, amely több elemi tulajdonság együttesére bontható. Ilyen tulajdonság például a lakcím, amely felbontható a város, utca, házszám, lakásszám elemi adatok együttesére. Az összetett tulajdonságot is ellipszissel jelöljük, melyhez hozzákötjük az illeszkedő elemi tulajdonságok szimbólumait.



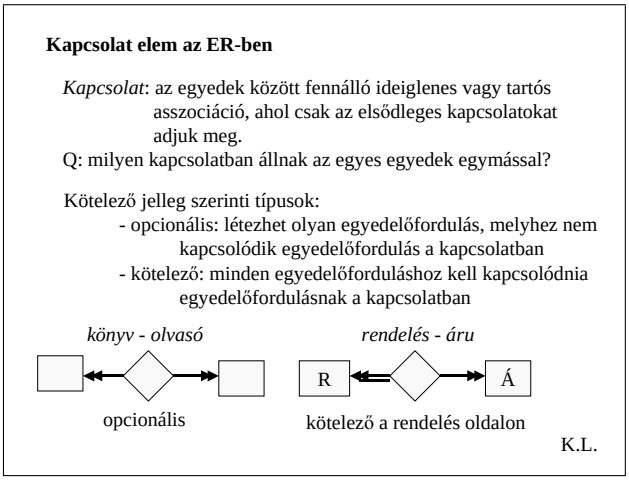
2.7. ábra. Tulajdonság elem az ER modellben

– *Kulcs tulajdonság*: az egyed egyértelmű azonosítására szolgáló tulajdonság. Az autó egyed esetén például a rendszám játszhat azonosító szerepet. Az ábrán a kulcstulajdonságot úgy jelöljük ki, hogy a tulajdonság azonosító nevét aláhúzzuk egy folytonos vonallal.
– *Többértékű tulajdonság*: olyan tulajdonság, amely nem egy elemi értéket, hanem több elemi értéket, az értékek egy tömbjét veheti fel. Így például a dolgozó egyed képzettség tulajdonságának leírására több elemi értéket is meg lehet adni, hiszen több képzettsége is lehet valakinek. Egy elemi érték egy sztringet jelent. A többértékű tulajdonságot egy dupla keretű ellipszissel reprezentáljuk.
– *Leszármaztatott tulajdonság*: olyan tulajdonság, melynek értéke más tulajdonságokból vezethető le, származtatható. Így például egy termék esetén az ÁFA kiszámolható a termék árából és az ÁFA-kulcs mértékéből. Az ER modellben szaggatott vonallal határolt ellipszis a leszármaztatott tulajdonság jele. A grafikonon nem jelöljük, hogy mely más tulajdonságokból és mi módon származtatható az érték.

Kapcsolat: az egyedek között fennálló viszonyt hordozza.

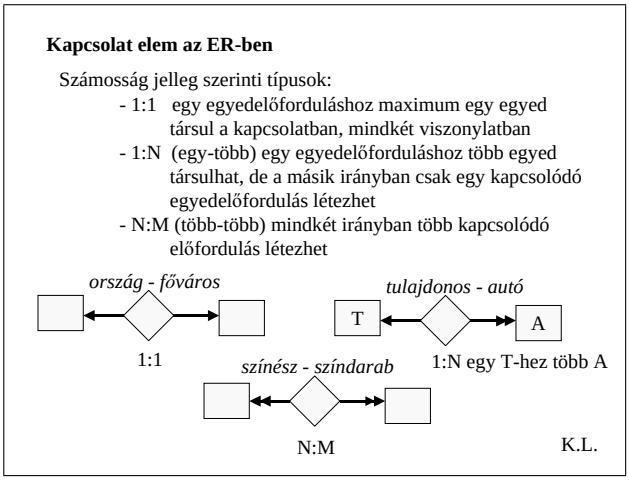
Altípusai:

– *1:1 kapcsolat*: a kapcsolatban mindkét egyedtípus előfordulásai csak egyetlen egy előforduláshoz rendelődnek a másik egyedtípusból. Így például a házassági kapcsolat a férfi és a nő egyedtípusok között egy-egy jellegű, hiszen egy házasságban csak egy férfi és egy nő előfordulás kerül kapcsolatba egymással. A kapcsolatot egy rombusszal szokás jelölni, melybe megadják a kapcsolatot leíró azonosító nevét. A négyszög átellenes csúcsaiból egy-egy nyilat húzunk, melyek a kapcsolódó egyedekhez vezetnek.



2.8. ábra. Kapcsolat elem az ER modellben I.

– *1:N kapcsolat*: annyiban különbözik az előző kapcsolattípustól, hogy az egyik, mondjuk A egyedtípus előfordulásai több előfordulással tarthatnak kapcsolatot a másik, mondjuk B típusból, de B egy előfordulása továbbra is csak egy A előforduláshoz kapcsolódhat. Az autó és ember kapcsolata lehet példa az 1:N kapcsolatra, hiszen egy autónak csak egy tulajdonosa lehet, de egy ember több autónak is lehet tulajdonosa. Az 1:N kapcsolat ábrázolásánál azon egyedbe, melyből több is kapcsolódhat a másik egyedhez, egy kettősnyíl vezet. Az autó-ember példa esetén az autó egyedbe kell a duplanyílnak vezetnie.
– *N:M kapcsolat*: olyan kapcsolattípus, melyben mindkét egyedtípus előfordulásai több előfordulással is tarthatják a kapcsolatot a másik egyedtípusból. Jó példa az N:M kapcsolattípusra a szereposztás kapcsolat a színészek és a színdarabok között, hiszen egy színész több színdarabban is játszhat, míg egy színdarabban is több színész szerepelhet. A kapcsolat ábrázolásánál mindkét kapcsolódó egyedbe kettősnyíl mutat.
– *N-ed fokú kapcsolat*: a kapcsolatban nemcsak kettő, hanem n egyed vesz részt. A valóságban ugyan a binér kapcsolat dominál, de előfordulhat tercier, vagy magasabb fokszámú kapcsolat is. A hármas kapcsolatra példa a rendelés kapcsolat, melyben a vevő, a szállító és az áru kapcsolódik össze, hiszen egy rendelésnél egy vevő, egy megadott terméket rendel egy megadott szállítótól. Az n-ed fokú kapcsolat ábrázolása abban különbözik a binér kapcsolatok ábrázolásától, hogy a rombuszból több nyíl fut ki.
– *Totális kapcsolat*: egy A egyed totálisan vesz részt a kapcsolatban, ha minden egyedelőfordulása az A-nak részt vesz egy kapcsolatelőfordulásban, azaz nincs olyan A-beli egyedelőfordulás, mely nem kapcsolódna a másik egyedtípus valamely előfordulásához.



2.9. ábra. Kapcsolat elem az ER modellben II.

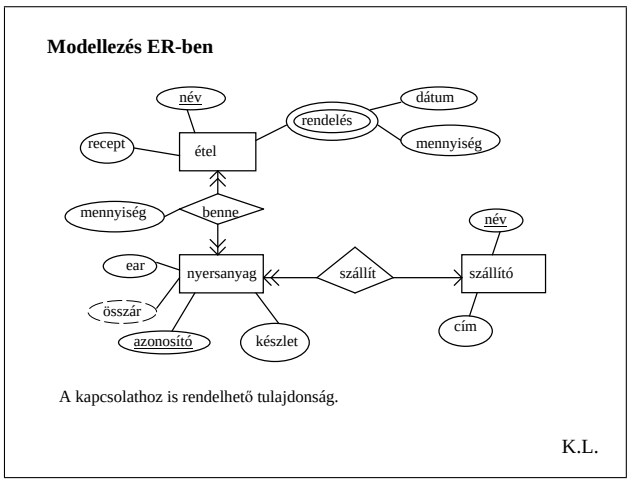
Ha feltesszük, hogy minden autónak van tulajdonosa, akkor az autó totális kapcsolatban van az emberrel a tulajdonosi kapcsolatban. Azon egyedek, melyek nem totálisan vesznek részt a kapcsolatban, parciális kapcsolatot alkotnak. Az előbbi példánál maradva, az ember csak parciálisan vesz részt a tulajdonosi kapcsolatban, mivel lehetnek olyan emberek, akiknek nincs autójuk. A totális kapcsolatban lévő egyedekhez egy dupla szárú nyíl vezet, míg a parciális kapcsolatnál az eddig is használt egyszeres él vezet.

Az ER modell használatára tekintsünk egy egyszerű feladatot, melyben egy képzeletbeli étterem nyilvántartásának egy részletét modellezzük.

A minta modellben három egyed van: az étel, a nyersanyag és a szállító. Mindhárom erős egyed, mivel vannak kulcs tulajdonságaik. A tulajdonságok, a rendelés tulajdonságot kivéve, egyszerű, elemi értékkel rendelkező tulajdonságok. A rendelés pedig összetett, többértékű tulajdonságként szerepel a modellben. Egy ételre tehát több rendelés is vonatkozhat, és minden rendelésnél a dátumot és a mennyiséget kell megadni. Az összár tulajdonság a többitől eltérően származtatott tulajdonság, hiszen a készlet és ár, azaz egységár tulajdonságokból, azok szorzataként megadható az értéke. A nyersanyag és a szállító között 1:N, míg az étel és nyersanyag között N:M típusú a kapcsolat. A 'benne' jelzésű kapcsolatnál látható, hogy tulajdonságot nemcsak egyedekhez, hanem kapcsolatokhoz is rendelhetünk az ER modell keretein belül (2.11. ábra). Ilyenkor a tulajdonság a kapcsolatpárost jellemzi.

2.3.1. Modellezés az ER modellel

Az adatbázis létrehozása során első lépésként rendszerint az ER modell segítségével hozzák létre a problémakör adatstruktúrájának szemantikai leírását. Mint minden összetett rendszert, az ER leírást is több különböző úton haladva hozhat-



2.10. ábra. Étterem ER modellje

juk létre. Itt is célszerű azonban azt a bevált módszert alkalmazni, hogy előbb a fontosabb, lényegesebb elemeket határozzuk meg, majd ezekre alapozva később finomítjuk, kibővítjük a modellt egyéb, kevésbé fontos elemekkel. Az ER modell esetén a központi szerepet az egyedek játszák, hiszen köréjük csoportosulva léteznek a tulajdonságok és a kapcsolatok is, azaz egyedek nélkül sem tulajdonság, sem kapcsolat nem létezik. Ezért a tervezés során célszerű elsőként a problématerületen megjelenő *egyedeket* számba venni, nevet és jelentést adva nekik.

Az egyedek felrajzolása után sorra vehetjük az egyedek között fennálló *kapcsolatokat*, kijelölve a kapcsolatok jellegét is. Az egyedek és a kapcsolatok együtt alkotják az ER modell gerincét, vázát. Alapvetően e váz határozza meg a későbbiekben létrehozandó DBMS adatmodell struktúráját is.

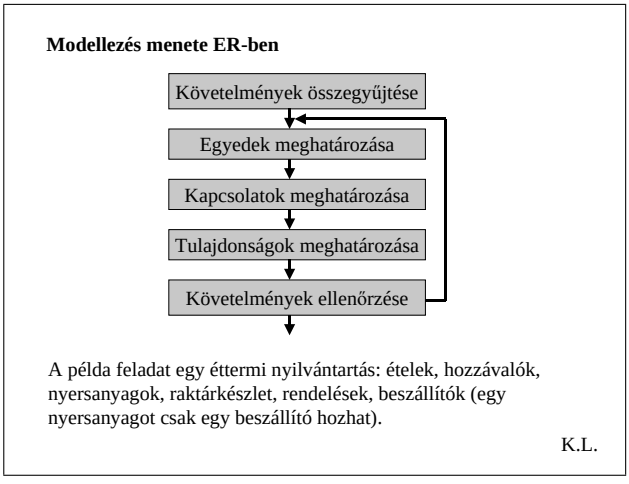
Az egyedek és kapcsolatok megadása után sorba vehetjük, hogy milyen információkra van szükség az egyes egyedekre és kapcsolatokra vonatkozóan. A *tulajdonságok* kijelölésénél ügyeljünk arra, hogy milyen értékeket vehetnek fel, megkülönböztetve a többértékű, összetett és származtatott tulajdonságokat.

A tervezés előbbiekben megadott lépései, azaz az

- egyedek meghatározása
- kapcsolatok meghatározása
- tulajdonságok meghatározása

nem szigorú szekvencia mentén mennek végbe, hiszen a bonyolultabb rendszerek esetén sokszor csak a későbbi lépésekben derülnek ki olyan módosítási igények, melyek korábbi lépések eredményeire vonatkoznak, azaz az ER modell tervezése esetén is, hasonlóan a szoftver termékek általános tervezési metodikájához, ciklikus, ismétlődő tevékenységek mentén haladunk előre.

A példánál maradva az elkészült grafikonon (2.11. ábra) három egyed szerepel,



2.11. ábra. Modellezés az ER modellben

melyek között egy egy-több és egy több-több kapcsolat él. A tulajdonságok közül a rendelés többértékű tulajdonságként szerepel és egyben összetett is, mivel a dátum és mennyiség együttesét kell hogy tartalmazza többszörösen is.

2.3.2. Az ER modellezés specifikumai

Az ER modellezés egyik sajátossága mutatkozik meg abban a tapasztalatban, hogy amikor egy hallgatói csoportnak maximálisnak vélt részletezettséggel kijelölünk egy adott modellezendő problémakört, sohasem lesznek az önállóan elkészült modellek egyformák. A jó, elfogadott modellek sem lesznek egyformák, vagyis egy problémakörre több, egymástól valamelyest eltérő megoldás is létezik. Nézzük meg, miből is fakadhatnak ezek a különbségek, a Codd által is kifogásolt hiánya a precizitásnak.

Az ER modell egyik jellemzője, hogy emberközeli fogalmakkal dolgozik, azaz egy tulajdonság megadása egyetlen emberi fogalommal történik, mint például név, cím. Az emberi fogalmakhoz viszont, mint közismert, igen gyakran bizonytalan, pontatlan értelmezések és jelentések társulnak. Vegyük például a cím fogalmát. A cím önmagában nagyon sok mindent jelenthet, például egy azonosító nevet vagy egy lakcímet. Ezenkívül a lakcím értelmezésére is több különböző megoldás kínálkozhat, hiszen ha akarjuk belevesszük az országkódot, az irányítószámot, ha akarjuk kihagyhatjuk őket. Hasonló problémával találjuk magunkat szemben a név mező esetén is. Talán még felsorolni és szép munka lenne, hogy hányféleképpen lehet megadni egy nevet; és az is igen érdekes kérdés, például hogy hol helyezkedjenek el a címek és rangok a néven belül. Egy másik gyakori félreértés forrásra mutat rá a videókölcsönző példánál előforduló eset.

Az elkészült modellben mindenki szerepelteti a kölcsönzés fogalmat. De a kölcsönzés fogalom mást-mást jelent az egyes modellekben, ugyanis valaki a kölcsönzés alatt az éppen élő kölcsönzéseket veszi, míg mások a már valaha megtörtént kölcsönzéseket értik a megjelölés alatt. E kéféle megközelítés pedig egészen másféle kezelési módot kíván a tervezés későbbi fázisaiban. Az ER modell egyes elemei tehát *többértelműek* lehetnek.

A kialakult modell szempontjából a bizonytalanság másik forrása az, hogy a tulajdonságok szerkezetét mennyire részletesen vesszük. A lakcím példájánál maradva a kérdés az, hogy egy egyszerű lakcím mezőt, vagy egy összetett lakcím mezőt szerepeltessünk, amely az irányítószám, város, ország, utca, házszám mezőkre bomlik fel. A tulajdonságok *rugalmas szerkezetűek*. Mi adhat itt útmutatást? Tanácsként azt javasolhatjuk, hogy abban az esetben, ha a feldolgozás során szükség van a részadatokra, mint önálló adatokra, akkor célszerű felbontani a tulajdonságot elemeire. Így például, ha szükség van az emberek város szerinti statisztikájára, akkor célszerű kiemelni a város tagot a lakcím tulajdonságból. Ha viszont semmi szükség sincs a részletek önálló elérésére, akkor feleslegesnek látszik a felbontás.

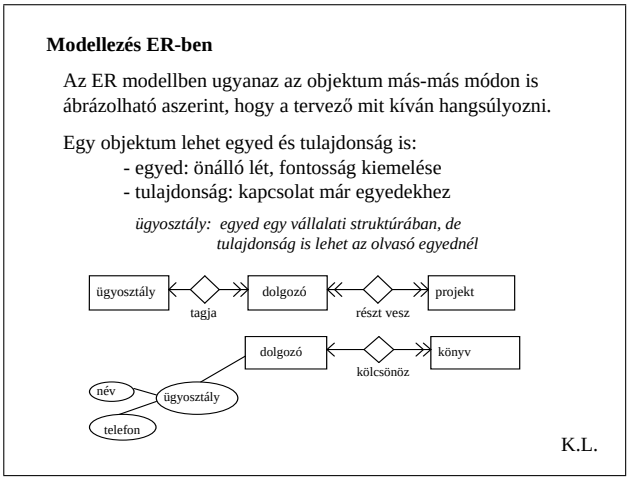
A tervezőnek a modellezés során egy fogalom ábrázolására több formai lehetősége is van. Számos esetben természetesen az egyes változatok nem teljesen egyenértékűek, de mindegyiknek lehet létjogosultsága.

Elsőként nézzük azt az esetet, amikor egy fogalom *egyed és tulajdonság is* lehet.

Ehhez vegyük az ügyosztály fogalmát példaként. Egy vállalati struktúra modellezésénél szerepelni kell a dolgozók, projektek mellett az egyes ügyosztályoknak is, tehát az ügyosztálynak, mint egyednek kell megjelennie, mint azt a 2.12. ábrán található modellrészlet is mutatja. Ha pedig a vállalat önállóan működő könyvtári rendszerét modellezzük, melyben szintén nyilvántartjuk a dolgozókat az adataikkal együtt, akkor a dolgozó munkahelye, az ügyosztály tulajdonságként is szerepelhet. Az ügyosztály tulajdonságot akár összetett tulajdonságként is szerepeltethetjük.

Mindkét modell elfogadható. Miért volt tehát ez egyik esetben egyed, a másikban tulajdonság ugyanaz a fogalom? Az első esetben az ügyosztály úgy szerepel, mint önálló egysége a problémakörnek, amely önállóan, a többi egyedtől függetlenül létezik. A másik esetben az ügyosztály nem létezik önállóan, csak a dolgozókhoz kötődve, a dolgozóhoz kapcsolódva, annak egyik jellemzőjeként él. Ezért ebben az esetben nem egyed, csak tulajdonság. A kétféle megközelítés tehát a fontosság, az önállóság tekintetében tér el egymástól.

A fogalmak ábrázolási lehetőségeit vizsgálva olyan esetek fordulhatnak elő, amikor a fogalmat *egyedként és kapcsolatként is* ábrázolhatjuk. Vegyük példaként a házasság fogalmát. Egy polgármesteri anyakönyvi hivatalban a nyilvántartás kiterjed a házasságkötésekre is. Ekkor a modellben a házasság, mint önálló léttel bíró fogalom, mint egyed szerepel, melyen önálló kérdéseket, műveleteket végezhetünk el. Például lekérdezhetjük, hogy mennyi házasságot kötöttek az elmúlt negyedévben. Ha pedig egy adónyilvántartási adatbázist dolgozunk ki, akkor nem a házasság, mint önálló egyed a lényeges, hanem csak az általa megvalósuló kapcsolat, hogy kik tartoznak egy családba. Ezért ebben az esetben a házasság, mint kapcsolat jelenik meg az adófizető polgárok között. Itt a házasság fogalomból azért lett kapcsolat, mert nem a fogalom közvetlen előfordulásai, hanem az általa kijelölt kapcsolatok játszanak szerepet a modellben.



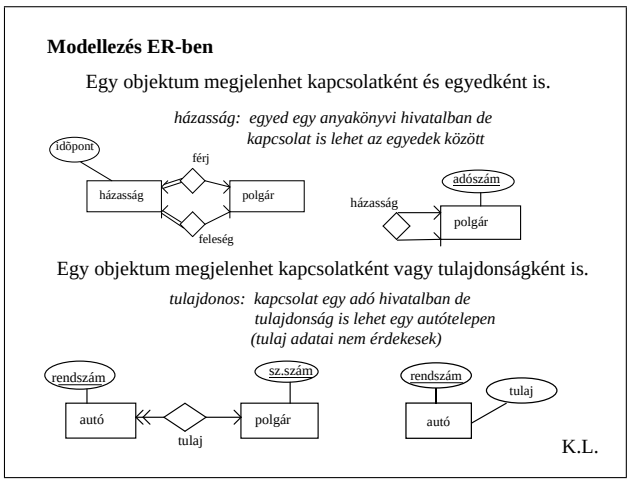
2.12. ábra. Az ER modellezés sajátosságai I.

A harmadik döntési helyzet, amikor egy fogalomról el kell dönteni azt, hogy *tulajdonság vagy kapcsolat* legyen. Vegyük példaként az autó-ember tulajdonosi viszonyt. Ekkor lehet úgy gondolkodni, hogy a tulajdonosi viszonyt úgy tartjuk nyilván, hogy az autóhoz (például annak forgalmi igazolásába) bejegyezzük a tulajdonos azonosítóját. Ekkor a tulajdonosi viszonyt egy tulajdonságon keresztül tartjuk nyilván.

A másik megoldás, hogy a tulajdonlást egy kapcsolaton keresztül ábrázoljuk. Ez a megoldás azt emeli ki, hogy a két egyed kapcsolatban áll egymással. Az irodalom ezt a megoldást javasolja az előző változattal szemben, hiszen ez szemantikailag helyesebb és többet mond. Alapszabályként elfogadhatjuk, hogy csak azon fogalmakat vesszük tulajdonságként, melyek csak az egyedhez kapcsolódnak, az egyed elválaszthatatlan részét képezik, melyek akkor is megvannak, ha az egyed önmagában létezik. Az első megoldás ennek az elvnek nem tesz eleget, hiszen a tulaj mező csak akkor kell az autóhoz, ha vannak tulajdonosok. Ha az autó csak önmagában létezne, akkor nem lenne szükség tulajdonos jellemzőre.

Nem követendő az a furcsa eset sem, amellyel szintén lehet találkozni a megoldások között, és amely ötvözi az első kettő változatot, azaz mind a kapcsolat, mind a tulajdonság megadás szerepel. Ezzel nemcsak felesleges tulajdonságot vittünk fel, hanem redundáns adatokat is megadtunk a modellben, ami zavarólag hat a tervezés későbbi fázisaiban, és a DBMS modell kialakításánál is.

Az ER modell bizonytalansága az előbbiekben vázolt problémák mellett még abban is megmutatkozik, hogy egyes, a valóságban megjelenő szituációkra nem tud megfelelő modellelemet biztosítani. Vegyük például azt az esetet, amikor egy autókat leíró információs rendszerben az autóhoz kapcsolódóan tudni szeretnénk azt is, hogy ki a tulajdonosa, milyen motor van benne, és milyen járműfajtához tartozik. Ha mindezen igényeket felvesszük a modellünkbe, akkor csak több kapcsolati

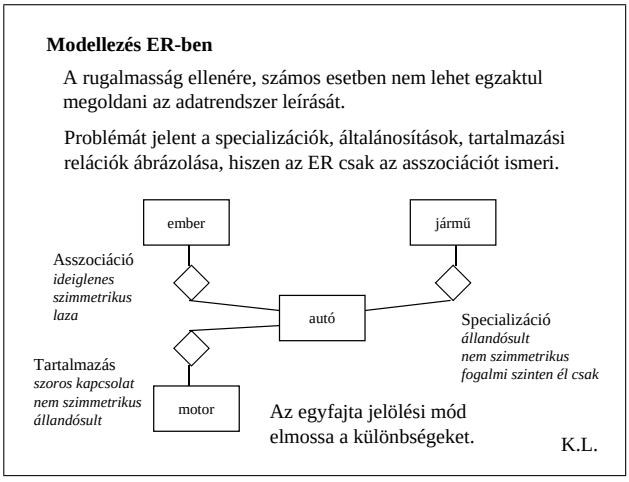


2.13. ábra. Az ER modellezés sajátosságai II.

elemmel oldhatjuk meg az igényelt információk nyilvántartását. Így összekötjük az autó egyedet az ember egyeddel, a motor egyeddel, illetve a jármű kategóriát leíró egyeddel. Mivel a kapcsolatra alapvetően egyféle szimbólum létezik az ER modellben, ezt a jelölést alkalmazhatjuk mindhárom esetben. Ez a megoldás viszont jelentős problémához vezet. Ugyanis a külső szemlélő számára az azonos modellelem miatt mindhárom kapcsolat azonos jellegűnek, vagyis asszociációnak (hozzárendelésnek) tűnik. A valóságban viszont csak az egyik jelent asszociációt, a másik kettő ettől eltérő jellegű kapcsolatot hordoz. A tartalmazás egy aszimmetrikus, állandósult kapcsolatot takar, a specializáció pedig nem az egyedelőfordulások, hanem az egyedtípusok között lép fel. Mindezen különbségeket a normál ER modell eltakarja, elfedi. A valóság pontosabb jellemzésére viszont jó lenne, ha ezek a jellegek is mind megjelennének a modellben. A fenti példákból látható, hogy az ER modell értelmezésében nem túl szigorú, bizonyos tekintetben hiányos modell, melyben számos esetben többféle leírás változat is készíthető egy megadott probléma modellezésére. Ezek a változatok egymástól bizonyos hangsúly eltolódásban különböznek, melyekre érdemes odafigyelni a tervezés során.

2.4. Az EER adatmodell

Az objektumorientált szemlélet elterjedésével egyre nőtt az igény az olyan SDM modellek iránt, melyek már tartalmaznak bizonyos eszközöket a fejlettebb modell elemek leírására is. Az ER modell ezirányú közvetlen továbbfejlesztésének eredményeként jöttek létre az EER modellek, amelyek az Extended ER, azaz *kibővített ER* modell elnevezést viselik. Az EER modelleknek több különböző változata létezik, melyek formalizmusban és elemkészletben különböznek egymástól. Mi most a Lawrence Berkley Laboratory által kidolgozott EER modellt mutatjuk be. A modell leírását egy 1994-ben megjelent cikk alapján adjuk meg. A megadott mo-

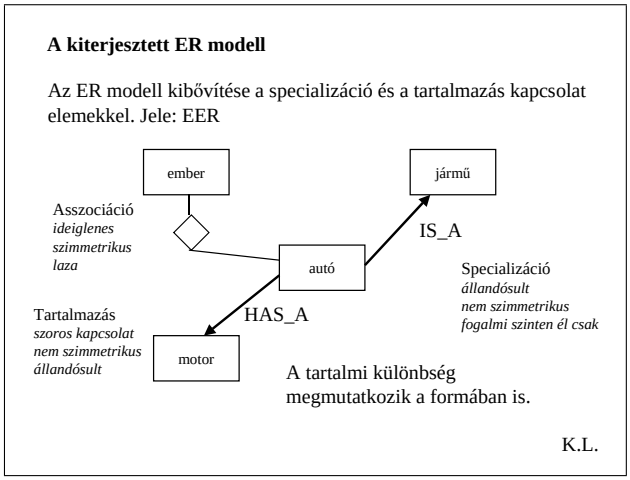


2.14. ábra. Az ER modellezés sajátosságai III.

dellhez hasonló elemeket tartalmaz a többi kiterjesztés is. A kiterjesztések alapvető eleme az osztályok és az osztályok közötti öröklési kapcsolatok figyelembe vétele. Az EER modell két új elemet tartalmaz az ER modellel összevetve, mindkettő az egyedek közötti kapcsolatokra vonatkozik. Az egyik új elem a tartalmazási kapcsolat, a másik pedig a specializációs kapcsolat.

A *tartalmazási kapcsolat* két egyedtípus között azt jelképezi, hogy az egyik egyed minden előfordulása tartalmazza a másik egyed előfordulásait. Például minden autónak van motorja, így a modellben az autó és a motor egyedek között egy tartalmazási reláció lesz. A tartalmazási reláció nem szimmetrikus, hiszen abból hogy az autó tartalmazza a motort, jön hogy a motor nem tartalmazhatja az autót. Így az EER modellben a tartalmazási relációt aszimmetrikus szimbólummal, egy nyíllal reprezentálják, amely a tartalmazó egyedből mutat a tartalmazott egyedbe. A nyíl mellé odaírják a *HAS_A* szimbólumot, mivel a nyíl másféle szerepben is előfordulhat még. A 2.15. ábrán az autó egyed tartalmazza a motor egyedet.

A másik bővítés a *specializáció*. Egy B egyed akkor specializációja az A egyednek, ha B úgy is viselkedik, mint A, azaz A minden tulajdonsága megvan B-ben is és A előfordulásai közé beletartoznak B előfordulásai is. Egy autó például specializációja a járműveknek, hiszen a járművek minden tulajdonsága (sebesség, tömeg, stb.) megvan az autónak is, és amikor a jármű előfordulásokat kell felsorolni, belevesszük az autó előfordulásokat is. A specializáció is aszimmetrikus kapcsolat, ezért ezt is egy nyíllal jelöli az EE/R modell, de most a nyíl mellé *IS_A* azonosító feliratot tesz. A 2.15. ábrán az autó egyed a jármű egyed specializációjaként szerepel. Mivel a specializáció megadásával azt is megadjuk, hogy A minden tulajdonsága egyben B-nek is tulajdonsága, ezért B-nél már nem tüntetjük fel az A-tól örökölt tulajdonságokat.



2.15. ábra. A kiterjesztett ER modell

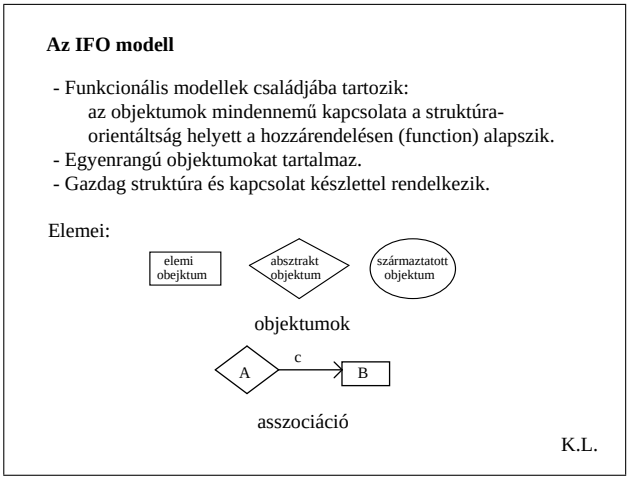
2.5. Az IFO szemantikai adatmodell

Az említett ER alapú modellek mellett számos egyéb megközelítésű modellek is kialakultak. Ezek közül a legismertebbek a *funkcionális adatmodellek*. A funkcionális adatmodell megjelölés arra utal, hogy a modellben a kapcsolatok függvényszerű formalizmussal adhatók meg. E modellek jellemzői, hogy a fogalmakat, objektumokat nem bontják szét egyedekre és tulajdonságokra. Mindkettő objektumként viselkedik. A kapcsolatok pedig ezen objektumok közötti leképzéseknek tekinthetők. Az autó példát véve, e megközelítés szerint, mind az autó, mind a rendszám egy-egy objektum. Azt a tényt, hogy minden autónak van rendszáma, a modellben egy függvénnyel adjuk meg, amely az autó objektum elemeit képezi le a rendszám objektumokra, azaz minden autó objektumhoz hozzárendel egy rendszám értéket. A többértékű tulajdonságok leképzésére e modellekben a többértékű függvényeket alkalmazzák.

Az 1970-es évek közepén kialakult funkcionális modellek módosított, kibővített változatának tekinthető az 1987-ben megjelentetett *IFO adatmodell*, mely Abiteboul és Hull nevéhez fűződik. E modell kisebb jelentőséggel bír napjaink gyakorlatában, de számos olyan eleme van, amely tovább él az újabb adatmodellekben is. Mi e modellnek csak az informális, intuitív részét vesszük át, a legfontosabb modellelemek bemutatására szorítkozva.

Az IFO modell is grafikus jelölésrendszert alkalmaz. A problémakör leírására szolgáló, elkészített modellt sémának nevezik. Az IFO séma egy irányított gráffal reprezentálható, melyben az egyes csomópontok az objektumokat, míg az irányított élek a kapcsolatokat jelölik ki.

Az objektumok ábrázolásánál háromféle objektumtípust különböztet meg az IFO modell. Az első csoportba tartoznak az *elemi* (printable) *objektumok*, melyek



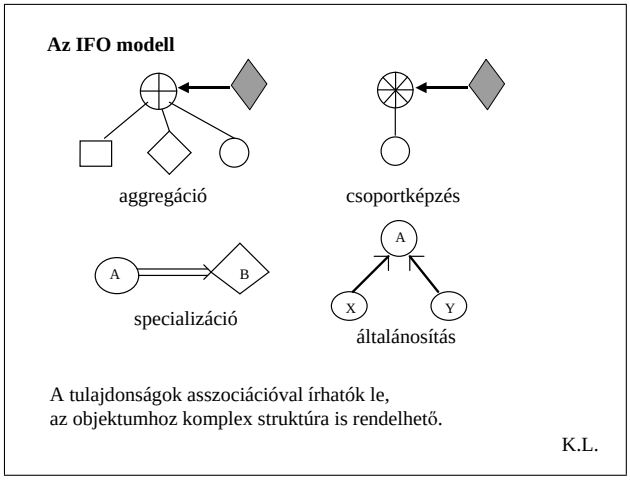
2.16. ábra. Az IFO modell I.

kiíratható, képernyőn megjeleníthető értékkel rendelkeznek. Ilyen elemi objektumnak tekinthető például a név vagy életkor, hiszen a név objektum egy sztringet, a kor objektum egy számot tartalmaz, melyek a képernyőre kiírathatók.

Az objektumok második csoportjába az *absztrakt objektumok* tartoznak. Az absztrakt objektumok olyan objektumokat jelölnek ki, melyek mögött nem egy elemi érték áll. Az ember, vagy autó egy ilyen absztrakt objektum, hiszen mindkettő nemcsak egyetlen elemi értékkel reprezentálható, hanem a hozzá kapcsolódó összetett értékek rendszerével.

A harmadik csoportba a *származtatott objektumok* tartoznak. Ezek olyan objektumok, melyek más, rendszerint absztrakt objektumokból származnak specializáció útján. Az IFO modellben a származtatott objektumok fő funkciója az, hogy ugyanazon objektumnak több különböző szerepben való megjelenését biztosítsák. Az autó-ember tulajdonosi rendszert az IFO modellben úgy ábrázolnánk, hogy az autó és ember absztrakt objektumok mellett létezik egy tulajdonos objektum is, amely az ember objektumból származtatható le. Az autó objektum e tulajdonos egyeddel állna kapcsolatban a tulajdonos viszony nyilvántartására. Az IFO formalizmusban az elemi objektumokat téglalappal, az absztrakt objektumokat rombusszal és a származtatott objektumokat ellipszissel jelölik.

A komplex értékstruktúrák ábrázolására két speciális konstruktor operátort tartalmaz az IFO modell: az aggregációt és a csoportképzést. Az *aggregáció* több különböző típusú objektum együttesét jelenti. A keletkezett struktúra egy rekordnak feleltethető meg, mely több mezőt is tartalmazhat. A lakcím például az irányítószám, város, utca, házszám objektumok együttesének, azaz aggregációjának tekinthető. Az aggregáció jelölésére egy körbe rajzolt + jel használható, melyből élek mutatnak az aggregációba bevont objektumokra.



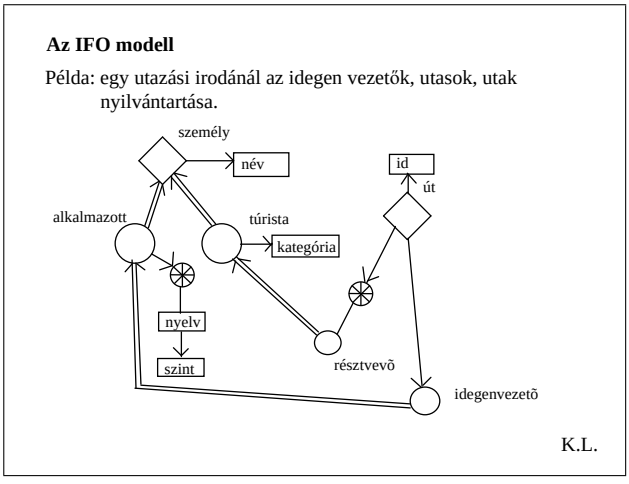
2.17. ábra. Az IFO modell II.

A *csoportképzés* ezzel szemben azt a műveletet jelenti, amikor egy másik objektum több előfordulását fogjuk össze. Ez a konstruktor a Pascal tömb fogalmához köthető. Mivel egy embernek több szakképzettsége is lehet ezért az emberhez nem egy szakképzettség előfordulás, hanem szakképzettségek csoportja köthető. A csoportképzés grafikus szimbóluma a körbe írt csillag jel, melyből él mutat a többszörözött alapobjektumra.

Az objektumok közötti *asszociációs kapcsolatok* ábrázolására szolgálnak az irányított nyilak, melyek jelentése és funkciója hasonló a funkcionális adatmodellnél említettel. A 2.16. ábrán az A objektumból indul ki nyíl a B objektum felé. Ez azt jelenti, hogy minden A-beli objektum előforduláshoz rendelhető B-beli objektum előfordulás. Az ER terminológia esetén a nyíl vagy tulajdonságként, vagy egyedek közötti kapcsolatként jelenne meg. Az IFO modellben minden asszociációs kapcsolat kap egy azonosító nevet, melyet a 2.16. ábrán a 'c' betű szimbolizál.

Egy adott objektum struktúráját és asszociációs kapcsolatrendszerét leíró gráfrészletet fregmentnek nevezik. A *fregment* elnevezés arra utal, hogy az így előálló gráfrészlet a séma egy önálló, logikailag összekapcsolódó részletét jelenti. Az irányítottságnak köszönhetően a fregmentek hierarchikus struktúrát alkotnak.

Az IFO modellből sem maradhatnak ki az objektumokhoz kapcsolódó specializációs operátorok. Az egyféle IS_A kapcsolat helyett itt kétféle módon is megadható az öröklés: specializációval és általánosítással. A *specializáció* egy létező objektumhoz különböző szerepköröket rendel. Ha az A objektum a B objektum specializációja, akkor B minden kapcsolata érvényes A-ra is, és a B-re való hivatkozás magába foglalja az A objektumokat is. A specializáció jele egy dupla vonalú nyíl, mely itt abba az objektumba mutat, amely általánosabb.



2.18. ábra. Utazási iroda IFO modellje

Az *általánosítás* azt jelenti, hogy több különböző objektumból alkotunk azokat átfogó új objektumot. Így például az autó és a vonat általánosításával létrehozható egy jármű objektum. Az általánosítás során, mely mintegy inverze a specializációnak, az új objektum felfelé örökli az alap objektumok közös tulajdonságait. Az általánosítás grafikus jele egy vastagított élű nyíl az alapobjektumokból az újonnan létrehozott objektumba.

Az IFO modell elemeinek együttes bemutatására vegyünk egy mintasémát (2.18. ábra), melyben egy utazási iroda nyilvántartási rendszerének részlete látható.

A modellben szerepel egy személy absztrakt objektum, amelyhez egy név rendelhető (asszociálható), azaz minden személynek van neve a modellben. A személynek kétféle specializációját is láthatjuk: az egyik a túrista, a másik az alkalmazott. A túrista annyiban gazdagabb a személynél, hogy hozzá már egy kategória objektum is kapcsolható tulajdonságként. Az alkalmazotthoz pedig nyelvek egy csoportja köthető, megadva hogy milyen nyelveket beszél az illető. Minden nyelvhez, mint elemi objektumhoz asszociálható egy szint, a beszédkészség szintje. Ezen hozzárendelés segítségével minden alkalmazotthoz megadható a beszélt nyelvek köre a hozzá tartozó készségszinttel együtt. Az út objektumhoz három másik objektum köthető tulajdonságként. Egyrészt az utat azonosító id száma, másrészt az idegenvezető és a résztvevők egy csoportja. Minden résztvevő egyben túrista is, ezért a résztvevő a túrista egy specializációjaként van feltüntetve. Mivel minden idegenvezető egyben alkalmazott is, ezért ő is, mint az alkalmazott specializációja van megadva. Ez utóbbit úgy is mondhatnánk, hogy az alkalmazott a modellben idegenvezető szerepben is fellép.

Az IFO modell egyik érdekessége, hogy benne minden kapcsolati elem aszimmetrikus, vagyis nem egyenrangúak a kapcsolódó elemek. Így például az utas-út kapcsolat esetében is kiemeltük az egyik elemet, az utat és ehhez rendeltük a másik objektumtípus egy előfordulási halmazát. Így a két objektumtípus közül az egyiket jelölő szerepben használjuk, míg a másik a kijelölt funkciót tölti be. Az ER modell esetében viszont az asszociációs kapcsolatokban szereplő egyedek egyenrangú szerepet töltöttek be. Az aszimmetria feloldásának egyik lehetséges módja, amikor külön kapcsoló objektumot hozunk létre, és ebből az objektumból mutatunk a kapcsolódó objektumokra.

2.6. Az UML modell osztálydiagramja

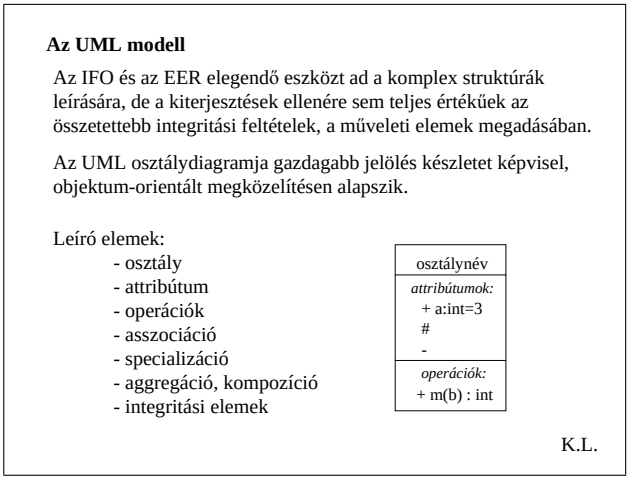
Az eddigiekben felvázolt adatmodellek egyik fő jellemzője, hogy azok az adatbázis tervezés világából nőttek ki, így csak az adatbázisok tervezése során felmerülő lehetőségekre koncentráltak. Az azonban már világos előttünk is, hogy az adatbázisok nem önmagukban létező rendszerek, melyek puszta léte a létrehozásuk célja. Az adatbázisok elsődlegesen valamely alkalmazás információs igényeinek a kielégítése céljából jöttek létre, biztosítva az információk, adatok hatékony, biztonságos és rugalmas kezelését. Ezt a körülményt is figyelembe véve, természetesnek tűnik, hogy az adatbázisok szemantikai alakjának meghatározása az információs rendszerek tervezésének szerves részét képezi. Így az általános információs rendszert tervező, modellező rendszerek is tartalmaznak komponenseket az adatmodell meg-

tervezésére.

Az általános információs-rendszer modellező rendszerek között számos olyan van, mint például az SSADM, amely átveszi a létező adatmodellező rendszerek valamelyikét, rendszerint az ER modellt. Ennek előnye, hogy az ER modell általánosan elterjedt, egyszerűen használható, jól ismert adatmodellezési formalizmus. A tervezőnek nem szükséges új, külön formalizmust megtanulnia. Ezen megoldás hátránya viszont, hogy elsődlegesen csak a relációs adatmodellhez kapcsolódó adatmodellek ismertek szélesebb körben, ezért alkalmazásukkal csak a relációs adatbázisok alkalmazása esetén lehetünk elégedettek. Egy, a relációson túlmutató adatbáziskezelő esetében viszont már gátat jelent az adatmodell funkcionális korlátozottsága. Továbbá azt sem szabad elfeledni, hogy a tiszta adatbáziskezelőorientált SDM modelleknél elsődlegesen csak a struktúra rész dominál, viszonylag szűkebb integritási, és még szűkebb viselkedési lehetőségek mellett. Ezért indokolt az a megoldás, amikor egy új adatmodellező rész kerül be az általános tervező rendszerbe. Mi most az UML rendszert vesszük példaként erre a megvalósítási alternatívára.

Az *UML* (Unified Modelling Language) egy általános célú, objektumorientált modellezési nyelv az információs rendszerek megtervezésére. Az UML több különböző komponensből áll, az információs rendszer különböző vetületeinek leírására. Az UML rendszerén belül az *osztálydiagram* áll a legközelebb a már megismert struktúra leíró modellekhez. Az ER modellel összevetve azonban itt egy sokkal gazdagabb jelölésrendszerrel találkozhatunk, melyben az OO szemlélet ötvöződik az adatkezelés statikusabb gondolatvilágával. Az UML osztálydiagramja az alábbi elemek definiálását teszi lehetővé.

– *Osztály*: az UML formalizmusában az osztályok felelnek meg az ER-nél megismert egyedtípusoknak. Itt is téglalapban adjuk meg az egyes osztá-

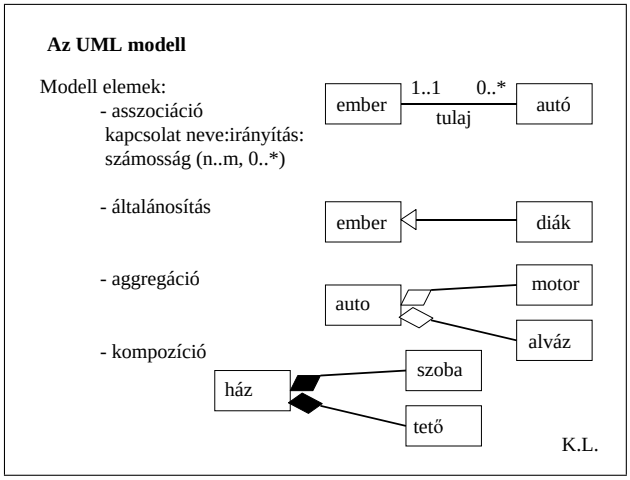


2.19. ábra. Az UML modell I.

lyok létét és elnevezését, viszont sokkal több információ kapcsolható egy osztályhoz, mint amennyit az ER modell megengedett. Az osztályt leíró téglalap magába foglalja a hozzá kapcsolódó további modellelemeket.

- *Attribútum*: az osztály előfordulásait jellemző tulajdonságok tárolására szolgál. Nem külön rajzi elemként jelenik meg, hanem az osztályleírás részeként. A tulajdonság lehet elemi és összetett is.
- *Operációk*: az OO programozás szemléletének megfelelően az osztályokhoz speciális kezelő metódusok is tartoznak. Az adatbáziskezelők fejlődésével mindinkább el fognak terjedni a metódusok, amik a kezelő függvények adatbázisban való tárolásának mechanizmusát jelentik. Így lehetővé válik a testre, azaz osztályra szabott kezelő eljárások megvalósítása az adatbázis keretein belül is.
- *Asszociáció*: az osztályok közötti asszociatív jellegű kapcsolatok megadására szolgál. Az alkalmazott formalizmus lehetőséget ad arra, hogy a kapcsolat létezésével együtt megadjunk egy finomabb számosság szabályozást is. Ennek során egy intervallummal leírható, hogy mennyi a kapcsolattal rendelkező objektumok minimális és maximális darabszáma. Az asszociáció egy szimmetrikus kapcsolatot leíró elem.
- *Általánosítás*: az osztályok közötti öröklési kapcsolatok megadására szolgál. Jelölése egy nyíl, amely az általánosabb osztály fogalom irányába mutat.
- *Aggregáció*: az alkotó osztályok együttes meglétével kialakuló összetett osztályt lehet vele megadni. Ez a kapcsolat lazább összetartozásra utal.
- *Kompozíció*: az aggregációhoz hasonlóan itt is összetett osztályokat lehet megadni, itt azonban szorosabb, strukturális kapcsolat él.

Az UML jelölésrendszer alkalmazására is veszünk egy kis mintapéldát (2.21.

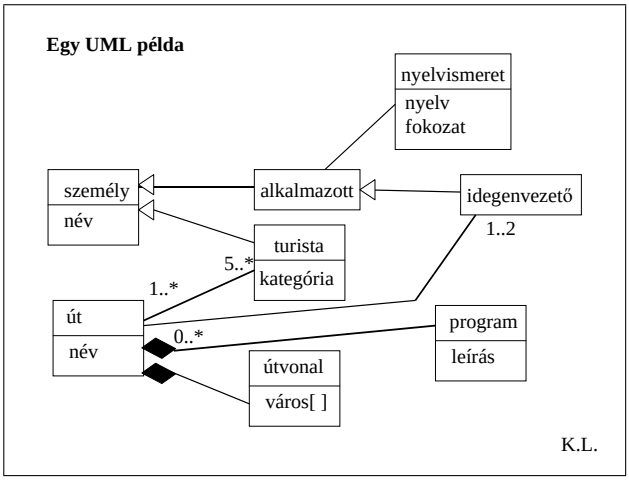


2.20. ábra. Az UML modell II.

ábra). Az IFO modellhez hasonlóan, most is egy utazási iroda információs rendszerének részletét fogjuk modellezni. A modellben első lépésként az osztályokat kell kijelölni, majd megadjuk azok belső felépítését és a köztük fennálló kapcsolódásokat.

A modellben osztályként szerepeltetjük a személyt, az alkalmazottat, az idegenvezetőt, a túristát, az utat és a nyelvismeretet. A korábbi példától eltérően itt két újabb elem is bekerült a feladatba. Az egyik az útvonal, a másik az út programja. Mindkettő majd az úthoz fog szorosan kapcsolódni. A kapcsolatok között találunk specializációs kapcsolatot, mely az osztályok között értelmezett viszonyt jelöli. Ide tartozik a személy és az alkalmazott kapcsolata, hiszen minden alkalmazott egyben személy is. A specializáció mellett találhatunk a modellben kompozíciós kapcsolatot is. A kompozíció esetében szoros strukturális kapcsolat él az elemek között. A példában az út osztályhoz két, hozzá szorosan társítható, azzal együtt létező osztály is tartozik: az útvonalat megadó és a programot leíró osztályok. A kapcsolatok harmadik csoportját az asszociációs kapcsolatok alkotják. Ennél a kapcsolatnál laza és szimmetrikus kapcsolatról van szó. Az út és a túrista, illetve az út és az idegenvezető kapcsolatát most így jelöltük a modellben, azt hangsúlyozva, hogy a túrista és az idegenvezető a konkrét úttól függetlenül is létezhetnek. Mint a modellben látható, az egyes osztályokhoz tartozó tulajdonságokat az osztály attribútumaiként adhatjuk meg. A kapcsolatok számosságát illetően itt sokkal finomabb számossági korlátot adhatunk meg, hiszen jelezhetjük a kapcsolódó egyedek számának alsó és felső korlátját is.

Látható, hogy a fejlődés során, a megvalósító rendszerek fejlődésével párhuzamosan egyre több lehetőséget, elemet vesznek be az SDM modellek elemei közé. Nem szabad azonban elfelejteni, hogy a funkciók bővítésének a megvalósítási lehetőségek szabnak korlátot, és egy modell jóságánál is érvényes az a megállapítás, hogy a tökéletes rendszer nem az, amihez már nem lehet semmit hozzátenni, hanem amiből már nem lehet semmit elvenni.



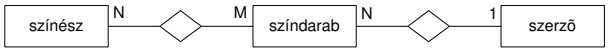
2.21. ábra. Utazási iroda UML modellje

Elméleti kérdések

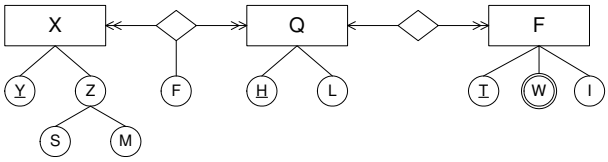
1. Mi az ER modell szerepe, milyen elemekből épül fel az ER modell?
2. Ismertesse az ER modell kapcsolat-típusait.
3. Ismertesse az ER modell tulajdonság-típusait.
4. Mutassa be az ER modell hiányosságait, kétértelműségeit.
5. Mit értünk az adatmodell fogalma alatt, és milyen komponensekből áll?
6. Ismertesse a szemantikai adatmodell fogalmát és adja meg néhány képviselőjét.
7. Adatmodellek típusai; milyen szempontok alapján lehet értékelni az adatmodelleket?
8. Mennyiben jelent problémát a specializáció és a tartalmazási kapcsolatok ábrázolása az ER modellben?
9. Ismertesse az EER modell elemeit (jelölés és jelentés).
10. Mutassa be az ER és az EER közötti konverzió lehetőségeit és az átalakítás szabályait.
11. Mutassa be az IFO és az EER közötti konverzió lehetőségeit és az átalakítás szabályait.
12. Adja meg az UML fogalmát, jellemzését és elemeit. Mely eleme használható az adatstruktúra megadására?
13. Sorolja fel az UML osztálydiagram elemeit és jelentésüket, jelölésüket.
14. Ismertesse az UML kapcsolati elemének típusait és jelentését.
15. Mutassa be az UML és az EER közötti konverzió lehetőségeit, és menetét.
16. Ismertesse az SDM rendszerek statikus elemeit.
17. Ismertesse az SDM rendszerek dinamikus elemeit.
18. Mi az SDM modellek szerepe?
19. Hogy viszonyul egymáshoz az ER és az EER modell?
20. Mutassa be az EE/R modell konverzióját E/R modellre.
21. Ismertesse az IFO modell egyed, struktúra elemeit.
22. Hasonlítsa össze az ER és az IFO kapcsolati elemének jelentését és jelölését.
23. Hogyan oldaná meg a specializáció ábrázolását az ER modellben? Milyen problémákat okozna a javasolt megoldás?
24. Mit jelent a gyenge egyed, és adjon példát előfordulására az ER modellben.
25. Milyen kapcsolat típusok vannak az UML modellben, és adjon példát az előfordulásukra.
26. Milyen struktúra típusok vannak az UML modellben, és adjon példát az előfordulásukra.
27. Milyen struktúra típusok vannak az IFO modellben, és adjon példát az előfordulásukra.
28. Milyen kapcsolat típusok vannak az IFO modellben, és adjon példát az előfordulásukra.
29. Hasonlítsa össze a tartalmazási kapcsolat megvalósítását a tanult szemantikai modellekben.

Feladatok

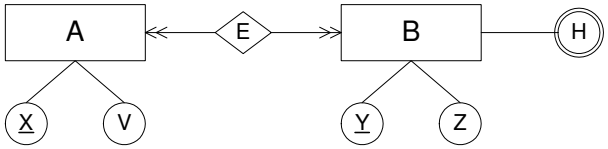
1. Egészítse ki tulajdonságokkal az alábbi ER modellt, és konvertálja át IFO modellre.



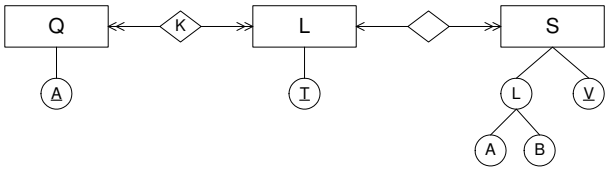
*2. Konvertálja az alábbi ER modellt IFO modellre:



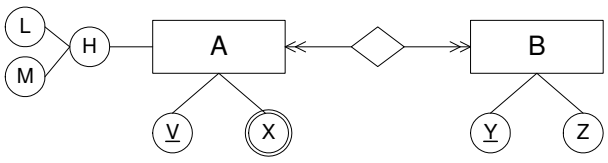
3. Konvertálja az alábbi ER sémát IFO modellre:



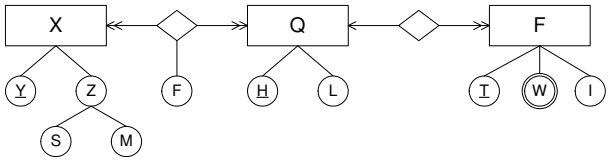
4. Konvertálja az alábbi ER sémát IFO modellre:



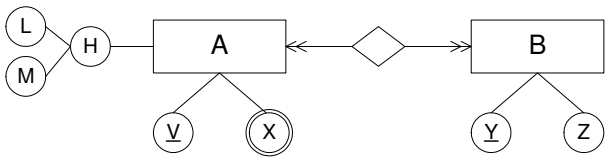
5. Konvertálja át a következő ER sémát IFO sémára:



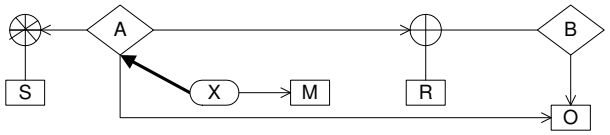
*6. Konvertálja az alábbi ER modellt UML modellre:



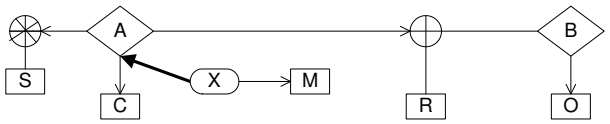
7. Konvertálja át a következő ER sémát UML sémára:



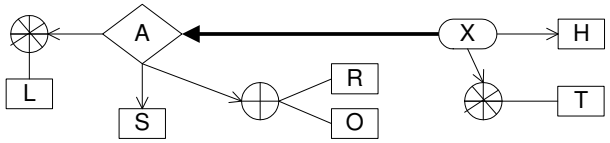
*8. Konvertálja át a következő IFO sémát EER sémára:



9. Konvertálja át a következő IFO sémát UML sémára:

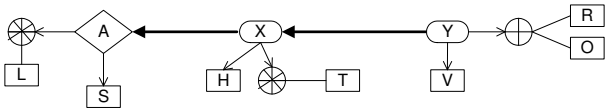


10. Konvertálja át a következő IFO sémát EER sémára:

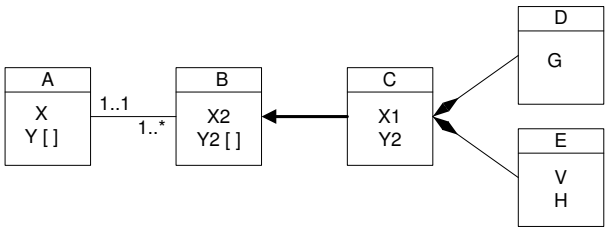


11. Készítsen ER modellt egy hallgatói index leírására.
12. Készítsen ER modellt egy könyvtár kölcsönzési rendszeréhez, melyben könyveket kölcsönöznek csak, és minden olvasónak be kell iratkoznia.

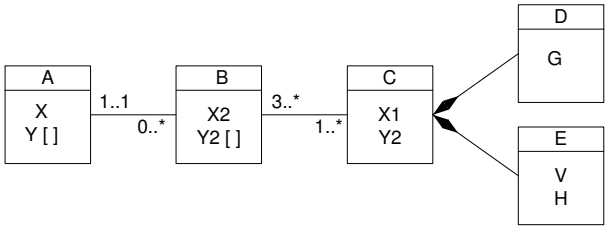
13. Konvertálja át a következő IFO sémát EER sémára:



*14. Konvertálja át a következő UML sémát EER sémára:



15. Konvertálja át a következő UML sémát EER sémára:



16. Készítsen ER modellt, mely egy csomagküldő szolgálat adatrendszerét modellezi. A megrendelt áru lehet könyv és CD, kazetta is. Nyilván kell tartani a rendeléseket, a szállításokat, a készletet és a befizetéseket is.
17. Készítsen EER modellt, mely egy csomagküldő szolgálat adatrendszerét modellezi. A megrendelt áru lehet könyv és CD, kazetta is. Nyilván kell tartani a rendeléseket, a szállításokat, a készletet és a befizetéseket is.
18. Készítsen IFO modellt, melyben egy tanfolyam szervező iroda adatrendszerét írja le.
19. Készítsen IFO modellt, melyben egy utazási iroda adatrendszerét írja le.
20. Készítsen UML modellt, mely egy vasúti helyjegyfoglalás adatrendszerét adja meg.
21. Készítsen UML modellt, mely egy verseny adatrendszerét adja meg, melyben a versenyszámok, versenyzők, eredmények kerülnek letárolásra.

3. fejezet

Pre-relációs adatbázis adatmodellek

Az előzőekben ismertetett SDM modellek után áttérünk a *DBMS adatmodellek* tárgyalására. Mint már említettük e modellek lényeges jellemzője, hogy létező DBMS rendszerek futnak mögöttük, ebből következően e modellek figyelembe veszik a megvalósítás lehetőségeit, hatékonysági problémáit. A várakozásnak megfelelően e modellek sokkal alacsonyabb absztrakciós szinten állnak, sokkal közelebb vannak a fizikai megvalósításhoz.

Mivel ezen adatmodellek leírásait ténylegesen meg kell valósítani az adatok szintjén, a rendszereknek már valós adatokkal kell dolgozniuk, itt már nem engedhető meg a szemantikai modelleknél jellemző nagyvonalúság, lényeg kiemelés. Itt már a részletekre is gondolni kell. Emiatt az adatbázis adatmodellek több ponton is eltérnek az SDM modellektől. A következő fejezetekben ezen adatbázis-adatmodelleket tekintjük majd át, hogy képessé váljunk a tényleges adatbáziskezelő rendszerekkel való munkára is.

Az adatbáziskezelő adatmodelleknek – a részletek pontos megadása és az implementálás követelménye miatt – számos olyan tulajdonsága van, mely megkülönbözteti őket a szemantikai adatmodellektől. E megkülönböztető tulajdonságok közé tartoznak többek között az alábbi jellemzők:

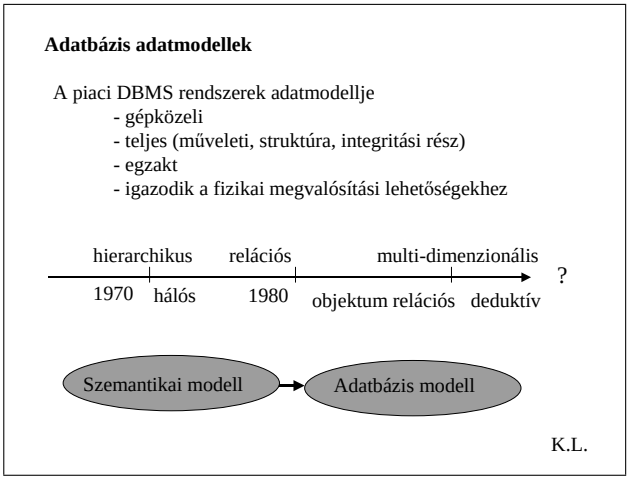
- *Gépközeliség*: Mivel a DBMS rendszerek értelmezik a megalkotott modellt, olyan formalizmusra van szükség, amely egyértelműen és hatékonyan feldolgozható, értelmezhető számítógépes programmal is. Ezért szöveges, gépközeli, egyértelmű, a formális elemekre nagy súlyt helyező modellt kell alkalmazni ezen a szinten.
- *Teljesség*: A modell legyen teljes abban az értelemben, hogy minden, a gyakorlat számára alapvetően fontos lehetőséget, funkciót tartalmazzon, hiszen csak modellen keresztül lehet majd az adatrendszert kezelni.
- *Egzaktság*: A gépközeli, formális és a részletekre is ügyelő formalizmus egyik fontos eleme az egzaktság, mely alapján minden modellmegvalósulás egyértelmű leírással rendelkezik.

– *Implementáció orientáltság*: Az elkészült adatmodell a DBMS segítségével adatstruktúrák alakjában fog létrejönni a számítógépen. A számítógépes megvalósításnál viszont van két nagyon fontos korlát, amire ügyelni kell: az időbeli hatékonyság és a tárolási hely korlátossága. Ezért csak olyan struktúrákat támogatnak az adatbázis-adatmodellek, melyek megfelelő válaszidővel feldolgozhatók és megvalósítható helyigénnyel rendelkeznek.

A technika fejlődésével természetesen egyre komplexebb rendszereket lehet hatékonyan implementálni, ezért az adatbázis modellek is állandó fejlődés részesei. Az adatbázis kezelés korai szakaszában az akkor jellemző egyszerűbb és kevésbé hatékony háttér tárolási lehetőségek miatt korlátozottabb, speciálisabb, kevésbé rugalmas adatmodellek jöttek létre, melyek erősen igazodtak az akkori adattárolási lehetőségekhez. E modellek közé tartozik a hierarchikus és a hálós adatmodell.

A fejlődés későbbi fázisaiban megjelentek a rugalmasabb és kevesebb fejlesztői munkát igénylő adatmodellek is, mint a relációs vagy az OO adatmodellek. Érdekes módon azonban nem beszélhetünk arról, hogy az új generációs DBMS rendszerek egyszerűen kisöpörnék a korábbi változatokat. A váltás kicsit komplexebb jelenség, ugyanis a régi rendszerek továbbra is megőrizték előnyüket, a hatékony végrehajtást. Emiatt több helyen továbbra is megmaradnak a régi fejlesztések, sőt az sem zárható ki, hogy egyes korábbi modellek újra elő fognak bukkanni új köntösben, hogy előnyeiket kidomborítva újra vezető szerepet töltsenek be a DBMS piacon. E jelenségnek egy kezdeti megnyilvánulása, hogy a szemi-strukturált adatforrások térhódításával a hierarchikus jellegű adatkezelési modellek ismét előtérbe kerültek.

Az egyes adatmodellek bemutatása során azonban nemcsak az adatmodell formális elemeinek az ismertetésére térünk ki, hanem áttekintjük használatának alaplépéseit is. Ennek során megnézzük, hogy a tervezés korábbi fázisaiban készített SDM modellek hogy kapcsolódnak az adatbázis adatmodellhez, mely konverziós lépéseken keresztül lehet az SDM formalizmust átalakítani adatbázis adatmodellé.



3.1. ábra. Adatbázis adatmodellek

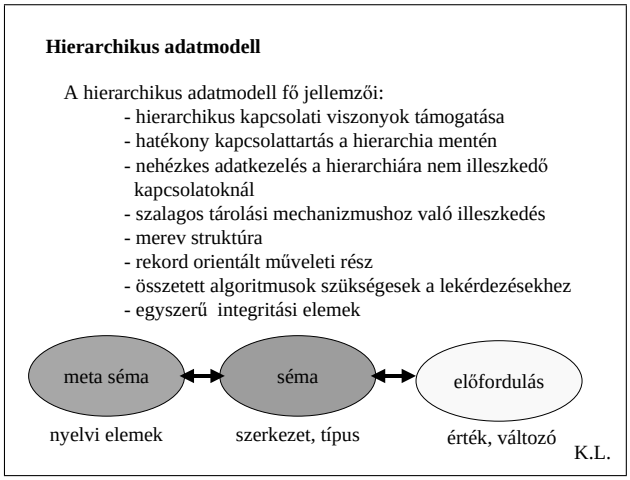
Az adatmodellek közül elsőként a hierarchikus, majd a hálós adatmodell kerül sorra.

3.1. A hierarchikus adatstruktúra

Ezt az adatmodellt az egyik legelső adatmodellként tartjuk számon, mely a hatvanas évek közepén alakult ki. Még ma is futnak elvétve hierarchikus modellen alapuló alkalmazások, de jelentőségük mára már nem domináns. Ennek megfelelően mi sem fogunk teljes részletességgel foglalkozni vele, ahogy a hálós adatmodell is, csak az átfogó ismertetés szintjén fog maradni, inkább csak bepillantást kívánunk adni pár oldal erejéig a legelső DBMS adatmodell felépítésébe és működésébe. Az igazi súlypont majd a relációs modellre fog esni, de e kis kitérő révén talán még jobban sikerül majd értékelni a relációs modell által nyújtott szolgáltatásokat, és sikerül jobban megérteni az adatbázis kezelés területén folyó fejlődést is.

A hierarchikus modell a valóságban előforduló hierarchikus szerkezetek leképzésére szolgál. A gyakorlati életben a társadalmi, vagy ipari rendszerek, de sokszor a természeti rendszerek is hierarchikus felépítést mutatnak. Egy vállalat például felbontható több üzemre, egy üzem felbontható több részlegre, és egy részleg felbontható több egységre, vagy munkacsoportra. A technológiai folyamatoknál egy gyártmány több technológiai folyamatból áll elő, és minden folyamat felbontható lépésekre.

A *hierarchia* olyan adatszerkezet, amikor az egyed előfordulások egy fastruktúrát alkotnak, és az egyed előfordulások különböző szinteken helyezkednek el. Egy egyed előforduláshoz, mint szülőhöz több gyermek előfordulás is kapcsolódhat, ahol a gyerekek a szülő alatt helyezkednek el a fában. A fa élei a szülőket



3.2. ábra. A hierarchikus adatmodell jellemzői

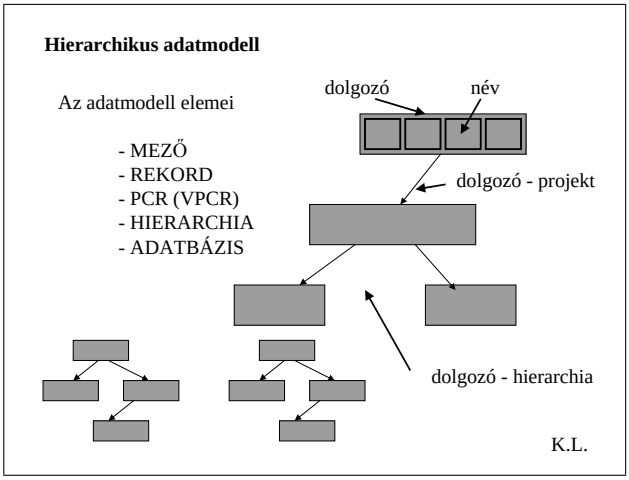
és a gyerekeket kötik össze. A fában egy szülőnek több gyereke is lehet, de egy gyereknek csak egy szülője van.

A hierarchikus modell egyik jellegzetessége, hogy nincs elméleti indíttatása. Ez a modell a különböző és egymástól viszonylag független gyakorlati alkalmazások során fejlődött ki fokozatosan. Elterjedését a sikeres és úttörő szerepű alkalmazások, mint például az IMS, segítették elő. Gyakorlatias szemléletmódja egyszerű kezelő felülettel párosul, de egyszerűsége később hátrányát növelte, ugyanis nem tudott a megnövekedett igényekhez alkalmazkodni, és így egyre jobban kiszorul a DBMS piacán folyó versenyből.

A hierarchiára épülő szerkezet egyik jelentős korlátja, hogy nemcsak az adatok tárolása, hanem az adatkezelés is alapvetően erre a sémára illeszkedik. Így a lekérdezések során is a fa élei mentén lehet csak mozogni, ami jelentősen megbonyolítja a lekérdezés összeállítását, hiszen ki kell találni, mely útvonalon lehet majd eljutni az egyik összekapcsolandó rekordból a másikhoz.

A fa struktúra elsődlegessége tehát nemcsak előnyöket ad, hanem korlátozásokat is jelent. A korlátozást jelentő hierarchia-struktúra kiválasztásának az oka, hogy e struktúra viszonylag hatékonyan megvalósítható a szalagos állománytároló eszközök mellett is. S ne feledjük, hogy ezen adatmodell létrejöttekor még a fent említett háttéreszköz dominált a piacon.

A hatékonyságot és implementálhatóságot szem előtt tartva, a lekérdezések idomulnak a meglévő rekord-menedzsment rendszer lehetőségeihez, így az adatok lekérdezése, a karbantartás is rekord egységekben történik. Ebben az értelemben az adatok elérése sok hasonlóságot mutat a hagyományos állománykezelés módjával. Ebben a modellben még viszonylag szűk az igényelhető integritási feltételek köre, így itt még elég sok tennivaló maradt meg az alkalmazó programozó számára.



3.3. ábra. A hierarchikus adatmodell elemei

A hierarchikus modellben az adatokat hierarchikus struktúrában, fákban tárolják. A modell építőkövei:

- mező,
- rekord,
- szülő-gyerek kapcsolat (PCR, Parent-Child Relationship),
- hierarchia,
- adatbázis.

A fenti struktúra elemek hierarchikus kapcsolatban állnak egymással. Az adatstruktúra legkisebb egysége a mező, amely egy elemi értéket tárol. Ez az érték lehet egy számérték, például egy ember életkora, lehet szöveges érték, mint az ember neve, de lehet dátum típusú is. Egy rekord több, összetartozó mezőt fog egységbe. A dolgozók egyes adatmezőit is egy adatrekordba hozzuk össze. Egy ilyen rekord egy komplex objektumot azonosíthat. Mivel az egyes objektumok között kapcsolat állhat fenn, mint ahogy minden autóhoz kapcsolható tulajdonos objektum, ezért e kapcsolatok megadására fog szolgálni a PCR és a belőle felépített hierarchia is. A fő különbség a két szint között az, hogy a PCR csak egy egyszintű kapcsolatot ad meg, melyben két egyedtípus vesz részt. A hierarchia ezzel szemben több egyedtípusnak a PCR elemeken keresztül felépített kapcsolatrendszerét jelenti.

A *mező*, mint elemi adattárolási egység az adatbázis legkisebb, tovább nem bontható egysége. Ezen atomiság következménye, hogy a mezőben nem tárolható olyan adat, mely összetett felépítésű lenne, és ezen struktúra felépítése a DBMS számára is ismert lenne. Ezért azt mondjuk, hogy a mezőben csak elemi értékeket lehet tárolni. Így egy mezőben foglalhat helyet a dolgozó neve, életkora, lakcíme, mivel ezek mindegyike egy-egy elemi adattal leírható. Az összetett értékeket, mint például a dolgozó összes iskolai végzettségét, több elemi mezőbe szétbontva tudjuk csak az adatbázisban elhelyezni.

Mivel egy rekordban több mező is van, melyek között lényeges különbséget kell tenni, hiszen tudnunk kell, hogy az egyes értékek mely mezőkhöz tartoznak, meg kell tudnunk fogalmazni az egyes mezőkre vonatkozó keresési feltételeket is, ezért minden mezőnek kell rendelkeznie egyedi *azonosító névvel*. A mező nevének csak a rekordon belül kell egyedinek lennie. Az elnevezés mellett a mezőben tárolt érték típusa is fontos az adatbázis megvalósításánál.

A mezők között rendszerint van egy vagy több olyan, amely kiemelt szerepet játszik a rekordon belül. Nevezetesen ezek a mezők szolgálnak a rekord azonosítására. Ugyanis a valós világban is van rendszerint legalább egy olyan tulajdonsága a modellezett objektumoknak, amely egyedi és azonosító szereppel bír, ahogy ezt az ER modellnél is láttuk. A hierarchikus modellen belül e mezőket szokás *kulcs mezőknek* nevezni.

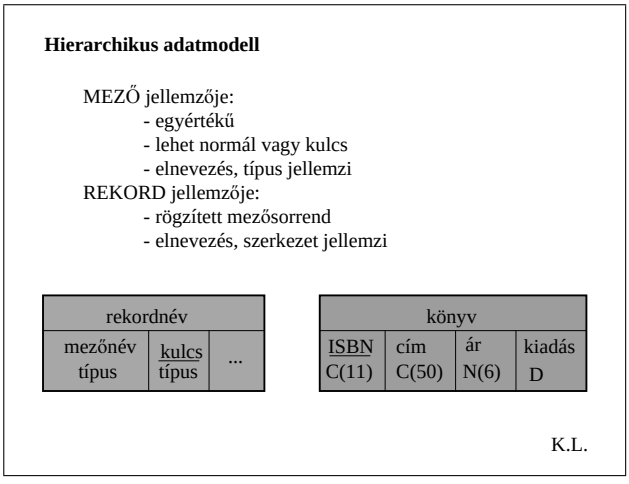
A mezőkből összeálló *rekord* megadásánál egyrészt egy azonosító nevet szerepeltetünk, másrészt megadjuk a rekord szerkezetét, *sémáját*, mely a benne tárolt mezők felsorolását jelenti. Az azonosító névre azért van szükség, hogy a későbbi kapcsolat kialakításnál, az adatkezelő műveletek során egyértelműen meg lehessen adni, hogy mely rekordból kell az adatokat kivenni. A rekordon belül a mezők

definiált sorrendje is lényeges, mert az a fizikai elhelyezésnek felel meg.

Az adatbázis adatmodell megadása a DBMS feldolgozás igényeinek megfelelően szöveges parancsmódban történik, hiszen ezt lehet a legegyszerűbben értelmezni egy számítógépes programmal. Ha viszont az elkészült szerkezetet, sémát nem a DBMS részére szeretnénk reprezentálni, hanem egy másik tervező kolléga részére, vagy az elkészült munka dokumentálása a cél, akkor a parancsok megadása helyett szokás egy tömörebb, az emberi gondolkodásmódhoz közelebb álló grafikus jelölésrendszert is használni. Így egy, az SDM rendszerekkel rokon szerepű formalizmust kapunk, amely információ tartalmában már egzakt és teljes, de formalizmusában még az absztraktabb szintet fedi le. A rekordok ábrázolása a *grafikus adatbázis séma* modell megadásánál egy téglalappal történik, amelybe beleírjuk a rekord nevét és a szerkezetét. A szerkezet megadásánál felsoroljuk a mezőket, megadva azok azonosító elnevezését és a hozzájuk tartozó adattípust.

Amikor egy rekord sémát megtervezünk, számba kell venni, milyen elemi adatokra lesz majd szükség a feldolgozás során. Ha például egy könyv rekordot veszünk, a leíró mező között szerepelhet az ISBN kódszám, a könyv címe, az ár és a kiadás dátuma. A mezők megadásánál az elnevezés mellett meg kell adni a hozzá tartozó adattípust is. Az ISBN és a cím lehet szöveges, az ár numerikus, míg a kiadás dátum típusú lesz. Kulcsnak az ISBN jelölhető ki.

A rekordok kapcsolatának leírására alkalmazható elemi egység a *PCR*, amely angolul a szülő-gyerek kapcsolatot (*Parent-Child Relation*) jelenti. Egy PCR séma egy szülő rekord típusból és egy gyerek rekord típusból áll. Mint korábban már említettük egy szülő egyed előforduláshoz több gyermek egyed előfordulás is tartozhat, például egy embernek több autója is lehet. Fordított irányban viszont az teljesül, hogy egy gyerek szerepű rekordhoz csak egy szülő szerepű rekord köt-



3.4. ábra. A HDM mező és rekord elemeinek jellemzői

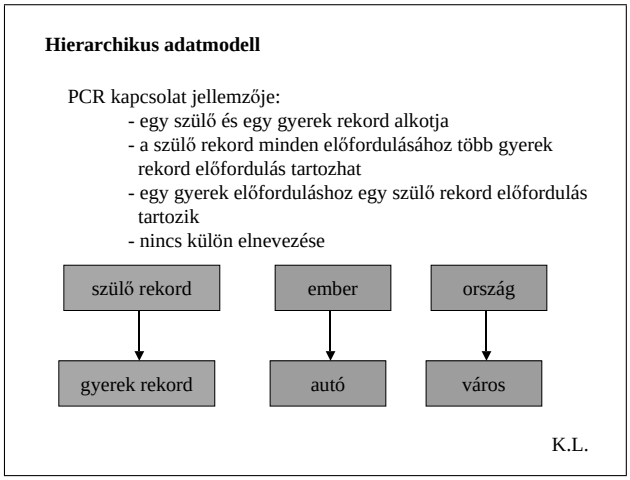
hető. Ezért a szülő és a gyermek egyedtípusok között 1:N típusú kapcsolat áll fent.

A PCR elemeknek a mező és a rekord szerkezeti elemektől eltérően nincs azonosító nevük. A névre azért nincs szükség, mivel a PCR mögött nincs közvetlen adatérték, a PCR csak a rekordok összetartozását tárolja. A PCR szerepe, hogy letárolja, mely irányban lehet továbbhaladni egy rekord feldolgozása után a hierarchián belül.

Az adatbázis séma megtervezésekor, a rekordtípusok meghatározása után fel kell tárni, hogy mely rekordtípusok között létezik közvetlen kapcsolat, melyek az 1:N jellegű kapcsolatok, és ezekben melyik lesz a szülő szerepkörű rekord. Ha például az ország és város egyedeket vesszük, akkor látható, hogy egy országhoz több város is tartozik, de egy város csak egy országhoz kapcsolódik. Ez 1:N kapcsolatot jelent, és ebben az esetben a PCR-en belül az ország lesz a szülő szerepkörű rekord, míg a városok a gyerek szerepkörűek.

Természetesen vannak olyan esetek, amikor nem lehet a kapcsolatot közvetlenül PCR-ként ábrázolni. Ha veszünk például egy rendelés nyilvántartást, akkor láthatjuk, hogy a rendelés és a termék kapcsolata nem 1:N jellegű kapcsolat. Vannak hármas, többes kapcsolatok is, amikor a kapcsolat több rekordtípus között értelmezett. A rendelésnél például a termék, vevő, gyártó hármasa alkot egy kapcsolati egységet. Ezekben az esetekben a modellünk belső struktúráját kell úgy átalakítani, hogy a meglévő kapcsolati rendszert le tudjuk írni a szokásos PCR elemek segítségével. Az ide tartozó konverziós lépéseket ezen fejezet későbbi pontjaiban fogjuk venni.

A PCR kapcsolat után a hierarchia jelenti a modell soron következő, magasabb szintű struktúra elemét. A *hierarchia struktúra* több rekordtípus PCR elemeken keresztül megvalósuló kapcsolat rendszerét tartalmazza. Az egyes egyedtípusok közötti kapcsolatok több szinten keresztül is lenyúlhatnak, tehát egy gyermekegyed



3.5. ábra. A PCR kapcsolat jellemzői

szerepelhet szülőegyedként egy harmadik egyedre vonatkozó relációban. Így több egymásba kapcsolódó PCR sémát kapunk. Egy közös gyökér elemből kiinduló PCR fát neveznek hierarchia sémának. A hierarchia séma tehát a rekordtípusok és PCR típusok együttese. A hierarchia séma szemléletesen egy fával reprezentálható.

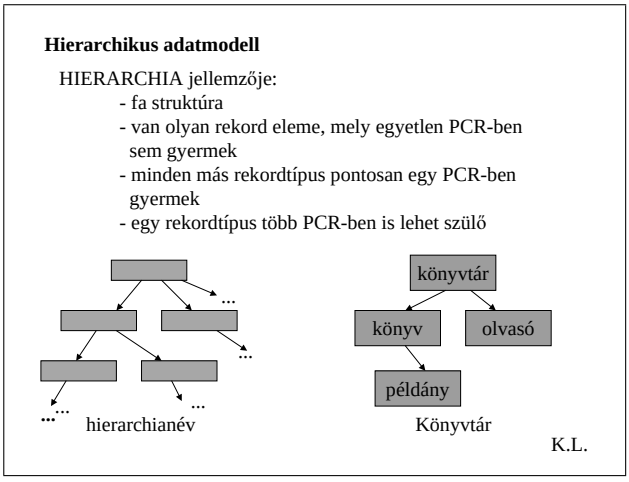
A fa struktúrából következően a sémában néhány megkötés érvényesül:

- A sémának van olyan eleme, amely egyetlen egy PCR-ben sem gyermek. A hierarchiában csak egyetlen egy ilyen elem létezik.
- Minden nem gyökér rekordtípus pontosan egy PCR-ben szerepel gyermekként, a rekordoknak nem lehet egynél több szülője.
- Egy rekordtípus több rekordban is lehet szülő.
- Levélnek nevezzük azokat a rekordtípusokat, melyek sehol sem szülők.
- Egy szülőhöz tartozó gyerekrekordok típus szerint rendezettek.
- Egy gyerek-előfordulásnak egy szülő-előfordulása van.

Az adatbázisban a hierarchia is rendelkezik azonosító névvel, hiszen ez az egység tartalmaz adatokat, melyek pozíciójának meghatározásához szükség van a befoglaló hierarchia kijelölésére is.

Mivel a modellezett világ sokszor bonyolultabb annál, hogy egyetlen hierarchikus fával le tudnánk írni, ezért megengedett, hogy több fát is használjunk. A *hierarchikus adatbázis séma* több hierarchia séma összességéből áll. A vállalat példánál maradva, külön hierarchikus sémát készíthetünk a technológiai folyamatokra, a szervezeti felépítésre, a raktározásra, a számlázásra.

A modell szemléletesebb leírására rendelkezésre áll egy grafikus jelölésrendszer, a *hierarchikus diagramm.* A diagrammal ábrázolhatjuk a fák szerkezetét, ez lesz a séma diagramm, illetve az egyes fa előfordulásokat is, amit *előfordulási diagramnak* neveznek. Egy séma diagramhoz több előfordulási diagramm is illeszthető. A könyvtár példában minden külön könyvtár előfordulás egy külön előfordulás diag-



3.6. ábra. A hierarchia jellemzői

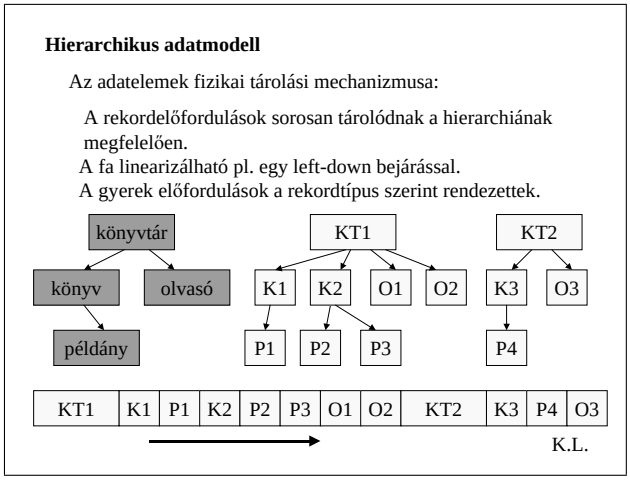
rammot határoz meg. Egy előfordulási diagrammban több gyerek, például könyv rekord előfordulás is szerepelhet.

A hierarchikus modell fő előnye az egyszerűség, ami nemcsak a modellezésben, a koncepcionális szinten jelentkezik, hanem a fizikai, belső szinten is. A hierarchikus sémához tartozó előfordulási fákat ugyanis egyszerű eszközökkel és egyszerű struktúrákban könnyen tárolhatjuk. Mindez különösen akkor volt fontos, amikor még nem álltak rendelkezésre a gyorsabb, közvetlen, direkt rekord elérést megvalósító lemeztárolók, és a szalagos adattárolással együttjáró kötöttségekhez kellett alkalmazkodni. A hierarchikus előfordulási fa előnye, hogy a szekvenciális szervezésben is hatékonyan letárolható, mivel fa struktúra, így *linearizálható*.

A fák bejárására több különböző módszer létezik, és mindegyik bejárás végülis linearizálja a fát, minden csomóponthoz egy bejárási sorszámot rendelve, mely sorszám megfeleltethető a csomópont szalagon elfoglalt pozíciójának. A fa például bejárható az 'elsőként baloldal utána jobboldal' elven, amikor egy adott részfánál előbb a gyökér, majd a gyermekek részfái következnek balról jobbra haladva. Ennél a bejárásnál egy csomópont leszármazottai mind az elem után helyezkednek el. A bejárás megvalósítható többek között az alábbi algoritmussal is:

```
bejar(cs) {
        output (cs);
        for (i=1; i<cs.gyerekszam; i++) {
            bejar(cs.gyerek[i]);
        }
}
```

A visszaolvasásnál az eredeti előfordulási fa egyértelmű helyreállítását úgy érhetjük el, hogy minden rekordtípushoz rendelünk egy típusjelzést és ezt is letároljuk az állományban. Ezáltal követni tudjuk, hogy a hierarchia mely szintjén



3.7. ábra. A HDM elemek fizikai tárolási mechanizmusa

járunk, és meg tudjuk határozni a testvéreket is. A típusjelzésnek azonban elegendő csak a fizikai nézetben szerepelnie, nem kell külön típusjelző tulajdonságot hozzáfűzni az egyes egyedtípusokhoz a koncepcionális modellen belül. Ez egyben jó példának tekinthető a fogalmi és fizikai nézetek tartalmi különbségének a szemléltetésére is.

A hierarchikus előfordulási fa tárolásakor a fa gyökér eleme kerül elsőként elhelyezésre. Ezután következnek a gyökér elem baloldali gyerekei, majd a jobboldali gyerekek kerülnek sorra. Mielőtt azonban a testvér rekord letárolásra kerülne, előbb a rekord alatt elhelyezkedő összes rekord letárolódik ugyanezen bejárási sorrendet követve. A rekordoknál az értékek mellett a rekordok típusa is letárolásra kerül, jelezve a hierarchiában történt szintváltásokat.

Nagyon lényeges, hogy lássuk, a leírt megvalósításban a rekordok közötti kapcsolatok a rekordok *fizikai pozícióján keresztül* valósulnak meg. Ez a megoldás előnyt jelent a gyerek rekordok megkeresésében, hiszen egy folyamatos szalagolvasással rögtön megkapjuk az összes leszármazottat. Másrészt hátrányos ez a megvalósítás több szempontból is: ezen negatív hatások közül kiemelhető például a rugalmasság kérdése. Ez alatt azt értjük, hogy milyen nagy munkát jelent a rekordok kapcsolatának átrendezése. Érzékelhető, hogy nagy végrehajtási költséggel tudnánk csak áthelyezni egy rekord előfordulást egy másik szülő alá, mivel a szalag tartalmának jelentős módosítása szükséges az új tárolási pozíció érvényesítéséhez.

A szekvenciális tárolási mód gyengesége mutatkozik meg akkor is, amikor a lekérdezési műveleteknek csak egy részfát kell visszaadniuk, illetve ha a fa olyan nagy, hogy a teljes struktúra nem fér be a memóriába, ezért sokszor kell a fa különböző szeleteiért visszanyúlni a szalaghoz. A hierarchia másik hátránya, hogy alapvetően csak egyféle keresési irányt támogat, a fentről lefelé irányt. Így például egy adott elem őseinek, azok elérési útvonalának a meghatározásához az egész fát át kell olvasni. A visszafelé keresés megkönnyítésére bizonyos módosításokat, bővítéseket kell végrehajtani az alapmodellen. Ennek egyik változata az lehet, amikor pointereket építünk be az egyed előfordulások struktúrájába, melyek a szülő felé mutatnak. Mindezek azt mutatják, hogy a szekvenciális elérési mód bizony igen kedvezőtlen lehet a rugalmas adatkezelés megvalósításánál.

3.1.1. ER modell konverziója hierarchikus adatmodellre

Mint korábban említettük, az adatbázisok tervezésének általános menete alapján az adatmodellt előbb SDM, majd DBMS adatmodellben fogalmazzuk meg, és az adatbázis fizikai létrehozásához ez utóbbi alapján létrehozzuk a megfelelő DDL utasításokat, melyeket a DBMS már megért és végre is hajt. A ma használatos SDM modellek között az ER modellnek van a legnagyobb jelentősége, ezért erre vonatkozólag követjük nyomon az átalakítás lépéseit, feltételezve egy ER SDM modellt és a hierarchikus DBMS modellt.

A két modell strukturális elemeit összevetve bizonyos párhuzamot lehet vonni a két modell között, mely alapján felállítható egy *általános konverziós szabály.*

Mindkét modellben fellelhető egy olyan struktúra elem, amely az elemi értékek együttesét jelenti. Az ER modellben az egyed, a hierarchikus modellben a rekord jelenti ezt a szerkezeti elemet. Ez alapján fő konverziós irányelvként e két egység egymásnak való megfeleltetését lehet megadni.

Magát az átalakítást nehezíti, hogy a tulajdonságok viszont nem minden esetben feleltethetők meg mezőnek. Ennek oka, hogy a hierarchikus modell csak az elemi értékű mezőket támogatja. Ezért az összetett vagy többértékű tulajdonságokat csak közvetve, több, egyszerű mezőkből álló struktúraként lehet tárolni. Az *összetett tulajdonság* esetén egy megoldás lehet, ha az összetett mező helyett rögtön a *komponenseit* adjuk meg mezőknek. A többértékű tulajdonság esetén pedig a megoldás az, hogy a tulajdonságokat egy új rekordként tároljuk, amely PCR révén kapcsolódik az E/R modellbeli egyednek megfelelő rekordhoz.

Nagyobb problémába ütközünk viszont akkor, ha a kapcsolatok átalakításával próbálkozunk. Az *1:N kapcsolatokat* közvetlenül ábrázolhatjuk *PCR* alkalmazásával, hiszen a PCR az 1:N kapcsolaton alapszik, de az N:M kapcsolat esetében nem alkalmazhatjuk közvetlenül a PCR mechanizmust.

Nézzük meg egy kicsit közelebbről, miért nem lehet az N:M kapcsolatot PCR-ként ábrázolni. Vegyük példaként a színészek és színdarabok kapcsolatát, ahol a kapcsolat egy évadra vonatkozóan megadja, ki mely darabban szerepel. Egy színész több színdarabban is játszhat, és egy színdarabban több színész is szerepelhet. Ha e két rekordtípus kapcsolatát PCR alakban adnánk meg, el kellene először is dönteni, melyik legyen a szülőrekord. Ha a színészt választjuk, akkor egy színdarab csak egy színészhez tartozhat, ha viszont a színdarab a szülő, akkor meg a színész köthető csak egy színdarabhoz. Egyik megoldás sem jó.

A problémára javaslatként felmerülhetne az, hogy vegyünk két PCR elemet. Az egyikben a színdarab, a másikban a színész lesz a szülőrekord, és a párja alkotja a gyerek rekordot. Ezzel a módszerrel valóban megadhatjuk az összes

Hierarchikus adatmodell

ER modell konverziója hierarchikus modellre

Elemi tulajdonság	→	Mező
Kulcs tulajdonság	→	Kulcs mező
Egyed	→	Rekord
Összetett tulajdonság	→	Több mező (a részekből)
Többértékű tulajdonság	→	?
1:1 kapcsolat	→	PCR
1:N kapcsolat	→	PCR
N:M kapcsolat	→	?

A hiányzó részek is megoldást igényelnek!

De ez csak kerülőúton lehetséges!

K.L.

3.8. ábra. ER konverziója hierarchikus modellre

kapcsolati párost, hiszen egy adott színész összes színdarabja megtalálható lesz a hozzá kapcsolódó gyerekek között, és egy színdarab színészei pedig a színdarab szülőjű PCR-ben fognak letárolásra kerülni. A probléma ezzel a megoldással ott van, hogy így minden rekord *többszörösen*, *redundánsan* fog letárolásra kerülni. Egy színészt véve alapul, a rekord letárolásra kerül egyszer mint szülő egy PCR-ben, majd többször is mindazon PCR-ekben melyekben a hozzá kapcsolódó színdarabok foglalnak helyet szülőként. Így tehát a rekordok redundánsan tárolódnak, ami többek között az alábbi hátrányokkal jár:

- felesleges helyfoglalás,
- módosítások nagyobb munkát igényelnek, és
- a módosítások során megnő az inkonzisztencia veszélye.

Ezért jobb lenne más megoldást találni a több-több jellegű kapcsolatok tárolására.

Sajnos azonban nemcsak az N:M kapcsolat az, amely problémát jelent a hierarchikus modellben. Nehézséget okoz minden olyan struktúra, mely nem illeszkedik az egyértékű mezők sémájára vagy a PCR kapcsolat típusra. Így külön meg kell vizsgálni és megoldást kell találni az alábbi problémákra:

- az egyednek többértékű tulajdonsága van;
- a modellben vannak többszörös kapcsolatok;
- egy egyed több 1:N vagy N:M kapcsolatban vesz részt a gyerek oldalon.

Az elmondottakból következik, hogy közvetlen reprezentáció ezen esetekben nem lehetséges. Mivel a nehézségek nem oldhatók meg közvetlenül a hierarchikus modell keretein belül, ezért ki kellett bővíteni a hierarchikus modellt egy új elemmel, mellyel elkerülhetők a felesleges redundanciák, és az összetettebb kapcsolatok is ábrázolhatóvá válnak.

A kapcsolatok tárolására a közvetlen fizikai elhelyezkedés helyett bevezetjük a *pointer* használatát, és a pointer segítségével kötjük össze a gyereket a szülővel. A pointert, mint egy megadott elemet kijelölő mutatót használjuk. A pointer tehát a hivatkozott adatelem fizikai címét tartalmazza. A pointeres kapcsolat ábrázolás esetén a hivatkozó és a hivatkozott rekord tárolási helye teljesen független lehet egymástól.

Tehát nem kell a rekordot még egyszer elhelyezni a szülő mögé, hogy jelezzük a kapcsolatot, azaz nem fogjuk fizikailag megsokszorozni az egyed-előfordulást. A kapcsolatot a pointer jelzi, melynek technikai okok miatt a gyerekből kell kiindulni a szülő felé. A szülőből ugyanis meghatározatlan számú pointer indulhatna ki a gyerekek felé, míg a gyerekből csak egy pointer indul ki és egy rögzített méretű struktúrát egyszerűbb implementálni, mint egy dinamikus struktúrát, főleg a régi, hagyományos programozási környezetben.

A pointerek átmutathatnak más hierarchia előfordulási fában elhelyezkedő rekordokra is. A pointerek tehát összeköthetik az egyes előfordulási fákat. A mutatók révén megvalósított PCR-t nevezik *virtuális szülő-gyerek kapcsolatnak* (*VPCR*). A VPCR is 1:N kapcsolatot valósít meg, de nem fizikai pozicionálással, hanem a pointerek felhasználásával. Egy rekord szerepelhet párhuzamosan PCR és VPCR kapcsolatban is. A VPCR jele a gyerek rekordból a *virtuális szülőhöz*

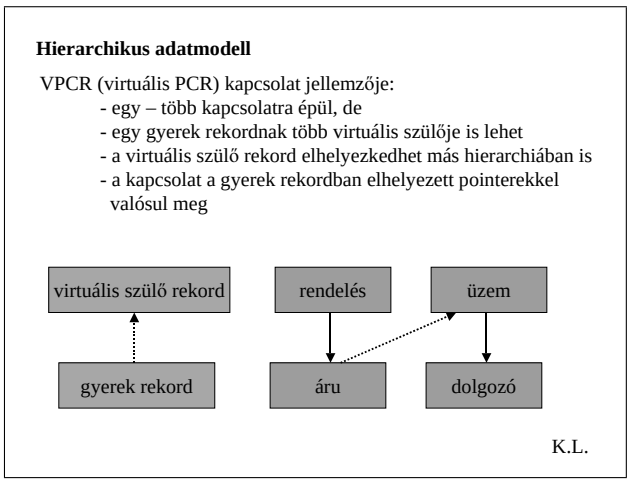
húzott szaggatott nyíl.

A 3.9. ábrán az áru rekord egyrészt a rendeléshez kapcsolódó PCR-ben foglal helyet gyerekként – jelezve, hogy egy rendeléshez több áru is tartozhat – másrészt az áru az üzem alatt is szerepel, mivel egy üzemben több árut is készítenek. Egy rekord azonban csak egy PCR-ben szerepelhet gyerekként. Most ez a rendelés rekord. Így az áru rekord fizikailag a hozzá tartozó rendelés rekord mögött fog helyet kapni. A másik irányú kapcsolathoz az áru rekord tartalmaz egy pointer mezőt, mely a hozzá tartozó üzem rekord előfordulási címét adja meg.

Nézzük, hogyan alkalmazhatók a VPCR-ek az N:M típusú kapcsolatok letárolására. A most vázolandó megoldást azért is érdemes jól megjegyezni, mert ez a fajta konverzió nemcsak a hierarchikus modellben jelenik meg, hanem majdnem mindegyik további modellben is, mivel azok is alapvetően csak a PCR jellegű kapcsolódást támogatják az adatbázis adatmodell szintjén.

Az N:M kapcsolat ábrázolásának megoldásához kanyarodjunk vissza az ER modellhez. Az N:M kapcsolat két egyedet kötött össze. Az ER jellemzése során azonban hangsúlyoztuk azt a tényt, hogy az ER modell nem egzakt, szigorú modell. A valóság egyes elemei bizonyos esetekben más és más ER elemmel is jelölhetők. Így például ugyanazt a dolgot ábrázolhatjuk kapcsolatként vagy egyedként is. Kövessük most is azt a módszert, amikor *egyedként* próbáljuk értelmezni az N:M kapcsolati elemet.

Ebben a megközelítésben a kapcsolati elem azt mutatja, hogy A és B típusú rekord előfordulások kapcsolódnak egymáshoz. Minden kapcsolatban egy A-beli és egy B-beli rekord előfordulás szerepel. Ez alapján mondhatjuk azt, hogy a kapcsolati elem az A és B rekordok *kapcsolódó párosait reprezentálja.* Így minden előfordulása egy-egy rekordpárost jelöl ki. Ha elfogadjuk, hogy az AB kapcsolati



3.9. ábra. A VPCR kapcsolat jellemzői

egyed kapcsolódó rekordpárosokat szimbolizál, akkor nézzük meg, hogyan kapcsolódik ezen új egyed a többi, már meglévő A és B egyedekhez. Mivel minden AB előfordulás mögött egy páros áll, és ezen pároshoz egyértelműen hozzátartozik egy A és egy B egyed előfordulás, ezért az AB egyed 1:N típusú kapcsolatokkal kötődik az A és a B rekordtípushoz.

Egy N:M kapcsolat egy új kapcsoló egyed bevezetésével két 1:N kapcsolatra vezethető vissza.

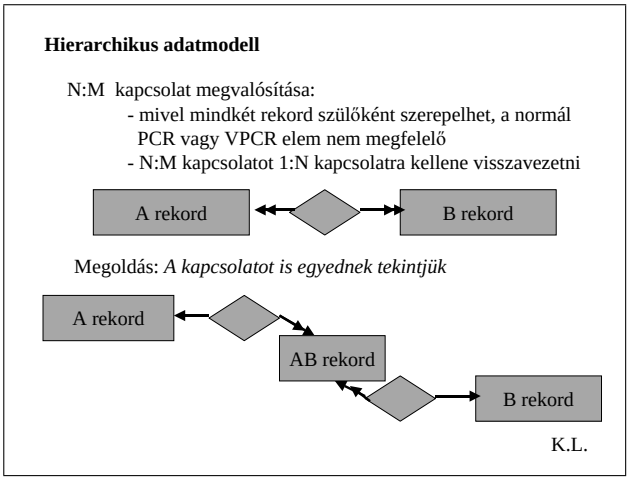
Az előzőekben felvázolt konverzió alapján most már csak olyan kapcsolatunk van, ami 1:N jellegű, tehát a PCR ábrázoláshoz illeszkedik. Az egyedüli, még megoldandó probléma az, hogy az AB rekordnak két szülője is van, mind az A mind a B rekordtípus. Szerencsére az ilyen jellegű problémát épp az előbb oldottunk meg a VPCR segítségével. Vagyis most is a VPCR lesz a megfelelő eszköz arra, hogy az AB-nak mindkét szülője ismert legyen az adatbáziskezelő számára. Az A és B rekordtípusok egyike lesz az AB normál szülője, míg a másik a virtuális szülő szerepében fog megjelenni.

A bemutatott konverziós lépéssorozat a hierarchikus modellre vonatkoztatva így foglalható össze:

Az N:M kapcsolat egy új kapcsoló egyed bevezetésével, egy PCR kapcsolattal és egy VPCR kapcsolattal reprezentálható.

Az N:M kapcsolat ábrázolásához tehát két hierarchia séma kell, melyek gyökerében a két kapcsolódó egyedtípus áll. A fenti séma megvalósulásakor egy A előforduláshoz több virtuális gyerek előfordulás is csatlakozik, melyek mindegyike egyetlen B előfordulásra mutat.

A séma alapján egy B előfordulásra több AB gyerek rekord előfordulás is mutathat. Az N:M kapcsolat tárolására tehát létre kellett hozni egy virtuális kapcsoló



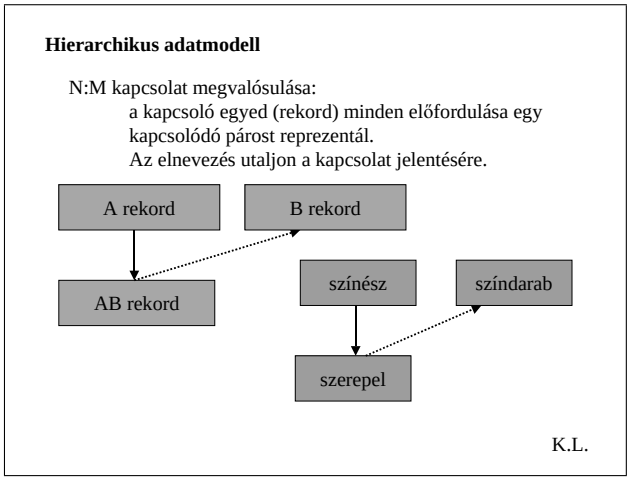
3.10. ábra. N:M kapcsolat visszavezetése 1:N kapcsolatokra

egyedtípust, mely egyrészt fizikailag kapcsolódik az egyik egyedhez, másrészt a pointeren keresztül kapcsolódik a másik egyedhez, tehát mindkettőnek a gyereke egyidejűleg. Az egyiknek fizikai, a másiknak virtuális gyereke.

Az 1:N kapcsolat megvalósításában a kapcsoló rekord segédszerepet tölt be, nem kapcsolható hozzá önálló jelentés. Az AB rekord magát a kapcsolatot reprezentálja. Ha e kapcsolatnak vannak saját tulajdonságai (mint például a szerep lehet a színész és színdarab vonatkozásában ilyen, a kapcsolatra magára jellemző tulajdonság), akkor ezen adatok a kapcsoló rekordban, a pointerekkel együtt tárolhatók. Ezen adatokat nevezik intersection, azaz metszet adatoknak.

A hierarchikus modellhez számos integritási feltétel kapcsolódik, melyek a megvalósítható hierarchia előfordulások körét korlátozzák. A hierarchikus modellek viszont nem támogatják új, egyedi integritási feltételek definiálását, tehát csak a beépített integritási feltételek kapcsolódhatnak az adatbázishoz. Ha az alkalmazásnak ezen felül még további, egyedre vagy kapcsolatokra vonatkozó integritási feltételt kellene tartalmaznia, például azt, hogy egy színész maximum 5 színdarabban játszhat, akkor minden ilyen jellegű egyedi megkötést a kezelő programban kell megvalósítani. Tehát ezek az integritási elemek nem az adatmodellben tárolódnak. Ahány alkalmazói programot érint ez a megkötés, annyi helyen kell ügyelni a betartására. A legfontosabb beépített integritási feltételek a következőkben foglalhatók össze:

- csak a gyökér rekordtípus előfordulás létezhet szülő nélkül;
- egy gyerek rekordtípus előfordulás csak a szülőkapcsolattal együtt vihető be az adatbázisba;
- a szülő rekord előfordulásának törlése a gyereket, azaz a leszármazottat is törli;
- VPCR addig nem törölhető, amíg mutat rá pointer;
- ha egy gyerek rekordtípus előfordulásnak több különböző típusú szülő-



3.11. ábra. N:M kapcsolat megvalósítása HDM-ben

rekord előfordulása van, akkor VPCR alkalmazható.

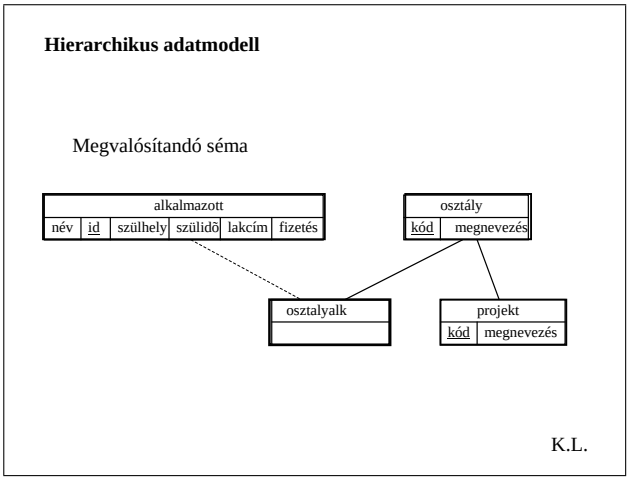
3.1.2. Hierarchikus adatdefiníciós nyelv

A következőkben az adatmodell jobb megértésére egy minta kezelő nyelvet mutatunk be, amely nem egy konkrét HDBMS nyelve, hanem egy példanyelv, mely a létező HDBMS nyelvek leglényegesebb elemeit emeli ki. A nyelv *beágyazott* használatra épül, ami annyit jelent, hogy az utasításokat nem lehet közvetlenül parancsablakban kiadni, hanem szükség van egy gazda nyelvre, melybe beágyazzuk az adatkezelő utasításokat. A gazdanyelv bármely hagyományos programozási nyelv lehet.

A HDDL a *hierarchikus adat definiáló nyelv* (Hierarchical Data Definition Language) rövidítése. A DDL nyelvek célja az adatbázisban tárolandó struktúrák leírása. A DDL nyelv segítségével lehet az adatbázisban létrehozni az adatstruktúrákat. A DDL emellett a létező struktúra elemek módosítására és megszüntetésére is tartalmaz utasításokat.

A hierarchikus modell definiálandó elemei a következők:

- séma,
- a sémát alkotó hierarchiák,
- rekord,
- a rekordot felépítő mezők,
- a rekord típusa:
  - gyökér,
  - gyerek (ekkor szülő kijelölése),
- a rekordok közötti kapcsolatok:



3.12. ábra. Vállalat hierarchikus modellje

– PCR,
– VPCR.

A HDDL nyelvet konkrét példán keresztül mutatjuk be. A példában egy grafikus séma mintához készítjük el a DDL utasítássorozatot. A mintapélda egy vállalati struktúra részletét mutatja: az adatbázisban nyilvántartjuk az alkalmazottakat, az ügyosztályokat és az ügyosztályokon futó projekteket. Mivel egy projekt csak egy osztályhoz tartozik, az osztály - projekt kapcsolat egy normál PCR szerkezettel adható meg. Ezzel szemben a példánkban egy alkalmazott több osztályhoz is kötődhet és egy osztályon több alkalmazott is dolgozhat, ezért itt N:M kapcsolat él.

Vegyük sorra a sémát leíró utasításokat. A séma megadásához előbb a keretet, az adatbázist kell létrehozni:

```
SCHEMA NAME = VALLALAT
```

Ezt követően ki kell jelölni, milyen hierarchia-fákat fog tartalmazni a séma, megadva az egyes hierarchiák azonosító nevét is:

```
HIERARCHIES = HIER1, HIER2
```

A hierarchiák kijelölése után következhet a hierarchiák részletes szerkezetének leírása. Ehhez ki kell jelölni a hierarchiában előforduló rekord típusokat, és definiálni kell az egyes rekord típusok szerkezetét is. A rekord típusok struktúrájának és az egymáshoz való kapcsolódásuknak a leírása egyazon nyelvi szerkezetben, a rekordot definiáló részben történik. A rekord definiálásakor ugyanis nemcsak a hozzá tartozó mezőket, hanem a szülő rekordtípust is meg kell adni. A hierarchia csúcsán álló rekord típusoknál a szülő helyett a hozzá tartozó hierarchiát adjuk meg. A soron következő utasításokban rendre az alkalmazott, osztály, projekt és a segédrekord szerepét betöltő *osztalyalk* rekordtípusok definícióját láthatjuk.

```
RECORD
    NAME = ALKALMAZOTT
    TYPE = ROOT OF HIER2
    DATA ITEMS =
            NEV         CHAR 15
            ID          INTEGER
            SZULHELY    CHAR 20
            SZULIDO     DATE
            LAKCIM      CHAR 30
            FIZETES     INTEGER
    KEY = ID
    ORDER BY NEV
RECORD
    NAME = OSZTALY
    TYPE = ROOT OF HIER1
    DATA ITEMS =
            MEGNEVEZ    CHAR 15
            KOD         INTEGER
```

```
        KEY = KOD
        ORDER BY MEGNEVEZ

    RECORD
        NAME = PROJEKT
        PARENT = OSZTALY
        CHILD NUMBER = 1
        DATA ITEMS =
                MEGNEVEZ  CHAR 15
                KOD       INTEGER
        KEY = KOD
        ORDER BY MEGNEVEZ
    RECORD
        NAME = OSZTALYALK
        PARENT = OSZTALY
        CHILD NUMBER = 2
        DATA ITEMS =
                KOD POINTER TO VIRTUAL PARENT ALKALMAZOTT
```

Ezzel teljes egészében ismertté vált a DBMS számára is a létrehozandó struktúra.

Összefoglalva, a HDDL nyelvben a következő kulcsszavak fordultak elő az alábbi jelentést hordozva:

- SCHEMA NAME: az adatbázis séma azonosító neve
- HIERARCHIES: az adatbázisban lévő hierarchiasémák nevei
- RECORD: egyedtípus azonosítás
- NAME: egyedtípus neve
- TYPE = ROOT OF: az egyed gyökér a hierarchiában
- DATA ITEMS: mező azonosítások
- KEY: kulcsmező
- ORDER BY: egyed előfordulások rendezési elve
- PARENT: szülő egyedtípus
- CHILD NUMBER: egyedtípus sorszáma a gyerekek között
- POINTER TO VIRTUAL PARENT: pointer a virtuális szülőre

A fenti utasításkészlet elegendő egy egyszerű hierarchikus adatbázismodell megadására. A minta is érzékelteti a modell viszonylag merev szerkezetét. Szembetűnhet például az, hogy az adatbázis megadásánál rögtön fel kell sorolnunk az oda tartozó hierarchiákat, tehát előre ismernünk kell az adatbázis szerkezetét. Egy rugalmasabb modelltől elvárnánk például, hogy széttagoltan, izoláltan lehessen definiálni a hierarchiasémákat.

3.1.3. Hierarchikus adatkezelő nyelv

Az utasítások másik nagy csoportja az adatkezelő utasítások (HDML) köre. A HDML a *hierarchikus adatkezelő nyelv* (Hierarchical Data Manipulation Language) rövidítése. A DML nyelv azon utasításokat fogja össze, melyek az adatbázisban letárolt adatok lekérdezését, módosítását, bővítését és kitörlését teszik

lehetővé. Más rendszerekben még további bontást is szokás megadni a műveletek osztályozásánál.

A hierarchikus modellek a rekordorientált műveleteket támogatják, ezért a kezelőnyelv is a rekordonkénti feldolgozásra épül. Mivel a műveletek rekord szinten hajtódnak végre, a feldolgozásnál lényeges szerepet játszik, hogy hol tart a feldolgozás, mi lesz a következő feldolgozandó rekord. Ezért a rendszer nyilvántartja az *aktuális rekord pozíciót*. Mivel a modellben a hierarchiasémák és a rekordtípusok is viszonylag önálló szerepet töltenek be, ezért a következő szinteken végzik a kurrens rekord nyilvántartását az adatbázison belül:

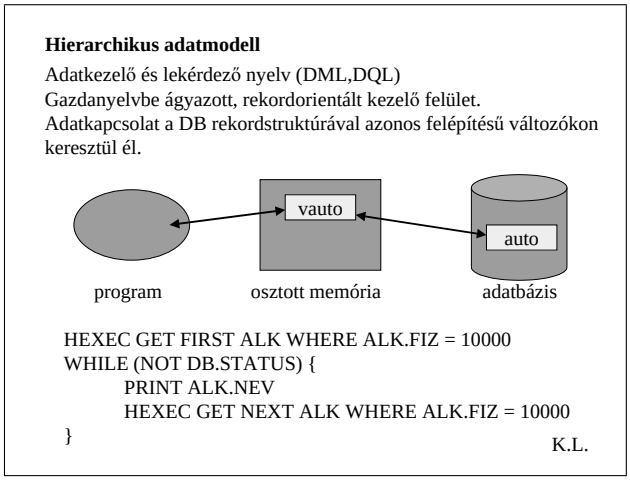
- adatbázis szintje;
- hierarchia-fa szintje;
- egyedtípus szintje.

A HDML parancsok jellegének szemléltetésére néhány parancsot emelünk ki:

- *GET*: az aktuális rekord beolvasása a kijelölt memóriaváltozókba.
- *INSERT*: új rekord beszúrása.
- *DELETE*: rekord törlése.
- *REPLACE*: rekord módosítása.
- *GET HOLD*: rekordpointer rögzítése.

Az egyes parancsok részletesebb szintaktikája megmutatja a műveletek jellegét, működési elveit is.

- GET FIRST típus WHERE feltétel: az első, feltételnek megfelelő egyedtípus előfordulás meghatározása.
- GET NEXT típus WHERE feltétel: a következő, feltételnek megfelelő egyedtípus előfordulás meghatározása.



3.13. ábra. A HDML és a HDQL nyelv

- GET HOLD FIRST | NEXT típus WHERE feltétel: egy rekord kiválasztása törléshez, vagy módosításhoz.
- DELETE típus: a kurrens előfordulás törlése.
- INSERT típus FROM változó: egyedtípus előfordulás beszúrása.
- REPLACE típus FROM változó: egyedtípus előfordulás módosítása.

A fenti műveletek értelmezéséhez figyelembe kell venni, hogy a DML és a DDL utasítások egy gazdanyelvbe beágyazva használhatók. A gazdanyelv és az adatbázis adatkapcsolata a program munkaterületén keresztül valósul meg. A programban az adatkapcsolat megvalósításához speciális rekordváltozókat, *kapcsolati változókat* kell definiálni. A változó azonosító neve megegyezik a hozzá tartozó egyedtípus nevével, és egy olyan memória területen foglal helyet, melyet a DBMS is elér. A program és a DBMS ezen változón keresztül küld át adatokat egymásnak. A kapcsolati programváltozó mindig a kurrens adatbázis rekord értékét tartalmazza minden egyedtípusnál. Így egy GET után ebben a változóban találhatjuk meg az éppen érintett adatbázisrekord értékét. Beszúrásnál és módosításnál az új értékeket egy másik, de ugyanilyen szerkezetű programrekordból veszi a rendszer. Az utasítások, mint látható, rekordorientáltak. Ennek szemléltetésére álljon itt egy példa az összes 10000 Ft-ot kereső alkalmazott egyedelőfordulás kiíratására.

```
HEXEC GET FIRST ALKALMAZOTT WHERE
        ALKALMAZOTT.FIZ = 10000;
        while(not DB_STATUS) {
            print ALKALMAZOTT.NEV;
            HEXEC GET NEXT ALKALMAZOTT WHERE
                ALKALMAZOTT.FIZ = 10000;
}
```

A példában a C nyelvet választottuk gazdanyelvként. A HEXEC kulcsszó jelzi, hogy az adott sor egy DDL vagy DML utasítást tartalmaz (ezt a megoldást nevezik *laza csatolás*nak). A DBMS és az alkalmazás közötti kapcsolat egyik lényeges, eddig nem említett elemét is megfigyelhetjük itt. Lényeges ugyanis, hogy az alkalmazás értesüljön a kiadott utasítások végrehajtásának sikerességéről, hiszen ennek függvényében adódnak a következő lépések. A *hibakezelés* egy foglalt változón, a DB_STATUS-on keresztül valósul meg; értéke jelzi, hogy sikeres volt-e a végrehajtás, vagy sem. A rekordorientáltság miatt a teljes előfordulás halmazra vonatkozó utasításokat ciklusszervezéssel oldhattuk meg.

3.2. A hálós adatstruktúra

A hierarchikus adatmodell tárgyalása során láthattuk, hogy mennyi korlátozást jelent a hierarchikus kapcsolati struktúra megkövetelése a különböző adatbázis programozási feladatokban. Egy valósághűbb adatstruktúra megvalósításához tehát olyan adatmodellre volna szükség, amely sokkal több lehetőséget ad a különböző kapcsolati formák ábrázolására. Mivel a számítógépes háttértárolási technikák fejlődése során elavulttá vált a soros tárolásra vonatkozó hatékonysági megkötés, ezért már lehet gondolkodni összetettebb, ugyanakkor rugalmasabb tárolási struktúrában is. A továbblépés következő állomása a hierarchikus struktúra

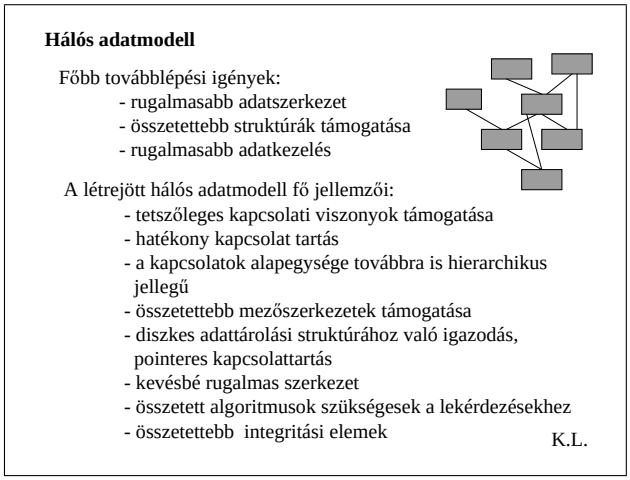
feloldására épülő hálós adatmodell megjelenése.

A *hálós adatmodell*, mint ahogy azt az elnevezése is mutatja, a háló jellegű kapcsolati struktúra megvalósításán alapszik. A háló struktúra is csomópontokból és kapcsolati elemekből, élekből áll, de itt a hierarchiától eltérően, egy elemnek több szülő szerepkörű kapcsolódó párosa is lehet. Így a rekordtípusok és előfordulásaik úgy helyezkednek el, mint egy halászháló különböző csomói, ahol az élek az összetartozást mutatják. Természetesen az adatmodellnek valamilyen kezelhető formában kell megvalósítania a hálót, ezért a hálót is lebontják elemibb egységekre, úgynevezett *set*-ekre, melyeket összekapcsolva alakul ki a teljes háló.

A rugalmasabb és összetettebb struktúra mellett a változás a műveleti részben is megnyilvánul. Továbbra is igaz az a megállapítás, hogy a rekordok összekapcsolása az élek mentén történhet, viszont itt sokkal rugalmasabban lehet mozogni, hiszen az élek is bővebb kapcsolatrendszert írhatnak le, ezért több különböző irányban is lehet navigálni a lekérdezés folyamatát. A hálós adatmodell legfontosabb jellemzői a következő pontokban foglalhatók össze:

- hálós kapcsolati struktúra,
- a háló set-ekből épül fel,
- gazdag kapcsolati viszony,
- hatékony kapcsolattartás,
- összetett mező szerkezetek támogatása,
- pointer alapú kapcsolattartás,
- a kialakult struktúra viszonylag körülményesen módosítható,
- a lekérdezés beágyazott környezetben, algoritmikus elemekkel történik,
- összetett integritási feltételek támogatása.

A hálós adatmodell valóban hatékony és robosztus adatbázis struktúrát szolgáltat, ezért nagyon sokáig ez az adatmodell volt az uralkodó az adatbázis piacon,



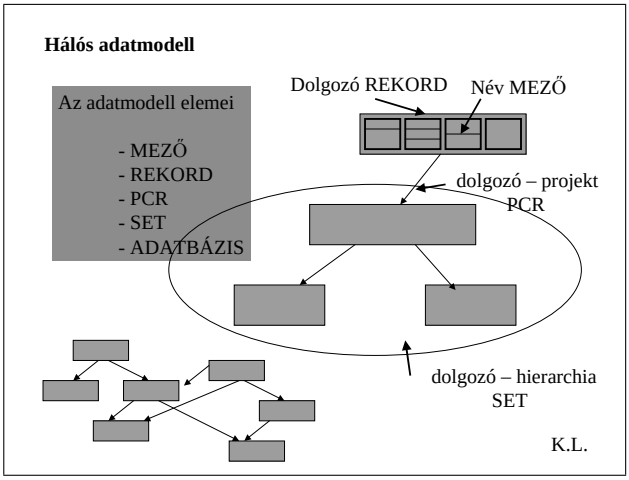
3.14. ábra. A hálós adatmodell jellemzői

és sok helyen még ma is alkalmazzák nagyobb adatbázisok esetében, hatékonyságának köszönhetően.

A hálós adatmodell is több struktúra építőelemet tartalmaz, melyet megint a legkisebb egységtől kezdve veszünk sorba. A legkisebb, névvel ellátott tárolási egység a mező. A mező megvalósításakor azonban feloldották azt a megkötést, hogy a benne tárolt érték elemi legyen. Ezzel természetesen bonyolultabb tárolási struktúrát kellett implementálni, viszont jóval egyszerűbb az adatbázis tervező és alkalmazásfejlesztő dolga, ha az adatmodellben összetett tulajdonság is előfordul.

A mezőkből itt is rekordok képezhetők, melyek az összetartozó mezők együttesét jelentik. Az adatbázisban több különböző rekordtípus jelenik meg, melyek között kapcsolatok léphetnek fel. A kapcsolatok segítségével tarthatjuk nyilván az egyes egyed előfordulások közötti asszociációt, ami igen lényeges információt hordozhat a felhasználó számára. Egy ingatlan nyilvántartási rendszerben például fontos, hogy az ingatlan és a lakosok adatai mellett a tulajdonosi viszony is ismert legyen. A kapcsolatok ábrázolása a hálós modellben is az elemi PCR kapcsolati egységen alapszik. A PCR itt is azonosító név nélküli elemi kapcsolati egység, amely 1:N kapcsolat ábrázolására alkalmas. A hierarchikus modelltől eltérően itt lehetőség van arra, hogy függetlenebbül kezeljük az egyes PCR kapcsolati elemeket. Míg a hierarchikus modellben egy fában kellett összehozni a kapcsolódó PCR egységeket, addig itt egy kisebb egység is belép a háló (adatbázis) és a PCR közé, a *set*.

A set, mint egység egy egyszintű hierarchiát jelent, hiszen a set az azonos szülőből kiinduló PCR egységeket fogja össze egy azonosító névvel ellátott struktúrába. A hálós DBMS különböző változataiban a set csak egy PCR-t tartalmazhat, de más változatokban több, összetartozó PCR alkot egy set egységet. Természetesen felmerülhet bennünk a kérdés, miért van szükség set egységekre, miért nem elegendő a PCR egységek megadása. Nos, tartalmi szempontból a PCR kapcsolatok



3.15. ábra. A hálós adatmodell elemei

ismerete elegendő a háló felépítéséhez, viszont míg a PCR egy kapcsolat leíró elem, addig a set egy struktúra egység a hálóban. A sémában a set az, aminek önálló azonosító neve van, így például a hálóban való navigációs mozgáskor a set azonosításával jelölhetjük ki a feldolgozási irányt. A gyakorlatban a set és a PCR sokszor azonos információs tartalmat hordoz, hiszen a set csak egy PCR-t tartalmaz. A felvázolt rétegződés viszont azt is megengedi, hogy hatékonysági megfontolásokból előre megszabjuk, hogy az adott típusú rekord mely más rekord típusokkal fog kapcsolatban állni a PCR viszonyon keresztül.

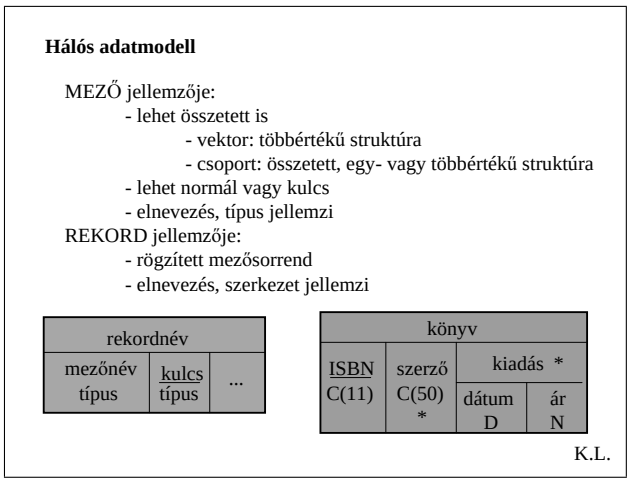
Rátérve a hálós modell elemeinek részletesebb leírására, elsőként a *mező* formátumát pontosítjuk. A sémában a mezőknek van

- neve,
- típusa, és
- integritási elemei.

A típus tekintetében lehetőség kínálkozik arra, hogy ne csak elemi szerkezetű mezőket definiáljunk, hanem összetett szerkezetűeket is. A modell az alábbi mező struktúrákat engedi meg:

- *Elemi* mező: csak egy elemi értéket tartalmaz, mint például a dolgozó neve vagy életkora.
- *Vektor* mező: többértékű mező, ilyen lehet a dolgozó esetében például a hobbi vagy a végzettség, hiszen mindkettő esetében több elemi érték is megadható egy rekordon belül.
- *Csoport* mező: több elemi vagy összetett mezőből álló struktúra. A csoport különböző szerkezetű részeket fog össze. Ilyen mező lehet például a lakcím, mert annak egy értéke több elemibb részre bontható.

A hierarchikus modellhez hasonlóan itt is fontos szerepe van a kulcs mezőnek,



3.16. ábra. Mező és rekord elemek NDM-ben

hiszen a rekord előfordulások egyediségét itt is biztosítani kell. A kulcs mezőket a sémarajzban aláhúzással emeljük ki a többi mező közül.

A logikailag összetartozó mezők alkotnak egy *rekordot*. A rekord esetében fontos a mezők sorrendje, ami rögzített sorrendiséget jelent, mivel az adatforgalom az adatbázis és az alkalmazói programok között az ugyanilyen szerkezetű segédváltozókon keresztül zajlik. A hálós adatmodell műveleti részében, mint majd látni fogjuk, a rekord jelenti az adatátvitel, a navigáció alapegységét.

Az adatbázisban a rekordok közötti kapcsolat itt is a PCR viszonyra épül, de az adatbázisban a set egység az, amely önálló névvel azonosítva foglal egységbe egy, vagy több összetartozó PCR kapcsolatot. A CODASYL bizottság, melyet az 1960-as évek végén hoztak létre az adatbázisok kezelő és sémaleíró nyelvének szabványosítására, foglalta össze elsőként egységes keretben a hálós adatmodell definícióját. A CODASYL bizottság értelmezését követve

> *a set (vagy CODASYL halmaz) nem más, mint egy névvel ellátott kétszintű fastruktúra. A set így azonos rekordtípusból kiinduló, PCR kapcsolatok névvel ellátott egységeként értelmezendő.*

A set-ben tehát több rekordtípus fordulhat elő, melyek közül van egy, amely kiemelt szerepet foglal el, ez a szülő szerepkörben lévő rekord. A többi rekord mind gyerek szerepben kell hogy szerepeljen. A szülő szerepű rekordot nevezik a set *tulajdonosának*, míg a többi rekord a set *tagrekordja* lesz.

Ha például egy vállalat információs rendszerét vesszük, abban az ügyosztályhoz több más egyed, illetve rekord típus is kapcsolódik. Az ügyosztály mind a dolgozók, mind a projektek viszonylatában 1:N kapcsolatban áll, mégpedig úgy, hogy mindkét viszonylatban az ügyosztály lesz a szülő rekordtípus. Ezért e két PCR kapcsolat összefogható egy set-be, melynek a tulajdonosa az ügyosztály rekord, tagjai pedig a dolgozó és a projekt rekordtípusok lesznek.

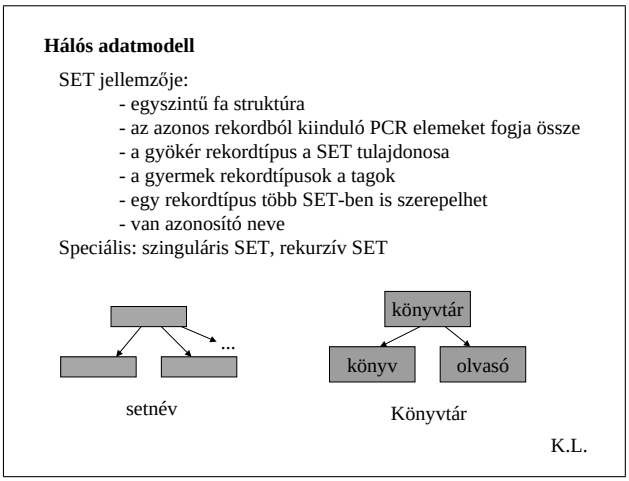
A set-ek között vannak speciális felépítésűek is, amelyek nem minden implementációban támogatottak, és külön elnevezést is kaptak.

- *Szinguláris set*: olyan set, amelyben nem létezik explicit tulajdonos, hanem a rendszert tekintik implicit tulajdonosnak, és rendszerint csak egy tagrekordja van. E set célja, hogy egy helyen érjük el a rekordtípus összes előfordulását mindennemű kapcsolatok figyelembe vétele nélkül.
- *Egytagú és többtagú setek*: olyan setek, amelyekben vagy csak egy tagrekord szerepelhet, vagy több tag is megjelenhet bennük.
- *Rekurzív set*: olyan set, amelyben a tulajdonos rekordtípus egyben tagrekord típus is. Ez a fajta set nagyon ritkán implementált a hálós DBMS rendszerekben. A rekurzív set-re például akkor lenne szükség, ha a vállalatnál a dolgozók egymás közötti hierarchiáját is tárolni kívánnánk. Ekkor ugyanis minden dolgozóhoz rendelhetünk egy főnök szerepű dolgozót, így a főnök-beosztott kapcsolat mindkét egyede a dolgozó egyed lesz. Ezáltal a PCR a dolgozó rekordból a dolgozó rekordba mutat, és a set-ben a dolgozó tulajdonos, és tag is lesz, egy rekurzív set-et alkotva.

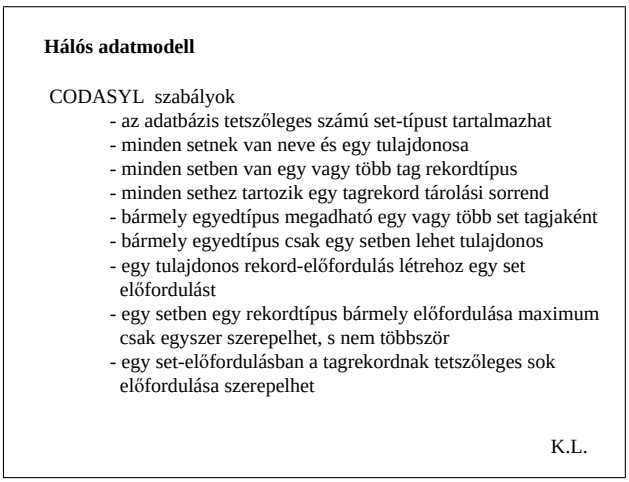
A set-ek kialakításában igen nagy a szabadságunk, mégis vannak bizonyos megkötöttségek a létrehozható set-ek körére vonatkozóan. Így például az alapul vett

CODASYL szabályok egyértelműen tiltják a rekurzív set-ek alkalmazását. A következőkben tehát a *CODASYL szabályrendszert* foglaljuk össze röviden. Ezek a szabályok nemcsak egyszerű tiltásokat jelentenek, hanem útmutatást adnak arra is, hogy hogyan értelmezzük a set-ek megvalósítását és működését. A szabályrendszer az alábbi elemekből áll:

– A set-típus rekordtípusok közötti, névvel ellátott kapcsolat.
– Egy adatbázis sémában tetszőleges számú set-típust lehet definiálni.
– Minden set-típusban léteznie kell egy tulajdonos rekordnak, kivéve a szinguláris set szerkezetet.



3.17. ábra. A set jellemzői

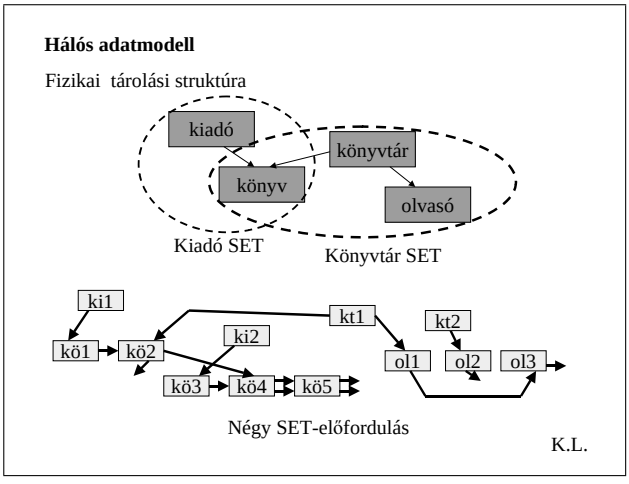


3.18. ábra. CODASYL szabályok

- Minden egyes set-ben léteznie kell egy vagy több tagrekordnak.
- Minden egyes set-ben adott a tagrekordok tárolási sorrendje.
- Egy tetszőleges rekordtípus több set-ben is definiálható tulajdonosként.
- Egy tetszőleges rekordtípus több set-ben is definiálható tagként.
- Egy tetszőleges rekordtípus nem szerepelhet ugyanazon set-ben tagként és tulajdonosként is, csak olyan set-ben lehet tag amelyben nem szerepel a rekord tulajdonosként.
- Egy setelőfordulás egy tulajdonos rekord előfordulásból és valamennyi számú tagrekord előfordulásból áll, a tagrekord előfordulások száma lehet nulla is (üres set).
- Egy setelőfordulásban minden rekordelőfordulás maximum egyszer szerepelhet csak.
- Egy setelőfordulásban a tulajdonos rekord pontosan egy előfordulásának kell szerepelnie.
- Egy setelőfordulásban bármely tagrekord típusához tetszőleges sok előfordulás tartozhat.

A fent említett CODASYL szabályok egyik fontos eleme, hogy nemcsak az adatbázis sémák szabályaival foglalkozik, hanem kitér a rekordelőfordulások és a set-előfordulások kérdéseire is. A *set-előfordulás* konkrét kapcsolódó rekordelőfordulásokat tartalmaz. A szabályrendszerből látható, hogy minden tulajdonos rekord előforduláshoz pontosan egy set-előfordulás rendelődik, melyben a tagrekordoknak tetszőleges előfordulása szerepel a *set sémában* meghatározott sorrendben. Az előfordulások vizsgálatával elérkeztünk a fizikai megvalósítás, a fizikai szintű set-előfordulás tárolási problémájához.

A korábban említett elveknek megfelelően a konkrét rekord előfordulások közötti kapcsolatok itt nem a fizikai tárolási pozícióval kódoltak, hanem kizárólag pointerek alkalmazásával kerülnek letárolásra. Ennek megfelelően a rekordelőfor-



3.19. ábra. Az NDM fizikai tárolási struktúrája

dulások között *pointer láncok* alakulnak ki, melyek mentén a kapcsolódó rekordok a definiált sorrendben bejárhatók lesznek. Így például, egy kiadó→könyv PCR-t tartalmazó set-et és egy könyvtár→könyv,olvasó set-et véve alapul az adatbázisnak tartalmaznia kell egy olyan pointer láncot, amely az egyes kiadó rekord előfordulásokból indul ki a hozzá kapcsolódó könyv előfordulásokat kijelölve. Ha a séma másik set-jét is figyelembe vesszük, akkor látni fogjuk, hogy a könyv rekord előfordulásokat a könyvtár rekordból is el kell tudni érni, ezért a könyv rekordoknak a könyvtárból kiinduló pointer láncolatban is szerepelniük kell. Mivel egy tetszőleges rekordtípus több set-ben is szerepelhet tag, vagy tulajdonosként, így az adatbázis előfordulásban egy igen kusza és összetett *pointer háló* jöhet létre, melyben minden lehetséges, pontosabban minden, a sémában előre definiált kapcsolathoz egy-egy saját pointer lánc jön létre, amelyben minden rekordelőfordulás több ilyen láncnak is tagja lehet. Ezek a pointer láncok és hálók fogják az alapját adni a rekordelőfordulások gyors elérésének, a kapcsolódó rekordok hatékony feltárásának.

A pointer láncok kialakítása során a tulajdonos rekordból minden tagrekord típushoz kiindul egy-egy láncolat. Ezt követve jutunk el az egyes tagelőfordulásokhoz. A felépítésből látható, hogy a k-adik tagrekord előfordulást csak akkor érhetjük el, ha már átolvastunk (k-1) darab tagrekordot. Ez nem tűnik a leghatékonyabb megoldásnak. Felmerülhet a kérdés, miért nem mutat közvetlenül pointer minden tagrekordra? Nos ennek az a magyarázata, hogy ez utóbbi esetben tetszőleges sok kiinduló pointerre kellene felkészülni, ami menetközben dinamikusan is változhat. Ez a megvalósítás érzékelhetően bonyolult és kevésbé hatékony megvalósító algoritmussal lenne csak működtethető. Ezzel szemben a pointer lánc esetén minden érintett rekordban pontosan egy vagy néhány, de mindenképpen rögzített számú pointernek kell helyet foglalni a rekordon belül. Ez a megvalósításnál egyszerűséget és nagyobb hatékonyságot jelent.

A sémák megfelelő kialakításával igen bonyolult háló alakítható ki, mégsem mondhatjuk azt, hogy a hálós adatmodellben minden lehetséges kapcsolat típus közvetlenül és egyszerűen ábrázolható. A korlátozás oka itt is lényegében ugyanaz, mint a hierarchikus modellnél volt: a rekordok kapcsolata a PCR jellegű kapcsolatokon nyugszik. Ennek oka – az előzőekben megadott okfejtéshez igazodva – a hatékony megvalósítás igénye. Tegyük fel, hogy a hálós modellünkben az N:M kapcsolat is közvetlenül letárolásra kerül. Ebben az esetben bármely oldali rekord előforduláshoz hatékonyan, vagyis pointer láncon keresztül el kell tudni érni a másik oldal kapcsolódó rekordjait. Ehhez elegendő egy pointer lánc, de nézzük csak meg, mennyi ilyen pointer láncban kell részt vennie egy rekord előfordulásnak? Mivel egy lánc egy szülő előforduláshoz kapcsolódik, ezért annyi ilyen láncra van szükség, amennyi kapcsolódó rekord előfordulás létezik a másik oldalon. Mivel ezek száma tetszőleges sok lehet az N:M kapcsolat jellegéből eredően, ezért egy rekordban tetszőleges sok pointer tárolására kellene felkészülni. Ez viszont sokkal rosszabb hatékonysággal oldható meg, mint az az eset, amikor csak egy pointernek kell helyet biztosítani kapcsolattípusonként.

A fentiekből következik tehát, hogy az *N:M kapcsolat* itt sem ábrázolható köz-

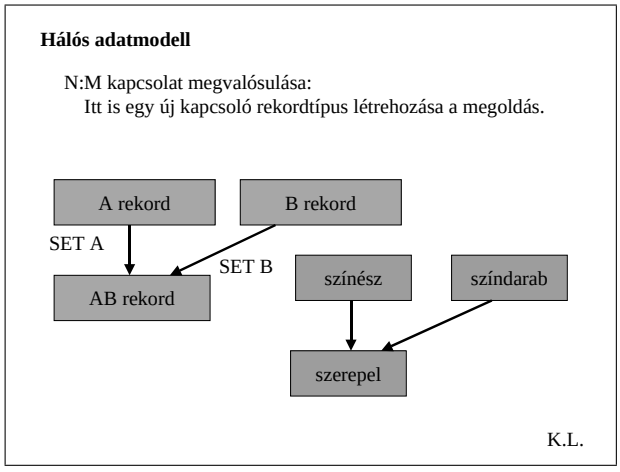
vetlenül. Azaz itt is egy kerülőutat kell keresni. Szerencsére a már megismert átalakítás itt is járható, vagyis az lesz a megoldás, hogy az N:M kapcsolatnál bevezetünk egy új kapcsoló rekordot, mely mindkét oldali rekordhoz immár PCR kapcsolatokon keresztül fog csatlakozni. Így két set lesz érintett a kapcsolat tárolásában. Vegyük például a színész-színdarab kapcsolatot a szerepel jelentéssel. Mivel egy színészhez több darab is társítható és egy darabban több színész is játszhat, ezért ez N:M kapcsolat. Az adatbázisban való ábrázoláshoz át kell alakítani a kapcsolatot PCR jellegű kapcsolatokra. Első lépésként létrehozunk egy kapcsoló egyedet. Ennek minden rekordja egy kapcsolódó párost reprezentál. Nevezzük el ezt a rekordtípust 'szerepel'-nek. A szerepel rekord PCR kapcsolatban áll mind a színész, mind a színdarab rekorddal, és mindkét kapcsolatban a gyerek szerepkört tölti be. A PCR ábrázolására a set szolgál a hálós modellben, ezért most két set-re van szükség. Az egyik set-ben a színész a tulajdonos, és a szerepel a tag, a másikban pedig a színdarab a tulajdonos, és ismét a szerepel lesz a tagrekord.

A hierarchikus modellben jelentkező egyéb problémák közül a többértékű mező és az összetett mező közvetlenül megoldható, mert a hálós modell támogatja a komplex mezők megvalósítását. A még említendő modellezési nehézségek közé tartozik a fentieken kívül a többszörös kapcsolatok kérdése. Ennek szokásos megoldása egy külön kapcsoló rekord alkalmazása, mely minden kapcsolódó egyed alatt gyerekként foglal helyet.

3.2.1. Hálós adatdefiníciós nyelv

Minden adatbáziskezelő rendszerhez tartozik egy parancsnyelv, amelynek utasításaival írható le a létrehozandó adatrendszer pontos szerkezete. A követelményeknek megfelelően a parancsnyelvnek az adatmodellhez kell igazodnia. A *hálós adatdefiníciós nyelvnek* (NDDL) ki kell térnie az alábbi komponensek megadására:

– adatbázis azonosítása,



3.20. ábra. N:M kapcsolat megvalósulása NDM-ben

– rekordtípusok megadása,
– mezők definiálása,
– set-ek definiálása.

A fenti elemek megadására a következő utasítások szolgálnak:

SCHEMA NAME IS sémanév	adatbázis séma azonosítása
RECORD NAME IS rekordnév	
szerkezet-leírás	rekord típus megadása
SET NAME IS setnév	
szerkezet-leírás	set típus megadása

A rekord típusának definiálásakor megadjuk a

– rekord nevét (NAME IS név),
– mezők nevét és típusát (szint név típus),
– tárolási paramétereket (LOCATION...).

A mezők megadásakor a szint a mező elem szintjét jelzi, ami arra utal, hogy az a mező a hierarchia mely szintjén áll. Ez a csoportmezők megadásakor fontos, amikor is vannak összesítő és részletező szintek. Például a lakcím mezőt véve, melynek elemei az irányítószám, város, utca elemek, a lakcím egy összefoglaló szintű mező. Ez a legmagasabb szint, ezért ez kapja a 01 szintszámot. Az irányítószám, város elemek lesznek a részletező tagok, ezért ezeknél a szintszám jelzése 02. A mező definíciója ez alapján az alábbi alakú lesz, ahol C(n) egy n hosszúságú karakter típusra utal:

```
01 lakcim
   02 irányitószám     C(6)
```

Hálós adatmodell

Séma leíró, definíciós nyelv (DDL)

```
SCHEMA NAME IS VALLALAT
RECORD NAME IS ALKALMAZOTT
01 NEV
02   CSALADNEV C(15)
02   KERESZTNEV C(15)
LOCATION MODE CALC USING KOD
RECORD NAME IS FELADAT
01 LEPES OCCUR 0 TO 10 TIMES
       ...
SET NAME IS DOLGOZO
ORDER IS SYSTEM DEFAULT
OWNER IS ALKALMAZOTT
MEMBER IS FELADAT
```

K.L.

3.21. ábra. A hálós sémaleíró nyelv

02 város	C(30)
02 utca	C(30)

A vektor típusú mezőknél az adatérték többszörösen is előfordulhat a mezőben (mint például az alkalmazottak esetén a végzettség). Ezért a tárolása nem egy skalárban, hanem egy vektorban történik. A vektor esetében több cellát kell lefoglalni, ahol a helyfoglaláshoz ismerni kell a lefoglalandó cellák darabszámát. Ezért a vektor típusú mezők esetében szerepeltetni kell az ismétlődési darabszámot. A megadás formátuma:

szint név típus OCCUR n1 TO n2 TIMES

A felírásban az ismétlődő elemek száma n1 és n2 között szerepelhet.

A set-ek esetében közölni kell az adatbázis kezelővel, hogy

- mi a set neve,
- mely rekord a tulajdonosa,
- mely rekordok alkotják a tagjait, és
- mi lesz a tagok letárolási sorrendje.

Ezen adatok megadására az alábbi kulcsszavak szolgálnak:

SET NAME IS	setnév:	a set nevének megadása
OWNER IS	rekordnév:	tulajdonos kijelölése
MEMBER IS	rekordnév:	tag rekord megadása
ORDER IS	mód:	láncolási sorrend meghatározása

A set és a rekordok típusának kijelölésével minden szükséges szerkezeti elem megadható a hálós adatbázis felépítéséhez. A szerkezeti elemek megadása után elkezdődhet az adatbázis érdemi használata, amikor adatokat viszünk fel, módosítunk vagy kérdezünk le. Ehhez a DDL utasítások mellett adatkezelő, adatlekérdező utasításokra is szükség van.

3.2.2. Hálós adatkezelő nyelv

Az adatkezelő nyelven belül mi most elsősorban az *adatlekérdező* komponenst (DQL, Data Query Language) fogjuk részletezni, hiszen ez a rész a legösszetettebb, legsajátosabb eleme a hálós adatmodellnek. E rész megismerésével még jobban elsajátítható lesz a hálós adatbázis kezelési szemlélet gondolkodásmódja, ezáltal a további adatmodellek lényegi elemei is jobban felismerhetőek lesznek. A DQL ismertetése előtt célszerű a nyelv működési logikáját egy kicsit absztraktabb megközelítésben megismerni. Úgy ahogy a DDL rész előtt a séma általános szerkezetét tárgyaltuk, most előbb áttekintjük a lekérdezési műveletek jellegét, típusait.

A DQL nyelv egyik fő jellemzője, hogy az utasítások gazdanyelvi környezetben működnek, hasonlóan a hierarchikus adatmodell lekérdezési felületéhez. Itt is léteznek kapcsoló változók, melyek mint osztott elérésű változók hidat képeznek

az adatbáziskezelő rendszer és a program, az alkalmazás között. Az adatbázisban rekord pointerek jelzik, hogy mely rekordok állnak éppen feldolgozás alatt. A kapcsoló változók ezeket, a pointerekkel kijelölt rekordok értékeit tartalmazzák. A különböző tárolási struktúra szinteknek megvan a saját rekord pointerük, ami megadja, hogy mely rekordot érintették utoljára ezen a szinten. Az adatbázis az alábbi szintű rekord pointereket tartalmazza:

- *Rekord szintű*: megadja, hogy mely rekordelőfordulást érintették utoljára a rekordtípus rekordjaiból.
- *SET szintű*: megadja, hogy mely rekordelőfordulást érintették utoljára a set-típus rekordjaiból.
- *DB szintű*: megadja, hogy mely rekordelőfordulást érintették utoljára az adatbázis összes rekordja közül.

A lekérdezés itt is *navigációs* jellegű, vagyis a lekérdezés megfogalmazása során azt kell meghatározni, hogy milyen irányban mozgassuk az egyes rekord pointereket, hogy a kívánt rekordelőfordulásokat érintsük a mozgás során.

A rekord pointer mozgatására a következő lehetőségek vannak:

$p^1{}_{felttel}$*(rekord)*	: a megadott rekordtípuson belül az első olyan rekordelőfordulásra áll rá, mely teljesíti a paraméterként megadott feltételt.
$p^n{}_{felttel}$*(rekord)*	: a megadott rekordtípuson belül a következő olyan rekord előfordulásra áll rá, mely teljesíti a paraméterként megadott feltételt.
o (set, rekord)	: a megadott set-típuson belül megkeresi a megadott rekordtípus kijelölt előfordulását, majd elmegy ezen előforduláshoz tartozó

Hálós adatmodell

Adatkezelő nyelv (DML, DQL)

- gazdanyelvi környezetbe ágyazott kezelő nyelv
- rekordorientált adatkezelés
- az osztott memóriában több kapcsoló változó (pointer)
  - -- SET szintű
  - -- Rekord szintű
  - -- DB szintű
- keresés művelete rekordonként a kapcsoló változókon át:

$p^1_{feltétel}$(rekord)	- rekordban első előfordulás
$p^n_{feltétel}$(rekord)	- rekordban köv. előfordulás
o (set,rekord)	- setben tulajdonos
$m^1_{feltétel}$(set,rekord)	- setben tag első előfordulás
$m^n_{feltétel}$(set,rekord)	- setben tag köv. előfordulás

K.L.

3.22. ábra. A hálós adatkezelő nyelv jellemzői

tulajdonos rekord előforduláshoz.

$m^1{}_{felttel}(set,\ rekord)$: a megadott rekordtípuson belül az első olyan rekord előfordulásra áll rá, mely teljesíti a paraméterként megadott feltételt és benne van a set tagrekord előfordulásai között, valamint gyereke a set tulajdonos rekord előfordulásának is.

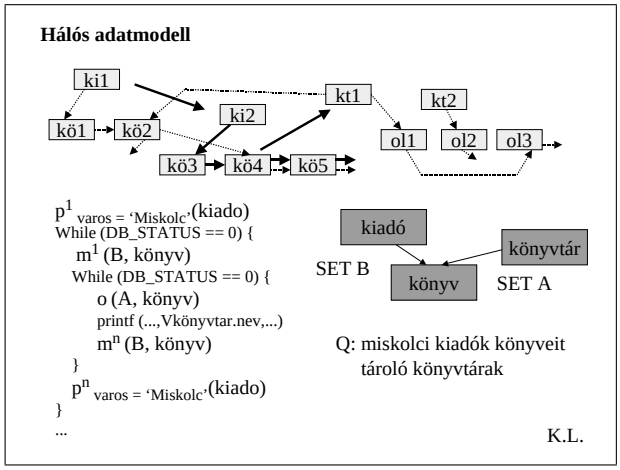
$m^n{}_{felttel}(set,\ rekord)$: a megadott rekordtípuson belül a következő olyan rekord előfordulásra áll rá, mely teljesíti a paraméterként megadott feltételt és benne van a set tagrekord előfordulásai között, valamint gyereke a set tulajdonos rekord előfordulásának is.

A fenti műveleteket egy gazdanyelvi programozási nyelvbe beágyazva használjuk, mert egyes lépéseket rendszerint ismételten is végre kell hajtani, egyes lépések pedig csak feltételesen kerülnek meghívásra. A hívó programban viszont tudni kell, hogy az előző lépés sikeresen végrehajtódott-e, hiszen egyes lépéseknek csak akkor van értelme, ha az előző lépések mindegyike lefutott. A sikeres végrehajtás jelzésére speciális osztott hozzáférésű változó szolgál, melyet az adatbázis állít be, és a program olvashatja, így értesülve a végrehajtás eredményességéről. Ezen változó azonosítására mi most a

DB_STATUS

jelölést alkalmazzuk, melynek értéke

- 0 : ha sikeres a végrehajtás,
- >0 : ha sikertelen a végrehajtás.



3.23. ábra. Példa lekérdezés hálós modellben

Példaként vegyük a korábban már említett rendszert, melyben a könyvtárakat, könyveket és kiadókat fogjuk össze. A megoldandó lekérdezés megfogalmazása:

> *Adjuk meg mindazon könyvtárakat, melyek tárolnak miskolci kiadójú könyveket.*

A feladat megoldásához előbb magunk előtt kell látni az adatbázisban megvalósuló, a kiadó, könyv, könyvtár rekord előfordulásokból felépülő hálót. A séma szerint kétféle set-típus él az adatbázisban. Az A SET-ben a könyvtár rekord előfordulásokból indulnak ki az egyes pointer láncok a könyv tagrekordok felé. A B SET szerint pedig a kiadó rekord előfordulásokból is ered egy pointer lánc a tagot alkotó könyv rekordok felé. Tehát minden könyv rekord két pointer láncban is szerepelhet.

Ha látjuk az adatbázis háló struktúrát (3.23. ábra), akkor következhet a lekérdezési navigációs műveletsor összeállítása. A feladat szerint bizonyos könyvtár rekordokat kell kiválasztani, mégpedig azokat, melyekben van miskolci kiadótól származó könyv. Ez az adatbázis szerkezetre lefordítva annyit jelent, hogy azon könyvtár rekordokat kell megkeresni, amelyekből van kapcsolat, összeköttetés valamely miskolci kiadóhoz. Ugyanis a feltétel szerint a könyvtárból el kell tudni jutni olyan könyvekhez, melyekből el lehet jutni miskolci kiadóhoz. A keresést célszerű a kiadótól elkezdeni. Ennek oka, hogy míg a könyvtárból elindított kereséseknél, csak azon könyvtárak lesznek jók, melyekből el tudunk jutni miskolci kiadóhoz, addig a fordított irányú navigáció esetén, vagyis a miskolci kiadókból való kiindulásnál bármely könyvtár jó lesz, amihez eljutunk a navigációs mozgás során. Ezért a kiadóknál kezdjük a lekérdezés végrehajtását. A navigáció menete:

- miskolci kiadóktól a könyvekig;
- könyvektől a könyvtárakig.

A fenti mozgás az alábbi lépéseken keresztül valósítható meg:

Mozgás a miskolci kiadók rekordjai között
Mindenilleszkedő rekordra
    Mozgás a könyv tagrekordok között
    Mindenilleszkedő rekordra
        Mozgás a könyv rekord könyvtár szülőjéhez
        A könyvtár rekord feldolgozása
    Mozgási ciklus vége
Mozgási ciklus vége

A fenti műveletsor még precízebb felírása a navigációs műveletekkel az alábbiakban adható meg, ahol a könyvtár rekordtípus kapcsoló változóját a 'Vkönyvtár' szimbólum jelzi:

$P^1{}_{varos='Miskolc'}(kiado)$
while(DB_STATUS == 0) {
    m^1*(B, könyv)*
    while(DB_STATUS == 0) {
        o(A, könyv)

printf (. . . ,Vkönyvtár.név,..)
m^n *(B,könyv)*
}
$P^n{}_{varos='Miskolc'}$ *(kiado)*
}

A fenti felírás már pontosan megadja a végrehajtandó műveletsort, viszont formailag nem illeszkedik még a gépi feldolgozásra, hiszen például indexeket is tartalmaz. Ezért a fenti műveleteket át kell alakítani egy parancssorrá, melyben az egyes navigációs lépéseket a megfelelő NDQL utasításokkal helyettesítjük. Mintarendszerünkben az alábbi navigációs parancsok értelmezettek:

– *FIND FIRST | NEXT rekordtípus USING feltétel*
Keresés egy rekordtípus rekord előfordulásai között a minta szelekciós feltétel alapján. A FIRST kulcsszó az első előfordulást, NEXT a következő rekordelőfordulást adja vissza. A parancs a $p^{1|n}{}_{felttel}$(rekord) műveletet valósítja meg.

– *FIND FIRST | NEXT rekordtípus IN CURRENT setnév SET USING feltétel*
Keresés egy rekordtípus rekord előfordulásaira a megadott set-előfordulás tagrekordjai között. A FIRST kulcsszó az első előfordulást, a NEXT a következő rekordelőfordulást adja vissza. A parancs az $m^{1|n}$ (set, rekord) műveletet valósítja meg. Az a set-előfordulás fog kiválasztódni, amely a pointerrel kijelölt tulajdonos rekordelőforduláshoz tartozik. A feltétellel szűkíthető az érintett rekordok köre.

– *FIND OWNER OF rekordtípus IN CURRENT setnév SET*
Keresés a megadott típusú rekordelőfordulás tulajdonos előfordulását adja vissza a megadott set-hez tartozóan. A parancs az o(set, rekord) műveletet valósítja meg.

Hálós adatmodell

Adatkezelő nyelv

- keresés művelete rekordonként a kapcsoló változókon át:
FIND FIRST\NEXT rekord USING feltétel
FIND FIRST\NEXT rekord IN CURRENT xx SET
USING feltétel
FIND OWNER OF rekord IN CURRENT xx SET

- módosítás rekordonkét a kapcsoló változókon át:
STORE rekordtípus
ERASE rekordtípus
MODIFY rekordtípus

K.L.

3.24. ábra. A hálós modell adatkezelő utasításai

A DBMS rendszerekben a DQL utasítások mellett létezniük kell olyan utasításoknak is, melyekkel módosíthatók a rekordokban tárolt mező értékek. A hierarchikus modellben a DML utasítások végrehajtási menete a következő:

- navigálással ráállunk arra a rekord előfordulásra, melyet módosítani kell;
- beállítjuk a kapcsoló változó értékét a kívánt új értékre;
- meghívjuk a megfelelő DML utasítást.

Az alap DML utasítások az alábbi műveletekre terjednek ki:

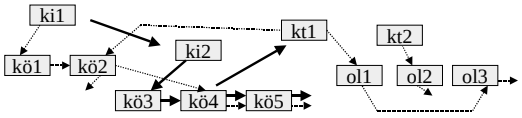
- *STORE rekordtípus*: új rekord felvitele.
- *ERASE rekordtípus* : rekord törlése.
- *MODIFY rekordtípus* : rekord módosítása.

Az új rekord felvitel műveletének sajátossága, hogy a STORE parancs kiadása előtt mindazon set típusokban, melyekben tagrekordként szerepel az új rekord típusa, a tulajdonos típusnál arra az előfordulásra kell navigálni, mely alatt tárolódnia kell az új rekordnak. Vagyis a STORE parancs azon set előfordulásba szúrja be az éppen felvitt rekordot, melyet a rekordtípus aktuális pointere kijelöl.

Például, ha az a feladatunk, hogy egy IN57657 kódszámú könyvet kell felvinni, melyet az Atlasz kiadó adott ki, akkor az alábbi lépéseket kell megtenni:

```
FIND FIRST kiado USING nev = 'Atlasz'
if (DB_STATUS == 0) {
        Vkönyv.kod = 'IN57657'
        Vkönyv.cim = 'Forró hógolyó'
        ...
        STORE könyv
}
```

Hálós adatmodell




```
NEXEC FIND FIRST kiado USING varos = 'Miskolc'
While (DB_STATUS == 0) {
   NEXEC FIND FIRST könyv IN CURRENT B SET USING kiadas<1975
   While (DB_STATUS == 0) {
      NEXEC FIND OWNER OF könyv IN CURRENT A SET
      printf (..., Vkönyvtar.nev, ...)
      NEXEC FIND NEXT könyv IN CURRENT B SET USING kiadas < 1975
   }
   NEXEC FIND NEXT kiado USING varos = 'Miskolc'
}
...
```

K.L.

3.25. ábra. Példa lekérdezés paracssori megoldása

Tekintsünk a DQL utasításokra egy újabb példát. Ehhez kanyarodjunk vissza a korábbi lekérdezéshez, azzal a különbséggel, hogy még egy feltételt kell teljesíteni: csak az 1975 előtt megjelent, miskolci kiadójú könyvek adatait kérdezzük le.

A megoldás során a korábban már megadott navigációs műveletek helyébe kell a megfelelő DQL utasításokat beírni. Az egyetlen módosítás, hogy a B set-beli tagrekord keresésekor egy év szerinti szűrő feltételt is meg kell adni.

Az egyes műveletek parancssori megfelelője:

$p^{1\|n}{}_{felttel}$(rekord)	FIND FIRST|NEXT rekord USING feltétel
$m^{1\|n}{}_{felttel}$(set, rekord)	FIND FIRST|NEXT rekord IN CURRENT x SET USING feltétel
o (set, rekord)	FIND OWNER OF rekord IN CURRENT x SET

Közben nem szabad megfeledkezni a műveleti státuszok lekérdezéséről sem az egyes műveletek végrehajtása után. A konvertált műveletsor alakja:

```
NEXEC FIND FIRST kiado USING varos = 'Miskolc'
while(DB_STATUS == 0) {
    NEXEC FIND FIRST konyv IN CURRENT B SET
        USING kiadas < 1975
    while (DB_STATUS == 0) {
        NEXEC FIND OWNER OF konyv IN CURRENT A SET
        printf (...,Vkonyvtar.nev, ...)
        NEXEC FIND NEXT konyv IN CURRENT B SET
            USING kiadas < 1975
    }
    NEXEC FIND NEXT kiado USING varos = 'Miskolc'
}
```

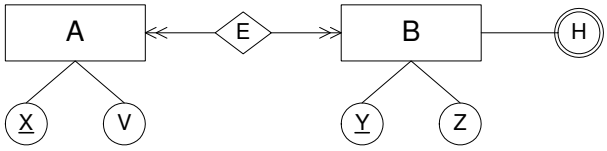
Még számos további lehetőség és parancs tartozna a hálós adatmodellhez, mi azonban nem vesszük ezeket. A fejezet célja elsődlegesen a relációs adatmodell előtti adatmodellek formátumának, működési módjának és lehetőségeinek áttekintése volt, mely révén szélesebb áttekintést kaphattunk az adatbáziskezelők világából és jobban megérthetjük a relációs modell funkcionalitását is. A következő részben már a relációs adatmodell tárgyalása következik.

Elméleti kérdések

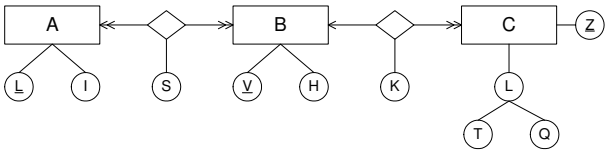
1. Hasonlítsa össze a szemantikai és az adatbázis kezelő adatmodelleket.
2. Ismertesse a hierarchikus modell strukturális részét.
3. Milyen előnyöket és hátrányokat hordoz a hierarchikus struktúra?
4. Milyen mechanizmusok vannak a hierarchikus modellben a kapcsolatok ábrázolására?
5. Ismertesse a VPCR fogalmát, szerepét, és megvalósulását.
6. Ismertesse a hierarchikus modell séma- és előfordulás-diagrammjainak felépítését és kapcsolataikat.
7. Mutassa be a hierarchikus adatmodell fizikai megvalósítását.
8. Ismertesse az ER modell hierarchikus modellre történő konverziójának lépéseit.
9. Ismertesse az ER modell tulajdonságtípusait, és ábrázolásukat a hálós adatmodellben.
10. Milyen lépéseken keresztül lehet az IFO adatmodellt hierarchikus adatmodellre konvertálni?
11. Milyen utasításokat tartalmaz a HDDL nyelv?
12. Milyen a hierarchikus adatmodellhez kapcsolódó adatlekérdező felület?
13. Milyen utasításokat tartalmaz a HDML nyelv?
14. Ismertesse a hálós adatmodell strukturális részét.
15. Mit jelent a set fogalma, és milyen speciális típusai vannak?
16. Ismertesse a CODASYL szabályokat.
17. Hasonlítsa össze a hálós és hierarchikus modell rekordstruktúráját.
18. Ismertesse a kapcsolatok ábrázolását a hálós adatmodellben.
19. Hogyan történik az N:M kapcsolatok tárolása a hierarchikus és a hálós adatmodellben?
20. Hogyan történik az 1:N kapcsolatok tárolása a hierarchikus és a hálós adatmodellben?
21. Ismertesse a hálós adatmodell fizikai megvalósításának alapjait.
22. Milyen utasításokat tartalmaz a hálós adatdefiníciós nyelv?
23. Hogyan jellemezhető az adatlekérdezés kezelő felülete a hálós adatmodellben?
24. Mire utal a navigációs jelző a hálós adatmodellben?
25. Mire szolgálnak a feldolgozási rekordpointerek és milyen típusai vannak?
26. Milyen navigációs utasítások társulnak a hálós adatmodellhez?
27. Milyen lekérdezéshez kapcsolódó (NDQL) utasítások vannak a hálós modellben?

Feladatok

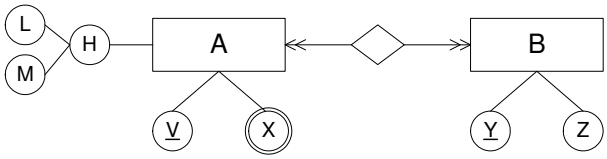
1. Konvertálja az alábbi ER sémát hierarchikus modellre:



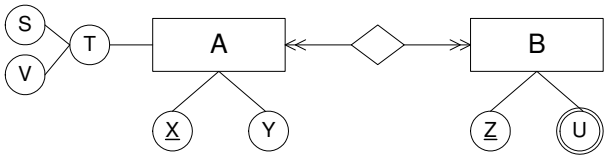
2. Konvertálja az alábbi ER sémát hierarchikus modellre:



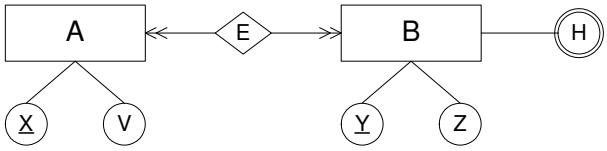
3. Konvertálja át a következő ER sémát hierarchikus sémára:



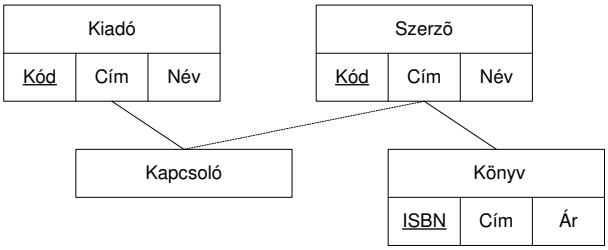
4. Konvertálja át az alábbi ER modellt hierarchikus modellre:



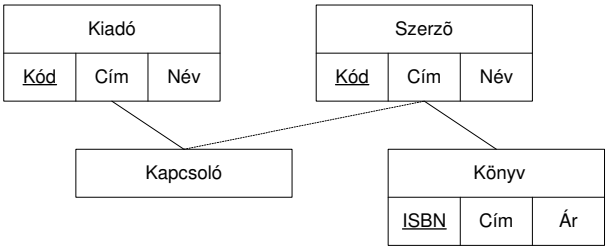
5. Konvertálja az alábbi ER sémát hálós modellre:



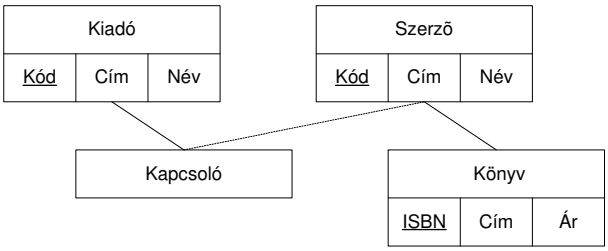
6. Adja meg az alábbi hierarchikus séma egy lehetséges előfordulás-sémáját.



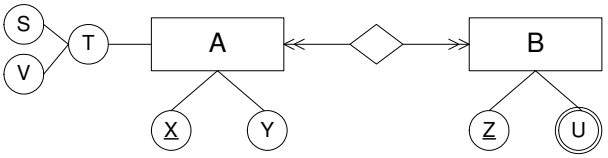
7. Adja meg az alábbi hierarchikus sémát előállító HDDL utasításokat.



8. Adja meg a szegedi székhelyű kiadók neveit lekérdező műveletsor utasításait a hierarchikus adatmodellben.



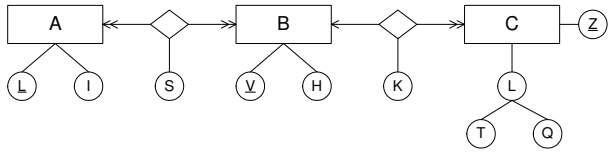
*9. Konvertálja az alábbi ER sémát hierarchikus modellre, és adja meg a létrehozó HDDL utasításokat.



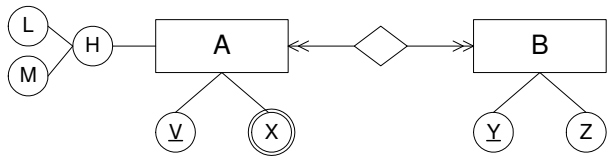
10. Adjon meg egy minta séma-előfordulást az alábbi sémához:

Rekord:	Szerző (név, kód, lakcím)
	Kiadó (név, cím)
	Könyv(cím, kód, ár)
Set:	S (tulajdonos: Szerző; tag: Kiadó, Könyv)

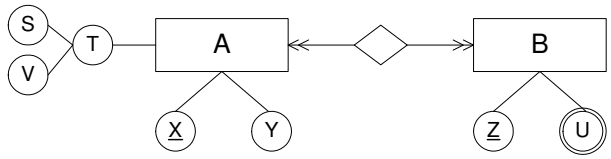
11. Konvertálja az alábbi ER sémát hálós modellre:



12. Konvertálja át a következő ER sémát hálós sémára:



13. Konvertálja át az alábbi ER modellt hálós modellre:



*14. Adja meg a sémát megvalósító NDDL utasításokat az alábbi sémához:

Rekord:	Szerző (név, kód, lakcím(város, utca, hsz))
	Kiadó (név, cím)
	Könyv(cím, kód, ár, téma(kulcsszavak*))
Set:	S (tulajdonos: Szerző; tag: Kiadó, Könyv)

15. Kérdezze le a 2000-nél drágább könyvek kiadóinak az adatait az adott séma mellett:

Rekord:	Szerző (név, kód, lakcím(város, utca, hsz))
	Kiadó (név, cím)
	Könyv(cím, kód, ár, téma(kulcsszavak*))
Set:	S (tulajdonos: Szerző; tag: Kiadó, Könyv)

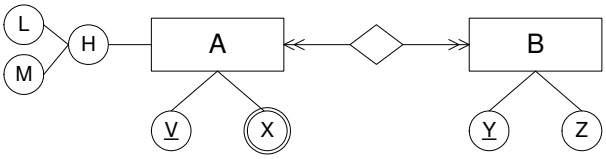
16. Kérdezze le a miskolci kiadók által kiadott könyvek címét az adott séma mellett:

Rekord:	Szerző (név, kód, lakcím(város, utca, hsz))
	Kiadó (név, cím)
	Könyv(cím, kód, ár, téma(kulcsszavak*))
Set:	S (tulajdonos: Szerző; tag: Kiadó, Könyv)

*17. Kérdezze le a miskolci kiadóknál megjelent könyvek szerzőinek a nevét az adott séma mellett:

Rekord:	Szerző (név, kód, lakcím(város, utca, hsz))
	Kiadó (név, cím)
	Könyv(cím, kód, ár, téma(kulcsszavak*))
Set:	S1 (tulajdonos: Szerző; tag: Könyv)
	S2 (tulajdonos: Kiadó; tag: Könyv)

18. Hozza létre a hálós adatmodellt az alábbi sémához, és kérdezze le azon B.Y adatokat, amelyek egy A.X = 1 értékű A előforduláshoz kapcsolódnak.



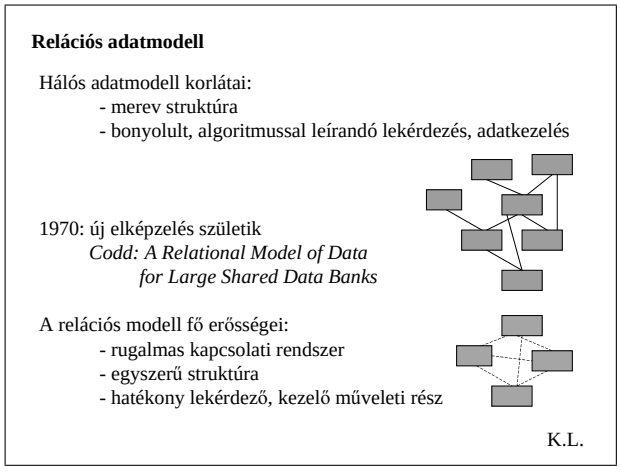
4. fejezet

A RELÁCIÓS ADATMODELL

4.1. A relációs adatmodell kialakulása

A relációs adatmodell napjaink legelterjedtebb adatmodellje. A modell alapjait 1970-ben fektette le Codd az "A Relational Model of Data for Large Shared Data Banks" cikkében. A cikk elsődleges célja az volt, hogy bebizonyítsa, létezik más alternatíva is az akkor elterjedőben lévő hálós adatmodell mellett, mely ráadásul matematikailag jobban megalapozott eszközöket, fogalmakat használ, így pontosabb, egzaktabb leírást, kezelést tesz lehetővé.

A megtervezett modellben egy igen egyszerű, könnyen megtanulható leírási módot sikerült megvalósítani. *Egyszerűségének* következtében gyorsan népszerűvé is vált a felhasználók körében, és sok implementációja született meg a személyi számítógépek piacán is. Másrészről az *elméleti megalapozottság* a kutatók, a szakemberek szimpátiáját is kiváltotta, és ez a modell számos új fejlesztési projekt



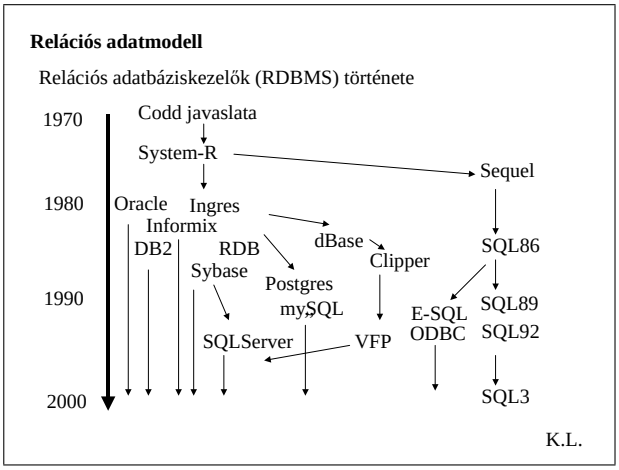
4.1. ábra. A relációs adatmodell kialakulása, jellemzése

alapját képezi. Az adatmodell fontos előnye az egyszerűség mellett a rugalmasság. A *rugalmasság* több szempontból is vizsgálható az adatbázisok esetében. Egyrészt tekinthető a kezelő felületek rugalmassága, ami annyit jelent, hogy a kezelő a meglévő utasítás elemek, parancsok segítségével tetszőlegesen állíthat össze bonyolultabb utasításokat is. A rugalmasság másik vetülete a fejlesztési eszközök rugalmasságában rejlik. Ennek mérője, hogy milyen gyorsan és milyen egyszerűen lehet az adatmodellben bekövetkező változásokat érvényesíteni az adatbázisban és az alkalmazásokban.

A korábbiakban már ismertetett speciális modellezési problémák, mint az N:M kapcsolatok, az integritási feltételek, itt viszonylag természetes, nem kilógó módon valósíthatók meg. Természetesen ezen modellnek is vannak gyengeségei, mely elsősorban a különböző szemantikai modellelemek hiányában rejlik, ezért is folynak igen intenzív vizsgálatok a relációs modellnek újabb, az SDM modellekhez közelebb álló komponensekkel történő kibővítésére.

A relációs modellt megalkotó Codd az IBM cégnél dolgozott, és abban az időben az IBM igen sok energiát ölt már bele a hálós adatmodellen alapuló DBMS rendszerének kialakításába. Mint vezető adatbázis technológiai cég, az IBM nem engedhette meg, hogy ne vegye figyelembe ezt az újszerű ötletet, ezért beindított egy kísérleti projektet, amelynek célja a relációs modell megvalósíthatóságának a vizsgálata, egy minta relációs DBMS (RDBMS) létrehozása volt.

A beindított projekt a *System/R* elnevezést kapta, ahol az R betű a relációs (relational) jelölésre utal. A System/R projekt két fázisban zajlott le. Az 1970-es évek közepére kifejlesztettek egy prototípus RDBMS-t, amely még egyfelhasználós környezetben működött. Az első részben a kutatások a fizikai elérési módszerek kidolgozására, a kezelő nyelv kialakítására, és a műveletek optimalizálására irányultak. A projekt második fázisában kibővítették a rendszert többfelhasználós



4.2. ábra. A relációs adatbáziskezelők története

platformra, így már egy igazi, gyakorlatban is alkalmazható RDBMS rendszert hoztak létre. A projekt 1976-ban zárult le, bizonyítva a relációs modell életképességét. A System/R sikereire felfigyeltek más szoftverfejlesztők is. Ennek következtében a 70-es évek végére kialakult néhány új önálló társaság, amelyek a piacon is eladható RDBMS rendszerek kifejlesztésébe kezdtek.

Az első termékként megjelenő RDBMS rendszer 1979-ben jelent meg. Ez egy PDP-n futó *Oracle* rendszer volt. Talán sokunknak ismerősen cseng ez a név, hiszen az Oracle ma is a világ legnagyobb RDBMS forgalmazó cége, melynek nyeresége napjainkban is drasztikusan fokozódik. Az Oracle rendszer mellett, nemsokára más cégek is jelentkeztek saját RDBMS termékeikkel, ezen nevek között megemlíthetők az Informix, Ingres, DB2, Sybase és az RDB.

A relációs modell sikerén és egyszerűségén felbuzdulva a PC-s rendszerek elterjedésével olyan cégek is felbukkantak, melyek a relációs modellen alapuló PC-s adatkezelő rendszereket jelentettek meg. Ezen rendszereket a szolgáltatásaik, képességeik és megbízhatóságuk miatt sokan nem tekintik igazi RDBMS rendszereknek, de strukturális alapjaikban a relációs adatmodellhez állnak a legközelebb és napjainkban e rendszerek egyre több elemet is megvalósítanak a relációs adatmodellből. A piacon e rendszerek igen jelentős szerepet töltenek be, hiszen megfizethetőek és a kisebb alkalmazások részére megfelelő szolgáltatást nyújtanak. Ezért mi is nagyobb teret fogunk szentelni e PC-s rendszereknek.

A későbbiekben kitérünk az Oracle rendszer mellett a PC-s VFP, azaz Visual FoxPro fejlesztő eszközre is, amelynek segítségével megismerkedhetünk a 4GL jellegű hatékony alkalmazásfejlesztés lehetőségeivel és szépségeivel. A szorosabban vett témakörhöz tartozik még a különböző SQL szabvány nyelvek megismerése, amelyek a relációs adatbázisok elérését biztosítják a különböző szituációkban. Az interaktív elérést támogató alap SQL verziók mellett kitekintést adunk a gazdanyelvi környezetből való adatbázis elérést szolgáló E-SQL és ODBC felületekre is.

A relációs modell, mint minden más adatmodell tartalmaz statikus és dinamikus elemeket is. A statikus elemek az adatok tárolási struktúráját írják le, míg a dinamikus oldal a struktúrákon végzendő operációkat, műveleteket adja meg. A relációs modell a struktúra és a műveletek mellett megjelentet egy újabb szempontot is, az adatintegritási komponenst, mely nem része sem a strukturális, sem a műveleti komponensnek, hiszen elsődlegesen az adatok értékét jellemzi, szabályozza. Az integritási feltételek tehát, az adatok értékei közötti kapcsolatokat szabályozzák. Így a *relációs adatmodell* az alábbi három komponens együttesét jelenti:

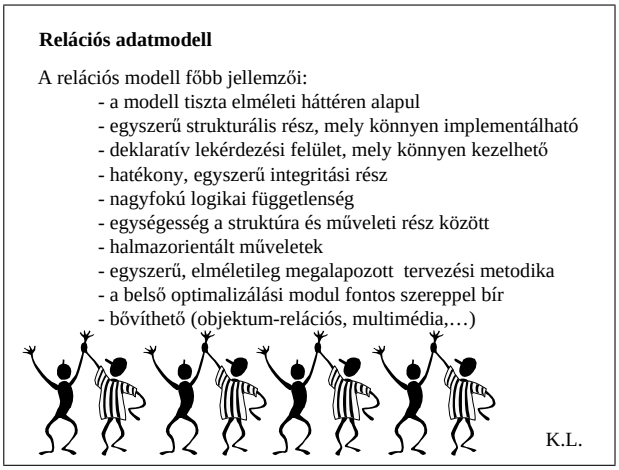
- *relációs adatstruktúra*,
- *relációs műveletek*,
- *relációs integritási feltételek*.

A relációs modell előnyei, főbb jellemzői tömören és leegyszerűsítve a következő pontokban foglalhatók össze.

- Az adatmodell *egyszerű és könnyen megérthető strukturális* részt tartalmaz. A strukturális rész egyszerűsége a közérthetőség mellett az alkalmazhatóságot is növeli, hiszen az egyszerűbb formulák általánosabban

fordulnak elő a különböző adatforrásokban. A relációs modell strukturális része nem tartalmaz a fizikai szinthez közel álló részleteket, célja olyan adatstruktúra modell megalkotása volt, amely nagymértékben független a fizikai megvalósítástól, egyszerű és könnyen érthető.

– A modellhez olyan műveleti rész csatlakozik, amely a programozási nyelveknél *egyszerűbb kezelői felületet* biztosít, hogy minél általánosabban, minél szélesebb körben lehessen használni. Az utasítások a procedurális jelleg helyett deszkriptív elemeket tartalmaznak, azaz magasabb szinten lehet megfogalmazni a megkívánt tevékenységet. Az alkalmazott nyelv egyik jellegzetessége, hogy nem rekordonként kezeli az adatokat, hanem egyszerre több rekord együttesét is érinti. A deszkriptív jelleg ellenére a kezelő nyelv biztosítani tudja az adatkezeléshez szükséges összes műveleti elemet.
– Az adatmodell integritási részében egyszerű, közérthető, de egyben hatékony feltételeket definiál. Az integritási feltételek előnye, hogy *deszkriptív* megadásukkal a *szabályok* magában az adatbázisban kerülnek letárolásra, és az RDBMS automatikusan ellenőrizni fogja e szabályok betartását. Pontosabban csak olyan tevékenységeket fog megengedni a relációs DBMS, amelyek nem sértik a megadott integritási előírásokat.
– *Egyszerű tervezési metodika.* Az adatbázis tervezése jól definiált, egyértelmű elméleti alapokon nyugszik. A tervezés elméleti alapjai az úgynevezett normalizálási szabályokon alapulnak, amelyek a relációs modell strukturális lehetőségeit és a modellezendő fogalmak közötti függőségi kapcsolatokat veszik figyelembe. A normalizáció végigkövetésével egy hatékony, áttekinthető adatmodellt kaphatunk.
– *Nagyfokú logikai függetlenség.* A relációs modell elrejti a felhasználók elől a megvalósítás részleteit. A felhasználó nyugodtan használhatja a magas



4.3. ábra. A relációs modell főbb jellemzői

szinten hozzá közel álló fogalmakat, a rendszer egy belső optimalizáló komponens segítségével ki fogja választani a számára leghatékonyabb fizikai megvalósítást. A függetlenség fokozására a rendszer lehetővé teszi a különböző logikai nézetek definiálását, azaz egy létrehozott struktúrát számos különböző nézetben fel lehet használni.

– *A relációs modell tiszta elméleti alapokon nyugszik.* A halmazelméletre és a matematikai logikára alapozva a relációs modell az első, mélyebb elméletre támaszkodó adatmodell. Az elméleti megalapozottság fő előnye, hogy általa a modell megbízhatóbb és kiszámíthatóbb lesz, és könnyebb a modell tulajdonságainak a vizsgálata is. A megbízhatóság biztosítja, hogy nem fogják kellemetlen események, váratlan hibák megzavarni az adatmodellen alapuló adatbáziskezelők működését.
– *Egységesség.* Az adatbázisban mind a normál, mind a struktúra leíró, azaz metaadatok, ugyanazon módon és formalizmussal kerülnek letárolásra, ezenkívül az adatok kezelése is ugyanazon utasításokkal valósítható meg. Ez nagyban megkönnyíti az adatbázis adminisztrációját is.

Az adatmodell három oldala közül elsőként a strukturális részt, majd az adatintegritási részt, végül a műveleti részt fogjuk elemezni az elkövetkező alfejezetekben.

4.2. A relációs adatstruktúra

A relációs adatmodell egyik fő eltérése a korábban ismertetett hierarchikus modellhez viszonyítva abban rejlik, hogy nem épít be a modellbe fix, rögzített kapcsolattípusokat, mint a PCR vagy a VPCR volt. Valójában semmilyen előző értelmezésben vett direkt kapcsolatleíró szerkezeti elem nem létezik a relációs modellhez, ami nem jelenti azt, hogy a kapcsolatokat egyáltalán nem lehet letárolni. Ha ez igaz lenne, akkor természetesen nem használná senki a relációs modellt, hiszen a kapcsolatok nélküli modell igencsak csonka képét adná a valóságnak. A relációs adatmodell a következő strukturális elemekből épül fel:

– domain,
– mezők,
– rekordok,
– relációk,
– adatbázis.

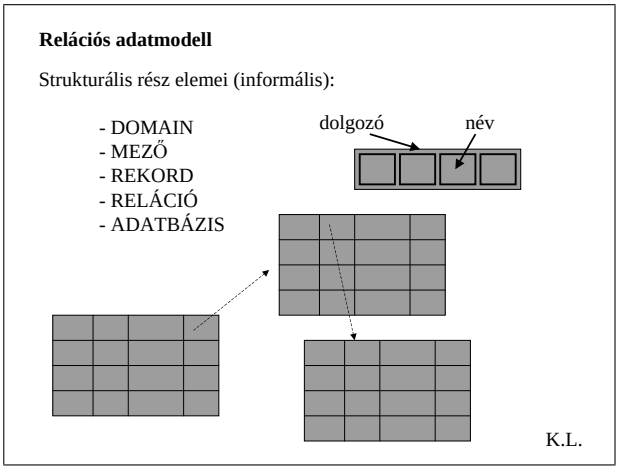
Ezek a strukturális elemek egyre bővülő szinteket jelentenek; a rekord ugyanis több mezőből épülhet fel, a reláció több rekordot tartalmaz, az adatbázisba pedig több reláció is tartozhat. A relációs modellben a mező, talán elsőre meghökkentő módon, újra csak elemi értékű lehet, ezért nem alkothatunk összetett vagy többértékű mezőket, mint ahogy azt a hálós modellben megtehettük. Ez a visszalépés azonban a modell egyszerűségét, a hatékonyabb adatkezelést szolgálja, ugyanis a relációs adatábrázolás nem használja a pointeres hivatkozásokat, ezért kénytelen minden hatékonyságot javító "fűszálba belekapaszkodni" és kihasználni.

Míg a mező és rekord fogalmai ismertek a korábbi adatmodellekből, addig a reláció szerkezeti egység egy teljesen új fogalmat jelent. Első megközelítésben a

reláció egy táblázatnak tekinthető, amelyben az azonos szerkezetű rekord előfordulások foglalnak helyet. A táblázat könnyen átlátható egység, melyben a sorok jelentik a rekordokat, és az oszlopokban az egyes mezők helyezkednek el.

A rekordok közötti kapcsolatok ábrázolása egészen újszerű módon valósul meg, nevezetesen az értékeken keresztül. Az alapelv a következő: minden rekordnak, vagyis sornak van olyan mezője, vagy vannak olyan mezői, amelyek értéke egyértelműen meghatározza az illető rekordot. Ha egy másik rekord kapcsolódik ezen rekord előforduláshoz, akkor a kapcsolat jelzésének módszere az, hogy a kapcsolódó rekordba beteszünk egy olyan mezőt, amelynek értéke a hivatkozott rekord azonosító értéke. Így a két rekord megfelelő mezőinek értékegyezősége fogja jelezni az összetartozást. Nos, ezen megoldásba belegondolva látszik, hogy itt egyáltalán nem volt fontos a hatékonyság, hiszen ez a megoldás sokkal több időt igényel a kapcsolatok feltárásakor, mint akár a pozíció, vagy akár a pointer alapú kapcsolódás. Mégis miért választottak egy rosszabbnak tűnő megoldást? Azért mert így sokat nyerhettek a rugalmasság oldalán. A modellben igen hatékonyan lehet módosítani a kapcsolatokat, és az adatok lekérdezésekor is rugalmasan össze lehet kapcsolni az egyes relációkat. Igen fontos már itt megjegyezni, hogy az adatok lekérdezésekor minden reláció *egyenértékű*, nincs alá- vagy fölérendelt rekord, mint a korábbi modellekben.

A relációs értelmezésben a mező egy tulajdonságnak felel meg. A mezőtípust a relációs terminológiában szokás domain-nek is nevezni. A *domain* alatt tehát a mező által felvehető értékek halmazát értjük. Az életkor mező esetén, a mezőhöz tartozó domain a lehetséges életkorokat jelentő egész számok halmaza lesz, például a [0-130] intervallum. Bizonyos értelmezésekben a domainhez még egy jelentéselemet is hozzácsatolnak az értékhalmaz mellett. A domain jelentése az azonosító nevében testesül meg. Ez alapján, az életkor és a testsúly domain-eknek hiába is lenne azonos az értelmezési tartományuk, értelmük más és más, ezért nem sze-



4.4. ábra. A relációs modell strukturális része

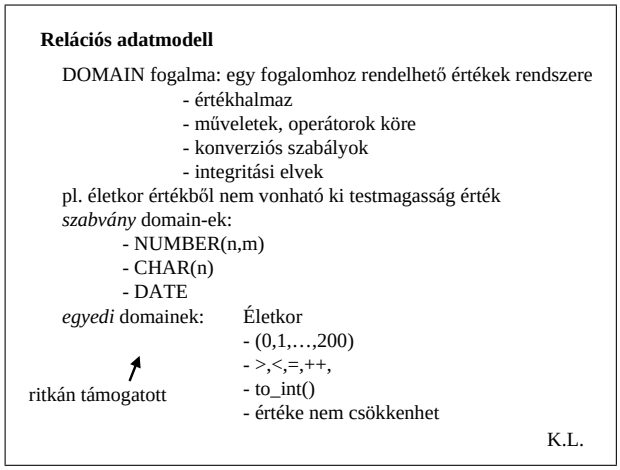
repelhetnek együtt bizonyos operátorok esetén. Így például értelmetlen dolognak tűnik egy életkor és egy testsúly érték összeadása.

A relációs modellben, hasonlóan a hierarchikus modellhez, nem lehet sem összetett, sem többértékű mezőket definiálni. Ezek a megkötések a relációs modell azon alapelemei közé tartoznak, melyek nagymértékben növelték a megvalósíthatóság hatékonyságát, a modell egyszerűségét, de napjainkra, a bonyolultabb objektumok tárolási igényeinek növekedtével e megkötések egyben korlátaivá is váltak az adatmodellnek.

A domain tehát a relációs modellben elemi típusok halmazát jelenti. Az atomiság arra utal, hogy tovább nem bontható az adat. Így domain lehet például a rendszámok halmaza, az életkorok halmaza. A domainek lehetnek standardok, kész domainek, mint például az egész számok halmaza vagy a sztringek halmaza, de a relációs modell lehetőséget nyújt saját domainek definiálására is.

A *saját domain* alkalmazása, mint például a rendszámok halmaza, több szemantikai értéket hordoz, szemléletesebb, és implicite más értékhalmazt is definiálhat, mint a standard típusok. A domain megadásához meg kell adni egy domain nevet és egy adattípust, esetleg az adatformátumot is.

A *mezők* az egyes rekordokon belüli elemi tárolási egységek. A mezőket nevükkel és a hozzájuk rendelt domain és integritási feltétel megadásával azonosíthatjuk. Egy autó egyednél például a rendszám tulajdonsághoz rendelhetjük a rendszámot azonosító mezőt, melyhez a magyar viszonyoknál maradva egy magyar-rendszámok domain tartozhat. E domain adattípusa olyan hatkarakteres sztringek halmaza lehet, melyben az első három karakter betű, az utolsó három pedig számjegy, azaz a formátuma "XXX999" alakú lesz. A sémában több mező is szerepelhet ugyanazzal a domainnel. Ennek előnye, hogy jobban érzékelhetők a mezők közötti kapcsola-



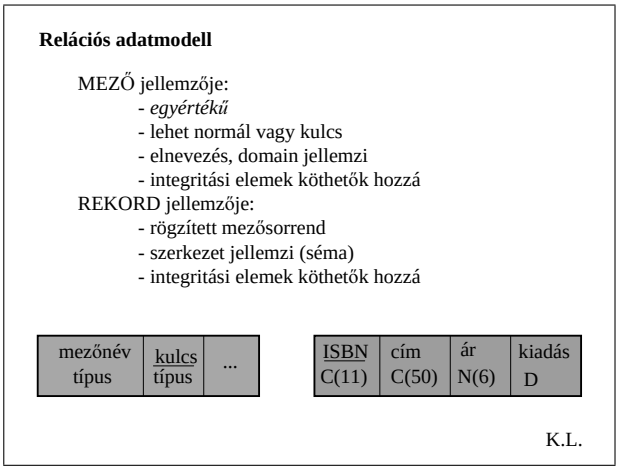
4.5. ábra. A domain fogalma

tok, másrészt hatékonyabbá teszi a karbantartást is, hiszen ekkor több mező helyett csak egyetlen egy domiannél kell az esetleges változtatásokat (például rendszám alakjának megváltoztatása) átvezetni. A mezők között kiemelkedő szerepet játszanak a rekordot meghatározó kulcs mezők.

A domainek jelentős szerepet játszanak az adatmodell szemantikai értelmezésében, hiszen a domain neveket a szemantikai tartalom figyelembe vételével hoztuk létre. Ha részletesebben elemezzük a szemantikai tartalom jelentőségét, akkor arra a következtetésre juthatunk, hogy a szemantikai tartalom felhasználható lehetne az adatmodell műveleti, operációs részében is. Hiszen például összehasonlítani, netán összeadni csak azonos értelemmel bíró mennyiségeket szabadna. E megkötést a modellünk úgy támogathatná, hogy csak az azonos domainhez tartozó mezők közötti összehasonlításokat, vagy összeadásokat engedné meg.

Másrészről a domainek létezése felhasználható lenne a domain-orientált műveletek elvégzésére is. Ez alatt azt értjük, hogy a rendszernek kiadhatnánk csak domain hivatkozásokat tartalmazó utasításokat is, mint például hogy az összes hosszúság adatot az adatbázisban szorozza meg 1.45-el, mert mértékegységváltás történt. A fenti szemantikai jellegű igények megvalósításával a legtöbb rendszer igencsak adós marad, így a domainek elsődleges szerepe ma még nagyrészt a szemléletesebb sémaleírásra korlátozódik.

A rekord egy összetartozó mezőcsoportot definiál. A rekord első közelítésben megfeleltethető az E/R modellbeli egyedeknek. Később látni fogjuk, hogy bizonyos esetekben az adatbázis hatékonysága, integritásának őrzése érdekében célszerű eltérni ettől az elvtől, és egy rekord nem feltétlenül egy egyedtípust fog reprezentálni. A rekordtípust a nevével és a hozzá tartozó mezők megadásával azonosíthatjuk. A mezők megadásánál megemlíthetjük, hogy a modell egyes formálisabb változataiban a rekordtípus a mező típusok halmazát, míg más, gyakorlatiasabb változata-



4.6. ábra. Mező, rekord jellemzői a relációs modellben

iban a mezőtípusok listáját jelenti. E kétfajta megközelítés alapvető különbsége, hogy egyikben sorrendiség áll fenn, míg a másikban, a halmazorientált leírásnál nincs semmiféle sorrendiség értelmezve a mezők között.

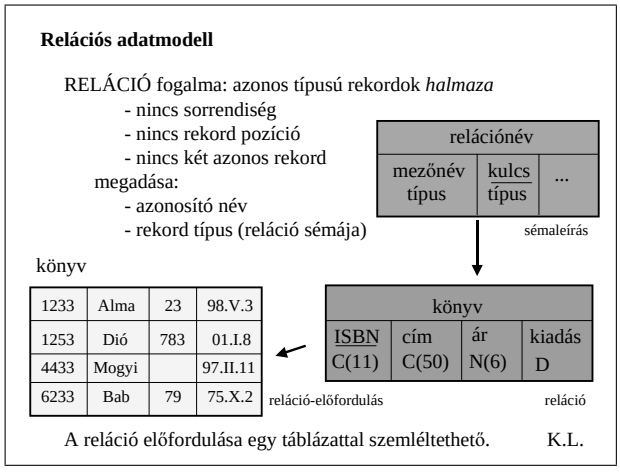
A *halmazorientált* módnál a mező nevével kell hivatkozni a mezőre, míg a listás megadásnál a mezőlistában elfoglalt pozíció jelöli ki a mezőt. Egy rekord előfordulás esetén a kétféle megközelítési, megadási mód az alábbiakban szemléltethető:

```
Listás mód :          INSERT INTO VEVO VALUES ('k1','Peter',25)
Halmazorientált mód:  INSERT INTO VEVO VALUES
                      (KOD='k1',NEV='Peter',KOR=25)
```

A relációs adatmodell legsajátosabb központi eleme a *reláció*. A reláció nem más, mint az azonos típusú rekord előfordulások összessége. Formálisabb megközelítésben, az igazi relációs adatmodellben ez az összesség halmazt jelent, vagyis a reláció az azonos típusú rekord előfordulások halmaza, és ilyenkor nem tehetünk különbséget a rekord előfordulások között a reláción belüli pozíció alapján, hiszen nem is értelmezhető pozíció a halmazelemek között.

Egyes megvalósított adatbáziskezelő rendszerek viszont listaként értelmezik a relációt, tehát feltételeznek valamilyen sorrendiséget a rekord előfordulások között. Ennek a szemléletesebb leírásnak köszönhetően a relációt egy táblázattal szokás reprezentálni. A halmazorientált szemlélet elvontabb, absztraktabb, logikai megközelítést tükröz, mely tudatosan szakít a fizikai nézettel. Viszont tudjuk, hogy számítógépeinkben az adatok tárolása rendezetten történik, az adatot megint a címe azonosítja, tehát a pozíció fizikailag elválaszthatatlan az adattól. A logikai és fizikai nézet különbségét emeli ki a halmazokon alapuló megközelítés.

A relációban szereplő egyed előfordulásokat a szakirodalomban *tuple*-nek nevezik, mi magyarul a sor vagy rekord kifejezést fogjuk használni. A relációra pedig alkalmazható a tábla kifejezés is. A következő táblázat egy, az autók adatait tar-



4.7. ábra. A reláció fogalma

talmazó relációt mutat be. A példában a reláció azonosító neve AUTÓK. E tábla célja a modellezett problémakörben előforduló autók adatainak a tárolása. Minden autóról a rendszám, a típus, a kor és az ár adatokat tartja nyilván. Az egyes mezők karakter és numerikus értékeket vehetnek fel.

rendszám	típus	kor	ár
FDT563	FIAT	6	821
AGY727	LADA	8	332
DVA824	SUZUKI	3	869
DGT763	LADA	4	766
AER772	SKODA	14	121
...			
GTR524	OPEL	6	753

A mezők és a domainek együttesen megadják a reláció struktúráját. Tehát az AUTÓK reláció struktúrája a következő:

(rendszám char(6), típus char(20), kor number, ár number)

A sorok a rekord előfordulásokat, az oszlopok a mezőket, a tulajdonságokat reprezentálják. A tábla, reláció szerkezetét egyértelműen meghatározza a benne foglalt rekordtípus, hiszen egy reláció csak egy rekordtípusbeli előfordulásokat tárolhat. A relációban tárolt rekordelőfordulások száma viszont tetszőlegesen sok lehet. Mivel a rekordtípus és a reláció nagyon szorosan, szinte elválaszthatatlanul összefonódik, ezért a relációs modellben rendszerint nem is említenek külön rekordtípust. A *relációtípus*, a reláció séma megadása a reláció nevével és a mezők felsorolásával történik.

A relációt a relációs modell kereteiben rekord előfordulások *halmazaként* értelmezzük, és a halmazjelleg kiemelésére röviden összefoglaljuk az elméleti relációk tulajdonságait:

- a relációban ugyanaz a rekord előfordulás nem fordulhat elő többször; egy halmaz egy elemét ugyanis csak egyszer tartalmazhatja, és ha két rekord minden mezője megegyezik egymással, akkor a két rekord ugyanazt az előfordulást jelenti, tehát csak egyszer fordulhat elő;
- minden mező atomi értéket tartalmaz;
- a reláció elemei között semmiféle rendezettség, sorrend nem értelmezett (gyakorlatban nem mindig teljesül);
- a mezők között semmiféle rendezettség, sorrend nem értelmezett (gyakorlatban nem mindig teljesül).

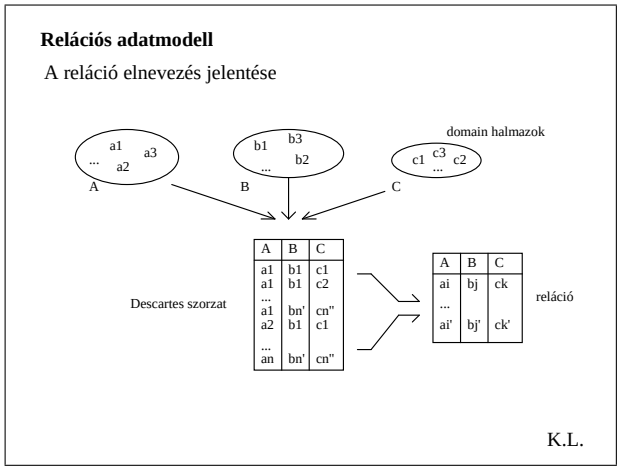
A reláció szó eredetileg kapcsolatot jelent, és a matematikában a reláció alatt több alaphalmaz Descartes-szorzatának egy részhalmazát értik. Az elnevezés ezen adatmodellben történő használatának jogosságához gondoljunk arra, hogy minden tábla a hozzá tartozó rekordtípus elméletileg lehetséges előfordulásainak egy részhalmazát tartalmazza, ahol minden rekord előfordulás a mezőkhöz tartozó domain halmazokból tartalmaz egy-egy elemet az egyes mezőiben, így egy rekord előfordulás nem más, mint a domain halmazok Descartes-szorzatának egy eleme.

Így maga a tábla, vagy reláció nem más, mint a domain halmazok Descartes-szorzatának egy részhalmaza, tehát az adatbázis reláció matematikai értelmében is megfelel a matematikai reláció definíciójának.

A relációk minőségi jellemzésére a reláció szerkezetét adjuk meg. A relációséma önmagában viszont csak egy üres váz, mely akkor nyer értelmet, ha feltöltődik adatokkal. A relációkat ekkor már mennyiségi oldalukról is vizsgálhatjuk. A *reláció fokszáma* alatt a relációsémához tartozó mezők darabszámát értik, míg a relációhoz tartozó rekordok darabszáma a *reláció számosságát* adja meg.

A relációk azonban nemcsak szerkezetükben és előfordulásaik méretében különböznek egymástól, hanem létezésük módjában is. Vannak ugyanis olyan relációk, melyek előfordulása és sémája az adatbázisban rögzítve van. Ezek a relációk a *bázis relációk*. A bázis relációk sémaleírása rögzített az adatbázis metaadatai között, illetve az előfordulásának adatai a fizikai adatbázisban is letárolásra kerültek. Mint majd látni fogjuk, a relációs modellben értelmezett műveletek relációkon értelmezettek és egy újabb relációt adnak eredményül. Az eredményreláció viszont rendszerint nem része az induló adatmodellnek, hiszen csak időlegesen van rá szükség, illetve letárolása felesleges, hiszen a benne tárolt adatok más relációkban megtalálhatók. Ezért az így keletkezett eredményrelációkat *ideiglenes eredmény relációknak* nevezik. Ha magát az előfordulást is megőrizzük, akkor a leszármaztatott relációt *snapshot*-nak nevezik. A snapshot rendszerint csak olvasható, tehát nem módosítható relációt jelent, és elő lehet írni, hogy tartalma milyen időközönként aktualizálódjon, ahol ez az aktualizálás automatikusan végbemegy.

A relációs modellben tehát nemcsak sémadefiniálással, hanem műveletsorral is lehet új relációt létrehozni. Ezen relációk rendszerint nem is kapnak külön nevet, sémájuk és előfordulásuk adatai sem tárolódnak a rendszerben. Az olyan relációkat, melyek megadása nem sémájuk leírásával, hanem már létező reláció-



4.8. ábra. A reláció elnevezés jelentése

kon értelmezett műveletsorral történik, *leszármaztatott relációknak* nevezik. Tulajdonképpen az ideiglenes eredményrelációk is ilyen leszármaztatott relációnak tekinthetők.

Viszont vannak olyan relációk is, melyek definíciós műveletsorát a metaadatok között megőrizzük, de magát az előfordulását nem tároljuk az adatbázisban. Az ilyen relációt nevezik *view*-nak. A felhasználó ugyanúgy kezelheti a view-kat is, mint a bázis relációkat, ezáltal a felhasználó elől rejtve marad, hogy voltaképpen a view nézeteken vagy a bázis relációkon dolgozik-e, tehát a felhasználó csak egyféle relációtípust lát maga előtt.

A bázistábla, view és snapshot viszonyának szemléltetésére vegyünk egy példát. Tegyük fel, hogy egy vállalat dolgozóiról kívánunk információt nyilvántartani, mely során a teljes lista mellett gyakran szükségünk lesz az 50 év feletti dolgozók listájára, valamint az éppen szabadságon lévő dolgozók listájára.

A tervezés során egyik döntési változat az, hogy mindhárom táblát megvalósítjuk. A probléma ekkor az, hogy ezen táblák adatai nem függetlenek egymástól, hiszen minden 50 év feletti dolgozónak is benne kell lennie a teljes dolgozói táblában és ott az 50 évnél nagyobb korral kell szerepelnie. Hasonlóan látható, hogy minden szabadságon lévő dolgozónak is szerepelnie kell a teljes táblában. A függőség azonban azt jelenti, hogy az egyik változása maga után vonja a másik tábla változását.

A három alaptábla helyett tehát jobb megoldás, ha csak azt tároljuk alaptáblában, ami feltétlenül szükséges, amiből a többi adat meghatározható, levezethető. A három listából a teljes lista szerepel ilyen bázisadatként, míg a másik kettő származtatott adat. A származtatott adatokat tehát nem érdemes külön alaptáblába tenni. A származtatott táblák tárolására kétféle megoldás kínálkozik: a view és a snapshot.

A view esetében nem hozunk létre külön táblát az adatbázisban, azaz nincs extra helyfoglalás. A view nem más mint a származtatási műveletsor, azaz azon műveletsor, amely megadja, hogy az alaptáblából milyen lépéseken keresztül hozható létre a kívánt lista. A view-ra történő hivatkozáskor a rendszer végrehajtja a kijelölt műveletsort, tehát előállítja az igényelt táblát.

A snapshot esetében viszont előáll induláskor a lista és meg is őrződik az adatbázisban, mint egy csak olvasható tábla. Mivel az alaptábla időközben megváltozhat, a snapshotban tárolt lista elavulhat, vagyis a snapshot már rég nem azt a listát tartalmazza, amit az aktuális alaptábla szerint mutatnia kellene. Ezért a snapshot-ok esetében időnként frissítést szoktak végezni, ami azt jelenti, hogy megadott időközönként a rendszer az aktuális alaptábla alapján újra elvégzi a lista előállítását és az eredménnyel helyettesíti a korábbi snapshot táblát.

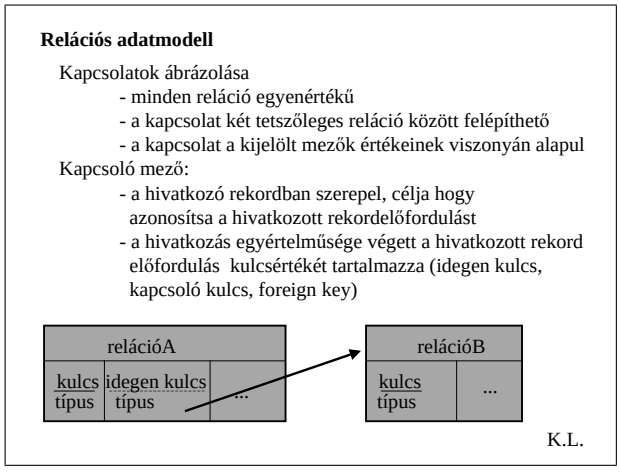
Mind a view-nak, mind a snapshot-nak vannak bizonyos hátrányai. A view hátránya, hogy hivatkozáskor a műveletsort végre kell hajtani, így a kiadott parancs végrehajtási idejét növeli. A snapshot hátránya viszont, hogy helyet foglal és emellett a frissítések is lekötik a rendszer idejét. Így mérlegelnünk kell, hogy mely megoldást is válasszuk. A view akkor célszerű, ha viszonylag ritkán van a

listára szükség és gyakran változik a lista értéke. A snapshot a gyakran igényelt, viszonylag ritkábban változó listák esetében lesz megfelelő megoldás. A példánkban így a szabadságon lévőket inkább view-nak, az 50 év felettiek adatait pedig snapshotban tárolhatjuk.

A relációk után maga az *adatbázis* jelenti a következő strukturális szintet. A relációs adatbázis szerkezetileg a logikailag összetartozó relációk összességét jelenti, ahol az összesség itt halmazként értelmezhető, azaz a relációk egy halmazát tekinthetjük adatbázisnak. Pontosabban fogalmazva a relációk halmaza az adatbázis szerkezetét definiálja.

A relációs adatbázisséma a relációs sémák egy halmazát jelenti. Az adatbázis séma egy előfordulása pedig a relációsémák előfordulásainak egy halmaza lesz. Az adatbázisséma megadása a nevével és a szerkezetével történik. Mivel a relációs modellben nem kell az adatbázis sémához fixen rögzíteni az oda tartozó relációk körét, az adatbázis séma tetszőlegesen bármikor módosítható, ezért egy új adatbázis séma létrehozásakor elegendő csak a séma azonosító nevét megadni. Tehát egy üres sémát hozunk létre, mely a későbbiekben bármikor dinamikusan változtatható. Ez úgy történik, hogy előbb kijelöljük az adatbázissémát, majd az ezt követő reláció séma létrehozások és törlések mind erre az adatbázissémára fognak vonatkozni.

Amint látható, a relációs adatmodell szerkezeti elemeiből hiányoznak a kapcsolatok leírására, tárolására vonatkozó elemek. Nincs olyan kapcsolatot rögzítő modellszerkezet, mint a set vagy a PCR volt az előzőekben ismertetett adatbázismodellekben. A relációs modellben a kapcsolatok nem szerkezeteken keresztül valósulnak meg, hanem adatokon keresztül, pontosabban adatértékeken keresztül. Ez azt jelenti, hogy ha valamely két egyedtípus között kapcsolat áll fenn, akkor a megfelelő rekordelőfordulásokban is léteznie kell olyan mezőknek, melyekben a



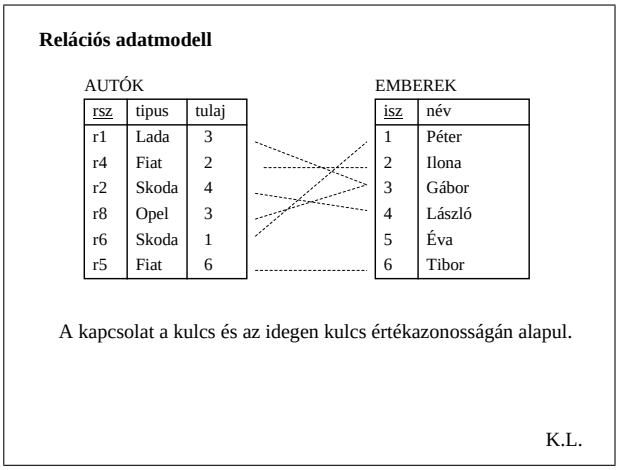
4.9. ábra. Kapcsolatok ábrázolása a relációs modellben

tárolt értékek utalnak a kapcsolatra. Ha például egy autó és egy ember közötti kapcsolatot, mondjuk a tulajdonosi kapcsolatot szeretnénk tárolni, akkor az autó rekordba teszünk be egy tulajdonos mezőt, melynek értéke a tulajdonost azonosítja.

A példában az autók és az emberek közötti tulajdonosi kapcsolatot a *mezők értékegyezősége* alapján tartjuk nyilván. Az autók táblában szerepel egy tulaj mező, amely a tulajdonos ember azonosító kódját tartalmazza. Az emberek táblában az 'isz' (igazolványszám) mező kulcsmező szerepet tölt be. Az ember egyed kijelölésére így ezen mező használható, ezért az autók tábla tulaj mezője is a megfelelő ember, vagyis a tulajdonos ember 'isz' tulajdonságának értékét tartalmazza. Mivel két azonos 'isz' nem fordulhat elő az emberek táblában, így a tulaj mezőben megadott hivatkozás egyértelmű.

A kapcsolat kijelölésére szolgáló mezőt kapcsoló mezőnek vagy *idegen kulcs* mezőnek nevezik. Az idegen kulcs elnevezés arra utal, hogy e mező értéke a hivatkozott táblában levő kulcs érték. Emiatt a tartalmazó táblában ez az érték egy idegen tábla kulcsa. Egy relációban több mező is lehet idegen kulcs, de minden idegen kulcs mező továbbra is csak egyetlen egy értéket tartalmazhat, tehát csak egyetlen rekord előfordulást jelölhet ki. A kijelölt rekord előfordulás tartozhat ugyanazon vagy más rekordtípusba is, mint a hivatkozó rekord. Egy dolgozó reláció rekord mutathat egy másik dolgozó rekordra, például egy főnök-beosztott kapcsolatban.

Általánosságban a kapcsolat jellegétől függ, hogy melyik rekordba tesszük be a kapcsolat leírására szolgáló mezőket. Mivel itt nincs rögzített kapcsolatszerkezet, azaz az adatok értékei hordozzák a kapcsolatot, ez sokkal nagyobb rugalmasságot jelent, hiszen elvileg bármely két mezőt formálisan adatkapcsolatba hozhatunk egymással. Itt is vannak azonban bizonyos korlátok, melyek elsősorban abból erednek, hogy egy mezőben csak egy elemi érték tárolható. Emiatt egy rekord kapcsolatait



4.10. ábra. Példa kapcsolódó táblákra

is csak szűkebben értelmezhetjük, és ha például a kapcsolat az adatok egyezőségén alapszik, ami a leggyakoribb eset, akkor egy rekord csak egy másikkal tud kapcsolatban állni, ami már jelzi, hogy lesznek bizonyos nehézségeink az M:N jellegű kapcsolatok ábrázolásánál.

A korábbi modellektől eltérően a relációs modellben a rekordok kapcsolódása, hivatkozása nem elérési korlátozás, hanem *érték korlátozás*, melynek semmilyen hatása sincs az elérési mechanizmusra. Gondoljunk vissza mondjuk a hálós adatmodellre. Vegyük példaként azt az esetet, amikor az A, B és C rekordtípusokhoz egy set-et definiálunk, amelyben az A a tulajdonos és B, C a tagok. Ekkor a rekordok lekérdezésénél a navigáció csak az A és B, illetve A és C között lehetséges közvetlenül. Az A, B és C rekord előfordulásokat csak az A-n keresztül lehet összekötni. Ezért a hálós modellben a kapcsolat elérési korlátozásnak is tekinthető.

Ezzel szemben a relációs modellben a kapcsolódás érvényessége azt jelenti, hogy a kapcsoló mező egy létező rekord kulcs értékét tartalmazza. Tehát egy érték megkötést határoz meg a kapcsolat. A rekordok lekérdezésénél a rekordok elérése, összekapcsolása viszont a definiált kapcsolókulcs-kulcs pároktól függetlenül hajtható végre. A relációs modellben bármely két rekordtípus rekordjai illeszthetők lesznek a lekérdezés során, tehát a kapcsolatnak nincs hatása a rekordok elérési sorrendjére, menetére.

Összefoglalva a strukturális rész leírását megállapíthatjuk, hogy az adatbázis relációk összessége, ahol egy reláció azonos felépítésű rekordok halmazát jelenti. Az adatbázis szerkezetének megadásakor az adatbázisba tartozó relációk azonosító nevét és szerkezetüket kell felsorolni. Egy reláció szerkezetét a hozzá tartozó mezők nevének és típusainak (domain), valamint integritási megkötéseinek a felsorolásával lehet megadni. A relációs modellben nem létezik külön kapcsolatleíró struktúra elem, mivel a rekordok közötti kapcsolatok a mezők értékein keresztül valósíthatók meg.

4.3. A relációs adatmodell integritási komponense

A relációs modellhez kapcsolódik néhány olyan alapfogalom is, amely nem közvetlenül az adatszerkezethez kapcsolódik, hanem az adatbázis integritásához. Az integritásőrzés fontosságát, egyenrangúságát mutatja, hogy Codd is külön önálló fejezetben tárgyalta az integritásőrzés problémáját.

Az integritási szabályok célja az adatbázis előfordulások lehetséges, megengedett körének behatárolása. Az *integritási szabályok* tehát az adatbázisban lévő adat előfordulásokra adnak megszorításokat. Az integritási szabályokat aszerint csoportosíthatjuk, hogy milyen szinten fogalmazzák meg a megkötéseket. A relációs adatmodellben az alábbi négy szintet szokás megkülönböztetni:

- domain és mező szint,
- rekord szint,
- reláció szint és
- adatbázis szint.

A *domain és mező szinten* egy mezőre vonatkozó érték előfordulások körét lehet megadni. A megkötést vagy egy logikai kifejezéssel lehet megadni, amely minden lehetséges domain értékre igaz vagy hamis értéket ad vissza, vagy annak előírásával, hogy a mezőben tárolt érték nem lehet üres. Az adatbázisba csak olyan mezőértékek tárolhatók le, melyekre a kijelölt feltétel igaz értéket ad vissza. Az a megkötés például, hogy a rendszám első három karaktere nem lehet szám, domain szintű integritási feltételt jelent, hiszen az ellenőrzés csak egyetlen egy domaint érint, egy önálló mező ellenőrzési feltételt szab ki.

A *rekord szint* esetén egy teljes rekord elfogadhatóságát döntjük el. Az ellenőrzési feltételben a relációsémában szereplő mezők szerepelhetnek. Az integritási feltétel célja az egy rekordon belül egymáshoz kapcsolódó mezők értékeinek vizsgálata. Példaként vehetünk egy olyan megkötést egy minta autókereskedő nyilvántartási rendszerből, mely szerint a katalizátoros autóknál az A vagy B adókulcs alkalmazható. E feltétel ellenőrzéséhez elegendő egyetlen egy rekord előfordulást önmagában vizsgálni.

A *reláció szintű* ellenőrzéshez a teljes relációt, azaz több rekord előfordulást is át kell vizsgálni. Az a megkötés például, hogy egy adott mezőben ugyanaz az érték nem fordulhat elő többször a relációban, csak úgy ellenőrizhető, ha ismerjük a reláción belül tárolt összes mezőértéket. Így az egyediséget előíró feltételt reláció szintű integritási feltételnek tekintjük. A reláció szintű ellenőrzési feltétel megfogalmazódhat egy aggregációs értékre vonatkozó megkötésben is, például amikor előírjuk, hogy az autók átlagéletkora 12-nél több nem lehet.

Az *adatbázis szintű* megkötések esetében a feltétel több relációban szétszórtan elhelyezkedő mezőkre vonatkozik. Ekkor az ellenőrzéshez több reláció adatait is át kell olvasni. Erre példa az a megkötés, amikor előírjuk, hogy az autó típuskódjának szerepelnie kell a típusok reláció valamely rekordjának kód mezőjében.

Az integritási feltételek egy másik szempont szerinti csoportosításában az osztályozás úgy történik, hogy a feltétel az adatbázis egy konkrét állapotára vagy egy állapot átmenetére vonatkozik. Az *állapotra vonatkozó* feltételeknél egy konkrét adatbázis előfordulást vizsgálunk, függetlenül a korábbi vagy későbbi adatbázis állapotoktól. Az a feltétel például, hogy egy mező értéke nem lehet üres, kitöltetlen, állapot integritási feltételnek tekintendő. Ha a feltétel azonban az állapotok megváltozását érinti, akkor egy *állapotátmenet* integritási feltételt kapunk. Erre példa lehet az a megkötés, mely szerint a fizetés értéke nem nőhet 25 százaléknál jobban. Ezt a feltételt úgy lehet ellenőrizni, hogy módosításkor az adatbázis módosítás előtti és módosítás utáni állapotából a két fizetésértéket összehasonlítjuk, és megállapítjuk a változás mértékét.

Az egyes integritási feltételeknél az előzőek mellett még abban is különbséget szoktak tenni, hogy mikor kerül sor az integritási feltétel ellenőrzésére. Vannak ugyanis olyan ellenőrzési feltételek, mint például a domain szintű műveletek, melyekben az ellenőrzés rögtön, az adatelem módosulásakor bekövetkezik. Ez a *közvetlen végrehajtási mód* jellemző a legtöbb integritási feltételre. Vannak emellett

azonban olyan több elemet érintő, összetettebb integritási feltételek, melyek nem feltétlenül teljesülnek a végrehajtás minden pillanatában. Vagyis az adatbázis nincs minden időpontban konzisztens állapotban. Az adatbáziskezelésben ezért a tevékenységeket nagyobb csoportokra szokták bontani, mely egységek egyfajta konzisztencia egységet is jelentenek. Eszerint az adatbázisnak nem szükséges minden időpontban konzisztensnek lennie, elegendő ezen tevékenység-egységek végén biztosítani a megkívánt állapotot. Az ilyen integritási feltételek esetén az ellenőrzés csak egy későbbi időpontban hajtódik végre, ezért ezeket *késleltetett ellenőrzésű* integritási feltételeknek nevezik.

A következőkben a relációs modellben megfogalmazott alapvető integritási feltételeket foglaljuk össze.

Mint már ismert, az egyedek megkülönböztetése tulajdonság értékeik alapján történik, és a helyesen megtervezett adatmodellben minden egyedtípushoz léteznie kell olyan tulajdonságcsoportnak, mely egyértelműen meghatározza az egyed előfordulásait. Az ilyen mezőcsoportot nevezik *kulcsnak*.

A kulcs lehet *egyszerű*, ha egyetlen mezőből áll, például az autó esetén a rendszám egy egyszerű kulcsnak tekinthető, vagy lehet *összetett*, amikor a kulcs több mezőből épül fel. Egy tanfolyam egyértelmű azonosításához a címe mellett szükség van a szervező intézmény és az időpont megadására is. Ebben az esetben e három mező együttese alkotja a kulcsot. A kulcsok kiemelésére a relációk táblázatos megadásánál a kulcsmezőket aláhúzással jelölik meg.

A kulcs azonban megfogalmazható más megközelítésben is: úgy mint egy integritási feltétel. Ugyanis egy mezőcsoportot akkor tekintünk kulcsnak, ha minden rekord előfordulásnál különböző értékeket vesz fel, azaz nem fordul elő egy érték soha sem többször a relációban. Ez formálisan azt jelenti, hogy a reláció minden

Relációs adatmodell

Integritási elemek:

- domain szintű vagy mező szintű	CHECK feltétel	értékellenőrzés
	NOT NULL	nem maradhat üres
- rekord szintű	CHECK feltétel	értékellenőrzés
- reláció szintű	PRIMARY KEY	kulcs
	UNIQUE	egyediség
- adatbázis szintű	FOREIGN KEY	idegen kulcs
	ASSERTION feltétel	összetett érték-ellenőrzés

K.L.

4.11. ábra. Relációs integritási elemek

elemére:

ha t1 nem egyenlő t2, akkor t1[kulcs] nem egyenlő t2[kulcs]

szabály teljesül, ahol t1 és t2 a reláció két rekordját (tuple) jelöli.

Adott egyedtípus esetén előfordulhat, hogy több mezőt is kiválaszthatnánk kulcsként, mint például az autók esetén a rendszám mellett az alvázszám is egyértelműen azonosít egy autót. Ahhoz, hogy egy mezőcsoport kulcsként szerepelhessen, természetesen teljesítenie kell a fenti megkötést, és ebben az esetben *jelölt* (candidate) *kulcsnak* is nevezik. A gyakorlatban ekkor természetesen választanunk kell, hogy melyik jelölt legyen a tényleges kulcs, hiszen ettől függően kell az adatmodell többi elemét is megválasztani. A hatékonysági megfontolásokat követve rendszerint a legrövidebb jelölt kulcsot választják valódi kulcsnak, ugyanis ekkor lesz más relációk mérete is a legrövidebb, illetve az indexszerkezet is kevesebb helyet foglal el, és ezáltal gyorsabb lesz az elérés is.

A fenti megfontolások alapján a kulcsra vonatkozólag a következő megkötéseket szokták tenni:

- *egyediség*, azaz nem ismétlődik az értéke a táblában;
- *nem lehet üres,* vagyis minden rekordban létezzen értéke;
- *minimalitás*, vagyis nincs olyan valódi részhalmaza, mely kielégítené az előző két feltételt.

A kulcs mezőhöz kapcsolódó fenti megkötés egy igen fontos integritási szabály a relációs modellben. Ezt a szabályt *egyed-integritási* (entity-integrity) szabálynak nevezik. E szabály tehát azt mondja ki, hogy

minden relációban legyen egyedi értékű kulcs-mezőcsoport és a kulcs-mezőcsoport nem lehet - még részlegesen sem - üres, azaz NULL értékű.

A részlegesen üres kulcs akkor lehetséges, ha a kulcs nem elemi, hanem több mező együtteséből áll. E szabály lényege tehát, hogy minden alaptáblában a tárolt egyednek legyen egyedi azonosító kulcsa, mely egyetlen egy esetben sem lehet üres. Másképp megfogalmazva azt is mondhatjuk, hogy a relációs modellben csak olyan rekordokat tárolhatunk, amelyeknek van egyedi azonosító kulcsmezője, tehát minden rekordja, sora egyértelműen megkülönböztethető egymástól.

Az integritási feltételnek eleget tevő kulcs-mezőcsoportot, azaz a kiválasztott, a fenti feltételeknek megfelelő kulcsot szokás *elsődleges kulcsnak* is nevezni.

A kapcsolatok tárolásához tehát a kulcsot más relációkban is megismétlik, ezért többszörösen is kifejti a hatását a kulcs hosszának növelése. Mindezek már sejtetik a relációs modell azon tulajdonságát, hogy a kapcsolatok tárolására mesterséges mezőket kell befűzni a relációsémába, amelyek nem az egyed természetes jellemzőit, hanem kapcsolatának jellemzőit hordozzák. Tehát a kapcsolatokat nem a rekordokon kívül írjuk le, mint ahogy a hálós és hierarchikus modelleknél tettük, hanem a rekordon belül, a mezők kibővítésével jelöljük a kapcsolat létezését.

A kulcshoz hasonlóan a kapcsoló kulcshoz is rendelhető integritási feltétel, mely szerint a kapcsoló kulcsnak létező egyedre kell mutatnia. Tehát amíg kulcs esetén teljesülnie kellett a reláción belüli egyediségnek, addig a kapcsoló kulcs esetén léteznie kell olyan rekordnak a másik relációban, melynek kulcsértéke megegyezik a kapcsoló kulcs értékével. A kapcsoló kulcsnak tehát a következő feltételeket kell teljesítenie:

- értéke vagy üres,
- vagy pedig a hivatkozott tábla kulcsmezői között szerepel.

Ez a szabály a referential integrity, azaz a *hivatkozási integritási szabály*, mely szerint

> *minden kapcsoló kulcs mező értéke vagy üres vagy egy létező kulcsértékre mutat.*

A 4.10. ábrán két elsődleges kulcsot és egy kapcsoló (idegen) kulcsot láthatunk. Az EMBEREK tábla elsődleges kulcsa az 'isz' mező, míg az AUTÓK tábláé az 'rsz' (rendszám) mező. Az AUTÓK tábla tartalmaz egy kapcsoló kulcsot, ez a tulaj mező. Ezen mező az EMBEREK táblához kapcsolja az AUTÓK táblát, és minden nem üres értéke valamely 'isz' mező értékével egyezik meg.

A két integritási feltételt összehasonlítva látható, hogy az egyikben, méghozzá a kulcsmező feltételnél, csak egy tábla adatait kell átolvasni a feltétel ellenőrzéséhez, a másik, a kapcsoló kulcs esetén a másik tábla adatait is át kell nézni. Ezt a különbséget alapul véve az integritási feltételeket felbontjuk lokális és globális feltételekre.

A *lokális integritási feltételek* köre azon feltételeket foglalja magába, melyek egyetlen egy tábla adataira vonatkoznak. A kulcs integritási feltétel egy ilyen lokális feltételt jelent.

A *globális integritási feltétel* esetén a feltétel több táblából származó adatokat is érint. A kapcsoló kulcs feltétel ebbe a csoportba esik.

Az integritási feltételekben megadott követelmények ellenőrzését az adatbáziskezelő rendszer automatikusan elvégzi. Így például, ha kijelölöm a rendszámot az autó reláció kulcsának, akkor a rendszer az új rekordok beszúrásakor, illetve a rekordok módosításakor automatikusan figyeli, hogy nem szerepel-e már valamelyik rekordban kulcsként a most megadott érték. Ha nem, elvégzi a kijelölt műveletet, ha pedig igen, azaz megsérülne a megadott integritási szabály, akkor pedig hibajelzéssel megáll a művelet végrehajtása. Természetesen a tervező a valós világ változásait követve módosíthatja az adatbázisban letárolt integritási feltételeket is. Az adatbázisban tehát nemcsak a struktúrák, a bennük tárolt adatok, hanem az adatokra vonatkozó integritási szabályok is változhatnak.

A kulcs esetén az egyik megszokott integritási feltételelem az, hogy a mezőnek mindig hordoznia kell egy valós értéket. Ezt úgy is mondhatjuk, hogy a mező nem lehet üres, azaz kitöltetlen. Az *üres, kitöltetlen érték*, amit az adatbáziskezelésben a *NULL* szimbólummal jelölünk, egy önálló érték a relációs modellben, mégpedig

fontos szerepet játszó érték. A NULL érték nem azonos a 0 értékkel, mert ez utóbbi értékes, valós adat lehet. Az RDBMS-ek rendszerint külön jelzővel jelölik, ha egy mező még nem kapott értéket, tehát ha nincs benne letárolt adat. Ekkor mondjuk, hogy a mező a NULL értéket tartalmazza.

A hagyományos programozási nyelvek változói rendszerint nem tudják megkülönböztetni az értékes, felvitt adatértékeket, a szemét, a véletlenül az allokáláskor ott maradt értékektől.

A relációs modellben viszont lehetőség van a NULL értékek megkülönböztetésére, ellenőrzésére.

A hivatkozási integritási szabály szoros kapcsolatban áll a NULL értékkel hiszen mint láttuk, e feltétel megengedi, hogy a kapcsoló mező üres, azaz NULL értékű legyen. Ez azt jelenti, hogy az egyed nem kapcsolódik másik egyedhez. Ezzel kapcsolatban egy érdekes probléma, hogy több mezőből álló kapcsoló kulcs esetén miként értelmezzük a NULL értéket. Mivel az elsődleges kulcs esetén az összetett kulcsnál egyetlen egy tagmező sem lehet üres, azaz a NULL érték sehol sem fordulhat elő, ezért egy olyan kapcsoló kulcs, amely bizonyos mezőiben nem üres, másokban meg üres, nem mutathat egyetlen egy létező elsődleges kulcsra sem. Így ez a kapcsoló kulcs érvénytelen, értelmetlen hivatkozás. Emiatt a relációs modellben nem engedünk meg olyan kapcsoló kulcsokat, melyek bármely tagmezője üres lenne. Az integritási feltételben a megengedett NULL érték tehát azt jelenti, hogy a kapcsoló kulcs minden mezője NULL értékű. Tehát összetett idegen kulcs esetén vagy minden tagmező üres, vagy egyik sem üres.

A hivatkozási integritási feltétel a NULL értékek kezelése mellett az adatok módosításánál is nagyobb körültekintést igényel. Az elviekben megengedettnek vélt módosítások során is előfordulhatnak olyan állapotok, amikor a feltétel nem teljesül. Ha ugyanis egy hivatkozott elsődleges kulcsérték megváltozik, akkor a korábbi kapcsoló kulcsok nem létező értékekre fognak mutatni, ami nem megengedett állapot. Ezért ebben az esetben a leggyakoribb kerülőút az, amikor a kapcsoló kulcsok értékét előbb NULL értékre állítjuk, majd elvégezzük az elsődleges kulcs módosítását, és legvégül beállítjuk a kapcsoló kulcs új értékét.

A relációs modellben a fenti integritási feltételek mellett megvalósult még két, a mezők értékére vonatkozó feltétel. Az *értékellenőrzési feltétellel* megadhatunk egy olyan logikai kifejezést, mely minden alkalommal, amikor a mezőbe új értéket viszünk be kiértékelődik és ha a kifejezés igaz értéket vesz fel, akkor letárolásra kerül az érték, ha pedig hamis lett a kiértékelés eredménye, akkor nem engedi a rendszer letárolni az új értéket.

Míg az előző feltétel egy új elem az ER modellben megismert elemekhez képest, addig a másik, eddig még nem említett feltétel már az ER modellben is megtalálható volt. Ugyanis szerepel az ER elemek között a leszármaztatott tulajdonság fogalma, melynek elterjedtsége miatt a relációs modellbe is átkerült a fogalma, és a legtöbb rendszer támogatja is, hogy egy mezőbe más mezőkben tárolt értékek alapján kiszámított értéket tároljunk. Úgymond egy *virtuális mezőt* létrehozva, hiszen adatai nem kerülnek letárolásra az adatbázisban, csak a kiszámítási képlete. Bizonyos rendszerekben a leszármaztatott mezőket nem lehet közvetlenül a bázis relációkba elhelyezni, csak a leszármaztatott relációkban szerepelhetnek.

Összefoglalva tehát a következő adatintegritási elemek állnak a rendelkezésünkre a relációs modellben:

- elsődleges kulcs,
- kapcsoló kulcs,
- egy mezőérték nem ismétlődhet,
- nem maradhat a mező üres,
- mező értékellenőrzése,
- mező kiszámított értéket tárol.

4.4. A relációs struktúra és az integritási feltételek formális megadása

Az eddigiekben a relációs adatmodellt szövegesen, informálisan írtuk le. Ez a leírási mód igen könnyen érthető, könnyen megjegyezhető, azonban van egy hátránya, hogy a hétköznapi szavaink sok esetben többértelműek, másrészről a szöveges leírások viszonylag nagy terjedelműek. Tömörség és egyértelműség tekintetében a formális leírások előnyösebbek az informális leírásoknál.

A relációs modell e tekintetben sem vall szégyent. Ez a modell volt tulajdonképpen az első olyan adatbázismodell, amely formális, matematikai alapokon nyugszik. A formális alapok lehetővé tették a modell egzakt elemzését, a modell későbbi továbbfejlesztését. A következőkben röviden összefoglaljuk a relációs modell strukturális és integritási részének formális leírását.

4.4.1. Attribútum és domain értelmezése

Legyen adott egy **U** nem üres, véges halmaz, amit univerzumnak nevezünk. E halmaz foglalja magába a vizsgált problématerület azon fogalmait, melyek értékeit egy skalár értékkel meg lehet adni. Egy videó kölcsönző esetén az **U** halmaz olyan fogalmakat foglal magába, mint például a klubtag neve, kazetta címe, kölcsönzés ideje, stb. E fogalmak nagyságát mind egy-egy skalár értékkel, egy számmal vagy egy sztringgel lehet leírni.

Az **U** halmaz tulajdonság leíró elemeit attribútumoknak nevezzük: $A \in \mathbf{U}$. Az A attribútumok halmaza az **A**. Az attribútum tehát a tulajdonság fogalomhoz áll közel, példánkban mind a klubtag neve, a kazetta címe, a kölcsönzés ideje egy-egy attribútum lesz.

Az egyedek tulajdonságait értékekkel írjuk le. Jelöljük az összes érték halmazát a **V** szimbólummal, ekkor $\mathbf{V} \subseteq \mathbf{U}$ teljesül, hiszen az értékek is elemei az univerzumnak. A **V** halmaz az összes értéket magába foglalja, amit valamely attribútum felvehet. Az egyes attribútumokhoz viszont lerögzíthető, hogy milyen értékeket vehet fel, hiszen attribútumonként más és más lehet a felvehető értékek köre. Az attribútumhoz kapcsolódó értékhalmaz lesz a domain. Minden domain-hez az értékek egy részhalmaza rendelhető, ezért $D \in 2^{\mathbf{V}}$ is teljesül.

Feltesszük, hogy létezik egy $dom : \mathbf{A} \rightarrow 2^{\mathbf{V}}$ leképzés, melyben minden elem képpé válik, azaz minden $A \in \mathbf{A}$-hoz hozzárendelhető egy $D_A \in \mathbf{D}$. Az A-hoz hozzárendelt D_A-t a $dom(A)$ szimbólummal jelöljük és az A attribútum domain

halmazának nevezzük. A w $\in$ *dom*(A) egy, az A attribútum által felvehető értéket jelöl. Példánkban 90 $\in$ *dom*(film hossza) teljesül, hiszen a film hosszát egész számokkal, percben mérve adjuk meg, és ez az érték lehet 90 perc. Ezzel szemben 'Jani' $\notin$ *dom*(film hossza), hiszen a megadott attribútum értéke csak egész szám lehet.

4.4.2. Relációséma és reláció értelmezése

Az $\mathbf{R} \subseteq \mathbf{A}$ halmazt relációsémának nevezik. Az **R** tehát nem más, mint attribútumok egy véges halmaza. A relációséma a reláció szerkezetét írja le, megadva a relációhoz tartozó attribútumokat. A példánkban a {klubtag neve, életkor, lakcím, azonosító szám} halmaz egy reláció szerkezetét adja meg. A definícióból látható, hogy nem kötöttük ki a nem üres jelleget, illetve egy attribútum több különböző relációsémában is szerepelhet.

Egy **R** felett értelmezett r reláció alatt, amit az r(**R**) szimbólummal jelölünk, az **R** felett értelmezett t leképzések egy véges halmazát értjük. Pontosabban megfogalmazva, ha $\mathbf{R} = \{A_1,..., A_m\}$, akkor $r(\mathbf{R}) = \{\ t : \mathbf{R} \rightarrow \cup_m D_i \mid \forall i\text{: } t(A_i) \in dom(A_i)\ \}$.

A t szimbólum, melyet angolul *tuple*-nek neveznek, mi pedig rekordnak, sornak fordítottunk, egy leképzés, amely a relációséma minden attribútumához hozzárendel egy elemet az attribútum értékkészletéből.

A reláció, melyet mi egy táblázattal reprezentáltunk ebben az értelmezésben leképzések halmaza. A táblázat minden sora egy önálló leképzés. E felírási mód magában foglalja a relációk azon tulajdonságát, hogy két azonos felépítésű sor (illetve leképzés) nem szerepelhet. A halmazban ugyanis minden elem csak egyszer fordulhat elő.

Relációs adatmodell

A modell formális felírása

U	univerzum
A $\in$ **A** $\subseteq$ **U**	attribútumok
V $\subseteq$ **U**	értékek halmaza
D $\in 2^{\mathbf{V}}$	domain
dom : **A** $\Rightarrow 2^{\mathbf{V}}$	attribútumok hozzárendelése domainekhez
R $\subseteq$ **U**	reláció séma
r(**R**)	reláció R felett és
t	egy tuple, ahol
r(**R**) = { *t* : **R** $\Rightarrow$ **V** | $\forall$ A $\in$ **R** : *t*(A) $\in$ *dom*(A)}	
$\underline{\mathbf{R}}$ = {**R**}	reláció sémák halmaza

K.L.

4.12. ábra. A relációs modell formális felírása I.

Egy $\mathbf{R}$ sémához több különböző r($\mathbf{R}$) reláció is tartozhat, hiszen több különböző halmaz is képezhető a lehetséges leképzésekből. Az $\mathbf{R}$ séma felett értelmezhető r($\mathbf{R}$) relációk halmazát $\mathbf{REL}(\mathbf{R})$ szimbólummal jelöljük, melynek pontos jelentése tehát: $\mathbf{REL}(\mathbf{R}) = \{ \text{r} \mid \text{r}(\mathbf{R})\}$.

4.4.3. Lokális integritási feltételek értelmezése

Egy $b : \mathbf{REL}(\mathbf{R}) \rightarrow \{\text{igaz, hamis}\}$ leképzést az $\mathbf{R}$ séma feletti lokális integritási feltételnek nevezünk. A b leképzés minden lehetséges $\mathbf{R}$ feletti relációhoz hozzárendeli az igaz vagy a hamis logikai értéket. A b tehát minden r($\mathbf{R}$)-ről eldönti, hogy elfogadható-e, azaz megfelel-e a kritériumoknak, vagy sem. A b így megszűri a lehetséges relációkat, megkülönböztetve a feltételeknek eleget tévő, illetve a feltételeket nem teljesítő relációkat.

Az $\mathbf{R}$ felett értelmezett b integritási feltételek halmazát $\mathbf{B}$-vel jelöljük. A $\mathbf{B}$ tehát magába foglalja az összes megalkotott integritási feltételt, amely az $\mathbf{R}$ relációsémára vonatkozik.

Egy adott $\mathbf{R}$ relációsémát a hozzá tartozó integritási feltételek $\mathbf{B}$ halmazával együtt kibővített relációsémának nevezünk. A kibővített relációséma jele $\boldsymbol{R}$. Ekkor $\boldsymbol{R} = (\mathbf{R}, \mathbf{B})$ teljesül. Egy $\boldsymbol{R}$ feletti relációnak, melynek jele r($\boldsymbol{R}$), olyan relációt nevezünk, amely egyrészt reláció $\mathbf{R}$ felett, másrészt teljesít minden $\mathbf{B}$-beli integritási feltételt: $\text{r}(\boldsymbol{R}) = \{ \text{r} \mid \text{r}(\mathbf{R})$ és $\forall b \in \mathbf{B} : b(\text{r}) = \text{igaz} \}$. Tehát r($\boldsymbol{R}$) egy megengedhető relációt reprezentál.

4.4.4. Adatbázis séma és adatbázis értelmezése

A relációs sémák egy halmazát nevezzük adatbázis sémának, melynek jele $\mathbf{D}$. A $\mathbf{D} = \{\mathbf{R}_1,\ldots\ , \mathbf{R}_n\}$ séma az adatbázis szerkezetét adja meg. Az adatbázis szerkezete tehát a benne tárolt relációk szerkezeteinek összességével egyenlő. Ez is mutatja, hogy a relációs modellben nincsenek különálló kapcsolatleíró elemek.

Relációs adatmodell

A modell formális felírása

$\mathbf{REL}(\mathbf{R}) = \{\text{r} \mid \text{r}(\mathbf{R})\}$	R feletti relációk
$\mathbf{B} = \{\text{b}\}$	lokális integritási feltételek, ahol
$b : \mathbf{REL}(\mathbf{R}) \Rightarrow (0,1)$	
$\boldsymbol{R} = (\mathbf{R}, \mathbf{B})$	kiterjesztett relációséma és reláció
$\text{r}(\boldsymbol{R}) = \{\text{r} \mid \text{r}(\mathbf{R}) \wedge\ \forall \text{b} \in \mathbf{B}: \text{b}(\text{r}) = 1\ \}$	
$\mathbf{D} \subseteq \{\mathbf{R}\}$	adatbázis séma
$\text{d}(\mathbf{D}) = \{\text{r}(\mathbf{R}) \mid \mathbf{R} \in \mathbf{D}\}$	adatbázis
$\mathbf{DAT}(\mathbf{D}) = \{\text{d} \mid \text{d}(\mathbf{D})\}$	D feletti adatbázisok
$\mathbf{B'} = \{\text{b'}\}$	globális integritási feltételek, ahol
$b' : \mathbf{DAT}(\mathbf{D}) \Rightarrow (0,1)$	

K.L.

4.13. ábra. A relációs modell formális felírása II.

Az adatbázis egy adatbázis sémára idomuló adatrendszer, jele d. Az adatbázis formális megadása a következő:

$$\mathrm{d}(\mathbf{D}) = \{\mathrm{r}_1(\mathbf{R}_1),...,\mathrm{r}_n(\mathbf{R}_n)\} \text{ ahol } \mathbf{D} = \{\mathbf{R}_1,... \ , \mathbf{R}_n\}.$$

Az r ∈ d relációkat a d adatbázis alap- vagy bázis relációinak nevezzük.

Ha a **D** sémát nem az egyszerű relációsémákkal állítjuk fel, hanem a kiterjesztett sémákkal, akkor az előálló adatbázist lokális integritási feltételekkel kiterjesztett adatbázisnak nevezzük:

$$\mathrm{d}(\boldsymbol{D}) = \{\mathrm{r}_1(\boldsymbol{R}_1),...,\mathrm{r}_n(\boldsymbol{R}_n)\} \text{ ahol } \boldsymbol{D} = \{\boldsymbol{R}_1,... \ , \boldsymbol{R}_n\}.$$

4.4.5. Globális integritási feltételek értelmezése

A több relációra kiterjedő feltételekre, mint az adatbázison értelmezett függvényekre tekinthetünk. Egy $\gamma : \mathrm{d}(\boldsymbol{D}) \rightarrow \{\text{igaz, hamis}\}$ függvényt nevezünk globális integritási feltételnek. A γ leképzés minden lehetséges **D** feletti adatbázishoz, reláció halmazhoz hozzárendeli az igaz vagy a hamis logikai értéket. A γ tehát minden $\mathrm{d}(\boldsymbol{D})$-ről eldönti, hogy elfogadható-e, azaz megfelel-e a kritériumoknak vagy sem, így a γ megszűri a lehetséges adatbázisokat, megkülönböztetve a feltételeknek eleget tévő illetve a feltételeket nem teljesítő adatbázisokat.

A $\boldsymbol{D}$ felett értelmezett γ integritási feltételek halmazát Γ-val jelöljük. A Γ tehát magába foglalja az összes megalkotott integritási feltételt, amely a $\boldsymbol{D}$ adatbázissémára vonatkozik.

Adott $\boldsymbol{D}$ adatbázis séma és Γ globális integritási feltételhalmaz esetén a feltételnek megfelelő, $\boldsymbol{D}$ séma feletti adatbázisok halmazát a $\mathbf{DAT}(\boldsymbol{D})$ szimbólummal jelöljük.

Relációs adatmodell

Egyed integritási szabály: minden relációs sémában létezzen kulcs (a kulcs nem üres, egyedi és azonosító)

$b_K(r(\mathbf{R})) = 1$, ha $K \subseteq \mathbf{R} \wedge \ \forall t_1,t_2 \in r(\mathbf{R})$: $t_1 \neq t_2 \Rightarrow t_1(K) \neq t_2(K)$
0, különben

Hivatkozási integritási szabály: az idegen kulcs vagy üres, vagy létező kulcs értékre mutat

$b'_{X,Y}(r1(\mathbf{R1}),r2(\mathbf{R2})) =$
1, ha $X \subseteq \mathbf{R1}$, $Y \subseteq \mathbf{R2} \wedge b_Y(r(\mathbf{R2})) \wedge$
$\{t(X)|t \in r(\mathbf{R1})\} \subseteq \{t(Y)|t \in r(\mathbf{R2})\}$
0, különben

K.L.

4.14. ábra. Kulcs, kapcsoló kulcs formális felírása

4.4.6. Kulcs integritási feltétel jelentése

A kulcs attribútumok felfoghatók egyfajta lokális integritási feltételként, amely akkor ad vissza igaz értéket, ha a megadott attribútum-csoport teljesíti a kulcs feltétel előírt követelményeit. A kulcs feltétel a következő integritási függvény formátumban adható meg:

$$b_K(\mathrm{r}(\mathbf{R})) = \text{igaz, ha } K \subseteq \mathbf{R} \text{ és } \forall\ t_1,t_2 \in r(\mathbf{R}) : t_1 \neq t_2 \Rightarrow t_1(K) \neq t_2(K)$$
$$\text{különben hamis.}$$

E kifejezés azt mondja ki, hogy a kulcsnak a relációs séma részének kell lennie és két különböző rekord, tuple esetén a kulcsban különbözni kell egymástól a rekordoknak.

4.4.7. Idegen kulcs integritási feltétel értelmezése

Az idegen, kapcsoló kulcs is értelmezhető integritási feltételként, mégpedig globális integritási feltételként. A kapcsoló kulcs szerkezet kijelöléséhez két reláció egy-egy mezőcsoportját kell kijelölni, így ezeket, mint paramétereket meg kell adni az integritási függvénynél:

$$\gamma_{X,Y}(\mathrm{r1}(\mathbf{R1}),\mathrm{r2}(\mathbf{R2})) = \text{igaz, ha } Y \subseteq R2,\ X \subseteq \mathbf{R1} \text{ és } b_Y(\mathrm{r2}) = \text{igaz és}$$
$$\{t(X)|t\in \mathrm{r1}\} \subseteq \{t(Y)|t\in \mathrm{r2}\}$$
$$\text{különben hamis.}$$

A feltétel azt mondja ki, hogy az Y attribútum az r2($\mathbf{R2}$) relációban kulcs legyen és emellett az X attribútumértékek legyenek benne az Y kulcs által felvett értékekben.

A most ismertetett formalizmusra támaszkodva a későbbi fejezetek során, a műveleteket leíró és az adatbázis tervezésről szóló részekben, pontosabban megfogalmazhatók lesznek a relációs algebra további elméleti alapjai. A formális felírást majd a relációs algebra és kalkulus keretében fogjuk újra elővenni és felhasználni.

4.5. ER modell konvertálása relációs adatmodellre

Az adatbázis tervezése során az elemzés eredményeképpen összegyűjtött információkat szemantikai modell-leírásban szokás megadni, és ez a szemantikai modell lesz az alapja a tervezésnél előállított adatbázis modellnek. Az adatbázis tervezés egyik fontos lépése tehát a szemantikai modellnek adatbázis modellbe történő konvertálása.

Mivel elsődleges feladatunk a relációs adatmodell elsajátítása, és a relációs modellhez legszorosabban kapcsolódó szemantikai modellek használata, ideértve az E/R modellt és az EE/R modellt, ezért a továbbiakban az E/R és a relációs modell konverziójának kérdésére koncentrálunk.

A konverzió során az E/R modellelemeket relációs modellelemekkel kell megvalósítanunk. Az átalakítás nehézsége abban rejlik, hogy egyrészt bizonyos E/R

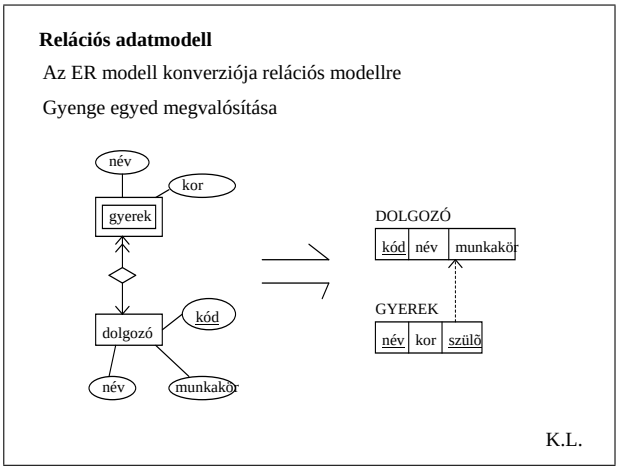
fogalmaknak nincs meg a relációs modellbeli megfelelője, másrészt míg az E/R modell szemantika orientált, addig a relációs modellben már más, hatékonysági, integritásőrzési szempontok is érvényesülnek. Elsőként azt nézzük meg, hogy az E/R modell egyes elemei hogyan képezhetők le a relációs modellre, majd később elemezzük a relációs modell hatékonyságát, jóságát.

A konverzió alapelve az, hogy *egy reláció egy egyedtípusnak* felel meg. A relációs adatmodell egy relációja tehát egy egyedtípus előfordulásait fogja tartalmazni. A reláció neve megegyezik az egyedtípus azonosító nevével. A reláció szerkezete is az E/R modellből jön. A reláció több mezőből épülhet fel hasonlóan ahhoz, ahogy az egyedtípus is a tulajdonságokból áll össze.

Tehát az első szabály a konverziónál az az irányelv, hogy az egyedekből készítsünk relációkat, és a hozzájuk kapcsolódó tulajdonságok legyenek a reláció attribútumai, mezői. A tulajdonság-mező megfeleltetés azonban nem alkalmazható automatikusan, hiszen a mező a relációs modellben csak atomi értéket hordozhat, tehát nem lehet összetett adatérték. Emiatt egy E/R összetett tulajdonság már nem valósítható meg egy relációs mezővel. A közvetlen leírás helyett más kerülőutat kell választani. Hasonló nehézségek fognak fellépni a kapcsolatok leírásánál is. A relációs modell a kulcs és kapcsoló kulcs segítségével tárolja a kapcsolatokat. Mivel a kapcsoló kulcs csak egy értéket tárol, ezért egy rekord előforduláshoz csak egyetlen egy egyed előfordulás köthető a kapcsolódó egyedtípusból. Tehát itt az N:M kapcsolatok tárolása fog nehézséget okozni. A következőkben szisztematikusan átnézzük az egyes EE/R elemeket és konverziós lehetőségeiket.

Egyed: megfeleltethető egy relációnak, a reláció neve megegyezik az egyedtípus azonosító nevével.

– *Normál egyed*: mivel van kulcstulajdonsága, ezért a tulajdonságok alkothatják a reláció mezőit, és nincs szükség kiegészítő mezőkre.

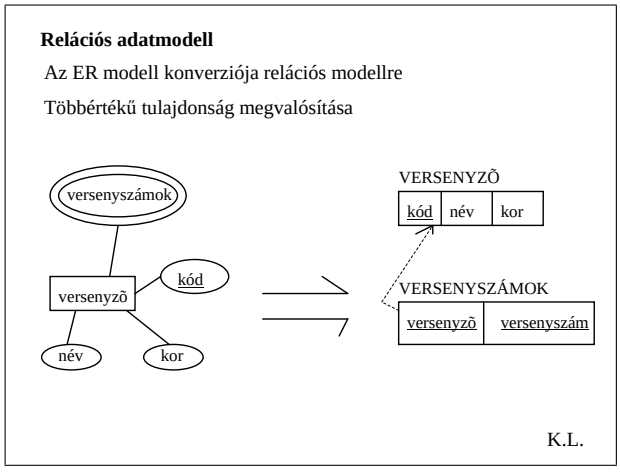


4.15. ábra. Gyenge egyed megvalósítása relációs modellben

– *Gyenge egyed*: mivel ezen egyedtípus tulajdonságai között nincs olyan, mely egyértelműen azonosíthatná az előfordulásokat, ezért a tulajdonságok önmagukban nem elegendőek a relációhoz, hiszen a relációs modellben minden relációnak kell hogy legyen kulcsértéke. A gyenge egyedet egy másik, normál egyedhez fűződő kapcsolatával azonosítjuk. Tehát úgy lehetne egy gyenge egyedhez tartozó relációban kulcsot létrehozni, hogy a mezők közé bevisszük a meghatározó egyedelőfordulás kulcsértékét. Ekkor ez a mező, együtt valamely más mezőkkel már használható kulcsként. A 4.15. ábra a dolgozókat és gyerekeiket ábrázolja. E modellben a gyerek egy gyenge egyed, mivel a dolgozó azonosítja őket egyértelműen. A példában a gyerek gyenge egyed, azonosítására a szülő és a saját név együttesen szükséges, így a gyerek reláció kulcsa összetett lesz. A szülő mező egyben kapcsoló kulcs, amely a dolgozó relációra mutat.

Tulajdonság: egy tulajdonság a reláció egy mezőjének feleltethető meg.

– *Egyszerű tulajdonság*: mivel az egyszerű tulajdonság elemi skalár értéket vehet fel, ezért megfeleltethető egy mezőnek. A mező neve megegyezik a tulajdonság azonosító nevével.
– *Kulcs tulajdonság*: a tulajdonság azonosító szerepét a relációs modellben az integritási feltételekkel lehet kijelölni. A relációs modellben létezik egy olyan integritási elem, mellyel kijelölhető, hogy melyik mezőcsoportot választottuk kulcsnak. A relációs séma megadásakor aláhúzással jelöljük ki a kulcs mezőcsoportot.
– *Összetett tulajdonság*: a relációs modell mezőszerkezete csak atomi értékeket tárolhat, amely nem bontható fel további részekre, ezért az összetett tulajdonság nem tárolható egyetlen mezőben. Az összetett tulajdonság leképezésére a legegyszerűbb mód, ha felbontjuk az összetett tulajdonságot alkotó elemeire, a felbontást addig végezve, amíg egyszerű



4.16. ábra. Többértékű tulajdonság megvalósítása relációs modellben

tulajdonságokat nem kapunk, és ezeket vesszük be a relációba mezőknek. Természetesen, az elnevezést esetleg módosítani kell, hiszen a relációban minden mezőnek egyedi elnevezést kell kapnia, míg a különböző összetett tulajdonságok esetleg tartalmazhatnak azonos nevű komponenseket.

– *Többértékű tulajdonság*: mivel a relációs modellben csak atomi értékeket tároló mezők létezhetnek, ezért a többértékű mezők közvetlenül nem tárolhatók a relációban. Mivel itt egy mezőhöz egy értékhalmaz tartozna, és a halmazszerkezet a relációkhoz áll közel, ezért az összetett tulajdonságok ábrázolásának szokásos módszere, hogy létrehozunk egy újabb relációt, melybe a tulajdonságértékeket tároljuk le. Mivel így több relációba szétkerülnek az egyedtípus tulajdonságai, ezért külön kell gondoskodnunk arról, hogy az összetartozó értékeket nyilvántartsuk, így ebbe az újonnan létrehozott relációba is bele kell tennünk az egyedtípus kulcsmezőjét, amelynek szerepe, hogy kijelölje melyik egyedelőforduláshoz tartoznak az egyes tulajdonságérték előfordulások. Itt találkozhattunk elsőként a relációs modellezés egyik sajátosságával; azzal, hogy a szemantikai tartalom megőrzésére a relációkat szétdaraboljuk több relációra.
A 4.16. ábrán szereplő példában a versenyszám, mint többértékű tulajdonság szerepel. Mivel többértékű mezőt nem támogat a relációs modell, így egy különálló relációt kell létrehoznunk, melyben ennek a tulajdonságnak az előfordulásai kerülnek majd letárolásra. Ahhoz, hogy tudjuk, hogy az egyes versenyszám értékek mely versenyzőhöz tartoznak, egy kapcsoló kulcsot is szerepeltetni kell az újonnan létrehozott relációban. E kapcsolókulcs a versenyző elnevezést kapta.

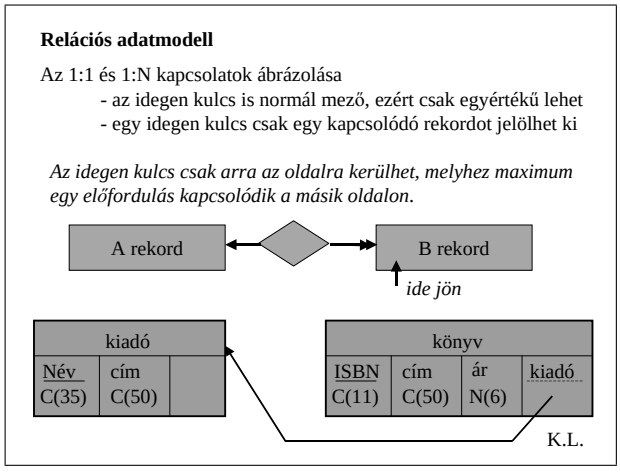
– *Leszármaztatott tulajdonság*: a tulajdonságot normál mezőként kell behozni a relációba, majd az integritási feltételekkel lehet kijelölni, hogy ez a mező nem közvetlen értéket tartalmaz, hanem más mezőkből származtatott értéket. A relációs modellben létezik tehát egy olyan integritási elem, mellyel kijelölhető, hogy melyik mező lesz leszármaztatott. A séma grafikus megjelenítésénél normál mezőként ábrázoljuk a származtatott mezőt is.

Kapcsolatok: a kapcsolatok jelzése a kulcs-kapcsolókulcs párosban tárolt értékek alapján történik.

– *1:1 kapcsolat*: mivel ennél a kapcsolattípusnál egy egyed előforduláshoz, azaz egy rekord előforduláshoz a másik egyedtípus egyetlen egy előfordulása kapcsolódhat, ezért rekord előfordulásonként elegendő egyetlen egy kapcsolóértéket tárolni. Tehát az 1:1 kapcsolat leképzése úgy történik, hogy az egyik rekordot kibővítjük egy új mezővel, mely a kapcsolókulcs szerepét fogja játszani. A mező domain-je megegyezik a másik rekordtípus kulcsának domain-jével, és értéke azon előfordulás kulcsértékét fogja tartalmazni, amellyel a kapcsolat fennáll. Az így megvalósított kapcsolatnak azonban van még egy szépséghibája, ugyanis a fenti szerkezetben semmi sem akadályozza meg, hogy a kapcsoló kulcsot tartalmazó relációban több rekord előfordulás is ugyanazt az értéket tartalmazza. Ennek megvalósulása egy 1:N kapcsolatot eredményezne, ami ellentmond a

szemantikai modellben felállított szerkezetnek. Az ellentmondást úgy lehetne megelőzni, hogy nem engedjük meg, hogy egy kapcsolókulcs érték többször is előforduljon. Szerencsére a relációs modell adatintegritási komponense rendelkezik ilyen eszközzel, ugyanis a mezőkhöz rendelhető egy érték egyediséget megszabó feltétel is, így ennek megadása után már nem fordulhat elő ugyanaz a kapcsoló kulcs érték kétszer.

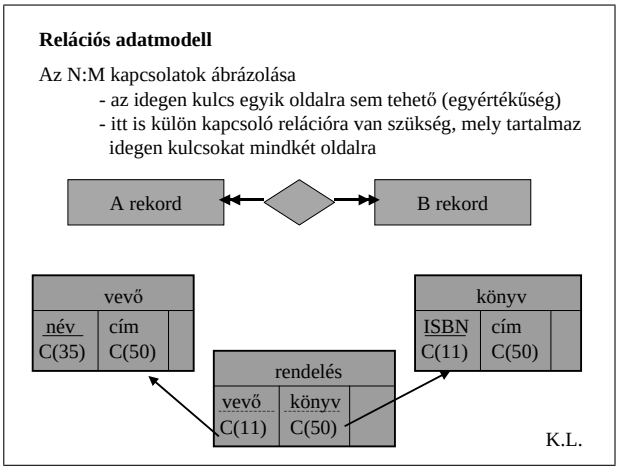
– *1:N kapcsolat*: mivel ennél a kapcsolattípusnál van egy olyan egyed előfordulás, mely a másik egyedtípus egyetlen egy előfordulásához kapcsolódhat, ezért rekord előfordulásonként elegendő egyetlen egy kapcsolóértéket tárolni. Tehát az 1:N kapcsolat leképzése is úgy történik, hogy az egyik rekordot kibővítjük egy új mezővel, mely a kapcsolókulcs szerepét fogja játszani. A mező domain-je megegyezik a másik rekordtípus kulcsának domain-jével, és értéke azon előfordulás kulcsértékét fogja tartalmazni, mellyel a kapcsolat fennáll. A két egyed közül abba kerül beépítésre a kapcsolókulcs, amelyik a másik egyednek maximum egy előfordulásával kapcsolódik. Ez a megkötés azért szükséges, mert a kapcsolókulcs mező, mint minden más mező, csak egyetlen egy értéket tárolhat. Mivel itt ugyanaz a kapcsolókulcs érték több rekord előfordulásban is szerepelhet egy reláción belül, ezért itt nincs szükség külön integritási feltétel definiálására.
  A 4.17. ábrán a könyv egyedhez kellett beépíteni a kapcsolókulcsot, hiszen egy könyvhöz csak egyetlen egy kiadó kapcsolódik. Azért nem lehet a kiadóhoz tenni a kapcsoló kulcsot, mert akkor a kapcsoló mezőnek több értéket is tárolnia kellene, hiszen egy kiadóhoz több könyv is tartozhat. A relációs modellben azonban egy mező csak egy elemi értéket tárolhat.
– *N:M kapcsolat*: mivel itt mindkét, a kapcsolatban előforduló egyedtípus előfordulásai több másik egyedtípusbeli előforduláshoz is kapcsolódhatnak, ezért mindkét kapcsolókulcsnak több értéket kellene felvennie, ami



4.17. ábra. 1:1 és 1:N kapcsolatok megvalósítása relációs modellben

nem megengedett. Így egyik sem tárolható a relációban mezőként. A probléma megoldása hasonlít a korábbi modellekben alkalmazott eljáráshoz, vagyis itt is létrehozunk egy új relációt, egy kapcsoló relációt, melynek minden rekord előfordulása egy konkrét kapcsolatot reprezentál. Ezen új reláció kulcsának a kapcsolatot kell azonosítania, erre a két kapcsolódó egyed előfordulás kulcsainak együttesét szokták használni. A kapcsoló reláció kulcsa tehát magában foglalja a két kapcsolókulcsot. Ez a megoldás lehetővé teszi, hogy a kapcsoló relációba mezőként bevegyük azokat a tulajdonságokat is, melyek magát a kapcsolatot jellemzik.
A 4.18. ábrán a kapcsolatot egy kapcsoló reláció létrehozásával lehetett realizálni a relációs modellen belül. A kapcsoló reláció, azaz a 'rendelés' reláció, kapcsoló kulcsként tartalmaz hivatkozást mindkét kapcsolódó relációra. A kapcsoló reláció minden rekordja egy konkrét vevő és könyv kapcsolatot tárol.
A tulajdonsággal ellátott kapcsolat bemutatására bővítsük a 4.18. ábrát úgy, hogy a rendelésnél a dátumot is tároljuk. Ez az új tulajdonság ekkor a kapcsolatot reprezentáló kapcsoló relációba fog bekerülni.

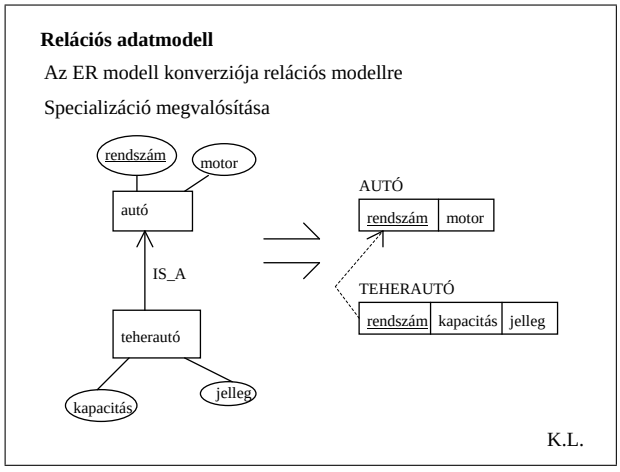
– *N-edfokú kapcsolat*: mivel a kapcsolat a kapcsolókulcs-kulcs párokon alapszik, ezért ez a megközelítés a kettes kapcsolatokon nyugszik. A magasabb fokú kapcsolatok leírására jó megközelítést ad, ha itt is bevezetünk egy kapcsoló relációt, melynek minden rekord előfordulása egy konkrét kapcsolatot reprezentál. Ezen új reláció kulcsának a kapcsolatot kell azonosítania, amire a kapcsolódó egyed előfordulások kulcsainak együttesét szokták használni. A kapcsoló reláció kulcsa tehát magában foglalja az n darab kapcsolókulcsot is. Ez a megoldás lehetővé teszi, hogy a kapcsoló relációba mezőként bevegyük azokat a tulajdonságokat is, melyek magát a kapcsolatot jellemzik.
– *Totális kapcsolat*: a kapcsolat totális voltának letárolása viszonylag egy-



4.18. ábra. N:M kapcsolat megvalósítása relációs modellben

szerűbb akkor, ha minden egyed előfordulás csak egy kapcsolat előfordulásban szerepel, tehát amikor az egyedhez tartozó reláció tartalmazza a kapcsolatra vonatkozó kapcsolókulcsot. Ekkor a kapcsolat totális jellege azt mondja ki, hogy egyetlen egy előfordulásban sem maradhat a kapcsolókulcs értéke NULL. Ez a megkötés előírható a megfelelő integritási feltétel kiadásával. Ha azonban egy olyan egyed kapcsolata totális, mely több másik egyed előfordulással is kapcsolatban áll, akkor a feltétel úgy szólna, hogy az adott reláció minden kulcsértékének elő kell fordulnia a másik relációban szereplő kapcsolókulcs értékek között. Ez viszont már egy összetettebb integritási feltétel, melyet csak bonyolultabb úton valósíthatunk meg, például triggerek segítségével. Még így is olyan problémákkal találkozhatunk, hogy miként kezeljük azt az esetet, amikor mindkét irányban totális a kapcsolat, és ekkor milyen lehet a rekordok felviteli sorrendje. Ha ugyanis előírjuk, hogy csak akkor vihetünk fel egy új rekordot, ha létezik kapcsolódó rekordja, akkor csak egyidejűleg vihetnénk fel a kapcsolódó rekordokat, amit már nehezebb lenne megoldani.

– *Specializáció, kategória*: az EE/R modellből származó két fogalom valójában igen távol áll az alap relációs modelltől. Hiszen ezek a fogalmak is kapcsolatokat írnak le, de nem egyed előfordulások közötti kapcsolatot, hanem típusok közötti kapcsolatot ábrázolnak. Ezért a kulcs-kapcsolókulcs szerkezet nem igazán illik erre a kapcsolatra, hiszen az kizárólag a rekord előfordulások közötti kapcsolatok leírására szolgál. Mivel a relációs modellben nincs megfelelő eszköz a típusok közötti kapcsolatok tárolására, ezért csak egy közelítést alkalmazhatunk a specializáció, kategória jelzésére a kulcs-kapcsolókulcs felhasználásával. Egyik lehetséges módszer az, hogy egy leszármaztatott egyedet két relációba szétbontva tárolunk. Ekkor külön relációba kerül az általános rész és

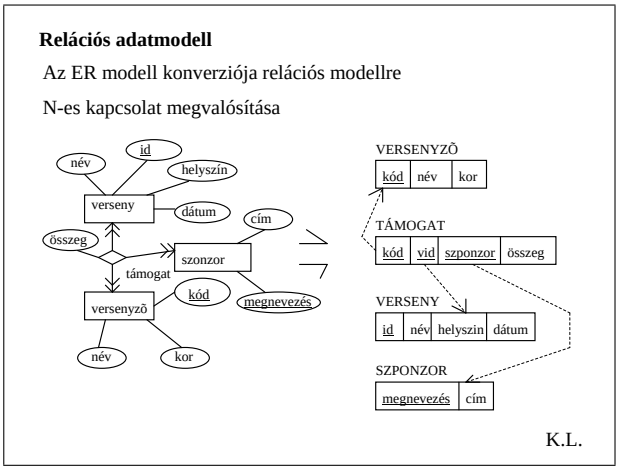


4.19. ábra. Specializáció megvalósítása relációs modellben

külön relációba a speciális rész. A két rész közötti kapcsolatot pedig egy kapcsoló kulcs segítségével valósítjuk meg. A specializált, leszármaztatott típushoz tehát egy olyan relációt hozunk létre, mely csak a speciális, egyedi tulajdonságok tárolására szolgáló mezőket tartalmazza, illetve emellett tartalmaz egy kapcsoló kulcsot a szülő relációhoz fűződő kapcsolat jelzésére is. Ekkor a specializált egyed összes tulajdonságának kiíratására több relációból kell a mezőket összegyűjteni.
A 4.19. ábrán a teherautó reláció csak a speciális tulajdonságokat tartalmazza a kapcsoló kulcs és elsődleges kulcs szerepet is játszó rendszám mező mellett. A teherautó általános adatai az autó relációban kerülnek letárolásra. Így például a teherautó motorjának adatait az autó tábla megfelelő rekordjából lehet kiolvasni.

– *Tartalmazás*: az EE/R modellben a specializáció mellett rendelkezésre állt egy tartalmazási kapcsolat is, amikor is az egyik egyed a másik egyed szerves alkotórészét képezi. Ebben az esetben is a szokásos idegenkulcs-kulcs páros áll rendelkezésre a kapcsolat nyilvántartására, hasonlóan a specializáció esetéhez. Mivel egy egység több részből állhat és egy rész csak egy egységhez kapcsolódik, a kapcsoló kulcsot a részt reprezentáló relációba kell elhelyezni.

Az E/R modell konverziója során létrejövő relációs sémában a strukturális elemek mellett szerepeltetni kell a hozzájuk kapcsolódó integritási elemeket is. A sémarajzon ezért mindig fel kell tüntetni a kulcs mezőket, az idegen kulcs mezőket a hivatkozott relációval együtt, az egyediséget, a nem üres megkötöttséget és az általános értékellenőrzési feltételt is. A megalkotott sémát nem szokták közvetlenül felhasználni, mert még egy ellenőrzési vizsgálatot elvégeznek rajta. Ez a folyamat a normalizálás, melyről a későbbiekben részletesebben fogunk szólni.



4.20. ábra. N-es kapcsolat megvalósítása relációs modellben

4.6. A relációs adatmodell műveleti része

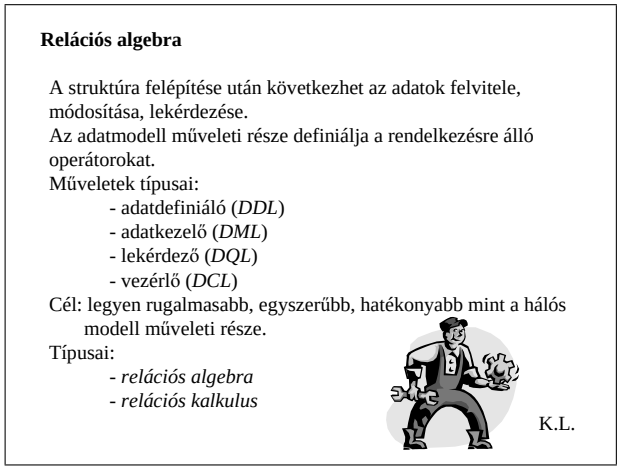
Az előzőekben megismerkedhettünk a relációs adatmodell strukturális és integritási komponenseivel. Ezek a komponensek vesznek részt a relációs modell felépítésében, e komponensekből áll össze a relációs adatstruktúra. Az ott megadott szabályok határozzák meg, hogy hogyan nézhet ki egy konkrét adatbázis. Mivel ezen komponensek az adatbázis bármely időpontbeli állapotaira vonatkoznak, tulajdonképpen időtől függetlenek, ezért statikus elemeknek nevezzük őket.

Az adatbázisok azonban nem holt, befagyott rendszerek, struktúrák. Az adatbázis értelme, hogy használják, hogy felhasználják. Eközben a különböző felhasználók módosíthatják és lekérdezhetik az adatbázis tartalmát. Az adatbázis akkor lesz tehát élő, ha csatlakozik hozzá egy olyan funkciócsoport is, amely lehetővé teszi az adatbázisban tárolt adatok módosítását és lekérdezését. Mivel ezek a komponensek az adatbázis változásaihoz, megváltoztatásához kapcsolódnak, ezért *dinamikus komponenseknek* nevezzük őket. A relációs adatmodell műveleti része ezen dinamikus adatkezelő és adatlekérdező lehetőségeket foglalja magába.

A relációs adatmodell műveleti része a relációkon alapul, azaz a műveletek mindegyike relációkon értelmezett. Ez azt jelenti, hogy a bemenő operandusai a relációk lehetnek. A lehetséges műveleteket számba véve a következő műveletköröket lehet kiemelni:

- adatok lekérdezése,
- adatok felvitele,
- adatok módosítása,
- adatok törlése.

A fenti négy műveletkörből az utóbbi hármat együttesen *adatkezelő* (data manipulation) utasításoknak nevezik. Az adatkezelő utasítások értelmezése viszonylag



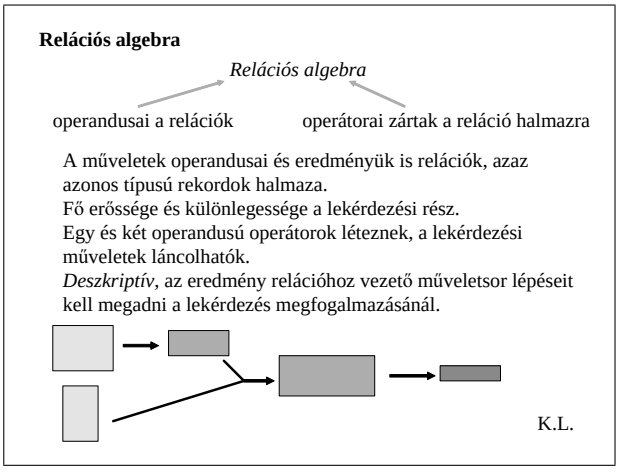
4.21. ábra. A relációs modell műveleti része

egyértelmű, hiszen mindenki sejti, mit is fog jelenteni, hogyan történik például az adatok felvitele. Ezzel szemben az *adatlekérdezés* egy sokkal szabadabb, tágabb értelmezést biztosít, hiszen itt elviekben nagyon sokfajta lehetőség kínálkozik arra, hogy mit és hogyan olvasok ki az adatbázisból, hogyan rakhatjuk össze az elemi információkat egy komplex kérdés esetén. Ezért az adatlekérdezési funkciót sokkal nagyobb terjedelemben fogjuk tárgyalni, mint az adatkezelő műveleteket. Mivel az adatkezelő műveleteknél bizonyos pontokon hivatkozni fogunk az adatlekérdezéshez kapcsolódó elemekre, így előbb az adatlekérdezés műveletét vesszük át.

A relációkból elméletileg igen sokféleképpen, sokféle mechanizmussal lehet adatokat kiolvasni. A gyakorlatban is több típusa létezik az adatlekérdezés mechanizmusának, habár az igazi relációs adatmodell kijelölt egyfajta kezelési technikát. A Codd által javasolt mechanizmus azonban sokkal erőforrásigényesebb, mint a hagyományos rekordorientált adatolvasási mechanizmusok. Ezen hatékonysági korlátok miatt tudott napjainkig is megmaradni ez az adatkezelési mód az egyszerűbb relációkat tároló adatrendszerekben.

A relációs adatmodellben a relációkból történő információ kinyerése is biztos elméleti alapokon nyugszik. A Codd által definiált adatlekérdező műveletcsoportot összefoglalóan *relációs algebrának* nevezik. A relációs algebra az alapja a ma már szabványként elfogadott és leginkább elterjed adatlekérdező relációs parancsnyelvnek, az SQL-nek is.

Nézzük, mit is jelent pontosan a relációs algebra kifejezése. Az algebra szó a matematikában azt a diszciplínát jelöli, ami egy halmazon értelmezett műveletek tulajdonságait vizsgálja. A mi esetünkben a műveletek a relációkon értelmezettek, így innen származik a relációs algebra kifejezése. A műveletek tehát relációkon értelmezettek és ami nagyon fontos és lényeges, hogy *relációkat is adnak eredményül.* Tehát egy lekérdezés eredménye egy újabb relációt szolgáltat, vagyis a relációs al-



4.22. ábra. A relációs algebra

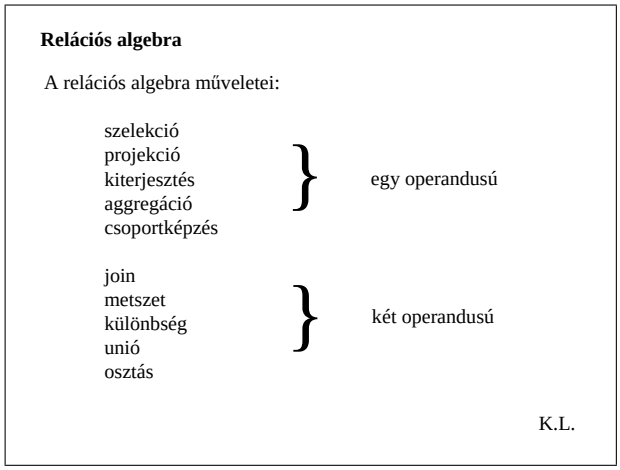
gebrai műveletek nem vezetnek ki a relációk halmazából, a kapott eredmény szintén az adatbázis részének tekinthető.

Ezen tulajdonság alapján szokták azt mondani, hogy a relációs algebrai műveletek zártak a relációk halmazára nézve, hiszen bárhogy is alkalmazzuk őket, újból egy adatbázisbeli relációt fogunk kapni. A zártság egy mellékkövetkezménye, hogy a rendszerben az egyes műveletek nagy szabadsággal ágyazhatók egymásba, hiszen egy eredményül kapott reláció szabadon felhasználható más műveleteknél, mint bemenő operandus. Ezáltal szinte korlátlan a lehetséges lekérdezési utasítások darabszáma, a felhasználók a rendelkezésre álló műveletelemeket szinte tetszőlegesen illeszthetik egymáshoz.

Mivel a relációkat halmazokként értelmeztük, ezért szokás a relációs algebrát is halmazorientáltnak nevezni. A műveleteket tehát a rekordok halmazán értelmezzük, azaz minden művelet a relációban lévő összes, feltételnek eleget tévő rekordra fog vonatkozni. A relációs algebra több különböző műveletet is értelmez a relációkra vonatkozóan. Ezek között vannak olyanok melyek egyparaméterűek és vannak kétparaméterű műveletek is. A piacon kapható RDBMS rendszerek zöme az SQL kezelő nyelven alapszik, mert ez a nyelv igen jól megvalósítja a most bemutatandó relációs algebrai műveleteket. Más kezelőfelületek, mint például az Xbase alapú rendszerek csak részben felelnek meg a halmazorientáltság követelményeinek, ott továbbra is a korábbi adatmodellekben megszokott rekordorientált műveletközelítést alkalmazzák.

4.6.1. A relációs algebra műveletei

Elsőként tekintsük át, milyen műveletek tartoznak a relációs algebra körébe, majd ezt követően részletezzük az egyes műveleteket. A relációs algebra a következő műveleteket tartalmazza:



4.23. ábra. A relációs algebra műveletei

– szelekció
– projekció
– join, egyesítés
– unió
– metszet
– különbség
– osztás
– kiterjesztés
– csoportosítás, aggregáció.

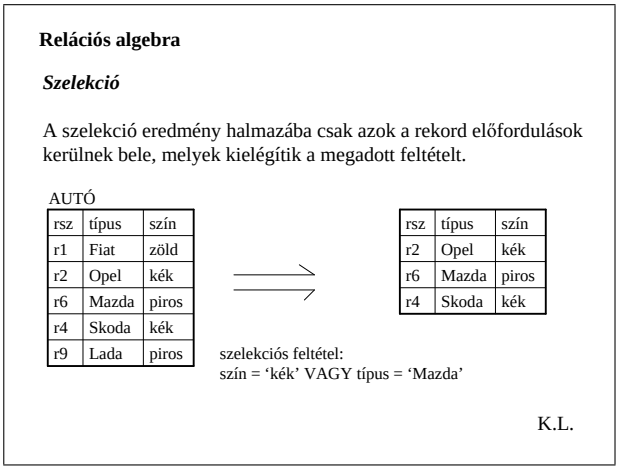
A megadott műveletek közül az első hét már Codd definíciójában is benne foglaltatott, míg az utolsó kettő csak később, a felhasználói igények alaposabb elemzése után került be a relációs algebrába.

A felsorolt lekérdező műveletek nem változtatják meg az operandusként megadott relációkat, csak olvassák azokat, majd az eredményt leteszik egy új, önálló eredményrelációba. A relációs algebra műveletei közül az első három játsza a legnagyobb szerepet, mivel ezek a műveletek fordulnak elő a leggyakrabban a lekérdezések során. Elsőként ezeket vesszük sorra.

A *szelekciós művelet* a relációban szereplő rekord előfordulások egy részhalmazának az előállítására szolgál. A művelet értelmezése:

> *A szelekció eredményhalmazába csak azok a rekordelőfordulások kerülnek bele, melyek kielégítik a megadott szelekciós feltételt.*

A szelekció elvégzéséhez tehát meg kell adni a relációt és a feltételt. A feltételnek olyannak kell lennie, hogy a reláció minden rekordjára kiértékelhető legyen, és logikai értéket adjon vissza. Azon rekordok, melyekre a feltétel igaz, bekerülnek az eredményrelációba. Azaz a szelekció a tábla bizonyos sorait adja eredményként.



4.24. ábra. A szelekció művelete

A 4.24. ábrán látható példában az AUTÓ táblából kiválasztjuk, szelektáljuk a kék színű vagy Mazda típusú autókat. Az eredményreláció tehát csak a kék színű vagy Mazda típusú autókat fogja tartalmazni.

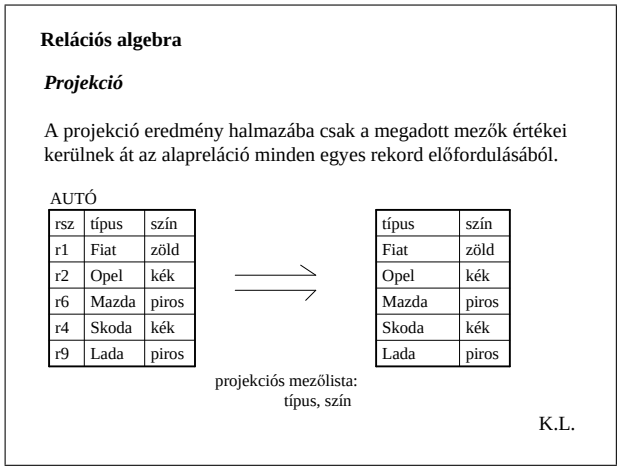
A szelekciós feltétel lehet egyszerű vagy összetett, melyben hivatkozhatunk mezőértékre és konstans adatelemekre is. Ha egy konstans értékkel hasonlítunk össze egy mezőt, akkor beszélünk *konstans szelekciós feltételről*, ha pedig két mezőt hasonlítunk össze, akkor *attribútum szelekciós feltételt* értékelünk ki.

Az előző példában szereplő *szín = 'kék'* feltétel konstans szelekciót ad meg, míg az emberek adatait tároló relációban az *életkor > 2*testsúly* feltétel két mező értékét hasonlítja össze, így ez attribútum szelekciót reprezentál.

Ha adott egy tanulókat tartalmazó reláció, akkor konstans szelekcióra fog épülni az a művelet, amely azon hallgatókat adja vissza egy eredménytáblába, melyek jeles eredményt értek el matematikából. A szelekciós feltétel itt az lesz, hogy a matematika jegy jeles értékű. Az attribútum szelekcióra lehet egy példa az, amikor azon hallgatókra vagyunk kíváncsiak, akiknek jobb jegye van matematikából, mint magyarból. Ekkor a szelekciós feltétel, hogy a matematika érdemjegy mező nagyobb értéket tartalmazzon, mint a magyar érdemjegy mező.

Az elemi feltételeket logikai operátorokkal összekötve *összetett szelekciós feltételt* kapunk. Erre lehet példa az a szelekció, amikor azon autók adatai kerülnek át az eredménytáblába, melyek vagy kék színűek vagy Mazda típusúak.

A felhasználható operátorok köréről majd az SQL nyelv tárgyalásakor beszélünk részletesebben. Annyi azonban előrebocsátható, hogy a szokásos operátorok (*AND, OR, NOT, <, >, =, +, -, *, /*) itt is alkalmazhatóak, azonban ezek köre ki fog bővülni néhány újabb, speciális reláció operátorral.



4.25. ábra. A projekció művelete

A *projekció* azt a műveletsort jelenti, amikor a relációban a rekord előfordulásokat leíró mezőkből csak bizonyos mezők értékeit kérdezzük le eredményként. Az eredményreláció csak a kijelölt mezőkre vonatkozó adatokat tartalmazza, viszont minden rekordelőfordulás szerepel az eredményrelációban. A projekció műveletének értelmezése:

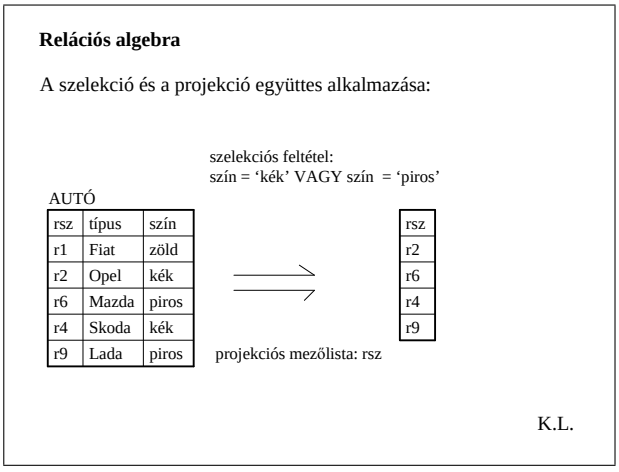
> *A projekció eredmény halmazába csak a megadott mezőértékek kerülnek át az alapreláció minden egyes rekordelőfordulásából.*

A projekció a tábla leszűkítését jelenti bizonyos oszlopaira. A projekció elvégzéséhez meg kell adni a kiinduló relációt és az eredménytáblába átkerülő mezők explicit felsorolását.

Projekciónak felel meg az a lekérdezési művelet, amikor az autók adatai közül csak az autó színére és típusára vagyunk kíváncsiak, azaz az eredményreláció csak a szín és a típus mezőt tartalmazza, viszont minden lehetséges autó rekordra vonatkozó értékpárost tartalmaznia kell az eredményrelációnak.

A projekcióval kapcsolatosan az a probléma merülhet fel, hogy mi történik akkor, ha a projekcióval kapott eredményrelációban egy rekordelőfordulás többször is előfordulna. A példánkban ez akkor következne be, ha létezne két azonos típusú és színű autó. Az elméleti relációs modell szerint egy relációban egy rekord előfordulás *nem szerepelhet kétszer*, ezért az eredménytáblának csak egyszer kell tartalmaznia minden eredményrekordot. Ezáltal az eredmény reláció számossága kisebb lehet, mint az induló reláció számossága.

A gyakorlati RDBMS rendszerek viszont rendszerint úgy implementálják a projekció műveletét, hogy minden előfordulást meghagynak az eredménytáblában, mivel ez a többszörösség hasznos információt jelenthet a felhasználónak, ezért külön kapcsoló áll rendelkezésre az egyes lekérdező nyelveknél a projekcióhoz, ha nem akarunk előfordulás többszörözést az eredményrelációban.



4.26. ábra. A szelekció és a projekció együttes alkalmazása

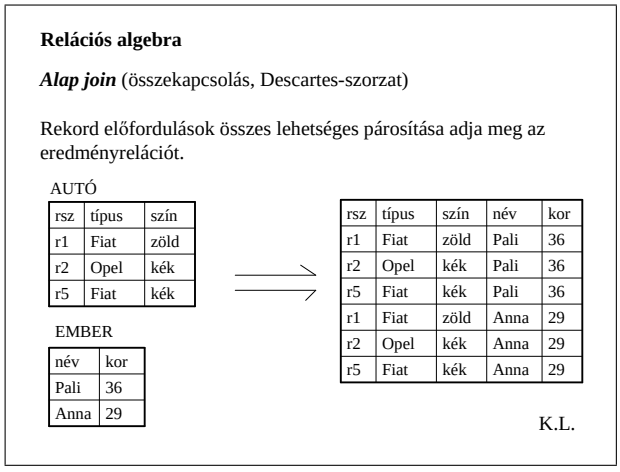
A szelekció és projekció együttesen is használható, például amikor azon autók rendszámára vagyunk kíváncsiak, amelyek színe piros vagy kék. Az eredményrelációt a 4.26. ábra mutatja.

Az *összekapcsolás (join)* művelete az előző műveletektől eltérően egy kétoperandusú operátor, azaz két relációból állít elő egy eredményrelációt. Az összekapcsolás a relációk Descartes-szorzatán alapszik, vagyis az eredménytáblában a két táblából vett rekord előfordulások minden lehetséges párosítása szerepel. Az egyesítés eredményeként kapott reláció mindkét reláció minden mezőjét tartalmazza. A join értelmezése:

> *Az alap join művelet eredményhalmaza az alaprelációk rekord előfordulásainak összes lehetséges párosát tartalmazza.*

Az összekapcsolás bemenő paramétereinek két relációnevet kell megadni. Ekkor az eredményreláció úgy áll elő, hogy az egyik reláció minden rekordját összekapcsoljuk a másik tábla minden rekordjával. Ha az egyik tábla N rekordot, a másik meg M rekordot tartalmaz, akkor N*M illesztés lehetséges, tehát az eredménytábla is N*M rekordot fog tartalmazni. Az eredményreláció szerkezete a két, összekapcsolt relációból származó minden mezőt magába foglalja. Azaz, ha az egyik reláció N darab mezőt, a másik meg M darab mezőt tartalmaz, akkor összesen N+M mezője lesz az előálló eredményrelációnak.

Az összekapcsolásra azért van szükség, mert a relációs adatbázisokban az adatok több relációba szétszórtan helyezkednek el, és a felhasználónak viszont gyakran van szüksége az elemi adatok mellett a kapcsolatban álló adatokra is. Az előző modelljeinkben például külön táblában tároltuk mind az autó, mind a tulajdonos adatait. Erre azért is szükség volt, mert mindkettő különböző adatszerkezettel írható le. Egy ezen modellre épülő információs rendszerben minden valószínűség



4.27. ábra. Az alap join művelet

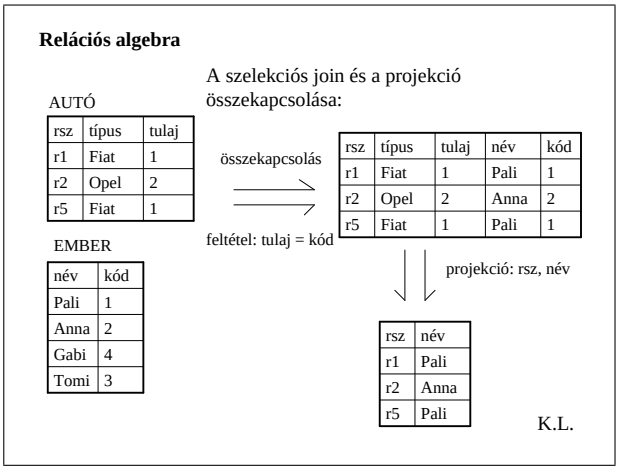
szerint gyakran igény jelentkezne olyan információra is, mely az autókat és a tulajdonos embereket együtt mutatja, vagyis az autót a tulajdonosával együtt kívánjuk megjeleníteni. A kapcsolódó, de különböző relációkban elhelyezkedő adatok együttes előállítására az összekapcsolás művelete ad lehetőséget. Az előzőekben említett példának egy mintarendszerre vonatkozó szemléltetését láthatjuk a 4.27. ábrán.

Természetesen ritkán fordul elő, hogy minden lehetséges párosításra szükségünk van, viszont annál gyakrabban szeretnénk azon párosításokat megkapni, melyek valamilyen feltételnek megfelelnek. Ez azért is olyan gyakori, mert a kapcsolatokat az adatértékeken keresztül, például a kapcsolókulcs és kulcs értékegyezőségével tároljuk.

Tehát amikor kapcsolatban álló rekordelőfordulásokat kívánunk előállítani, szükség van mindkét rekordra, melyeket az köt össze, hogy azonos értékek vannak az illeszkedő mezőkben. Azaz a lehetséges, összes párosítás közül csak azokra vagyunk kíváncsiak, melyeknél a két illeszkedő mező azonos értéket vesz fel.

A kapcsolat azonban nemcsak értékegyezőségen, hanem más összetettebb feltételeken keresztül is megvalósulhat. Ezért az összekapcsolás műveletéhez hozzákötethetünk egy illesztési feltételt is, mely megadja, hogy mely párosítások kerüljenek át a Descartes-szorzatból az eredménytáblába. Az egyesítési feltétel itt is lehet egyszerű és összetett feltétel. Ekkor tehát a két alaptábla mellett egy illesztési feltételt is megadnak a művelet végrehajtásához. Mivel ez a műveletpáros igen gyakran fordul elő a gyakorlatban, ezért külön elnevezést is kapott: a join ezen megvalósítását nevezik *szelekciós join*-nak. A művelet értelmezése:

> *A szelekciós join művelet eredményhalmaza az alap join eredményrelációján végrehajtott szelekció révén áll elő.*



4.28. ábra. A szelekciós join és a projekció összekapcsolása

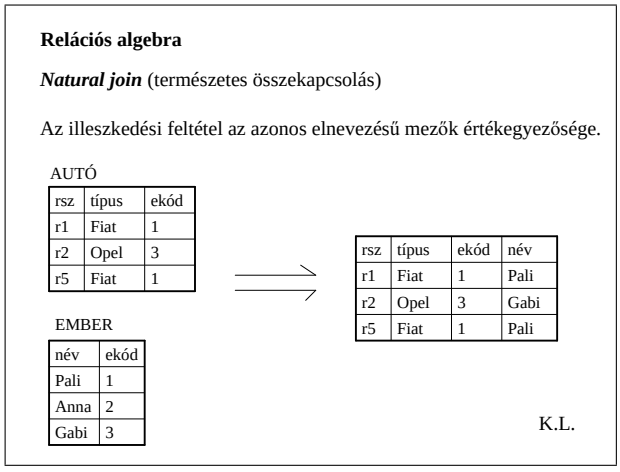
Példaként továbbra is azt az esetet tekintsük, amikor az autók és az emberek külön-külön relációban helyezkednek el, és a két egyed között egy tulajdonosi relációt modellezünk, és a kapcsolatot az autó relációba helyezett kapcsoló mezővel reprezentáljuk. Az autó reláció tartalmaz egy tulajdonos mezőt, mely a tulajdonosának az azonosító kódját tartalmazza. Az elvégzendő művelet, az autók rendszámának és a tulajdonos nevének a kiírása az eredménytáblába.

Mivel két táblából kell az adatokat összeszedni, ezért használjuk az összekapcsolás műveletét, az illesztési feltételként azt adhatjuk meg, hogy az autó rekord tulajdonos mezőjének értéke legyen egyenlő az ember rekord azonosító kód mezőjének értékével. Ha utánagondolunk, láthatjuk hogy az eredmény előállításához nem elegendő csupán az összekapcsolás művelete, hiszen annak eredménye más mezőket is tartalmaz. Ezért az összekapcsolás után még egy projekciót is végre kell hajtani. A 4.28. ábra példájában a műveletek egymásután fűzésére is kaphatunk példát.

Az összefűzés egyik speciális esete a *természetes összefűzés* (natural join), amikor az egyesítési feltétel a két tábla azonos elnevezésű és azonos domain-nel rendelkező mezőire követeli meg az értékegyezőséget. Az eredménytábla a felesleges redundancia elkerülése miatt a kapcsoló mezőket csak egyszer tartalmazza.

Az összekapcsolásnak a natural join mellett még számos más változata is van, melyekre később még visszatérünk.

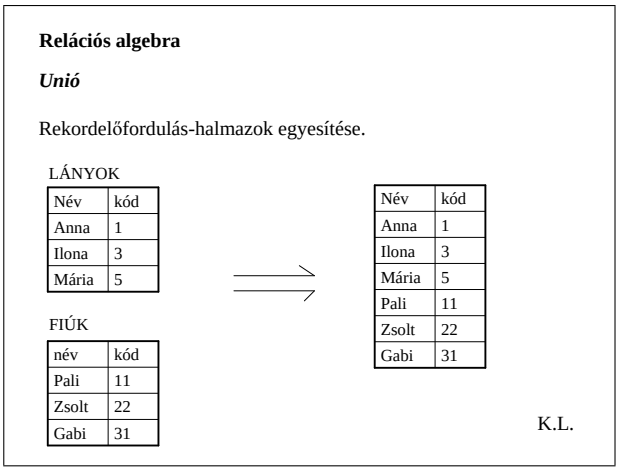
Az *unió* halmazegyesítést jelent, ami szintén kétoperandusú művelet, viszont itt mindkét táblának kompatibilis szerkezetűnek kell lenni, hiszen az eredményreláció mindkét reláció rekord előfordulásait tartalmazza, és hogy azok egy relációba legyenek elhelyezhetők, mindkét kiinduló relációnak kompatibilis felépítésűnek kell lennie. Az eredményreláció struktúrája megegyezik a bemenő relációk struktúrájával.



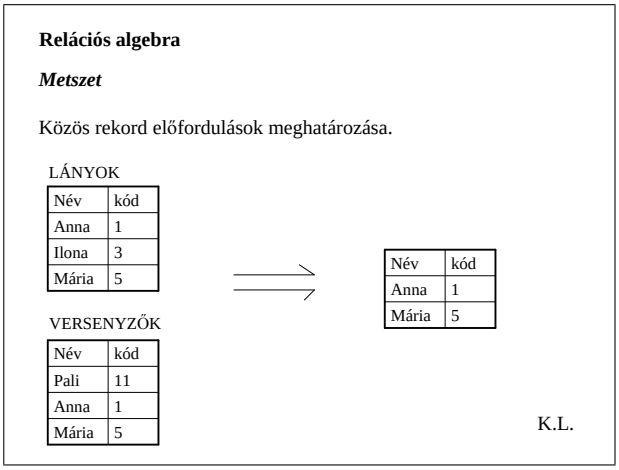
4.29. ábra. A natural join művelet

A *metszet* két relációban mindkét helyen előforduló rekord előfordulásokat adja vissza az eredménytáblában. Itt is mindkét táblának kompatibilis szerkezetűnek kell lennie, hiszen az eredményreláció a mindkét relációban meglelhető közös rekord előfordulásokat tartalmazza.

A *különbség* is egy újabb kétoperandusú művelet, mely az elsőként vett relációban megtalálható, de a másodikban nem szereplő rekord előfordulásokat adja vissza az eredménytáblában. Itt is mindkét táblának kompatibilisnek kell lennie. A különbség művelete nem szimmetrikus, azaz az eredmény függ az operandusok



4.30. ábra. Az unió művelete



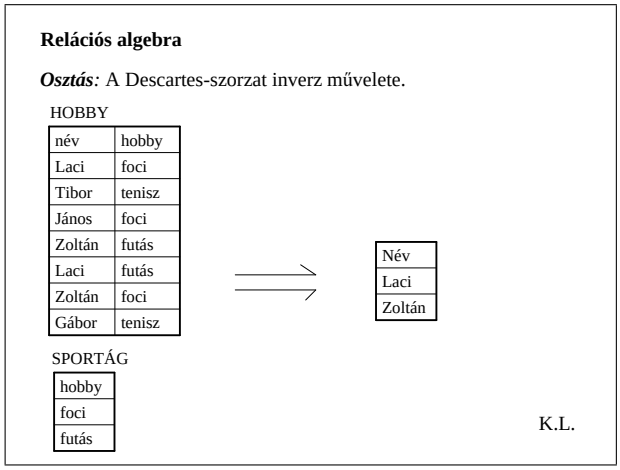
4.31. ábra. A metszet művelete

megadási sorrendjétől.

Az *osztás* egy kissé bonyolultabb műveletet jelent. Ez is kétoperandusú az előző műveletekhez hasonlóan. Az R1 és R2 relációk hányadosa az a reláció, amelybe R1 mindazon rekordjainak projekciói beletartoznak, amelyeknek az R2-vel való Descartes-szorzata a legnagyobb részhalmazát alkotja az R1-nek. Gyakorlatban igen ritkán használt és kevés RDBMS-ben megvalósított művelet, hiszen végrehajtása igen időigényes és gyakorlati haszna sem túlzottan jelentős.

A 4.32. ábrán bemutatott példában azért lett a (Laci, Zoltán) értékeket tartalmazó tábla az eredménytábla, mert ez az a reláció, melynek összekapcsolása (Descartes-szorzata) a SPORTÁG relációval benne van a HOBBY relációban és ez adja a legnagyobb ilyen módon előálló részhalmazát a HOBBY táblának. Az osztás eredményét egyébként az alábbi módon állíthatjuk elő. Előbb vesszük a HOBBY tábla azon mezőkre vett projekcióját, melyek nem szerepelnek a SPORTÁG relációban (komplementer mezők). Ezután képezzük ezen rekordok join-ját a SPORTÁG táblával, és a kapott eredményből kivonjuk a HOBBY relációt. Ezen a táblán egy újabb projekciót végzünk el a komplementer mezőkre. Az így előálló relációt kivonva az első projekció eredményéből, megkapjuk az osztás művelet eredményét.

A *kiterjesztés* művelete valójában kicsit később került be a köztudatba és még ma sem terjedt el igazán, mint önálló művelet, pedig a gyakorlatban igen sokszor használjuk. Ezért célunk, hogy most megérdemelt helyére emeljük a többi művelettel egyenrangú szintre. A művelet létrehozásának indíttatása az volt, hogy egyes lekérdezéseknél nem csak a mezőértékekre, hanem azokból képzett valamilyen kifejezések értékeire is kíváncsiak vagyunk. Például ha egy relációban van egységár és mennyiség mező, akkor bizonyos esetekben érdekelhet minket az összár értéke is, mely e két mező szorzataként adódik ki. Ezt úgy érhetjük el, hogy létrehozunk egy összár mezőt, melybe az egységár és a mennyiség mezők szorzatát írjuk. Így



4.32. ábra. Az osztás művelete

új relációt kapunk, mely a régi mezők mellett tartalmaz egy összár mezőt is.

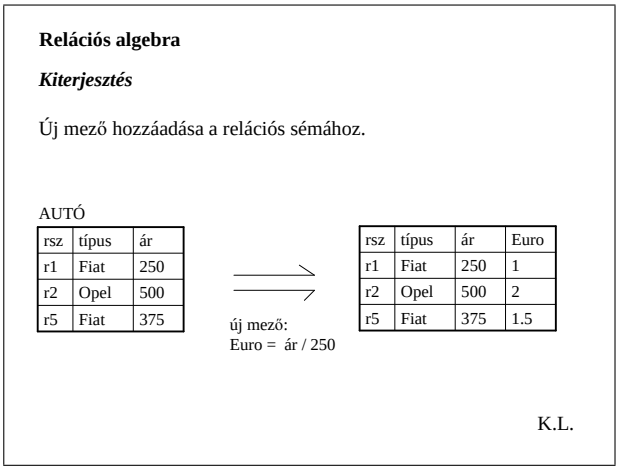
A kiterjesztésre példaként vegyük azt az esetet, amikor az autó táblánál a Forint-ban megadott árat átszámoljuk más pénznemre (4.33. ábra).

A megvalósított RDBMS rendszerekben is találkozhatunk hasonló reláció kiterjesztésekkel, igaz az eredményreláció rendszerint csak egy ideiglenes eredménytábla lesz, amely nem tárolódik le az adatbázisba. Ekkor rendszerint nem is adunk külön azonosító nevet az újonnan létrehozott mezőnek, ehelyett a rendszer a kiszámítási formulájával azonosítja ezt az oszlopot.

A másik ilyen jellegű művelet a *csoportosítás és aggregáció.* Bizonyos esetekben ugyanis nem magára a konkrét rekordelőfordulásra vagyunk kíváncsiak, hanem a rekordelőfordulások valamilyen összesítő adataira. A statisztikai adatok sokszor elegendő információt nyújtanak és jobban is kezelhetők, mint a részletes adatlisták. A dolgozók anyagi helyzetét egy adott vállalatnál az átlagfizetés jól jellemezheti, illetve ha részletesebb információkra van szükségünk, akkor lekérdezhetjük hogy hány dolgozó esik az egyes fizetési osztályokba. Az eredményből nem fogjuk megtudni, hogy ki mennyit keres, de a dolgozók csoportjára már értékelhető eredményt kapunk.

A relációs műveletek egységessége értelmében az ilyen kérdésekre adott válaszoknak is relációban kell tárolódnia. Az eredményreláció viszont nem részhalmaza az induló relációnak, ugyanis az eredményreláció minden egyes rekordja összesítő adatokat tartalmaz az induló relációk megadott rekord előfordulásainak egy-egy csoportjára. A csoportképzés és aggregáció értelmezése:

> *A csoportképzés és aggregáció művelete során előbb csoportokba válogatjuk szét a rekord előfordulásokat. A szétválogatás során azon rekordok kerülnek egy csoportba, melyekre egy megadott kifejezés megegyező értékű. Minden csoportra egy rekordot fog tartalmazni az eredményreláció. Ez a rekord a csoportbeli rekordelőfordulásokból számított aggre-*



4.33. ábra. A kiterjesztés művelete

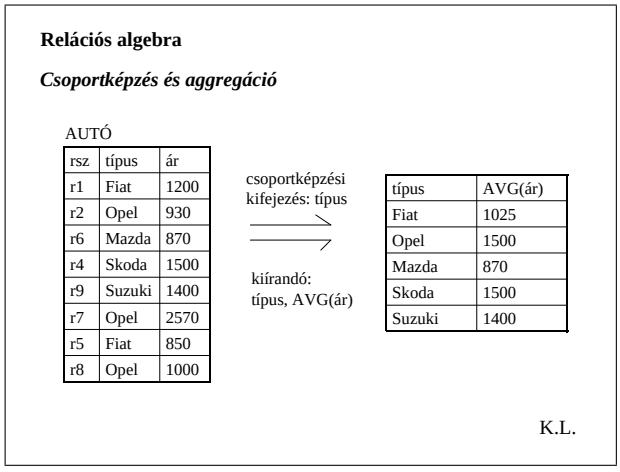
gációs értékeket tartalmaz.

A csoportképzés művelete tehát három bemenő paramétert is igényel. Az első annak a relációnak az azonosító neve, amelyre a csoportképzés vonatkozik. A második paraméter a csoportképzés alapjául szolgáló kifejezés. A csoportképzés értelmezése alapján a rendszer minden alaptáblabeli rekordra kiértékeli a megadott kifejezést, és azokat a rekordokat, melyekre a kifejezés ugyanazon értéket szolgáltatja, egyazon csoportba osztja be. Így tehát annyi különböző csoport jön létre, ahány különböző értéket ad a kifejezés a tábla rekordjainál. A harmadik paraméter az egyes csoportokra kiszámítandó kifejezéseket határozza meg. Ezeknek az értékeknek is egyértelműeknek, egyértékűeknek kell lenniük. Mivel egy csoporton belül több rekord is elhelyezkedhet, az alaptábla mezőinek is több értéke lehet, így ezek a mezők csak akkor maradhatnak meg az eredményrelációban, ha azok csoportképzés alapjául is szolgáltak, mert csak ebben az esetben biztosítható az értékek elemisége. A csoportképzés alapjául szolgáló mezők mellett e csoportokra a csoport egyes elemeiből képzett összesítő értékeket szokták még szerepeltetni az eredményrelációban. Minden egyes csoportra lehet összesített adatokat képezni, mint például

- *count*, az előfordulások darabszáma
- *sum*, az előfordulások valamely mezőjének összege
- *max*, az előfordulások valamely mezőjének maximuma
- *min*, az előfordulások valamely mezőjének minimuma
- *avg*, az előfordulások valamely mezőjének átlaga.

Az eredménytábla állhat egy sorból és lehet üres is, attól függően, hogy hány csoport képződött a rekordelőfordulásokból.

Példaként vegyük azt a műveletet, amikor az AUTÓ tábla alapján az egyes autótípusok átlagárait szeretnénk lekérdezni. Az eredményt úgy kaphatjuk meg,



4.34. ábra. A csoportképzés művelete

hogy előbb csoportokat képezünk az autókra az autótípus alapján. Egy csoportba az azonos típusú autók fognak kerülni. Ezután minden csoportra meghatározzuk a hozzá tartozó autók átlagárát. A csoportképzés három paramétere tehát a következő lesz: alaptábla: AUTÓ, csoportképzés kifejezése: a típus mező értéke, és a kiírandó adatok: a típus és az ár értékek átlaga.

A csoportképzés során a csoportképzés kifejezése maradhat üresen is. Ekkor, vagyis ha semmilyen feltételt nem adunk meg, az összes rekord előfordulás egy csoportba kerül. Például az autók darabszámát is ilyen aggregációval lehet megoldani.

Az előbbiekben felsorolt műveletek a megszabott kereteken belül tetszőlegesen kombinálhatók egymással, azaz valamely művelettel előállított eredmény reláció felhasználható egy következő művelet induló relációjaként. Ilyenkor egy kifejezésbe is összevonhatók a műveletek, mint azt már az összekapcsolás műveleténél is láttuk.

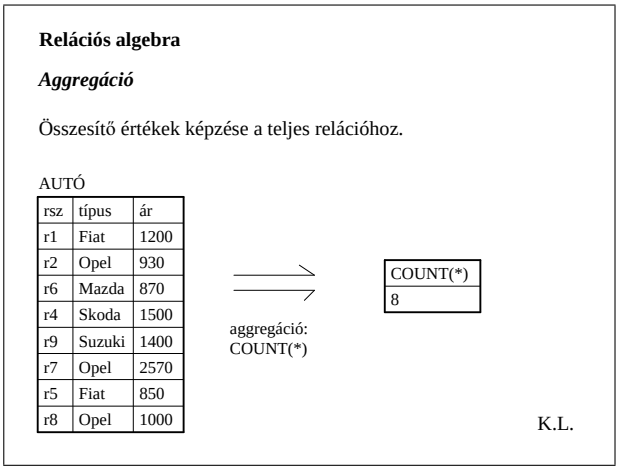
4.6.2. A relációs algebra formális leírása

Mint ahogy a relációs adatmodell strukturális része is formális nyelven adható meg a legtömörebben és legegyértelműbben, úgy a műveleti rész, a relációs algebra is megfogalmazható formális, matematikai szimbólumokkal. A műveletek felírásánál a korábban bevezetett strukturális jelölésekre fogunk támaszkodni.

A *konstans szelekció*: a relációból egy megadott feltételnek eleget tévő rekordok kiválogatása egy eredmény relációba. Szimbóluma a σ jel, melynek paraméterei egy r alapreláció és egy szelekciós feltétel. A művelet formális felírása:

$$\sigma_X \Theta_x(r) = \{\ t \mid t \in r \wedge t(X)\ \Theta\ x\}$$

ahol X egy attribútum csoport az r reláción belül, x egy az X típusához illeszkedő



4.35. ábra. Az aggregáció művelete

konstans és Θ egy relációs operátor.

Az *attribútum szelekció*: a relációból egy megadott feltételnek eleget tévő rekordok kiválogatása egy eredmény relációba, ahol a feltételben két attribútum összehasonlítása szerepel, szimbóluma szintén a σ jel, melynek paraméterei egy r alapreláció és egy szelekciós feltétel. A művelet formális felírása:

$$\sigma_{X\Theta Y}(r) = \{\ t \mid t \in r \wedge t(X)\ \Theta\ \mathrm{t}(Y)\}$$

ahol X és Y is egy-egy attribútum csoport az r reláción belül, és Θ egy relációs operátor.

A *projekciónál*, vagyis a relációnak bizonyos attribútumaira való leszűkítésénél az alapreláció azonosítóját és a kiválasztott mezők azonosítóit kell megadni paraméterként. A projekció szimbóluma a π jel. A paraméterezett művelet formális felírása:

$$\pi_X(r) = \{\ t(X) \mid t \in r\}$$

ahol r az alaprelációt és X az r egy attribútum csoportját jelöli.

Az *összekapcsolás*, a relációk Descartes-szorzata, olyan relációt eredményez, melyben a két bemenő reláció rekordjainak minden lehetséges párosítása benne foglaltatik. Az összekapcsolás műveletének jele a $\bowtie$ szimbólum. Az összekapcsolás műveleténél a két alaprelációt kell megadni paraméterként, és a művelet értelmezése a következő:

$$r_{\ 1} \bowtie r_2 = \{\ t \mid t \in r(R_1{+}R_2) \wedge t(R_1) \in r_1 \wedge t(R_2) \in r_2\}$$

ahol $r_1(R_1)$ és $r_2(R_2)$ teljesül a két alaprelációra.

Relációs algebra

Szelekció: a megadott feltételnek eleget tévő rekordok kerülnek át az eredmény relációba.

Jele: $\sigma_{felt}(r)$

$\sigma_{X\ \theta\ Y}(r) = \{\ t \in r \mid t(X)\ \theta\ t(Y)\}$

AUTÓ

rsz	típus	szín
r1	Fiat	zöld
r2	Opel	kék
r6	Mazda	piros
r4	Skoda	kék
r9	Lada	piros

rsz	típus	szín
r2	Opel	kék
r6	Mazda	piros
r4	Skoda	kék

$\sigma_{szín='kék'\ OR\ típus='Mazda'}$ (AUTO)

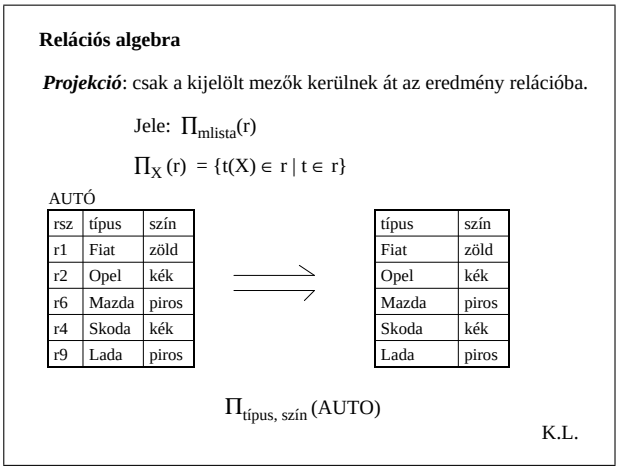
K.L.

4.36. ábra. A szelekció formális felírása

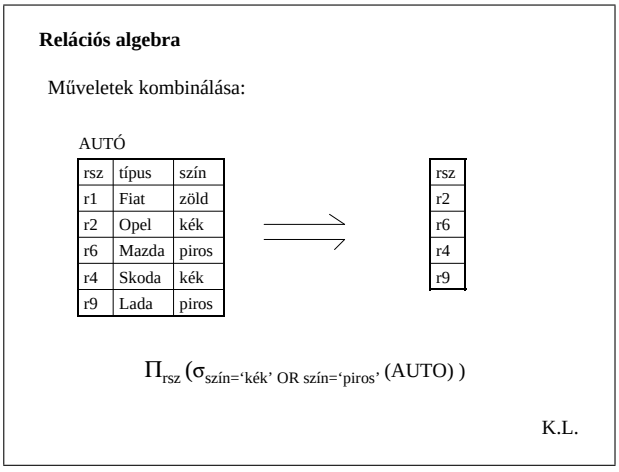
Mint említettük, az összekapcsolást legtöbbször egy szelekciós feltétellel kapcsoljuk össze, azaz azon rekordok kerülnek bele az eredményrelációba a Descartes-szorzatból, melyek kielégítik a megadott feltételt. A szelekciós összekapcsolás műveletének értelmezése:

$$\mathrm{r}_1 \bowtie_{expr} r_2 = \{\ t \mid t \in r(R_1{+}R_2) \wedge t(R_1) \in r_1 \wedge t(R_2) \in r_2 \\ \wedge\ \mathrm{expr(t)} = \mathrm{igaz}\ \}$$

ahol $r_1(R_1)$ és $r_2(R_2)$ teljesül a két alaprelációra. Az expr az összekapcsolt rekordokon értelmezett logikai értékű kifejezést takar.



4.37. ábra. A projekció formális felírása



4.38. ábra. A projekció és a szelekció kombinálása

A szelekciós összekapcsolás esetében csak azon rekordok kerülnek be az eredmény relációba, melyekre létezik kapcsolódó rekord a másik oldalon. Így a kapcsolat nélküli rekordok nem lesznek benne az eredmény relációban. Sokszor azonban az is hasznos információt ad, ha látjuk a kapcsolat nélküli egyedeket. Hogy ennek eléréséhez ne kelljen még egy kivonás műveletet végrehajtani, bevezették az outer vagy külső join operátorát. A *külső join* a szelekciós join-ra épülő művelet, melynek az eredményében a kapcsolat nélküli rekordok is szerepelnek egy NULL értékű résszel kiegészítve. A külső összekapcsolás (outer join) műveletének típusai:

Relációs algebra

Join: a két reláció rekordjainak párosait adja eredményként.

Jele: $r_1 \rhd\lhd r_2$

Alap join (Descartes-szorzat): $r_1 \rhd\lhd r_2 = \{(t,s) \mid t \in r_1 , s \in r_2\}$

AUTÓ

rsz	típus	szín
r1	Fiat	zöld
r2	Opel	kék
r5	Fiat	kék

EMBER

név	kor
Pali	36
Anna	29

rsz	típus	szín	név	kor
r1	Fiat	zöld	Pali	36
r2	Opel	kék	Pali	36
r5	Fiat	kék	Pali	36
r1	Fiat	zöld	Anna	29
r2	Opel	zöld	Anna	29
r5	Fiat	kék	Anna	29

AUTÓ $\rhd\lhd$ EMBER

K.L.

4.39. ábra. Az alap join formális felírása

Relációs algebra

Szelekciós join: a két reláció rekordpárosaiból a feltételnek eleget tévő párosokat adja eredményként.

Jele: $r_1 \rhd\lhd_{felt} r_2$

$r_1 \rhd\lhd_{felt} r_2 = \sigma_{felt}(r_1 \rhd\lhd r_2)$

Natural join: olyan szelekciós join, mely az azonos elnevezésű mezők értékegyezőségén alapszik.

Jele: $r_1 \rhd\lhd_{=} r_2$

$r_1 \rhd\lhd_{=} r_2 = \Pi_{(R1 \cup R2) \setminus X}(\sigma_{r1.X = r2.X}(r_1 \rhd\lhd r_2))$

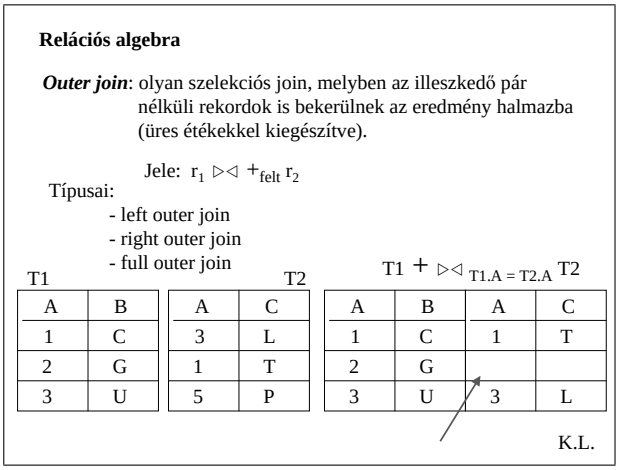
Ritkán implementált.

K.L.

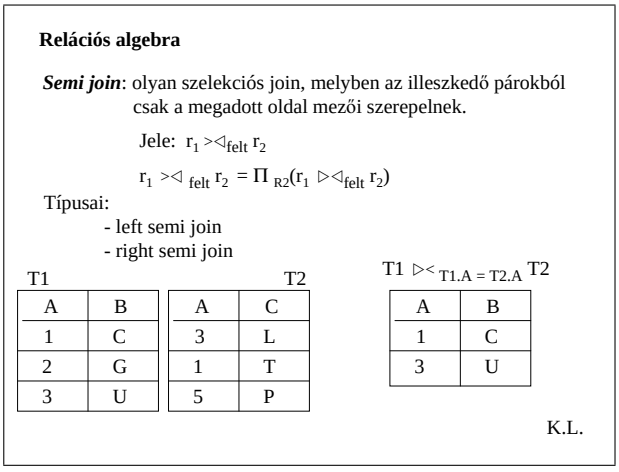
4.40. ábra. A szelekciós és a natural join formális felírása

$r_1 +\bowtie_{expr} r_2$	bal oldali külső join, a nem illeszkedő r_1 rekordok is megjelennek az eredményben
$r_1 \bowtie_{expr} + r_2$	jobb oldali külső join, a nem illeszkedő r_2 rekordok is megjelennek az eredményben
$r_1 +\bowtie_{expr} + r_2$	két oldali külső join, a nem illeszkedő r_1 és r_2 rekordok is megjelennek az eredményben

ahol az expr az összekapcsolt rekordokon értelmezett logikai értékű kifejezést takar.



4.41. ábra. Az outer join formális felírása



4.42. ábra. A semi join formális felírása

A *szemi-összekapcsolás* (semi-join) eredménye egy olyan reláció, melyet a szelekciós join eredményéből a kijelölt oldalra való projekciójával állítunk elő:

$$r_1 \ltimes_{expr} r_2 = \pi_{R2}(r_1 \bowtie_{expr} r_2)$$

ahol $r_1(R_1)$ és $r_2(R_2)$ teljesül a két alaprelációra. Az expr az összekapcsolt rekordokon értelmezett logikai értékű kifejezést takar.

Két reláció *uniója* a szokásos halmazalgebrai unió műveletére épül, így szimbóluma is a szokásos unió jel. A művelet értelmezése:

$$r_1 \cup r_2 = \{ t \mid t \in r_1 \vee t \in r_2 \}$$

ahol r_1 és r_2 a két alapreláció. Az unió végrehajthatóságához a két alaprelációnak ekvivalens domain-nel kell rendelkeznie.

Két reláció *metszete* is a hagyományos halmazműveletre épül, értelmezése:

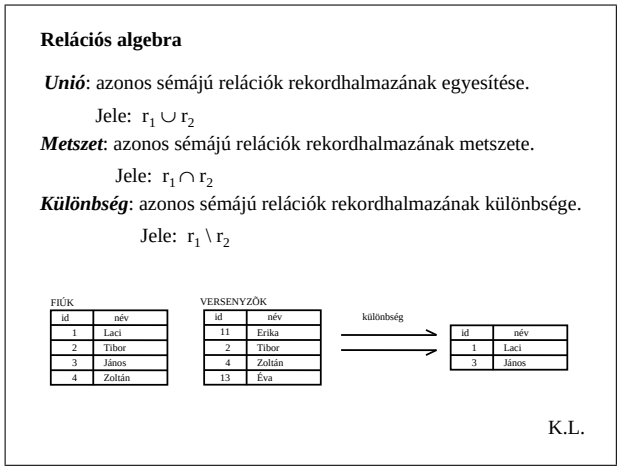
$$r_1 \cap r_2 = \{ t \mid t \in r_1 \wedge t \in r_2 \}$$

ahol r_1 és r_2 a két alapreláció. A metszet végrehajthatóságához a két alaprelációnak ekvivalens domain-nel kell rendelkeznie.

Két reláció *különbsége* is ismert értelmezésen alapul:

$$r_1 - r_2 = \{ t \mid t \in r_1 \wedge t \notin r_2 \}$$

ahol r_1 és r_2 a két alapreláció. A különbség végrehajthatóságához a két alaprelációnak ekvivalens domain-nel kell rendelkeznie.

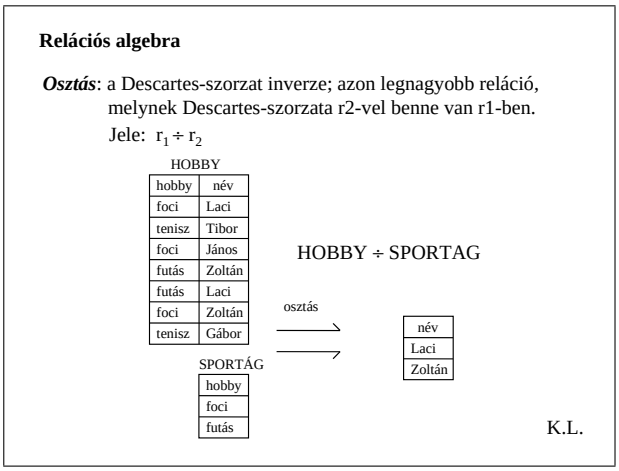


4.43. ábra. Unió, metszet, különbség formális felírása

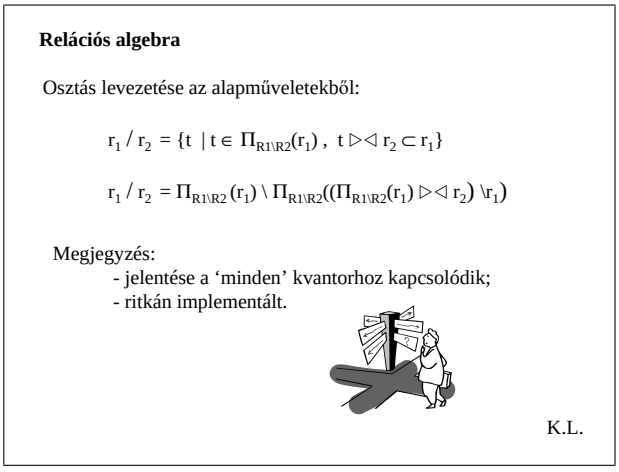
Az *osztás* művelete viszont már csak bonyolultabb formában írható fel. Az osztás paramétereként két alaprelációt kell megadni, és eredménye az osztandónak az osztóhoz nem tartozó attribútumaira vett projekciójának azon legnagyobb részhalmaza lesz, melynek az osztóval vett összekapcsolása (Descartes-szorzata) az osztandó egy részhalmaza. A művelet formális felírása:

$$r_{1/}\ r_2 = \{\ t \mid t \in \pi_{A1}(r_1) \wedge \{t\} \times r_2 \subset r_1\ \}$$

ahol $r_1(A)$ és $r_2(A2)$ a két alapreláció úgy, hogy $A = A1 + A2$ is teljesül, azaz az r_1 attribútumai magukba foglalják r_2 attribútumait. Az $A1$ attribútum csoport tehát r_1 azon attribútumainak összességét jelenti, amely benne van r_1-ben, de



4.44. ábra. Az osztás formális felírása



4.45. ábra. Az osztás levezetése az alapműveletekből

nincs benne r_2-ben. Az osztást jelölhetjük az $\div$ szimbólummal is.

Az osztás az $r_1(R1 \backslash R2)$ azon elemeit adja, amelyek minden r_2 elemmel társulnak az r_1 relációban. Az osztás visszavezethető a korábbi relációs algebrai műveletekre. A visszavezetés alapelve, hogy előbb meghatározzuk, melyek azok az $r_1(R1 \backslash R2)$ elemek, melyeknek van olyan párosa valamely r_2 elemmel, mely nincs benne az r_1 relációban. Ezen elemek komplementere adja a keresett értékhalmazt. Az r_1-ben nem szereplő párosok:

$$(\pi_{R1 \backslash R2}(r_1) \bowtie r_2) \backslash r_1$$

A párosok nélküli $r_1(R1 \backslash R2)$ rész:

$$\pi_{R1 \backslash R2}((\pi_{R1 \backslash R2}(r_1) \bowtie r_2) \backslash r_1)$$

és a keresett komplementer rész, mint megoldás:

$$\pi_{R1 \backslash R2}(r_1) \backslash \pi_{R1 \backslash R2}((\pi_{R1 \backslash R2}(r_1) \bowtie r_2) \backslash r_1).$$

A *kibővítésnél* egy újabb attribútummal növeljük meg a reláció sémáját. Az attribútumhoz tartozó mezőértéket a rekord meglévő mezőinek értékéből határozzuk meg. A bővítésnél tehát az alapreláció mellett az új attribútum azonosító nevét és a hozzá tartozó mezőérték kiszámítási függvényt kell megadni paraméterként. A művelet értelmezése:

$$\epsilon_{aexp}\ (\mathrm{r}) = \{\ t \mid t \in r\text{'}(A + a) \wedge t(a) = \exp(\mathrm{t}(A)) \wedge t(A) \in r\ \}$$

ahol $r(A)$ az alapreláció, A jelöli e reláció sémáját, az a szimbólum pedig az új attribútum azonosító neve.

A *csoportképzés / aggregáció* esetén több elemi tevékenységet kell elvégeznie a rendszernek. Előbb egy csoportképzési feltétel alapján diszjunkt csoportokba

Relációs algebra

Kiterjesztés: a reláció kibővítése származtatott mezőkkel.

Jele: $\varepsilon_{\text{mlista}}(\mathrm{r})$

Az új mező értékének a többi mező értékéből kell meghatározódnia.

AUTÓ

rsz	tipus	ár
r1	Fiat	1200
r2	Opel	930
r6	Mazda	870
r4	Skoda	1500
r9	Suzuki	1400

kiterjesztés
új mezõ: ár/100

rsz	tipus	ár	ár/100
r1	Fiat	1200	12
r2	Opel	930	9.3
r6	Mazda	870	8.7
r4	Skoda	1500	15
r9	Suzuki	1400	14

$\varepsilon_{\text{ar/100}}(\text{AUTO})$

K.L.

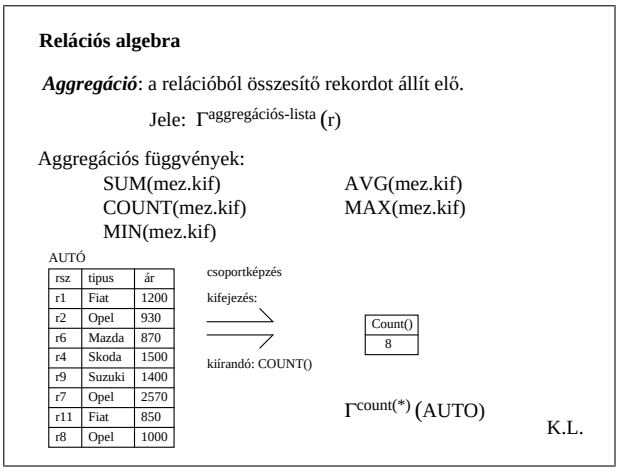
4.46. ábra. A kiterjesztés formális felírása

osztja a reláció rekordjait, majd minden csoportra kiszámol egy aggregált érték n-est, amely bekerül az eredményrelációba rekordként. A művelet jelölése:

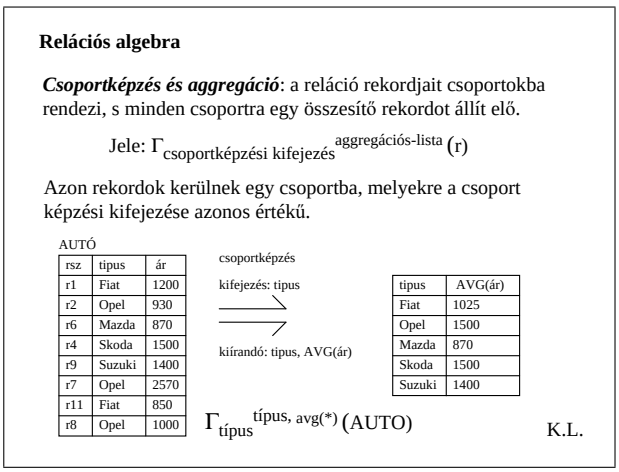
$$\Gamma_{\ gexp}{}^{\ a\ exp}(r)$$

ahol r az alaphalmaz, gexp a csoportosítási feltétel, a_i az eredménytáblába kerülő attribútumok és az $\exp_i$ az attribútumok kiszámítására szolgáló kifejezések szimbóluma. Ha gexp elmarad, a reláció összes rekordja egy csoportba kerül.

Az eredmény relációt meghatározó aggregációs listában csak olyan kifejezések szerepelhetnek, melyek minden csoportra nézve egyértelmű, egyértékű értéket adnak vissza. Ezért az aggregációs lista eleme lehet:



4.47. ábra. Az aggregáció formális megadása



4.48. ábra. Csoportképzés és aggregáció formális megadása

– aggregációs függvény és/vagy
– csoportképzési kifejezés.

Egyéb kifejezés nem elfogadott. Így például egy típus szerinti csoportképzésnél nem lehet az autó árát megjelentetni az eredményben, mivel egy csoportban több különböző árú autó is szerepelhet.

A bemutatott relációs algebrai kifejezések szemléltetésére vegyünk két alaptáblát, melyekre sorra bemutatjuk az ismertetett műveletek eredményét. Az alaptáblák egy borkereskedő információs rendszeréből származnak: legyen egy BOROK és egy TERMELŐK reláció az itt megadott struktúrával és alapadatokkal.

BOROK

bid	megn.	fajta	termelő
1	Olaszrizling	3	3
3	Muskotály	2	1
2	Kadarka	9	6
4	Bikavér	2	2
6	Szürkeb.	4	3
7	Hárslevelű	5	1
8	Szilváni	7	2
9	Piot Noire	2	4
10	Rizling	5	3
11	Aszú	6	5

TERMELŐK

tid	név	cím
1	Péter	Eger
2	Zoltán	Badacsony
3	Lajos	Gyöngyös
4	Gábor	Kiskörös
5	Lajos	Kecskemét
6	Ferenc	Miskolc

A következőkben sorra vesszük a műveleteket és a hozzájuk tartozó eredménytáblát.

– Konstans szelekció: a 2-es fajtához tartozó borok adatai.

$\sigma_{fajta=2}(BOROK)$

bid	megn.	fajta	termelő
3	Muskotály	2	1
4	Bikavér	2	2
9	Piot Noire	2	4

– Attribútum szelekció: a fajtakód megegyezik a termelőkóddal.

$\sigma_{fajta=termelo}(BOROK)$

bid	megn.	fajta	termelő
1	Olaszrizling	3	3
4	Bikavér	2	2

– Projekció: a termelők nevei.

$\pi_{nev}(TERMELŐK)$

név
Péter
Zoltán
Lajos
Gábor
Ferenc

– Összekapcsolás: a borok és a termelőjük adatainak kiírása.

$BOROK \bowtie_{termelo=tid} TERMELŐK$

bid	megn.	fajta	termelő	tid	név	cím
1	Olaszrizling	3	3	3	Lajos	Gyöngyös
3	Muskotály	2	1	1	Péter	Eger
2	Kadarka	9	6	6	Ferenc	Miskolc
4	Bikavér	2	2	2	Zoltán	Badacsony
6	Szürkeb.	4	3	3	Lajos	Gyöngyös
7	Hárslevelű	5	1	1	Péter	Eger
8	Szilváni	7	2	2	Zoltán	Badacsony
9	Piot Noire	2	4	4	Gábor	Kiskörös
10	Rizling	5	3	3	Lajos	Gyöngyös
11	Aszú	6	5	5	Lajos	Kecskemét

– Kibővítés: a BOROK reláció kibővítése egy kategória mezővel, amely a fajta mezőtől függ.

$\epsilon_{kategria,fajta<3?'A':'B'}(BOROK)$

bid	megn.	fajta	termelő	kategória
1	Olaszrizling	3	3	B
3	Muskotály	2	1	A
2	Kadarka	9	6	B
4	Bikavér	2	2	A
6	Szürkeb.	4	3	B
7	Hárslevelű	5	1	B
8	Szilváni	7	2	B
9	Piot Noire	2	4	A
10	Rizling	5	3	B
11	Aszú	6	5	B

– Csoportképzés: hány fajta bort szállítanak az egyes termelők. Ehhez a borokat a termelő alapján csoportosítjuk, és minden csoportra kiszámítjuk a benne található rekordok darabszámát.

$\Gamma_{termelo}{}^{termelo,count(*)\ darab}(BOROK)$

termelő	darab
3	3
1	2
6	1
2	2
4	1
5	1

A többi művelet bemutatása előtt vegyünk fel egy újabb táblát, amely szerkezetében a TERMELŐK táblához hasonlít. A reláció neve legyen ISMERŐSÖK. A tábla felépítése a következő:

ISMERŐSÖK

tid	név	cím
1	Péter	Eger
7	Lajos	Dorog
3	Lajos	Gyöngyös
9	Tibor	Győr
11	Zoltán	Szeged
6	Ferenc	Miskolc

– Unió: az ismerősök és termelők együttese.

TERMELŐK ∪ ISMERŐSÖK

tid	név	cím
1	Péter	Eger
2	Zoltán	Badacsony
3	Lajos	Gyöngyös
4	Gábor	Kiskörös
5	Lajos	Kecskemét
6	Ferenc	Miskolc
7	Lajos	Dorog
9	Tibor	Győr
11	Zoltán	Szeged

– Metszet: azon termelők, akik ismerősök is egyben.

TERMELŐK ∩ ISMERŐSÖK

tid	név	cím
1	Péter	Eger
3	Lajos	Gyöngyös
6	Ferenc	Miskolc

– Különbség: azon termelők, akik nem ismerősök.

TERMELŐK - ISMERŐSÖK

tid	név	cím
2	Zoltán	Badacsony
4	Gábor	Kiskörös
5	Lajos	Kecskemét

– Osztás: a művelet bemutatásához egy újabb relációt veszünk fel, amely az ismerős városokat tartalmazza, és felépítése a következő:

VÁROSOK

cím
Dorog
Gyöngyös

A lekérdezés azon nevekre irányul, amelyek minden ismerős városban előfordulnak az ismerősök között.

$\pi_{nev,cim}$ (ISMERŐSÖK) / VÁROSOK

név
Lajos

A bemutatott műveletek tetszőlegesen láncolhatók egymásba a relációs algebra zártságának köszönhetően. Néhány összetett példa bemutatására vegyünk fel két újabb relációt, a vevők és rendelések relációit. A két tábla szerkezete és tartalma a következő:

VEVŐK

vid	cég	cím
1	Alfa	Szolnok
2	C+C	Győr
3	Star	Hatvan
4	Billa	Pécs
5	Agria	Eger
6	Penny	Tata
7	Vino	Siófok

RENDELÉS

vevő	termék	menny.	dátum
2	1	2	1
3	2	5	4
4	6	7	4
1	9	2	5
1	7	1	6
5	2	2	6
2	2	5	6
7	3	1	7
3	1	3	7

Az első lekérdezés azon vevők cégneveit adja vissza, akik rendeltek bikavért:

$\pi_{ceg}(\sigma_{bid=termek\ AND\ vevo=vid\ AND\ megn='Bikaver'}$ *(BOROK* $\times$ *RENDELÉS* $\times$ *VEVŐK)*

A következő lekérdezésnél a vevők cégneveit és a rendelt mennyiség összértékét iratjuk ki:

$\Gamma_{ceg}^{ceg\ 'ceg',\ sum(menny)\ 'osszrendeles'}$ $(\sigma_{vevo=vid}$ *(VEVŐK* $\times$ *RENDELÉS))*

A legnagyobb rendelési összérték meghatározása:

$$\Gamma^{max(osszrendeles)\ 'max'}\ (\Gamma_{ceg}{}^{ceg\ 'ceg',\ sum(menny)\ 'osszrendeles'}\ (\sigma_{vevo=vid}\ (VEVŐK\ \times\ RENDELÉS)))$$

Mivel a külső csoportképzésben nem szerepel csoportképzési feltétel, így az alaptábla minden rekordja egyetlen egy csoportba kerül.

Az utolsó példában még további műveleteket adunk az előző kifejezések köré. A végrehajtandó feladat most azon vevő cégek nevének megadása, akiknek az összes rendelése értékben megegyezik a maximális összes rendeléssel :

$$\pi_{ceg}(\sigma_{osszrendeles=}\ (\Gamma^{max(osszrendeles)\ 'max'}\ (\Gamma_{ceg}{}^{ceg\ 'ceg',\ sum(menny)\ 'osszrendeles'}\ (\sigma_{vevo=vid}(VEVŐK\times RENDELÉS))))$$

$$(\Gamma_{ceg}{}^{ceg\ 'ceg',\ sum(menny)\ 'osszrendeles'}(\sigma_{vevo=vid}(VEVŐK\times RENDELÉS))))$$

Igen, ez a kifejezés már első pillantásra is elrettentőnek tűnik. Valóban áttekinthetetlen ez a művelet. Szerencsére a gyakorlatban nézeti táblák, a *view*-k segítségével sokkal áttekinthetőbb alakra lehet majd hozni a fenti kifejezést. A mintapéldát ugyanis az tette kuszává, hogy nem léteznek olyan alaptáblák, amelyekből közvetlenül, egyszerűbb módon származtathatók lennének a kívánt mennyiségek. Ha bevezetünk egy új szimbólumot a vevők összrendeléseit tartalmazó táblára, akkor máris sokkal áttekinthetőbb lesz a művelet felírása is.

$$X = \Gamma_{ceg}{}^{ceg\ 'ceg',\ sum(menny)\ 'osszrendeles'}(\sigma_{vevo=vid}(VEVŐK\times RENDELÉS))$$

$$\pi_{ceg}\ (\sigma_{osszrendeles=}(\Gamma^{max(osszrendeles),\ 'max'}(X))\ (X))$$

Relációs algebra

Mintapélda:

BOROK(kod, nev, gyarto, fajta, ar)
VEVO(vkod, nev, cim)
RENDELES(vevo, bor, mennyiseg,datum)

1. Azon vevők, akik rendeltek Bikavért:

$\Pi_{nev}(\sigma_{borok.kod = rendeles.bor\ AND\ rendeles.vevo = vevo.vkod\ AND\ borok.nev = 'Bikaver'}$
(BOROK $\triangleright\triangleleft$ RENDELES $\triangleright\triangleleft$ VEVO))

2. Melyik gyártó termel 5-nél több 3-as típusú bort:

$\Pi_{gyarto}(\sigma_{count(*)>5}(\Gamma_{gyarto}{}^{gyarto,count(*)}(\sigma_{fajta=3}(BOROK))))$

K.L.

4.49. ábra. Példalekérdezések relációs algebrában I.

A relációs algebrai kifejezéseink azt írták le, hogy milyen lépéseken, műveleteken keresztül állíthatjuk elő a kívánt eredményt a megadott alaphalmazokból. A kifejezéseink felírásakor tehát tudnunk kellett, hogy mely relációs algebrai műveletek együttese szolgáltatja a kívánt eredményt a lekérdezésnél. Ezért a relációs algebrát egy előíró, utasító (prescriptív) nyelvnek tekinthetjük, hiszen előírjuk, hogy hogyan jutunk el az eredményhez. Ha például meg szeretnénk tudni azon dolgozók neveit, akik többet keresnek 100000 forintnál, akkor a lekérdezés leírása a következőkkel adható meg:

– szelektáljuk a dolgozók relációt azokra a rekordokra, ahol a fizetés nagyobb mint 100000;
– végezzünk projekciót a részeredmény relációban a név mezőre.

A megadott műveleteket elvégezve, megkapjuk az eredményt, tehát a relációs algebra egyben le is írja a megoldás lépéseit.

A relációs algebrához kapcsolódóan nemcsak a szokásos összehasonlító operátorokat lehet alkalmazni, hanem vannak speciális, a relációkhoz kapcsolódó operátorok is. Ezek között kiemelhetők az alábbi műveletek:

$\in$: tartalmazás, egy kifejezés értéke benne van egy halmazban vagy listában;
$\notin$: a tartalmazás tagadása;
$\forall$: minden kvantor;
$\exists$: létezik kvantor.

Ha például azon vevők adatait kell megjeleníteni, akik nem rendeltek egy adott dátumú napon, akkor ezt megtehetjük olyan módon, hogy előbb előállítjuk azon vevők kódhalmazát, akik rendeltek az adott napon, majd megnézzük, mely vevők kódja nincs benne ebben a halmazban. A műveletsor alakja:

Relációs algebra

Speciális szelekciós operátorok:
$\in$: tartalmaz
$\notin$: nem tartalmaz
$\forall$: minden kvantor
$\exists$: létezik kvantor

A szelekciós feltételben is szerepelhet relációs algebrai kifejezés.

3. Az átlagnál drágább borok darabszáma:
$\Gamma^{count(*)}(\sigma_{ar>} (\Gamma^{avg(ar)}(BOROK))^{(BOROK)})$

K.L.

4.50. ábra. Példalekérdezések relációs algebrában II.

$$\pi_{nev}(\sigma_{\ vid \notin (\pi_{vevo}(\sigma_{datum=nap}\ (rendeles)))^{(vevo)}})$$

Az algebrai műveletek segítségével, a műveletek megfelelő sorrendben való kombinálásával igen összetett lekérdezések is megfogalmazhatók igen tömören és áttekinthetően.

4.6.3. A relációs kalkulus

A relációs algebra azonban nem az egyedüli megadási formalizmusa a relációs lekérdezési műveleteknek. Létezik olyan megközelítése is a relációs műveleteknek, mely nem részletezi a megoldás lépéseit, csak a kívánt eredményt fogalmazza meg. Ezen leírási módot nevezik relációs kalkulusnak.

A *relációs kalkulus* a logikai kalkulushoz hasonló formalizmus, amely egy formulával jellemzi a kívánt végeredményt. Ebben a megközelítésben azon rekordok lesznek a lekérdezés eredményei, melyek kielégítik a formulát, vagyis a formulára igaz helyettesítési értéket adnak vissza. A relációs kalkulusnál a felhasználónak csak az eredményt jellemző formulát, kifejezést kell megadnia, és nem kell törődnie a formulát kielégítő rekordok meghatározásának módjával. Ebből a megközelítésből nézve a relációs kalkulust úgy tűnik, mintha könnyebb lenne használni, mint a relációs algebrát. Az előbbi példa – azon dolgozók nevének lekérdezése, akik 100000 forintnál többet keresnek – a relációs kalkulusban a következő alakot ölti:

> Kérem azon név értékeket a dolgozók táblából, ahol a fizetés nagyobb mint 100000.

Látható, hogy ez a leírás nem mond semmit arról, hogy hogyan tudom előállítani az eredményrelációt. Ezért a relációs kalkulust descriptív, leíró nyelvnek nevezik, hiszen itt csak leírom, hogy mire van szükségem, de nem adom meg a megoldást. A relációs algebra ezzel szemben előírta a szükséges lépéseket is.

Relációs kalkulus

Nyelvi elemek

Az eredmény jellemzését kell leírni.

Háttér: predikátum kalkulusok elmélete

AUTO (rsz, típus, tulaj, ár)
EMBER (kód, név)

Azon ember rekordok kellenek, melyekhez létezik olyan autó rekord, melyben a típus mező Fiat és a tulaj mező megegyezik az ember kód mezőjével.

$\exists x,y,z:$ *auto(x, "Fiat", y, z)*

K.L.

4.51. ábra. A relációs kalkulus nyelvi elemei I.

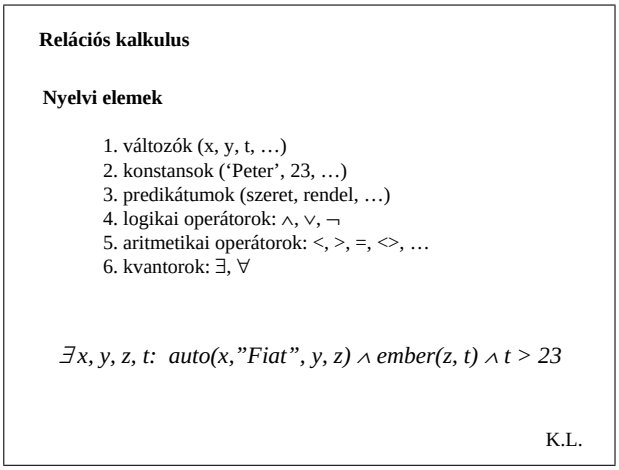
A relációs kalkulus alapja a matematikai logikában megismert predikátum kalkulusok elmélete, a relációs kalkulus a predikátum kalkulusok területének egy részhalmazát foglalja magába. Ennek megfelelően a relációs kalkulus a predikátum kalkulusoknál megismert jelölésrendszert alkalmazza a kifejezések megadásánál. A következőkben áttekintjük e terület legfontosabb alapfogalmait és alkalmazását.

A relációs kalkulusban a felhasználó egy megfelelő struktúrájú kifejezéssel adja meg az eredményrekordokat jellemző tulajdonságot. A rendszer e kifejezést kielégítő rekordokat vesz be az eredményrelációba. A kifejezés nem lehet tetszőleges formátumú, bizonyos strukturális megkötöttségeket kell teljesíteni, hogy értelmezhető legyen. A formális feltételeket teljesítő kifejezést *jól formált formulának* (well formed formula, wff) nevezik. A wff-ek a következő nyelvtani elemeket tartalmazhatják:

- csoportosító zárójelek
- változó szimbólumok (például x, y, z, ...)
- konstans szimbólumok (például 24, 12, 'Peter')
- predikátum szimbólumok (például Szeret, Rendel)
- logikai operátorok ($\wedge$, $\vee$, $\neg$)
- logikai kvantorok ($\forall$, $\exists$)
- reláció operátorok ($=$, $>$, $<$, $\leq$, $\geq$, $\neq$, ...).

A kifejezéseknek tehát a fenti elemekből kell felépülniük, hogy szintaktikailag helyesek legyenek. Az építkezés módjára, vagyis arra nézve, hogy hogyan lehet ezen elemekből a megfelelő kifejezéseket megformálni, a következő szabályok adnak útmutatást.

1. A wff kifejezések lehetnek elemiek és összetettek.
2. Elemi kifejezéseknek az alábbi alakú kifejezéseket tekintjük:



4.52. ábra. A relációs kalkulus nyelvi elemei II.

(a) $P(t_1,...,t_n)$, ahol P egy n paraméterű predikátum szimbólum, t_i pedig lehet változó vagy konstans szimbólum;

(b) $t_1 \ \Theta \ t_2$, ahol Θ egy reláció operátort jelöl, t_i pedig lehet változó vagy konstans szimbólum.

3. Az elemi kifejezésekből az alábbi szabályok alapján képezhetünk összetett kifejezéseket:

(a) Ha F_1 és F_2 helyes kifejezések, akkor

$F_1 \vee F_2$
$F_1 \wedge F_2$
$\neg F_1$
(F_1)

is helyes kifejezések.

(b) Ha F egy helyes kifejezés, melyben előfordul az x változó szimbólum, akkor a

$\exists x \ (F)$
$\forall x \ (F)$

is helyes kifejezések.

A helyes formulák csak a fenti lépések sorozatával képezhetők, más módon nem állítható elő helyes formula.

Azon változókat, amelyekhez nem kapcsolódik kvantor, *szabad változóknak* nevezzük. A kvantorhoz kötődő változók alkotják a kötött változókat. A

$$\forall x \ (\neg P(x,y)) \vee (Q(y) \wedge R(x,y))$$

Relációs kalkulus

Helyes formátumú kifejezések (wff)

- atomi wff:
  1. P (a, b, ...) ahol P predikátum és a, b változó vagy konstans.
  2. a Θ b, ahol a, b változó vagy konstans, és Θ relációs operátor.

- összetett wff:
  1. F1 $\wedge$ F2
  2. F1 $\vee$ F2
  3. $\neg$ F1
  4. (F1)
  5. $\exists$x (F1)
  6. $\forall$x (F1)

K.L.

4.53. ábra. Helyes formátumú kifejezések (wff)

kifejezésben egyetlen kötött változó van, mégpedig az x változó a $\forall$x ($\neg$P(x,y)) tagban. Az y változó viszont szabad a $\forall$x ($\neg$P(x,y)) tagban, ugyanúgy mint az x és y szimbólummal jelölt változók a Q(y) és R(x,y) tagokban.

A logikában minden kifejezéshez logikai igaz vagy hamis igazságértéket lehet rendelni. A logikai érték meghatározása a formulában lévő elemi kifejezések logikai értékétől függ. Az elemei kifejezések mindig rendelkeznek logikai értékkel. A predikátum ugyanis egy logikai állítást jelent, és a relációs operátorok is logikai értéket határoznak meg. Ha ismerjük tehát az elemi kifejezések logikai értékét, akkor az összetett formulák logikai értéke az alábbi szabályok révén határozható meg:

- $W(F_1 \vee F_2) = W(F_1) \vee W(F_2)$
- $W(F_1 \wedge F_2) = W(F_1) \wedge W(F_2)$
- $W(\neg F) = \neg\, W(F)$
- $W(\exists\, x(F)) = W(F|x{=}c_1) \vee \ldots \vee W(F|x{=}c_n)$
- $W(\forall\, x(F)) = W(F|x{=}c_1) \wedge \ldots \wedge W(F|x{=}c_n)$

ahol az utolsó két kifejezésben a helyettesítés végigfut a változó szimbólum összes lehetséges helyettesítési értékén.

A relációs algebra esetén a predikátumok az egyes relációknak feleltethetők meg. Vagyis minden *reláció egy predikátumot reprezentál.* A reláció attribútumai jelentik a predikátum paramétereit. Egy predikátum tehát annyi paraméterrel rendelkezik, ahány attribútuma van a mögötte lévő relációnak. A reláció megvalósulása, a tábla az igaz értékű állításokat fogja össze. Vegyük példaként az alábbi relációt:

Relációs kalkulus

Változók típusa

Egy wff-en belül lehetnek: kötött és szabad változók.
Egy x változó kötött, ha kvantor kapcsolódik hozzá.

1. Egy atomi wff-ben szereplő változó szabad.
2. Az előző ábra 5. és 6. pontjában megadott wff-ben szereplő x változó kötött.

A kötött változók jelentése a mögötte álló részre lokalizált, és más szimbólummal helyettesíthető.

$\exists$y, z: auto(x,"Fiat", y, z) $\exists$t, z: auto(x,"Fiat", t, z)

K.L.

4.54. ábra. A relációs kalkulus változói

ISMERŐS

név	cím
Péter	Eger
Lajos	Dorog
Lajos	Gyöngyös
Tibor	Győr
Zoltán	Szeged
Ferenc	Miskolc

A reláció szemantikailag az ismerősöket tartalmazza. A fenti tábla alapján igaznak tekinthető az az állítás, hogy:

Péter az egyik ismerősöm neve, aki Egerben lakik.

Hasonlóan igaz az is, hogy:

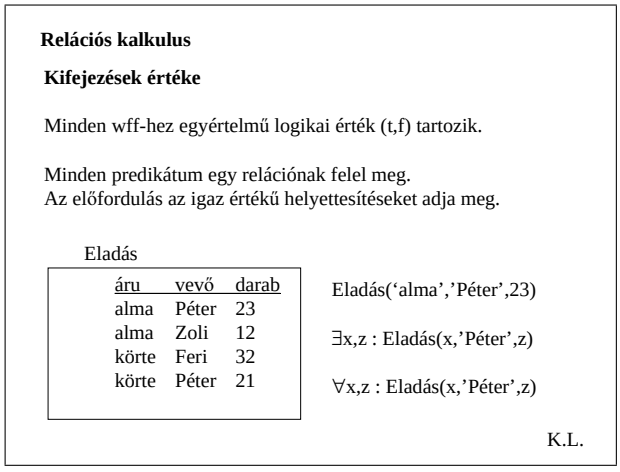
Lajos az egyik ismerősöm neve, aki Dorogon lakik.

A fenti állítást tehát a reláció minden rekordjában szereplő adatokra megismételhetnénk. Így a fenti reláció tulajdonképpen azon konstans változókat fogja össze, amelyekre az

x az egyik ismerősöm neve, aki y városban lakik

teljesül, ahol x és y két változó szimbólum volt. Az x szimbólum a név domainhez, míg az y szimbólum a cím domainhez kapcsolódik. Így az ISMERŐS relációhoz egy

x az egyik ismerősöm neve, aki y városban lakik



4.55. ábra. A relációs kalkulus kifejezéseinek értéke

predikátum kapcsolható, és maga a tábla pedig azon helyettesítéseket foglalja magába, amelyekre a predikátum igaz értéket ad.

Gyakorlásképpen nézzük meg néhány formula logikai értékének kiszámítását. A példák továbbra is az ISMERŐS relációra vonatkoznak. A kifejezésekben a változó szimbólumok neve megegyezik a mögöttük álló attribútum nevével, és a predikátumok neve is azonos a hozzá tartozó reláció nevével.

Az alábbi példa a 'Létezik olyan nevű ismerős, akinek a lakhelye Dorog' állítást fogalmazza meg:

$$W\ (\exists\ név\ (ISMERŐS(név,\ 'Dorog'))) = T\ (igaz)$$

Mivel találhatunk olyan behelyettesítést (Lajos), ahol az ISMERŐS predikátum igaz értéket ad, így a kifejezés értéke is igaz.

A következő formula által megfogalmazott állítás: minden városra teljesül, hogy nem lakik ott Zoltán nevű ismerős és lakik ott Tibor nevű ismerős.

$$W(\forall város\ (\neg\ ISMERŐS\ ('Zoltán',\ város) \wedge ISMERŐS('Tibor',\ város)) = F\ (hamis)$$

A kiértékelésnél, minden lehetséges városnevet be kell helyettesíteni a domainből (ami nagyobb halmaz lehet, mint a táblában szereplő mezőértékek köre). A kifejezés értéke hamis, hiszen például az Eger érték behelyettesítése is hamis értéket ad.

A formulákat azonban nemcsak egy állítás igaz értékének eldöntésére lehet alkalmazni, hanem lekérdezések megfogalmazásra is. A lekérdezéshez szabad változókat használunk fel, és a lekérdezés eredménye a szabad változók azon összetar-

Relációs kalkulus

Lekérdezés

{ x_1,x_2,… | F(x_1,x_2,…) }
ahol x_1,x_2 az F wff összes szabad változója.

Eredménye:
az x_1,x_2,… azon helyettesítési értékei, melyre az F(x_1,x_2,…) igaz értéket ad.

AUTO(rsz, típus, tulaj, ár)

{x | ∃z : Auto(x,'Fiat', 2, z) ∧ z > 23}
{x | Auto(x) ∧ x.ar > 23}

K.L.

4.56. ábra. Lekérdezés relációs kalkulusban

tozó helyettesítési értékei, melyek mellett a megadott formulák igaz értéket adnak. Egy *lekérdezés* tehát

$x_1,...,x_n\ F(x_1,...,x_n)$

alakú kifejezés, melyet

$\{x_1,...,x_n \mid F(x_1,...,x_n)\}$

alakban is szokás jelölni. A lekérdezés eredménye azon $\{e_1,...,e_n\}$ helyettesítési érték n-esek, amelyeket rendre behelyettesítve az $x_1,...,x_n$ változókba, W ($F(x_1,...,x_n)$) igaz értéket szolgáltat. Az előző példánál maradva a

{ név | ∃ város (ISMERŐS (név, város)) }

lekérdezés azon neveket adja vissza, melyekhez létezik városnév az ISMERŐS relációban. Az eredmény reláció a következő lesz:

név
Péter
Lajos
Tibor
Zoltán
Ferenc

A lekérdezés természetesen lehet összetettebb is és vonatkozhat több relációra is. Térjünk vissza egy korábbi példához, amelyben a VEVŐK és RENDELÉS relációk szerepeltek. Legyen a feladat az egyes rendelések darabszámának és a kapcsolódó vevő cégnevének kiíratása. A lekérdezés formátuma:

{ menny, cég | ∃ vid, cím, vevő, termék, dátum (VEVŐK(vid, cég, cím) ∧ RENDELÉS(vevő, termék, menny, dátum) ∧ vevő = vid) }

Az eredményreláció:

cég	menny.
C+C	2
Star	5
Billa	7
Alfa	2
Alfa	1
Agria	2
C+C	5
Vino	1
Star	3

Ez utóbbi lekérdezésnél kicsit zavaróan hatott, hogy milyen bőbeszédűen kellett fogalmazni, hiszen meg kellett adni pontosan mindkét reláció szerkezetét, azaz

szerepeltetni kellett a relációkban szereplő összes attribútumot. Sokkal egyszerűbb lenne a kifejezés, ha elegendő lenne a relációkra hivatkozni. A relációs kalkulusnak erre is van egy változata.

A relációs kalkulus eddig ismertetett formátumát *domain kalkulusnak* nevezik. Az elnevezés azon alapul, hogy a kifejezésekben szereplő változók attribútumokat helyettesítettek, így azok helyettesítési értékei egy domain-hez kapcsolódtak. Ebben az esetben a lekérdezéseket az attribútumok szintjén kellett megfogalmazni. A relációs kalkulus másik változata a *tuple kalkulus*, amelyben a kifejezés változói teljes rekordokat reprezentálnak. Így helyettesítési értékeik rekordok lesznek. Mivel a feltétel megfogalmazásában továbbra is szükség van az attribútumokra vonatkozó megkötésekre, ezért a tuple kalkulus változóit olyan rekordváltozóknak tekintik, amelyeknek lehetnek attribútumai. Egy t tuple változónak pontosan megegyezik a szerkezete az általa reprezentált reláció szerkezetével.

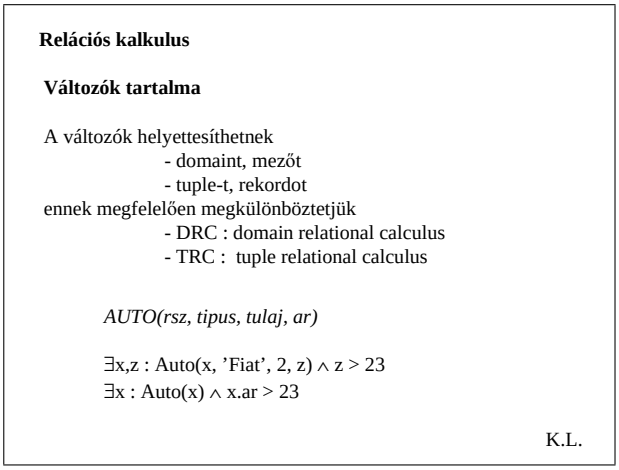
A kétféle megközelítési mód összehasonlítására vegyünk néhány minta lekérdezést, melyet mind domain mind tuple kalkulusban megadunk. A minta lekérdezés egy vállalat ügyosztály és dolgozó nyilvántartásához kapcsolódik. A táblák szerkezete a következő:

DOLGOZÓ (azon, név, beosztás, kor, osztály)
OSZTÁLY (azon, cím, főnök)

A végrehajtandó műveletek megfogalmazása:

- A 30 évnél idősebb dolgozók kódja.
- A bérügyön dolgozók neve.
- Mely beosztások szerepelnek minden osztályon.

Vegyük előbb a tuple kalkulust. Az első lekérdezésnél venni kell a dolgozó relációt, és ki kell választani azon tuple elemeket, melyeknél a kor nagyobb, mint 30.



4.57. ábra. A relációs kalkulus változóinak tartalma

A művelet alakja:

$$\{t.azon \mid dolgozo(t) \wedge t.kor > 30\}$$

A második feladatnál szintén a dolgozóhoz tartozó tuple előfordulásokat keresünk, de itt most a dolgozónak a bérügy osztályon kell dolgoznia, ezért a dolgozóhoz tartozó osztály tuple-ra is meg kell adni egy feltételt. Ehhez be kell hozni egy osztály típusú tuple-t is a lekérdezés kifejezésébe. A lekérdezés alakja:

$$\{t.nev \mid dolgozo(t) \wedge \exists o\ (osztaly(o) \wedge o.azon = t.osztaly \wedge o.cim = \text{'berugy'})\}$$

A harmadik feladatban a dolgozó tuple értékeiből kell kiválasztani azon beosztás értékeket, melyekhez minden osztálynál találunk olyan dolgozót, akinek a beosztása szintén ilyen értékű. A minden és létezik kvantorokat alkalmazva az alábbi alakú lesz a lekérdezés:

$$\{t.beosztas \mid dolgozo(t) \wedge \forall o\ (osztaly(o) \wedge \exists k\ (dolgozo(k) \wedge o.azon = k.osztaly \wedge k.beosztas = t.beosztas))\}$$

Ebben a példában már látszik, hogy az összetett feltételek esetén a kalkulus formalizmusa nem egyszerűbb az algebrai alaknál, hiszen elég korlátozott a megadható kifejezések köre, amire le kell fordítani a szöveges lekérdezést.

A domain kalkulus esetében, a tuple változók helyett mező, attribútum változókat kell venni, és ezen mezőkre kell megfogalmazni a szelekciós feltételeket. Az első művelet megfogalmazása:

$$\{a \mid \exists n,b,k,o\ (dolgozo(a,n,b,k,o) \wedge k > 30)\}$$

Relációs kalkulus

Minta lekérdezések – Tuple kalkulus

dolgozó (azon, név, beosztás, kor, osztály)
osztály (azon, cím, főnök)

A 30 évnél idősebb dolgozók azonosítója.
{ t.azon | dolgozo(t) ∧ t.kor > 30 }

A bérügyön dolgozók neve.
{ t.nev | dolgozo(t) ∧
∃o (osztaly(o) ∧ o.azon = t.osztaly ∧ o.cim = 'berugy')}

Melyik az a beosztás, amelyik minden osztályon előfordul.
{ t.beosztas | dolgozo(t) ∧
∀o (osztaly(o) ∧ ∃k (dolgozo(k) ∧
k.osztaly = o.azon ∧ k.beosztas = t.beosztas)) }

K.L.

4.58. ábra. Példa lekérdezések - Tuple kalkulus

Ebben a felírásban minden olyan mezőt szerepeltetni kell, amelyek az érintett relációhoz kötődnek, hiszen a predikátum a mezőszámmal megegyező paramétert vár. Csak azon változók maradnak szabadok, amelyek értékét eredményként kívánjuk megjeleníteni.

A második lekérdezésnél

$$\{n \mid \exists a,b,k,o,f\ (dolgozo(a,n,b,k,o) \wedge osztaly(o,'berügy',f))\}$$

Mint látható, az összekapcsolás az azonos elnevezésű változók alkalmazásával valósul meg, mely egyben értékegyenlőséget is jelent.

A harmadik művelet viszonylag tömören megadható a következő formában:

$$\{b \mid \forall o\ (\exists f,c\ (osztaly(o,c,f)) : \exists\ a,n,k\ (dolgozo(a,n,b,k,o)))\}$$

Olyan b beosztás mezőértékeket kell visszaadni, melyeknél minden o osztály esetén létezik ide kapcsolódó dolgozó.

Természetesen a fenti műveletek mindegyikét meg lehet fogalmazni a relációs algebra formalizmusával is:

$$\pi_{kod}\ (\sigma_{kor>30}(dolgozo))$$
$$\pi_{nev}\ (\sigma_{cim='berugy'}(osztaly) \bowtie_{osztaly=osztaly.azon}\ dolgozo)$$
$$\pi_{beosztas,osztaly}\ (osztaly \bowtie_{osztaly=osztaly.azon}\ dolgozo) \div \pi_{osztaly}\ (dolgozo)$$

Hogy a kért lekérdezések mind az algebra mind a kalkulus nyelvén leírhatók, az nem csupán a véletlen műve, mivel bebizonyítható, hogy a kétféle leírás - bizonyos feltételek esetén - *ekvivalens* egymással. Ez a megkötés egy nem túl szigorú megszorítást jelent a kalkulusbeli kifejezésekre. A megkívánt feltétel annyit igényel,

Relációs kalkulus

Minta lekérdezések – Domain kalkulus

dolgozó (azon, név, beosztás, kor, osztály)
osztály (azon, cím, főnök)

A 30 évnél idősebb dolgozók azonosítója.
{ a | ∃ n,b,k,o (dolgozo(a,n,b,k,o) ∧ k > 30) }

A bérügyön dolgozók neve.
{ n | ∃ a,b,k,o,f (dolgozo(a,n,b,k,o) ∧
osztaly(o, 'berugy',f))}

Melyik az a beosztás, amelyik minden osztályon előfordul.
{ b | ∀o (∃ f,c (osztaly(o,c,f)) :
∃ a,n,k (dolgozo(a,n,b,k,o))) }

K.L.

4.59. ábra. Példa lekérdezések - Domain kalkulus

hogy a kalkulusbeli kifejezés véges eredményt szolgáltasson. Az ilyen kifejezéseket nevezik *safe kifejezésnek*. Erre a megkötésre azért van szükség, mert míg a relációs kalkulusban mindig véges eredményt kapunk, hiszen véges értékű relációkból indulunk ki és véges eredményt előállító lépéseket hajtunk végre, addig a kalkulusban megadhatók olyan feltételek, melyeket végtelen, nem meghatározott értékek is kielégíthetnek. Példaként nézzük az alábbi egyszerű kifejezést:

$\{t \mid \neg\ dolgozo(t)\}$

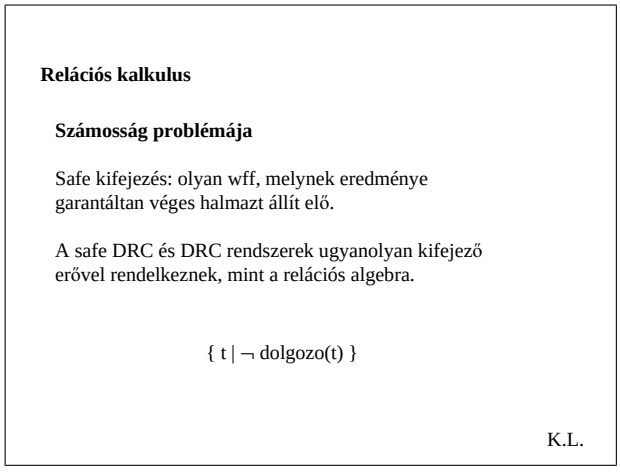
A lekérdezés azon tuple értékeket kéri, melyek nem elégítik ki a dolgozó predikátumot, tehát nem dolgozó táblabeli rekordok. Az ilyen lekérdezések az alapuniverzum függvényében végtelen sok értéket is visszaadhatnak, és a relációs algebrában sem találunk ehhez kapcsolható operátort. Ezért ez a kifejezés nem konvertálható relációs algebrai alakra.

A fentiek figyelembe vételével a kétféle megközelítés ekvivalenciájára vonatkozó megállapítás a következő alakban fogalmazható meg:

> *A safe DRC és TRC rendszerek ugyanolyan kifejező erővel rendelkeznek, mint a relációs algebra.*

Így a safe kalkulusbeli kifejezéseket mindig át lehet alakítani algebrai alakra. A következőkben a tuple orientált kalkulus kifejezéseinek az algebrai alakra történő átalakítási lépéseit tekintjük át. Az átalakítás menete nagyban függ a kalkulusban meglévő operátoroktól, kifejezésektől. Egy általános konverziós formula igen összetett lenne, ezért mi most csak az alap kalkulusbeli kifejezésekre koncentrálunk, hogy a konverzió általános jellegét és menetét egyszerűbben bemutathassuk.

A konverzió általános menete az alábbi lépésekből áll:



4.60. ábra. Safe kifejezések, számosság problémája

1. a kifejezéshez tartozó relációk meghatározása a tuple változók értelmezési tartományainak megkeresésével;
2. az egyedülálló tuple változókra vonatkozó szelekciós feltételek átalakítása a megfelelő relációhoz tartozó szelekcióra;
3. a szelektált relációk összekapcsolása alap join művelettel;
4. a több tuple változót érintő szűkítések megadása a join eredményre vonatkozó szelekcióval;
5. a létezés kvantor előfordulása esetén egy megfelelő projekció elvégzése, mert a létezés esetén nem lényeges a változó értéke csak a létezése (a projekcióban a létezéshez kapcsolódó kötött változó részt kell eliminálni a join eredményből);
6. a minden kvantor előfordulása esetén egy megfelelő osztás elvégzése, mert a minden kvantor esetén nem lényeges a változó értéke csak a létezése (az osztásban a kapcsolódó kötött változó részt kell eliminálni a join eredményből);
7. az eredményből a szabad változókhoz tartozó rész kiemelése projekcióval.

A fenti lépés sorozat eredményeképpen az induló kalkulus alakból egy vele ekvivalens, azaz ugyanazon eredményt előállító relációs algebrai kifejezés hozható létre.

A műveletsor végrehajtásának szemléltetésére vegyünk egy egyszerűbb példát, melyben az induló kalkulus formula két tuple változót tartalmaz:

$$\{t.x \mid A(t) \wedge t.x > 5 \wedge \exists y\ (B(y) \wedge y.z = t.v)\}$$

Mivel két változó szerepel a kifejezésben, és ezek különböző tartományhoz tartoznak, ezért most két alaprelációt kell meghatározni. A változók jele t és y, a relációik A és B. Az első pont utáni eredmény a két reláció:

Relációs kalkulus

A relációs kalkulus konverziója

1. Tuple változók domain-jei → relációk
2. Elemi szelekció → relációra vonatkozó szelekció
3. Relációk Descartes-szorzata
4. Összetett szelekció → Joinra vonatkozó szelekció
5. ∃ kvantor → projekció
6. ∀ kvantor → osztás
7. Projekció

K.L.

4.61. ábra. A relációs kalkulus konverziós lépései

$A,\ B$

A második lépés után az elemi szelekció:

$\sigma_{x>5}\ (A)$

A join művelete:

$\sigma_{x>5}\ (A) \bowtie B$

Az összetett feltétel kijelölése:

$\sigma_{z=v}(\sigma_{x>5}\ (A) \bowtie B)$

A létezik kvantor eliminálása:

$\pi_y(\sigma_{z=v}(\sigma_{x>5}\ (A) \bowtie B))$

Mivel nincs minden kvantor, közvetlenül jöhet az eredmény mezők leválogatása:

$\pi_x\ (\pi_y(\sigma_{z=v}(\sigma_{x>5}\ (A) \bowtie B)))$

4.6.4. Adatkezelő műveletek

A relációk előfordulásai az adatbázis élete során folytonosan változhatnak, azaz új rekordok kerülhetnek be a relációba, létező rekordok kerülhetnek ki a relációból. Emiatt a kezelő nyelvnek a rekordok beszúrásáról, kitörléséről illetve módosításáról is gondoskodnia kell.

A módosítás során a relációban tárolt adatértékek módosulnak, azaz egy új relációelőfordulás fog szerepelni az adatbázisban a régi reláció előfordulás helyett. A módosításnál meg kell adni paraméterként, hogy mely relációban, mely rekordok

Relációs kalkulus

A relációs kalkulus konverziója

$\{t.x \mid A(t) \wedge t.x>5 \wedge \exists y\ (B(y) \wedge y.z = t.v)\}$

1. A, B
2. $\sigma_{x>5}(A)$
3. $\sigma_{x>5}(A) \triangleright\triangleleft B$
4. $\sigma_{z=v}(\sigma_{x>5}(A) \triangleright\triangleleft B)$
5. $\Pi_A(\sigma_{z=v}(\sigma_{x>5}(A) \triangleright\triangleleft B))$
6. -
7. $\Pi_x\ (\Pi_A(\sigma_{z=v}(\sigma_{x>5}(A) \triangleright\triangleleft B)))$

```
SELECT x FROM A, B WHERE x>5 AND z=v;
```

K.L.

4.62. ábra. Példa a relációs kalkulus konverziójára

mely mezőit milyen értékre kell módosítani. A módosításnál nem szabad tehát elfelejteni, hogy egy relációra vonatkozik, de ott egyszerre több rekordot is érinthet. A módosítás jele:

$$\upsilon_{felt}{}^{a_1=kif_1,\ldots,\ a_n=kif_n}(r)$$

ahol r a reláció azonosítója, *felt* egy feltétel, mely kijelöli a módosítandó rekordokat, a_i a módosítandó attribútumokat, és kif_i az új attribútum értékeket adja meg.

Példaként az ISMERŐS táblában változtassuk meg a nevet Lalira ott, ahol korábban Lajos volt. Az utasítás formátuma:

$$\upsilon_{nev='Lajos'}{}^{nev='Lali'}(\text{ISMERŐS})$$

Az eredmény reláció:

ISMERŐS

név	cím
Péter	Eger
Lali	Dorog
Lali	Gyöngyös
Tibor	Győr
Zoltán	Szeged
Ferenc	Miskolc

Mint látható, mindkét Lajos érték átíródott a táblában.

A törlés során a relációból kikerül egy rekord előfordulás. A törléshez meg kell adni a reláció azonosítóját és egy törlési feltételt. A reláció minden olyan rekordja kikerül a táblából, amelyek teljesítik a megadott feltételt. A törlés jele:

$$\delta_{felt}(r)$$

ahol r a reláció azonosítója, *felt* egy feltétel, mely kijelöli a törlendő rekordokat.

Példaként az ISMERŐS táblából töröljük ki a Lali nevet tartalmazó rekordokat. Az utasítás formátuma:

$$\delta_{nev='Lali'}(\text{ISMERŐS})$$

Az eredmény reláció:

ISMERŐS

név	cím
Péter	Eger
Tibor	Győr
Zoltán	Szeged
Ferenc	Miskolc

A beszúrás során a reláció bővül egy rekord előfordulással, azaz egy új relációelőfordulás fog szerepelni a táblában. A beszúrás során meg kell adni a reláció azonosító nevét, és az új rekord adatait. A beszúrás egy relációba tesz be egy új rekordot. A beszúrás szimbóluma:

$$\iota^{a_1=kif_1,\ldots,\ a_n=kif_n}(r)$$

ahol r a reláció azonosítója, a_i az új rekord attribútumait, és kif_i az egyes attribútum értékeket jelölik.

Példaként az ISMERŐS táblába vegyünk fel egy új rekordot, amelyben a név Béla, és a város Gyula. Az utasítás formátuma:

$$\iota^{nev='Bela',cim='Gyula'}(\text{ISMERŐS})$$

Az eredmény reláció:

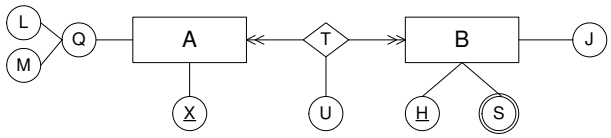
ISMERŐS

név	cím
Péter	Eger
Lajos	Dorog
Lajos	Gyöngyös
Tibor	Győr
Zoltán	Szeged
Ferenc	Miskolc
Béla	Gyula

Az előzőekben bemutatott műveleti résszel lezártuk a relációs adatmodell alapjainak az ismertetését. A relációs modell, mint a legelterjedtebb adatmodell megvalósulása - az egyes RDBMS rendszerek ismertetése és kezelésének bemutatása - előtt még egy elméleti problémára térnénk ki. Ez a probléma a relációs adatbázisok tervezésének a kérdését érinti majd.

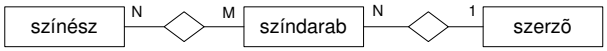
Elméleti kérdések

1. Mikor és hogyan alakult ki a relációs adatmodell?
2. Sorolja fel néhány reprezentánsát a relációs adatbáziskezelő rendszereknek.
3. Jellemezze a relációs adatmodell három fő komponensét.
4. Ismertesse a relációs adatmodell strukturális elemeit.
5. Adja meg a domain fogalmát, szerepét és alkalmazási módját.
6. Ismertesse a reláció fogalmát és tulajdonságait.
7. Miben nyilvánul meg a reláció halmazorientáltsága?
8. Ismertesse az egyes reláció típusokat és felhasználásuk módját.
9. Ismertesse a relációs modell integritási feltételeinek típusait.
10. Milyen lokális integritási feltételek értelmezettek a relációs adatmodellben?
11. Miért lehet szükség a késleltetett érvényesítésű integritási feltételekre?
12. Ismertesse a relációs modellhez kapcsolódó egyed integritási és hivatkozási integritási szabályokat.
13. Mit jelent a NULL kifejezés és mi a használatának jelentősége?
14. Adja meg a relációséma és a reláció formális felírását.
15. Ismertesse a relációs integritási elemek formális megadását.
16. Ismertesse az ER kapcsolattípusok átalakítását a relációs modellbe.
17. Mutassa be, hogy lehet az ER modell tulajdonságtípusait ábrázolni a relációs modellben.
18. Adja meg az alábbi sémához kapcsolható integritási feltételeket:
    PILÓTA (pkód, név, kor, beosztás); JÁRAT(cél, jkód, dátum, pilóta).
19. Ismertesse az EER modell elemeit (jelölés és jelentés), és az elemek relációs modellbeli megfelelőit.
20. Adja meg a tulajdonságok lehetséges típusait, és ábrázolásukat az ER, a relációs, és a hierarchikus modellben.
21. Az egyedek közti kapcsolatok típusai és ábrázolásuk az ER, a relációs és a hierarchikus adatmodellben.
22. Osztályozza a relációs adatmodellben definiált integritási feltételeket.
23. Milyen domain-ek fordulnak elő az alábbi ER sémához tartozó relációs modellben?



24. Hogyan ábrázolható az N:M kapcsolat a relációs modellben?
25. Hogyan ábrázolható az 1:N kapcsolat a relációs modellben?
26. Hogyan ábrázolható a többértékű tulajdonság a relációs modellben?
27. Hogyan ábrázolható a kötelező kapcsolat a relációs modellben?
28. Hogyan ábrázolható a gyenge egyed a relációs modellben?
29. Hogyan ábrázolható az 1:1 kapcsolat a relációs modellben?
30. Hogyan ábrázolható a specializáció a relációs adatmodellben?
31. Hogyan ábrázolható egy n-es kapcsolat relációs modellben?

32. Egészítse ki tulajdonságokkal az alábbi ER modellt, és konvertálja át relációs modellre.



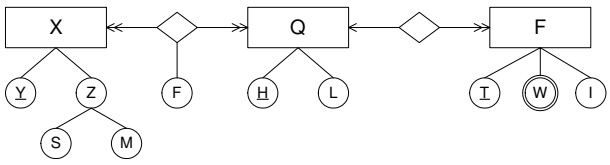
33. Végezze el a TABLE, VIEW és SNAPSHOT összehasonlítását, és adja meg kezelésük jellemzését.
34. Ismertesse az EER modell kapcsolati elemeinek konverzióját a relációs adatmodellre.
35. Jellemezze a relációs algebrát, jelentőségét és előnyeit.
36. Adja meg a relációs algebra egy operandusú műveleteinek jelölését és jelentését.
37. Adja meg a relációs algebra több operandusú műveleteinek jelölését és jelentését.
38. Adja meg a projekció értelmezését, jelölését és formális felírását.
39. Adja meg a szelekció értelmezését, jelölését és formális felírását.
40. Adja meg az alap és szelekciós join értelmezését, jelölését és formális felírását.
41. Adja meg az outer és szemi join értelmezését, jelölését és formális felírását.
42. Adja meg a csoportképzés értelmezését, jelölését és formális felírását.
43. Adja meg az aggregáció és kiterjesztés értelmezését, jelölését és formális felírását.
44. Adja meg az osztás értelmezését, jelölését és formális felírását.
45. Adja meg a relációs algebrai halmazműveletek értelmezését, jelölését és formális felírását.
46. Adja meg az adatkezelő műveletek értelmezését és működését.
47. Ismertesse a relációs algebra és kalkulus kapcsolatát.
48. Mutassa be a csoportképzés és aggregáció részletes végrehajtási lépéseit.
49. Mutassa be a join műveletének különböző típusait a relációs algebrában.
50. Hogyan értelmeztük a lekérdezést a relációs kalkulusban.
51. Adja meg a szelekció, projekció, és a join algebrai műveletek relációs kalkulusbeli megfelelőit.
52. Adja meg az alábbi relációs algebrai műveletsor relációs kalkulusbeli megfelelőjét:

$$\pi_{\ A.B,Z.D}\ (\sigma_{A.B<3\ \ AND\ \ A.K=Z.H}\ (\mathrm{A} \bowtie \mathrm{Z}))$$

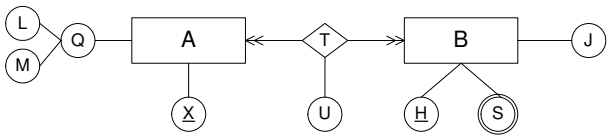
53. A wff kifejezés jelentése és definíciója, szerepe a relációs kalkulusban.
54. Relációs kalkulus leírása, a kalkulus és algebra kapcsolata. Konvertálja át a $\pi_c(\sigma_{t.a=s.k}(\mathrm{t} \bowtie \mathrm{s}))$ kifejezést.
55. Mit jelent a safe kifejezés a relációs kalkulusban és mi a jelentősége?
56. Mi a relációs algebra és kalkulus kifejező erejének kapcsolata?
57. Milyen lépéseken keresztül konvertálható egy kalkulus kifejezés relációs algebrai alakra?
58. Írjon fel egy szelekciós join-t kalkulus alakban.
59. Írjon fel egy szelekciót kalkulus alakban.
60. Írjon fel egy projekciót kalkulus alakban.
61. Írjon fel egy outer join műveletet kalkulus alakban.
62. Milyen típusú kalkulus alakok vannak és a mi kapcsolatuk?
63. Vezesse le az osztást más algebrai műveletekre.

Feladatok

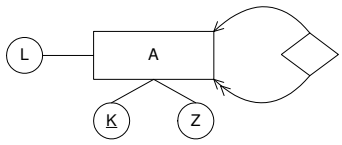
*1. Hozza létre a relációs modellt az alábbi ER sémához:



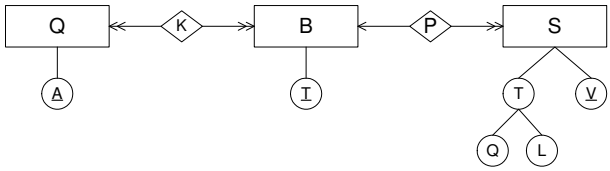
2. Hozza létre a relációkat, táblákat az alábbi ER sémához:



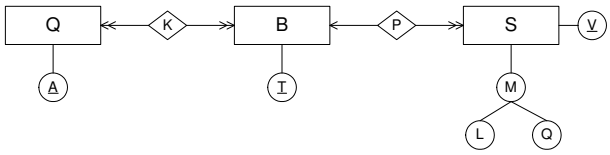
3. Hozza létre a relációkat, táblákat az alábbi ER sémához:



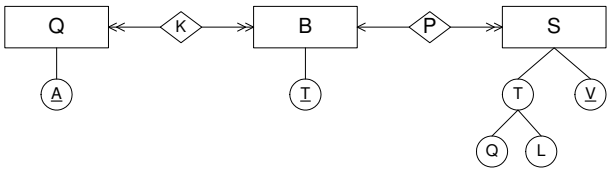
4. Konvertálja az alábbi ER sémát relációs modellre:



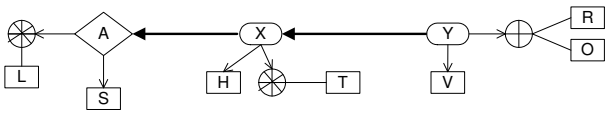
5. Konvertálja az alábbi ER sémát relációs modellre:



6. Konvertálja az alábbi ER sémát relációs modellre:



*7. Konvertálja át az alábbi IFO sémát relációs adatmodellre:



8. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a relációk: **AUTÓ** [RSZ, TÍPUS, SZÍN, ÁR, TULAJ] és **EMBER** [KÓD, NÉV, VÁROS].
   - A piros autók tulajdonosai.
   - A 100000 forintnál drágább autók darabszáma.
   - Az átlagnál drágább autók tulajdonosai.
9. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].
   - A könyvek címe.
   - A kiadók neve és a kiadott könyvek átlagára.
   - Azon szerzők neve, akiknek 5-nél több könyve van.
   - Az átlagnál drágább könyvek szerzőinek neve.
10. Adja meg a tuple relációs kalkulusbeli lekérdezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].
    - Könyvek címe és a kiadó neve.
    - Kiadók neve és kiadott könyvek darabszáma.
    - Azon szerzők neve, akiknek 5-nél több könyve van.
11. Adja meg a következő műveletek relációs algebrai megfelelőjét, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].
    - A 2000 forintnál olcsóbb könyvek ISBN száma és címe.
    - Az átlagnál olcsóbb könyv(ek) címe és a kiadó neve.
    - Mely szerzők adtak ki az UNIVERSUM nevű kiadónál.
12. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

- A könyvek átlagára.
- Mely szerzők nem jelentettek meg könyvet az UNI kiadónál.
- A kiadók és könyveik összára.

13. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

    - Könyvek címe, a kiadó neve és a szerző neve.
    - A 2000 forintnál olcsóbb könyvek szerzőinek neve.
    - A szerzők és könyveik darabszáma.

*14. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

    - Könyvek címe és szerző neve.
    - Mely szerzőknek nincs könyve.
    - Az UNIVERSUM kiadónál megjelent könyvek átlagára.
    - Azon szerzők neve, akiknek 5-nél több könyve van.

*15. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM, KOR].

    - A kiadók neve és könyveik átlagára.
    - Az átlagnál fiatalabb szerzők könyvei.
    - Mely szerző adott ki minden kiadónál.

*16. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

    - Mely szerzők nem adtak ki 2000 forintnál olcsóbb könyvet.
    - A szerzők neve és könyveik darabszáma (akinek nincs ott 0 álljon).
    - A legtöbb könyvet írt szerző neve.

17. Adottak az **OKTATÓ** [ID, NÉV, FOKOZAT] és **TÁRGY** [TID, CÍM, OKTATÓ, ÓRASZÁM] relációk. Adja meg a relációs algebrai formulákat az alábbi lekérdezésekhez:

    - A tárgyak címe.
    - A 8 órás tárgyak címe és oktatójuk neve(i).
    - Milyen fokozatok vannak és hány oktató tartozik oda.
    - Azon oktatók, akiknek 4-nél több tárgyuk van.
    - Azon 'H' fokozatú oktatók darabszáma, akiknek összóraszáma kisebb, mint 20.

18. Adottak a **ZENESZÁM** [KÓD, CÍM, TÍPUS, ELŐADÓ] és **ELŐADÓ** [KÓD, NÉV, LAKCÍM] relációk, adja meg a következő műveletekhez a relációs algebrai lekérdezést:

    - A miskolci előadók számainak címe.
    - Hány zeneszám van az egyes típusokból.
    - Mely városokbeli előadóknak nincs 'népdal' típusú számuk.

19. Adott **DOLGOZÓ** [KÓD, NÉV, FIZETÉS, BEOSZTÁS, ÜZEMKÓD] és **ÜZEM** [KÓD, NÉV] relációkhoz adja meg a következő lekérdezések relációs algebrai alakját:
    - Dolgozók összlétszáma.
    - A 100000 forintnál többet keresők neve.
    - Dolgozók neve és üzemük neve.
    - Mennyi az egyes beosztásokban az átlagfizetés.
    - Az átlagnál többet keresők neve.

*20. Adottak az alábbi relációk: **VERSENYZŐ** [VKÓD, NÉV, KOR, CSAPAT] és **CSAPAT** [ID, NÉV, CÍM]. Adja meg a megfelelő relációs algebrai kifejezést:
    - Versenyzők neve és csapatuk neve.
    - A nem a SASOK csapatban játszó játékosok átlagéletkora.
    - Azon játékosok, akik idősebbek csapatuk átlagéletkoránál.

21. Adottak a **DOLGOZÓ** [KÓD, OSZTÁLY, BEOSZTÁS, FIZETÉS, NEM] és **OSZTÁLY** [OKÓD, NÉV, CÍM] relációk. Adja meg a lekérdezéseket relációs algebrában:
    - Mely osztályokon dolgoznak Operátor beosztású dolgozók.
    - A női dolgozók aránya az egyes osztályokon.
    - Mely beosztásokban nagyobb az átlagfizetés 150000-nél.

22. Adottak a **DOLGOZÓ** [KÓD, OSZTÁLY, BEOSZTÁS, FIZETÉS, NEM] és **OSZTÁLY** [OKÓD, NÉV, CÍM] relációk. Adja meg a lekérdezéseket tuple relációs kalkulusban:
    - Mely osztályokon dolgoznak Operátor beosztású dolgozók.
    - A női dolgozók aránya az egyes osztályokon.
    - Mely beosztásokban nagyobb az átlagfizetés 150000-nél.

23. Adottak a **DOLGOZÓ** [KÓD, OSZTÁLY, BEOSZTÁS, FIZETÉS, NEM] és **OSZTÁLY** [OKÓD, NÉV, CÍM] relációk. Adja meg a lekérdezéseket domain relációs kalkulkusban:
    - Mely osztályokon dolgoznak Operátor beosztású dolgozók.
    - A női dolgozók aránya az egyes osztályokon.
    - Mely beosztásokban nagyobb az átlagfizetés 150000-nél.

24. Adottak az alábbi sémák: **TANFOLYAM** [TID, NÉV, TÍPUS, FELELŐS] és **TÁRGY** [ID, CÍM, ÓRASZÁM, TANFOLYAMID]. Adja meg a relációs algebrai alakot a lekérdezésekhez:
    - Tárgyak átlagos óraszáma.
    - Tárgyak címe és tanfolyamuk neve.
    - Tanfolyamtípusok, és hány tanfolyam tartozik oda.
    - A 60 óránál rövidebb összóraszámú tanfolyamok.
    - A legtöbb tárggyal rendelkező tanfolyam neve.

*25. Adja meg a tuple relációs kalkulus kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

   - Könyvek címe, kiadó neve és a szerző neve.
   - A 2000 forintnál olcsóbb könyvek szerzőinek neve.
   - A szerzők és könyveik darabszáma.

*26. Adja meg a domain relációs kalkulus kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

   - Könyvek címe, kiadó neve és a szerző neve.
   - A 2000 forintnál olcsóbb könyvek szerzőinek neve.
   - A szerzők és könyveik darabszáma.

27. Adja meg a domain relációs kalkulus kifejezéseket az alábbi műveletekhez, ha a relációk: **AUTÓ** [RSZ, TÍPUS, SZÍN, ÁR, TULAJ] és **EMBER** [KÓD, NÉV, VÁROS].

   - A piros autók tulajdonosai.
   - A 300000-nél olcsóbb autók darabszáma.
   - Az átlagnál drágább autók tulajdonosai.

28. Adja meg a tuple relációs kalkulus kifejezéseket az alábbi műveletekhez, ha a relációk: **AUTÓ** [RSZ, TÍPUS, SZÍN, ÁR, TULAJ] és **EMBER** [KÓD, NÉV, VÁROS].

   - A piros autók tulajdonosai.
   - A 300000-nél olcsóbb autók darabszáma.
   - Az átlagnál drágább autók tulajdonosai.

5. fejezet

Az SQL nyelv alapjai

5.1. Általános áttekintés az SQL nyelvről

Az előző fejezetben a relációs adatmodell alapjait jelentő strukturális, műveleti és integritási részeket vettük át. A következő fejezetben eljutunk odáig, hogy a megtervezett adatbázisokat ténylegesen is megvalósíthassuk egy relációs adatbáziskezelő rendszerben. Ehhez meg kell ismerni azt a parancsnyelvet, amelyet a relációs adatbáziskezelő rendszerek mindegyike megért, nevezetesen az SQL szabvány adatkezelő nyelvet.

Mint már korábban említettük, a relációs adatbáziskezelők a mini gépek kategóriájából fejlődtek ki. Az első RDBMS, a System/R kidolgozásánál fontos szempont volt, hogy minél tökéletesebben valósítsák meg Codd elveit: a relációs adatmodell és a relációs algebra elemeit. A kezelő felület esetében tehát olyan parancsnyelvet kellett kidolgozni, amely lehetővé teszi a relációknak, mint rekord halmazoknak a kezelését, ahol a lekérdezésekben deklaratív formában lehet megadni a kívánt eredményreláció tulajdonságát, felhasználva a relációs algebra ismert műveleteit, például a szelekciót, a projekciót vagy az összekapcsolást.

Ez a kezelő felület lényegesen eltér a hagyományos rekordorientált, ciklusokat és elágazásokat tartalmazó lekérdező felületektől. A System/R-ben megvalósított, a relációs algebrán alapuló kezelő felületet *SEQUEL*-nek nevezték el, utalva e nyelv néhány alapvonására. A SEQUEL, mint rövidítés a Structured English Query Language kifejezésre utal, azaz jelzi, hogy ez egy

- *strukturált*,
- *angol nyelvre* épülő,
- *lekérdező* nyelv.

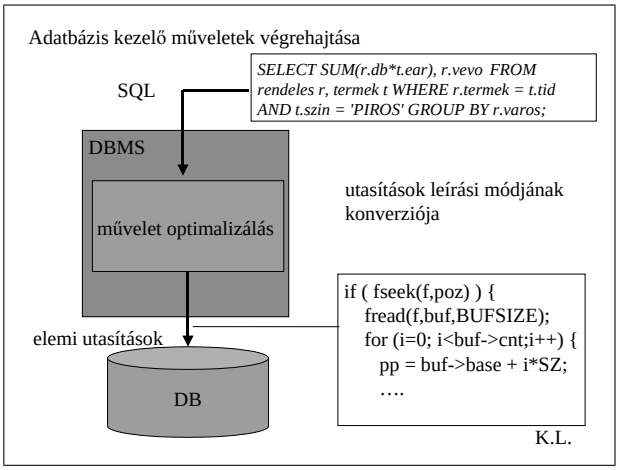
A parancsok kulcsszavai értelmes, a művelet jelentéséhez közel álló angol szavak, melyekből a parancsok mint mondatok hozhatók létre. Természetesen igen szigorúan kötött e mondatok szerkezete, emiatt is nevezhető strukturált nyelvnek. A lekérdezés szó azért szerepel a rövidítésben, mert igaz, hogy ez a nyelv többre is alkalmas, mint puszta lekérdezésre, azonban ennek a nyelvnek az igazi ereje a lekérdezési részben, a relációs algebrára épülő komponensben rejlik. A nyelv részleteit

elsőként 1974-ben publikálták. Az elnevezés a későbbiekben *SQL*-re (Structured Query Language) változott, de a nyelv felépítése, működési filozófiája változatlan maradt.

A System/R után következő RDBMS rendszerek a belső működési elveken kívül ezt a kezelő nyelvet is átvették, megőrizve a kulcsszavakat és az operátorokat is. Emellett természetesen voltak egyéni, egyedi megvalósítások is, mint például a DEC RDB kezelő nyelve, amelyek más szintaktikára épültek, azonban ezek a nyelvek idővel mind eltűntek, nem bírták a versenyt az SQL nyelvvel szemben. Az *SQL* lassan egyeduralkodóvá vált az RDBMS piacon, ezért az igazi adatbáziskezelő szolgáltatásokat nyújtó rendszereket szokás *SQL adatbázisoknak* is nevezni. Az SQL elterjedésének és előnyeinek köszönhetően *szabvánnyá* vált a relációs adatbáziskezelők világában, ami miatt elterjedése méginkább fokozódik, és lassan kiszorít minden más kezelő felületet.

Az SQL első *szabványosítási* lépése az ANSI szervezet nevéhez és az 1986-os évhez kötődik. A rákövetkező évben a nemzetközi szabványhivatal, az ISO is átvette és megjelent az ISO SQL szabvány. A gyakorlati élet fejlődéseit követve többször, pontosabban napjainkig kétszer módosították a szabványt, először 1989-ben, majd utána 1992-ben. Az egyes SQL szabványok megkülönböztetésére meg szokták adni az évszámát is, és így hivatkozhatunk *SQL86*-ra, *SQL89*-re, vagy éppen *SQL92*-re.

Az SQL szabványok valójában a piacon megjelenő RDBMS-ek kezelő felületeit kívánják mederben tartani, vagy éppen követni. Az SQL86 éppenséggel csak a legfontosabb elemeket gyűjtötte össze a megvalósulásokból, ezért már a megalkotásakor is több olyan RDBMS volt, melynek kezelő nyelve többet tudott a szabványnál. Az azóta készült RDBMS megvalósítások egyre többet nyújtanak a műveletek, az integritási feltételek és a vezérlő utasítások terén is. A lemaradást igyekeztek bepótolni az újabb SQL89 szabvánnyal, az SQL92-t pedig már előre-



5.1. ábra. Az SQL szerepe az relációs adatbázis rendszerekben

mutató céllal készítették. Az SQL92 egy sor olyan elemet is tartalmaz, amelyet a legfejlettebb RDBMS-ek is csak a közeljövőben tudnak megvalósítani. Éppen ezért az SQL92 szabvány különböző megvalósulási fokokat állapít meg:

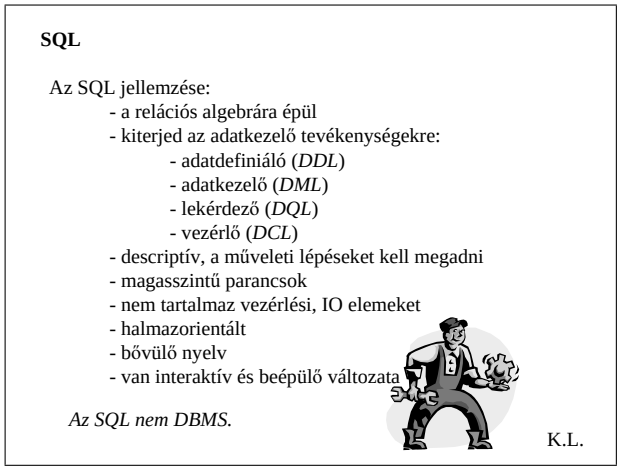
- *Entry level*: alapszint, ami az SQL89 kiegészítve a kulcsra és értéktartományra vonatkozó integritási feltételekkel.
- *Intermediate level*: közbenső szint.
- *Full level*: a teljes szint, mely minden elemet tartalmaz.

Az egyes SQL szabványokról elmondható, hogy viszonylag kompatibilisek egymással alulról felfelé. Így az SQL86 ismerete felhasználható például az SQL92-ben is, ugyanis az újabb szabványok a régi elemeket rendszerint változatlanul hagyják, a változások az újabb elemek befűzéséből erednek. Az SQL szabványokra épülő, a piacon megjelenő adatkezelő nyelveket szintén SQL nyelveknek nevezik, azonban tudnunk kell, hogy a megvalósított nyelvek a szabványos elemek mellett egyedi elemekkel is rendelkeznek, hiszen az SQL szabvány átfogó jellege ellenére több olyan dologra nem tér ki, melyek viszont a megvalósított rendszerekben szükségesek, mint például az adattípusok fajtái és jelölésének kérdése.

Így valójában egy konkrét RDBMS kezelő nyelvének használatához három szintet kell bejárni. Mindenek előtt ismerni kell a relációs modellt, a relációs algebrát. Emellett ismerni kell az SQL szabványt, majd legvégül meg kell ismerkedni a kiválasztott RDBMS SQL alapú kezelő nyelvével. Ha ezt az utat követjük, akkor egy másik SQL alapú rendszerre való átállást igen hatékonyan és gyorsan megoldhatjuk.

Az SQL részletezése előtt azonban célszerű előbb az SQL általános leírását, jellemzését áttekinteni illetve összefoglalni. Mint már az előzőekből kiderült

az SQL a relációs adatmodellen alapuló adatbázisok kezelő nyelve, méghozzá szabványosított nyelve.



5.2. ábra. Az SQL jellemzése

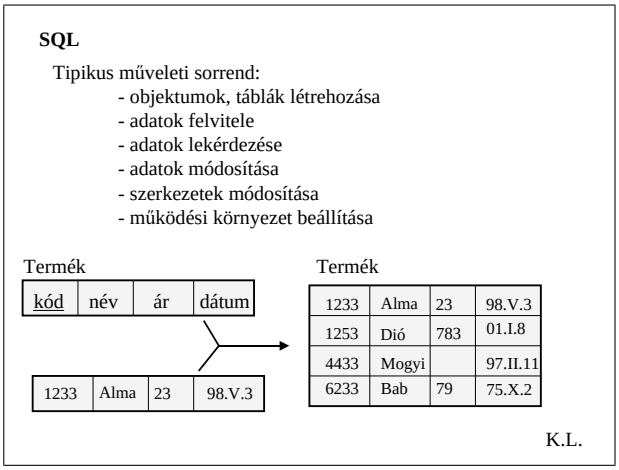
Ebből egyrészt következik az, hogy az SQL-t nem tekinthetjük adatbáziskezelő rendszernek, hiszen az SQL annak csak egy komponense, a kezelő nyelvezete. Egy RDBMS az SQL mellett rendelkezhet más kezelő nyelvezettel is, mint például az RDB rendszernek (DEC VMS alapon futó RDBMS) volt SQL alapú és saját kezelő nyelve is.

A másik lényeges jellemzője az SQL-nek, hogy kimondottan az adatbázis kezelés, az adatkezelés megvalósítására szolgál, tehát nem tartalmaz algoritmikus elemeket, például ciklusszervezést vagy elágazásokat, illetve nincsenek benne a felhasználói képernyőkezelésre, a normál állomány kezelésre vonatkozó utasítások. Emiatt mondhatjuk, hogy az SQL *nem algoritmikus nyelv*, ellentétben a C vagy a Pascal nyelvvel. Az adatkezelésnél az SQL a relációs algebrára épül, tehát a relációkat halmazokon végzi, és az SQL-ben egész relációkra vonatkozó műveletek adhatók ki. Mint ismert, a hagyományos programozási nyelvek a rekordorientált adat megközelítést használják. E különbség kihangsúlyozására nevezik az SQL-t *halmazorientált* nyelvnek.

Az SQL és a relációs adatmodell kapcsolatára megemlíthetjük, hogy az SQL több különböző szinten keresztül közelíti meg az elméleti relációs adatmodellt, amiben benne van, hogy bizonyos elemeket nem tartalmaz, ezért az SQL a relációs adatmodellnek csak részleges leképzése.

Az adatkezelési funkciókon belül az SQL-nek a lehetséges legszélesebb igényhalmazt kell kielégítenie, hogy az SQL önmagában használható legyen, ne kelljen mellé más kiegészítő nyelv.

Az eddig ismert adatkezelési funkciókra hivatkozva az SQL-ben is elvégezhetőnek kell lennie az adatszerkezetek definiálásának, ami megfelel a már általánosságban említett *DDL* (Data Definition Language) nyelvi komponensnek, amelynek segítségével létrehozhatók és módosíthatók többek között a különböző relációk, domainek és integritási feltételek. Mivel a létező RDBMS-ekben a relációk el-



5.3. ábra. Tipikus műveleti sorrend

nevezésére inkább a táblázat kifejezés honosodott meg, ezért a továbbiakban, ha az RDBMS-ben fizikailag tárolt relációra gondolunk, akkor a táblázat elnevezést fogjuk használni.

A DDL mellett az SQL része az adatkezelő, úgynevezett *DML* (Data Manipulation Language) nyelv is, amelyben a táblázatokban tárolt adatok módosítása, törlése vagy új adatok felvitele végezhető el.

Ugyan legtöbbször az adatok lekérdezését is ebbe a körbe veszik, de az SQL-ben és a relációs modellben betöltött fontossága miatt külön csoportot alkotnak a lekérdező utasítások, amelyre a Query elnevezést használjuk. E műveletcsoport jele *DQL* (Data Query Language).

A megvalósított RDBMS kezelő nyelvek hatásának köszönhetően az SQL tartalmaz a relációs adatmodellhez szorosan nem kötődő utasításokat is, melyekkel a műveletek végrehajtását szabályozhatjuk, vezérelhetjük. Emiatt a negyedik utasításcsoportot adatvezérlő csoportnak nevezik, melynek jelölése a *DCL* (Data Control Language).

Az SQL utasításokat több különböző módon is eljuttathatjuk az RDBMS-hez. A legegyszerűbb módszer egy *interaktív SQL parancsértelmező* használata, mely közvetlenül kapcsolódik az RDBMS-hez. Ebben a megjelenő prompt után kiadhatjuk az SQL parancsot, mely rögtön végre is hajtódik, és az eredményt a terminálra kiírva kapjuk meg. Mivel így csak adatkezelő utasításokat adhatunk meg és a feltétel, hogy a felhasználó ismerje az SQL-t, ez a módszer nem alkalmas normál alkalmazói programok készítésére. Ehhez az SQL utasításokat algoritmikus elemekkel kell kibővíteni, melyre kétféle lehetőség is kínálkozik. Egyrészt magát az SQL-t lehet egy létező algoritmikus nyelvbe beépíteni, beágyazni, másrészt az SQL-t lehet kibővíteni algoritmikus elemekkel. Természetesen ez utóbbi esetben a kapott nyelv már igen messze esik az SQL szabványtól. A valóságban mindkét eset előfordul.

SQL

Utasítások csoportosítása

DDL:	- objektum létrehozás	CREATE
	- objektum megszüntetés	DROP
	- objektum módosítás	ALTER
DML:	- rekord felvitel	INSERT
	- rekord törlés	DELETE
	- rekord módosítás	UPDATE
DQL:	- lekérdezés	SELECT
DCL:	- védelem	GRANT, REVOKE
	- tranzakció kezelés	COMMIT, ROLLBACK

K.L.

5.4. ábra. SQL utasítások csoportosítása

Összefoglalóan a következőket állapíthatjuk meg az SQL-ről:

- relációs DBMS kezelő nyelv szabvány, nem RDBMS;
- relációs algebrán alapszik;
- szöveges, nem algoritmikus, előíró jellegű utasításokat tartalmaz;
- halmazorientált;
- négy utasításcsoportot tartalmaz:
  - adatdefiníciós
  - adatlekérdező
  - adatkezelő
  - adatvezérlő
- az utasítások kiadhatók interaktívan és algoritmikus környezetbe építve.

Az SQL általános áttekintése után az SQL utasításokat vesszük sorra, mégpedig elsőként az egyszerűbb SQL89 szabványt, mivel sok esetben már ez is elegendő erővel rendelkezik a kívánt műveletek végrehajtásához kiadásához, köszönhetően a kompatibilitásnak, és egyszerűsége miatt könnyebben elsajátítható. Az SQL89 ismertetése után röviden összefoglaljuk az SQL92 legfontosabb változásait és lehetőségeit.

5.2. Az SQL szabvány DDL utasításai

Elsőként a DDL komponenst vesszük át, hiszen az alkalmazások során is előbb meg kell alkotni a struktúrákat, az üres adatszerkezeti elemeket, hogy a későbbiek folyamán fel tudjuk tölteni azokat adatokkal, illetve fel tudjuk használni a bennük letárolt adatokat. A relációs adatmodell esetére vonatkoztatva ez azt jelenti, hogy előbb létre kell hoznunk a táblázatokat: tehát üres, azaz adatok nélküli táblázatok jönnek létre. Szemléletesen kifejezve úgy is mondhatnánk, hogy elsőként a táblázatok fejlécét alkotjuk meg, a táblázat sorait csak későbbi műveletekkel hozzuk létre.

A táblázat szerkezete, *sémája*, a már ismert, a táblázathoz tartozó mezőkkel írható le. Azaz, ha megadjuk, hogy milyen mezőkből épül fel a táblázat, akkor ezzel egyértelműen megadjuk a táblázat szerkezetét is, tehát két táblázat szerkezete különbözik egymástól, ha található olyan mező, mely az egyikben benne van, és a másikban nincs. A mezők megadása pedig a mező *nevének* és a mező *adattípusának*, valamint az *integritási feltételeknek* a kijelölésével történik. Mivel a szerkezetek megadása nem azonosít egyértelműen egy táblázatot, hiszen több táblázat is létezhet ugyanazzal a szerkezettel, másrészről a szerkezet leírása igen hosszadalmas, ezért minden táblázat kap egy *egyedi azonosító nevet* az adatbázison belül. Ezzel a névvel egyértelműen lehet azonosítani a táblázatokat a műveletek során. A táblázat nevének tehát az adatbázison belül, a mezőnévnek pedig a táblázaton belül kell egyedinek lennie. Összefoglalva: a táblázat létrehozásakor meg kell adni a táblázat nevét valamint az őt alkotó mezők nevét, típusát és érték megkötéseit.

Mivel a relációs modellnek az integritási feltételek is szerves részei, és az integritási feltételek szorosan kötődnek a táblázatokhoz, ezért a táblázatok létrehozásakor

megadott paraméterek részei a hozzájuk kapcsolódó integritás feltételek definiálásának. Az induló SQL86 szabvány nem tartalmazta az összes általunk említett integritási feltételt, azok együttesen csak a későbbi szabványokban jelentek meg. Így itt még nincs lehetőség sem a kulcs, sem a kapcsolókulcs integritási feltételek megadására. Az átmeneti helyzetet jól mutatja, hogy a korábbi RDBMS változatok már elfogadták a PRIMARY KEY integritási feltételt, azonban figyelmen kívül hagyták ezt a megkötést, azaz nem tudták biztosítani az elsődleges kulcs feltétel ellenőrzését.

A továbbiakban az alábbi két táblán mutatjuk be az utasításokra vonatkozó példákat:

AUTO

TUL	RSZ	TIP	SZIN	EVJ	AR
1	bkx720	Opel	bordo	1991	1000000
1	cmt111	Golf	piros	1981	350000
2	aaa156	Trabant	feher	1985	100000
3	lui999	Opel	kek	1991	450000
1	kjs234	Lada	kek	1989	275000

EMBER

ID	NEV	SZULEV	CIM
1	Bela	1975	Budapest
2	Geza	1979	Miskolc
3	Feri	1974	Pecs

Az SQL utasításait az angol nyelvhez igazítva rögzítették le, ennek megfelelően az adatszerkezetek létrehozásának utasítása a CREATE utasítás. A *táblázat létrehozása* a következő utasítással lehetséges:

CREATE TABLE táblázatnév (m_1 t_1 [i_1][,..., m_i t_i [i_i]],[i_g]);

Az SQL utasításokat mindig pontosvessző határolja, zárja le. Az utasítás több soron keresztül is folytatódhat, csak a pontosvessző jelzi az utasítás végét. Az utasítás leírásában a következő jelölések fordulnak elő:

m_i: mezőnév,
t_i: típus,
i_i: mezőhöz kötött integritási feltétel,
i_g: mezőcsoporthoz kötött integritási feltétel.

Az integritási feltételt lehet mezőhöz illetve táblázathoz kötve is megadni. A mezőhöz kötött integritási feltétel a következő elemeket tartalmazhatja:

PRIMARY KEY : elsődleges kulcs, azaz a mező egyedi, nem üres és erre az értékre hivatkoznak majd az idegen kulcs mezők.

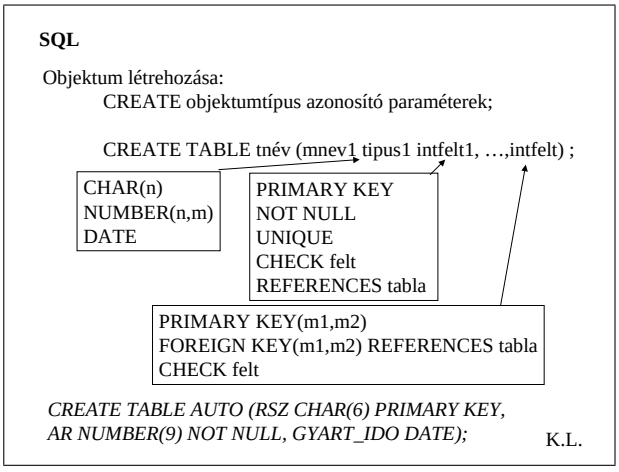
REFERENCES tábla: idegen kulcs, mely a megadott táblára hivatkozik; értéke vagy üres, vagy létező kulcsérték.
UNIQUE : a mező egyedi értékeket tartalmaz, azaz egy érték nem fordulhat elő egynél több rekordban.
NOT NULL : a mezőnek tartalmaznia kell értéket, nem maradhat üres egyetlen egy rekordban sem.
CHECK feltétel : általános érték ellenőrzési feltétel, melyben a mezőre a mező nevével hivatkozhatunk.
DEFAULT érték : alapértelmezési érték, ha a mező nem lenne kitöltve az új rekord beszúrása során.

Egy mezőre több integritási feltétel is megadható, ekkor egymás után írjuk az egyes integritási elemek kulcsszavait. A rekord szintű, mezőcsoportra vonatkozó integritási feltétel esetében külön meg kell adni, hogy a feltétel mely mezőkre vonatkozik. A mezőneveket rendszerint a kulcsszót követő zárójelben adhatjuk meg:

PRIMARY KEY($m_1,\ldots,m_n$) : összetett elsődleges kulcs.
UNIQUE($m_1,\ldots,m_n$) : egyedi értékek.
CHECK feltétel : általános érték ellenőrzés.
FOREIGN KEY($m_1,\ldots,m_n$) *REFERENCES tábla* : összetett idegen kulcs.

Az adattípusok esetében sajnos nem lehet szabványosított hivatkozást megadni, mivel a legtöbb elterjedt RDBMS rendszer más-más elnevezést használ. Mi most csak a legfontosabb alaptípusokat vesszük át a következő jelölésekkel:

CHAR (n) : n hosszúságú szöveg.
NUMBER (n [,m]) : n hosszú szám, ahol m a tizedesjegyek száma.



5.5. ábra. Objektum létrehozása

DATE : dátum.

A példaként megadott táblák az alábbi utasításokkal hozhatók létre:

```
CREATE TABLE AUTO (
        TUL NUMBER(3) NOT NULL,
        RSZ CHAR(6) NOT NULL UNIQUE,
        TIP CHAR(10),
        SZIN CHAR(10),
        EVJ NUMBER(4),
        AR NUMBER(8));

CREATE TABLE EMBER (
        ID NUMBER(3) NOT NULL UNIQUE,
        NEV CHAR(20),
        SZULEV NUMBER(4),
        CIM CHAR(20));
```

Ami első ránézésre is szembetűnhet az az, hogy a táblázatok definiálásakor nem szükségszerű integritási feltételt is megadnunk. Ez egy eltérés a relációs modell elméleti követelményeitől, mivel például a relációs modellben a tervezés során minden táblázathoz kötelezően hozzá kell rendelni egy kulcsmező csoportot, míg az SQL-ben nem kell. Az SQL nyelvben tehát nem kötelező kulcsmező létezése, hiszen a táblázat megadásakor elhagyhatunk bárminemű integritási feltételt, ami a relációs adatmodell elmélete szerint ugyan megengedett, de ez egy újabb esetleges hibaforrást jelenthet az adatbázis tervezése során.

A létrehozott táblázatok szerkezetét az SQL86 szabványban még nem lehetett módosítani, erre csak a későbbi változatok adnak lehetőséget. Az SQL89-es szabványtól kezdve erre a műveletre szolgál az ALTER parancs. A *táblaszerkezet módosításának* utasítása:

ALTER TABLE tábla ADD (m_1 t_1 [i_1]) | *MODIFY* (m_1 t_1 [i_1]);

ahol az ADD kulcsszó hatására egy új mező épül be a relációba, míg a MODIFY egy létező mező definíciójának a módosítására szolgál. Az ALTER parancs nemcsak mezőket, hanem integritási elemeket is módosíthat. Ennek a parancsnak az egyik jellegzetessége, hogy a módosítást leíró rész sajnos nem minden RDBMS rendszerben azonos formátumú, ezért célszerű az alkalmazott rendszer SQL kézikönyvét is tanulmányozni mielőtt e parancsot kiadnánk.

Ha módosítani nem is, de táblát megszüntetni már lehet az SQL86 keretében, így ez a parancs az SQL89-ben is megtalálható. A *táblák megszüntetésének* parancsa a DROP. A tábla megszüntetés utasítása formailag igen egyszerű, csak meg kell adni a törlendő táblázat nevét:

DROP TABLE táblázatnév;

A CREATE TABLE utasítással létrehozott táblázat a *bázis-táblázatok* (alap táblák) közé tartozik, azaz a tartalmazott rekordok fizikailag is letárolásra kerülnek. Mint már említettük, a bázistáblázatokon kívül létezik egy úgynevezett *view* is, ami nevesített leszármaztatott táblázatnak tekinthető. A view-hoz tartozó rekordok tehát nincsenek közvetlenül letárolva az adatbázisban, eltérően a bázistáblázat rekordjaitól. Természetesen a view rekordjainak minden forrás mezője benn van az adatbázisban, valamely bázistáblázatban, de nincs olyan fizikai adatsor, mely a view-nak megfelelő struktúrában és rekord előfordulásokkal tárolná a view egy előfordulását. A view tartalma mindig egy, a táblázatokon értelmezett lekérdezési műveletsor eredménye, és a rendszer a view-hoz kapcsolódóan, a megnevezés mellett az előállító műveletsort tárolja és szükség esetén, azaz amikor hivatkozunk a view-ra, elvégzi a kijelölt műveletsort és a kapott eredménytáblázatot adja meg, mint a view aktuális előfordulását. A view létrehozásának parancsa:

CREATE VIEW viewnév [(m_1 [,m_2...,m_i])] AS műveletsor;

Az m_i szimbólum mezőnevet jelöl, ugyanis lehetőség van arra, hogy a viewban az egyes mezőket a felhasznált alaptáblázatokban megadott nevüktől elérő, új névvel azonosítsuk. Ha nem adunk meg mezőneveket, akkor a view mezők ugyanazt az elnevezést viselik, mint az alaptáblázatban. A műveletsor egy lekérdezési műveletsort jelent, melynek a SELECT utasítás lesz a kulcsszava. A SELECT utasítást később vesszük, most csak előrevetítésként a példában egy olyan auto2 view-t kreálunk, mely csak az OPEL típusú autók rendszámait tartalmazza:

CREATE VIEW auto2 AS SELECT rsz FROM auto WHERE tip LIKE 'Opel%';

A létrehozott view hasonlóan *szüntethető meg*, mint a bázistáblázat:

DROP VIEW viewnév;

SQL

Objektum megszüntetése:
DROP objektumtípus azonosító paraméterek;

DROP TABLE tnév ;

Objektumséma módosítás:
ALTER objektumtípus azonosító paraméterek;

ALTER TABLE tnév ADD | MODIFY (mnev tip intfelt | intfelt);

ALTER TABLE AUTO ADD (TULAJ REFERENCES EMBER);
DROP TABLE AUTO;

A műveletek nem tranzakció hatáskörben futnak le!
Séma lekérdezése: lásd SELECT.

K.L.

5.6. ábra. Objektum módosítása, megszüntetése

5.3. Az SQL DML utasításai

Ha már léteznek táblázatok, melyek a létrehozáskor még üresek, akkor elkezdhetjük őket adatokkal feltölteni, majd a későbbiekben ezeket az adatokat módosíthatjuk. Ezen tevékenységek végrehajtására szolgálnak a DML utasítások. A három idetartozó alaptevékenység:

- az adatok bevitele,
- az adatok törlése,
- az adatok módosítása.

Az *adatok felvitele* az INSERT utasítással történik, mely során meg kell adni, hogy mely táblázatba visszük fel az adatokat, illetve meg kell adni, hogy milyen mezőértékeket vegyen fel az új rekord. Az utasítás leggyakrabban használt alakja:

INSERT INTO táblázatnév VALUES (e_1 [,e_2...,e_i]);

Az utasításban az e_i szimbólum a mezőértéket jelenti. A numerikus adatértéknél számjegyeket és tizedespontot használhatunk, míg a szöveges adatkonstansokat idézőjelekkel határoltan adhatjuk meg. A dátum típusú adatok megadása rendszerint egy kicsit körülményesebb, mivel egy konverziós függvény közbeiktatásával állíthatjuk elő a dátumtípusú adatot. A zárójelben megadott adatértékek sorban hozzárendelődnek az egyes mezőkhöz. Az első érték a táblázat létrehozásakor elsőként megadott mezőhöz, míg az utolsó érték a felsorolásban utolsónak megadott mezőhöz rendelődik. Ebből következik, hogy az értéklistában ugyanannyi elemnek kell szerepelnie, mint amennyi mezőt a táblázat tartalmaz. Példaként bővítsük az autó táblázatot egy rekorddal:

INSERT INTO auto VALUES (3,'bhk546','Fiat','kek',1989,NULL);

A listában szereplő *NULL* érték arra utal, hogy üresen hagyjuk az ár mezőt. A felvitt rekord tul mezője a 3 értéket, az rsz mezője a bhk546 értéket, a tip mezője a Fiat, a szin mezője a kek, míg az evj mezője az 1989 értéket kapja. Az ár üresen marad.

A rekordok felvitelének van egy másik útja is az SQL-ben, amikor a beszúrandó rekordokat nem egyenként visszük fel, hanem már létező táblákból állítjuk elő azokat, azaz egy lekérdezési eredménytáblázat rekordjait tároljuk le egy táblázatban. Ebben az esetben a beszúrásnál meg kell adni a lekérdezési műveletsort is, mely az előzőekhez hasonlóan SELECT kulcsszóval fog kezdődni. Az utasítás alakja:

INSERT INTO táblázatnév műveletsor;

ahol a művelet egy későbbiekben tárgyalásra kerülő SELECT lekérdezési utasítás lehet.

A rekord felvitel, az INSERT utasítás egy harmadik alakja arra szolgál, hogy ne kelljen a rekord minden mezőjét megadni a definíciós sorrendben, hanem elegendő legyen csak az értéket kapó mezőket szerepeltetni egy tetszőleges kijelölési sorrendben. Az INSERT utasítás ezen alakja:

INSERT INTO táblázatnév VALUES (m_1=e_1 [, m_2=e_2..., m_i=e_i]);

ahol a kifejezésben az m_i szimbólum az e_i értéket felvevő mező nevét adja meg. A listában nem kell szerepelni minden táblabeli mezőnek és nem kell követnie a tábla definíciós mező sorrendjét sem. Vegyünk egy példát erre az utasításra is. Ismét az autó táblát vesszük elő:

INSERT INTO auto VALUES (rsz='bhk546', tipus='Fiat', evj=1989);

A felvitt *rekordok kitörlésére* a DELETE utasítás szolgál. A törlés egyértelműen elvégezhető, ha megadjuk, hogy melyik táblázatból, mely rekordokat töröljük ki. A rekordok kiválasztása egy szelekciós feltétellel lehetséges. A szelekciós feltétel a rekordokon értelmezett logikai kifejezés, amely igaz vagy hamis értéket vehet fel. Ha a kifejezés értéke igaz, akkor törlődik a rekord, ha a feltétel hamis, akkor nem törlődik a rekord. Ha a szelekciós feltétel nem szerepel az utasításban, akkor alapértelmezés szerint minden rekord törlődik a táblázatból. Ez nem azt jelenti, hogy maga a táblázat törlődik, mert az megmarad csak éppen üresen. Az utasítás alakja:

DELETE FROM táblázatnév [WHERE feltétel];

A feltétel egy összetett logikai kifejezés is lehet, melyben az elemi kifejezéseket az

AND : logikai és
OR : logikai vagy
NOT : logikai tagadás

köti össze, illetve módosítja (negálja). Az egyes elemi kifejezésekben az alábbi relációs operátorok szerepelhetnek, ahol most a reláció kifejezést matematikai értelemben használjuk:

SQL

Rekord felvitele:

INSERT INTO tabla VALUES (ert1, ert2, …,ertn);
INSERT INTO tabla SELECT …;
INSERT INTO tabla VALUES (mezo=ertek,…);

Az érték lehet NULL is.
Nem maradhat ki mező.
Fontos a mezősorrend.

INSERT INTO AUTO VALUES ("bju564",234,"FIAT");
INSERT INTO AUTO VALUES ("bju564",234,NULL);
INSERT INTO AUTO VALUES (RSZ="bju564",...);

Dátumok felvitele konverziós függvénnyel:

INSERT INTO AUTO VALUES(DATUM =
TO_DATE("2002.02.12","YYYY.MM.DD"),…);

K.L.

5.7. ábra. Rekord felvitele

=	: egyenlő,
>	: nagyobb,
<	: kisebb,
>=	: nagyobb egyenlő,
<=	: kisebb egyenlő,
<>	: nem egyenlő,
BETWEEN	: értéktartományba esik,
IN	: valamely listaértékkel megegyezik,
LIKE	: hasonló,
IS NULL	: üres.

A szokásos relációjelek mellett négy speciális operátor található a felsorolásban. A *BETWEEN* operátorral megvizsgálhatjuk, hogy egy érték valamely intervallumon belül helyezkedik-e el vagy sem. Alakja:

o1 BETWEEN o2 AND o3

ahol o2 kisebb vagy egyenlő mint o3, és a kifejezés akkor szolgáltat igaz értéket, ha o1 értéke az o2-nél nagyobb vagy egyenlő és az o3 értékénél pedig kisebb vagy egyenlő.

Az *IN* operátorral egy lista elemeivel hasonlíthatunk össze egy értéket, alakja:

o1 IN (o2 [,o3...,oi]).

A fenti kifejezés akkor ad vissza igaz értéket, ha o1 értéke megegyezik valamely listabeli értékkel.

SQL

SELECT - speciális szelekciós operátorok

kif [NOT] LIKE minta	sztring kifejezés illeszkedése mintára
	Speciális karakterek a mintában: % több karaktert helyettesít _ egy karaktert helyettesít
	… *WHERE nev LIKE "%Peter%"* …
kif IS [NOT] NULL	üres-e a kifejezés
kif1 [NOT] BETWEEN kif2 AND kif3	intervallumba esés vizsgálata

K.L.

5.8. ábra. Speciális szelekciós operátorok

A LIKE operátor a sztringek összehasonlítására szolgál. A normál '=' operátortól eltérően alkalmas nem szigorú egyezőség vizsgálatra is. Ugyanis az '=' operátor esetén akkor teljesül az egyenlőség, ha a két oldal byte-ról byte-ra megegyezik egymással. A LIKE operátorral lehet közelítő egyezést is vizsgálni, azaz amikor csak bizonyos rész-sztringek egyezőségét, azaz bizonyos minták előfordulását követeljük meg. Használata:

o1 LIKE 'sssss' [ESCAPE 'x'].

A fenti kifejezés az o1 értékét az 'sssss' mintához hasonlítja. A minta tartalmazhat normál karaktereket és úgynevezett joker karaktereket is, hasonlóan a DOS-ból ismert állományspecifikációs '*' és '?' karakterekhez. Itt is két ilyen speciális karakter létezik, melyek jelölése:

% : egy tetszőleges karaktersorozatot helyettesít,
_ : egyetlen egy karaktert helyettesít.

Ekkor az o1 és a minta összevetésekor a normál karaktereknél pontos egyezőséget követel meg a rendszer, míg a % karakter tetszőleges szövegrészt helyettesíthet illetve az _ pedig egyetlen egy karaktert helyettesít. Mivel a fenti két karakter szerepe kötött, probléma léphet fel, ha ezeket normál karakterként kívánjuk felhasználni. A probléma megoldására vezették be az ESCAPE opciót, mellyel kijelölhetünk egy olyan karaktert, mely nem fordul elő a normál karaktereink között, és ennek az lesz a szerepe, hogy a fenti két joker karakter elé írva a mintában jelezze, hogy azok nem helyettesítőként, hanem normál karakterekként szerepelnek. Ez az ESCAPE karakter mindig csak a közvetlenül utána álló joker karakterre vonatkozik. A leírásban x jelentette a kiválasztott ESCAPE karaktert. A LIKE kifejezés akkor ad igaz értéket, ha az o1 illeszkedik a mintára.

Példaként vegyük előbb azt az esetet, amikor azon o1 sztring értékeket keressük, melyekben a második karakter 'R':

o1 LIKE '_R%'

majd azt, amikor a keresett szövegnek a 'DOS_' résszel kell kezdődnie:

o1 LIKE 'DOS^_%' ESCAPE '^'.

A negyedik operátor az *IS NULL*, mely azt vizsgálja, hogy az előtte álló kifejezés üres-e vagy sem. Alakja:

o1 IS NULL.

A kifejezés igaz értéket vesz fel, ha o1 üres. A fenti operátorok közül a normál relációjelek mindegyikének van tagadása is, viszont ez utóbbi négy operátor egyikének sincs negált párja az operátorok között. Ezért ezek negáltját a *NOT* tagadást jelölő kulcsszóval képezhetjük, méghozzá a következő formában:

o1 NOT BETWEEN o2 AND o3
o1 NOT IN (o2 [,o3...,oi])
o1 NOT LIKE 'sssss' [ESCAPE 'x']

o1 IS NOT NULL

A NOT szó az IS NULL operátor kivételével az operátor szó elé kerül. A fenti kifejezések jelentése sorban a következő:

o1 nincs o2 és o3 között;
o1 nincs benne a listában;
o1 nem hasonlít a mintára;
o1 nem üres, azaz tartalmaz adatot.

A relációs operátorok operandusai, azaz az o1, o2 értékei jelenthetnek konstansokat, és jelenthetnek mezőket is. A mezők megadásánál azonban nemcsak a bázistáblázat mezői szerepelhetnek, hanem a kiterjesztett táblázat mezői is, azaz azon mezők, amelyek értékei az alapmezőkből származtathatók. A származtatás során hivatkozhatunk az alapmező mellett konstans adatokra is, amelyeket különböző nem-logikai operátorokkal köthetünk össze. Ezek az operátorok megőrizhetik az operandusok adattípusát, de lehetnek konverziós operátorok is. Az operátorok nagy része különböző rendszer specifikus függvényben nyilvánul meg, de emellett megtalálhatók a közismert aritmetikai operátorok is, mint

\+ : összeadás,
\- : kivonás,
* : szorzás,
/ : osztás.

A többi operátor leírását már az SQL szabvány nem tartalmazza, így az alkalmazott RDBMS kézikönyvekhez kell nyúlni a további operátorok illetve függvények megismeréséhez.

Az előbbiekben ismertetett szelekciós feltétel szemléltetésére azt az utasítást mutatjuk be, amely kitörli az autó táblázatból azon rekordokat, ahol az autó típusa vagy Trabant, vagy pedig 300000 Ft alatt van az ára:

DELETE FROM auto WHERE tip LIKE 'Trabant%' OR ar < 300000;

A DELETE utasítással kapcsolatban még egyszer felhívjuk a figyelmet arra, hogy a

DELETE FROM auto;

utasítás az auto táblázat minden rekordját kitörli. Ha véletlenül mégis elkövetnénk ezt a hibát, szerencsére akkor sem kell megijedni, mivel az SQL RDBMS rendszerek rendszerint rendelkeznek tranzakció visszagörgetési opcióval, így a törlési parancs után még van esély az adatok visszaszerzésére. Ehhez ki kell adni a

ROLLBACK

utasítást, amely minden, a tranzakcióban korábban végrehajtott tevékenységünket visszagörgeti, tehát újra el kell végezni a szükséges lépéseket, cserébe viszont

visszakapjuk az elveszett adatokat.

A DML utasítások harmadik eleme az *adatok módosítására* vonatkozik. A módosítás egyértelmű elvégzéséhez meg kell adni, hogy mely táblázatban, azon belül mely rekordokban mely mezők értékét kell módosítani és hogy milyenek legyenek az új értékek. Ezen adatok a következő utasítással adhatók meg:

UPDATE táblázat SET $m_1 = e_1$ *[,*$m_2 = e_2 \ldots, m_i = e_i$*] [WHERE feltétel];*

A parancsban az m_i jelenti a módosítandó mező nevét, e_i a mező új értékét, és a WHERE után álló feltétel pedig az előbb már megismert szelekciós feltételt jelenti. Hasonlóan a DELETE parancshoz, itt is igaz, hogy ha elhagyjuk a szelekciós feltételt, akkor a táblázat minden rekordjában módosulnak a mezőértékek. Az autó táblában az OPEL típusú autók árának 25%-os emelése a következő utasítással érhető el:

*UPDATE auto SET ar = 1.25*ar WHERE tip LIKE 'Opel%';*

Igen érdekes kérdés a DML utasításoknak view-ra történő alkalmazási lehetőségeinek vizsgálata. A relációs adatmodell elmélete szerint a táblázatok között nincs különbség a kezelhetőség tekintetében, ezért ugyanúgy lehetne dolgozni a bázistáblázatokkal, mint a view táblázatokkal. Azonban a gyakorlati problémák miatt ez sokszor csak igen nehezen megoldható, sokszor nem is egyértelmű az utasítás eredménye a bázistáblázatokra vetítve, hiszen minden módosításnak a bázistáblázatoknál kell végrehajtódnia. Ezen nehézségek miatt az SQL86 csak a bázistáblázatok vonatkozásában engedi meg a DML utasítások végrehajtását. A későbbi SQL verziókban is csak azon view-k módosíthatók, melyekre igaz, hogy az elvégzett módosítás egyértelműen visszavezethető az alaptáblák módosításaira. Ennek megfelelően rendszerint *csak az egy alaptáblára épülő nézeti tábláknál megengedett* a rekordok módosítása vagy törlése.

SQL

Rekord törlése:
DELETE FROM tabla WHERE feltétel;

A feltételnek eleget tevő rekordok törlődnek.
Ha elmarad a WHERE tag, minden rekord törlődik.
A feltétel rész részletesen a SELECT-nél kerül ismertetésre.

DELETE FROM AUTO WHERE AR<1200000;

Rekord módosítása:
UPDATE tabla SET mezo= érték,... WHERE feltétel;

A feltételnek eleget tevő rekordok módosulnak.
Ha elmarad a WHERE tag, minden rekord módosul.

*UPDATE AUTO SET AR = AR * 1.2*
WHERE AR<1200000;

K.L.

5.9. ábra. Rekord törlése, módosítása

5.4. Az SQL lekérdezési utasítása

Az utasítások harmadik csoportja a *lekérdezésekhez* kapcsolódik. A lekérdezések mindegyike a *SELECT* utasításhoz kötődik, igaz ennek az utasításnak számos variációja létezik. A SELECT utasításnak ugyanis alkalmasnak kell lennie a megismert relációs algebrai lekérdező műveletek, például a szelekció, a projekció vagy a join végrehajtására. A jobb megértés végett nem egyszerre tekintjük át a teljes lehetőségskálát, hanem fokozatosan bővítjük az utasításkört. A SELECT utasítás használata közben ne feledjük el, hogy eredménye mindig egy újabb táblázat lesz.

Elsőként a *projekciót* vesszük. A projekció a táblázatnak bizonyos mezőire történő leszűkítését jelenti. Meg kell tehát adnunk, hogy mely táblázatból mely mezők értékeire van szükségünk. Az utasítás alakja:

SELECT m_1 *[,* m_2*, ...,* m_i*] FROM táblázatnév;*

A fenti parancsban az m_i mezőnevet jelent. Ha tehát az autó táblából csak a rendszámok adatai érdekelnek bennünket, akkor a

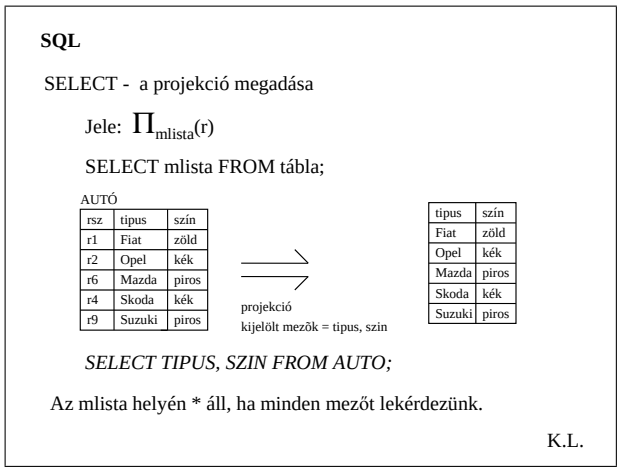
SELECT rsz FROM auto;

utasítást kell kiadnunk. Az utasítás eredménye az alábbi listával adható meg:

RSZ
bkx720
cmt111
aaa156
lui999
kjs234

Az alábbi utasítás pedig az emberek nevét és címét adja eredményül:

SELECT nev, cim FROM ember;



5.10. ábra. Projekció megadása SQL SELECT-el

NEV	CIM
Bela	Budapest
Geza	Miskolc
Feri	Pecs

A projekcióval kapcsolatban még két megjegyzést érdemes megemlíteni. Az első arra irányul, hogy hogyan jelöljük azt, amikor minden mezőre kíváncsiak vagyunk. Ekkor ugyanis nem szükséges minden mezőnevet felsorolni, elegendő egy * karaktert megadni a mezőneveknél. Ha tehát az emberek minden adatát tudni szeretnénk, akkor a

*SELECT * FROM auto;*

utasítást kell kiadni. Ekkor a következő listát kapjuk vissza:

ID	NEV	SZULEV	CIM
1	Bela	1975	Budapest
2	Geza	1979	Miskolc
3	Feri	1974	Pecs

A másik megjegyzés arra vonatkozik, hogy lehetnek olyan projekciók, amelyeknél az eredménytáblázatban bizonyos rekordok többször is előfordulhatnak. Az elméleti relációs adatmodell szerint ennek nem lenne szabad bekövetkeznie, de mint már mondtuk, az SQL csak egy közelítése az elméleti modellnek. Bizonyos esetekben hasznos lehet ez a duplázódás, máskor viszont feleslegesnek tűnik. Ezért az SQL lehetőséget ad az ismétlődések elkerülésére: a *DISTINCT* opcióval az eredménytáblázat minden sora különböző lesz. Az opciót közvetlenül a SELECT kulcsszó után kell megadni. Ha tehát az autó típusokat szeretnénk ismétlés nélkül kilistázni, akkor ehhez az alábbi parancsot kell kiadni:

SELECT DISTINCT tip FROM auto;

A kapott eredménylista:

TIP
Opel
Golf
Trabant
Lada

A táblázat kiterjesztésének lehetőségét már korábban tárgyaltuk a szelekciós feltétel megadásakor, így most csak visszautalunk rá, megjegyezve, hogy a származtatott mezőt a projekciós részben is megadhatjuk. Például az autók árainak dollárban történő kiírásához az alábbi utasítás tartozik:

SELECT rsz, tip, ar/150 FROM auto;

ahol minden autóra még a rendszámot és a típust is kiírtuk.

A másik nagy műveletcsoport a *szelekció*. Ebben ki kell jelölni azon rekordokat, amelyek átkerülnek az eredménytáblába. A szelekciós feltétel jellege és megadásának formája megegyezik a DELETE és UPDATE parancsoknál bemutatott szelekciós feltétellel. Mivel egy lekérdezési utasítás több relációs algebrai részműveletet tartalmazhat, ezért a SELECT utasításban szerepelhetnek a már korábban megismert elemek is az éppen vizsgált műveleti rész mellett. A részművelet tárgyalásánál viszont már csak a rövidített jelölést fogjuk használni. A szelekció megadása ennek megfelelően a következő alakban történik:

SELECT ... FROM ... WHERE feltétel;

ahol a SELECT és FROM utáni részek a korábban megadott elemeket tartalmazhatják. A piros színkódú autók rendszámainak lekérdezése így a következőképpen alakul:

SELECT rsz FROM auto WHERE szin = 'piros';

Az eredménytábla:

RSZ
cmt111

A szelekciós műveletek bemutatására vegyük a következő példákat.

– Írassuk ki az 1977 előtt született emberek neveit.

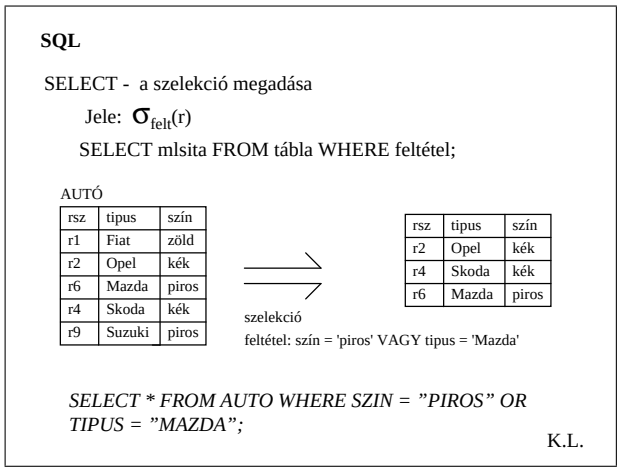
SELECT nev FROM ember WHERE szulev<1977;

Az eredménylista:

NEV
Bela
Feri

– Írassuk ki a nem piros autók összes adatát.

*SELECT * FROM auto WHERE szin<>'piros';*



5.11. ábra. Szelekció megadása SQL SELECT-el

Az eredménylista:

TUL	RSZ	TIP	SZIN	EVJ	AR
1	bkx720	Opel	bordo	1991	1000000
2	aaa156	Trabant	feher	1985	100000
3	lui999	Opel	kek	1991	450000
1	kjs234	Lada	kek	1989	275000

– Írassuk ki az 1975 és 1980 között született emberek adatait.

*SELECT * FROM ember WHERE szulev BETWEEN 1975 AND 1980;*

Az eredménylista:

ID	NEV	SZULEV	CIM
1	Bela	1975	Budapest
2	Geza	1979	Miskolc

A BETWEEN után megadott két paraméter esetében az egyenlőség is megengedett.

– Írassuk ki a piros, kék, vagy zöld színű autók rendszámait.

SELECT rsz FROM auto WHERE szin IN ('piros','kek','zold');

Az eredménylista:

RSZ
cmt111
lui999
kjs234

Az IN operátor után egy halmazt adhatunk meg a zárójelek között az elemek felsorolásával.

– Listázzuk ki azokat a sorokat, ahol a CIM mező üres.

*SELECT * FROM ember WHERE cim IS NULL;*

Az eredménylista most üres lesz:

ID	NEV	SZULEV	CIM

– Listázzuk ki a "G" betűvel kezdődő emberek adatait.

*SELECT * FROM ember WHERE nev LIKE 'G%';*

Az eredménylista:

ID	NEV	SZULEV	CIM
2	Geza	1979	Miskolc

Ebben az esetben a %-jel minden további karaktert lefed (joker karakter).

– Most írjuk ki azokat az embereket, akik nevének első karaktere bármi lehet, a név többi része pedig az "eza" szöveggel egyenlő.

*SELECT * FROM ember WHERE nev LIKE '_eza%';*

Az eredménylista:

ID	NEV	SZULEV	CIM
2	Geza	1979	Miskolc

Ebben a példában az "_" (aláhúzás) jel egy karakter helyen töltött be joker szerepet.

– Most listázzuk ki azokat a neveket, amelyekben szerepel az "ez" szöveg.

*SELECT * FROM ember WHERE nev LIKE '%ez%';*

Az eredménylista:

ID	NEV	SZULEV	CIM
2	Geza	1979	Miskolc

– Hozzunk létre egy ideiglenes mezőt az életkorra, ha most 1996-ot írunk.

SELECT nev, 1996-szulev FROM ember;

Az eredménylista:

NEV	1996-SZULEV
Bela	21
Geza	17
Feri	22

– Most írassuk ki az autók típusait és árait dollárban megadva (az aktuális átváltási arány: 1USD=150Ft).

SELECT tip, ar/150 FROM auto;

Az eredménylista:

TIP	AR/150
Opel	6666.6667
Golf	2333.333
Trabant	666.6667
Opel	3000
Lada	1833.33

Az előbb már említettük, hogy az SQL közel áll a fizikai tároláshoz, a gyakorlati élethez, jobban valósítja meg a táblázatok kezelését, mint az elméleti relációs algebra. Ennek másik példája, amikor egy olyan opciót mutatunk be, amely nem szerepel az eredeti modellben, mégpedig a rekordok rendezését. Az elméleti modell ugyanis halmazorientált, amelyben nem értelmeztük a rendezettséget. A gyakorlatban viszont adatainkat nem halmazokban, hanem inkább listákban tároljuk,

ahol rendszerint valamilyen rendezettséget valósítunk meg, például a hallgatók neveit ABC sorrendben tároljuk. A rendezettség gyakorlati jelentőségének tükrében az SQL is lehetővé teszi az eredményrekordok megadott sorrendbe történő rendezését. A *rendezés* opciójának kulcsszava az ORDER BY és használata a következő:

SELECT... FROM... [WHERE...]
ORDER BY k_1 *[ASC|DESC][,*k_2 *[ASC|DESC]...,*k_i*[ASC|DESC]];*

A kifejezésben a k_i egy kifejezést takar, melyet valamely mezőből vagy mezőkből kapunk a már ismertetett operátorok segítségével. A legegyszerűbb esetben a k_i egy mezőnévvel egyezik meg. A kifejezés mögött álló ASC illetve DESC a rendezettség irányára utal. Az ASC jelenti a növekvő sorrendet, míg a DESC a csökkenő érték szerinti sorrendet jelöli ki. Az alapértelmezés a növekvő sorrend, amely akkor érvényesül, ha nem használjuk sem az ASC sem a DESC kulcsszavakat.

A rendezéshez több kifejezést is felhasználhatunk. Az elsőnek megadott kifejezés lesz az elsődleges rendezési elv. Azaz a k_1 értéke dönt elsőként a sorrend megállapításánál. Ha viszont két rekordnál a k_1 értéke azonos lenne, akkor kap szerepet a következő, k_2 kifejezés. Ebben az esetben ez a másodlagos kifejezés dönti el a sorrendet. Ha k_2 értéke is azonos lenne, akkor a k_3 kifejezés kerül sorra a sorrend meghatározásánál, és így tovább hasonlóan lehet végighaladni egészen az utolsó tagig. Ha például a nem piros színű autók adatait kívánjuk kilistázni szín szerint csökkenő ABC rendben és azonos szín esetén típus szerint növekvően, akkor az alábbi utasítást kell kiadni:

*SELECT * FROM auto WHERE szin<>'piros' ORDER BY szin DESC, tip;*

Az eredménylista:

TUL	RSZ	TIP	SZIN	EVJ	AR
1	kjs234	Lada	kek	1989	275000
3	lui999	Opel	kek	1991	450000
2	aaa156	Trabant	feher	1985	100000
1	bkx720	Opel	bordo	1991	1000000

Az ORDER BY opció a halmaz orientált szemlélettől való eltérés következtében csak korlátozottan használható, ugyanis csak a végső, kiírt eredménytáblázatra vonatkozhat.

A gyakorlatban nagy jelentősége van a csoportképzés műveletének is. A *csoportképzés* annyit jelentett, hogy a táblázat rekordjaiból csoportokat képeztünk, ahol egy csoportba a valamilyen szempont szerint összeillő rekordok tartoznak, és az eredménytáblázatban egy rekord egy csoportnak fog megfelelni, azaz egy rekord egy csoport összesítő (aggregált) adatait írja le. Mivel az eredménytáblázat is kielégíti az 1NF feltételt (első normálforma, lásd a következő fejezetben), ezért minden mezőnek egyértékűnek kell lennie, azaz olyan értékeket tartalmazhat csak, melyek a csoportra nézve egyértelműek.

Az SQL-ben léteznek a csoportokra olyan operátorok, amelyek a csoportokra ilyen egyértékű (összesített, aggregált) mezőt hoznak létre az eredménytáblázatban. Az alábbi operátorok állnak rendelkezésre, melyek formailag függvényszerűen használhatók:

MIN(k) : minimumérték,
MAX(k) : maximumérték,
AVG(k) : átlagérték,
SUM(k) : összeg,
COUNT(k) : darabszám.

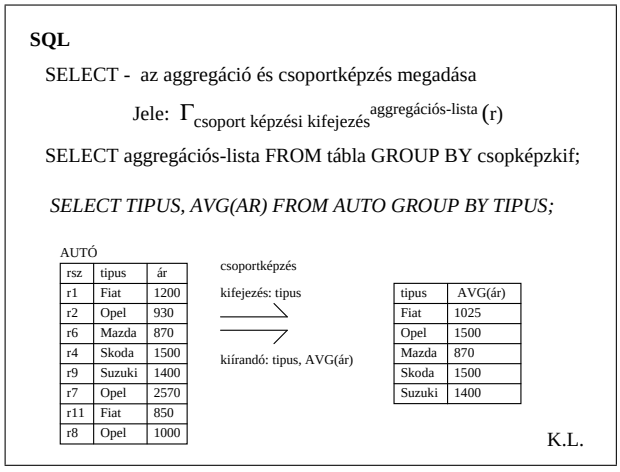
Az operátoroknál szereplő k szimbólum egy tetszőleges, előzőekben már értelmezett kifejezés lehet. Ekkor a kifejezés kiértékelődik a csoport minden rekordjára, és megkapjuk az értékek minimumát, maximumát, átlagát, összegét és darabszámát. Itt a darabszám azon rekordok darabszáma, melyeknél a kifejezés nem ad üres értéket. Ha magára a csoportban elhelyezkedő rekordok darabszámára vagyunk kíváncsiak, akkor a

COUNT(*)

kifejezést használhatjuk. A csoportok képzése a *GROUP BY* opcióval történik, melyben meg kell adni egy kifejezést is. Azon rekordok kerülnek egy csoportba, melyekre a kifejezés azonos értéket vesz fel. A csoportképzés általános alakja:

SELECT... FROM... [WHERE...] GROUP BY k [ORDER BY...];

A csoportképzésnél arra kell elsősorban figyelni, hogy csak olyan mezők szerepelhetnek az eredménytáblázatban, amelyek minden csoportra nézve egyértékűek. A csoportképzésre példaként vegyük azt az esetet, amikor az egyes autótípusok



5.12. ábra. Aggregáció és csoportképzés megadása

átlagárait szeretnénk megkapni. Ehhez ugyanis előbb csoportokba bontjuk az autókat a típus szerint, majd minden csoportra képezzük az 'ar' mező átlagát és az eredménytáblázatba a típus és az átlagár mezőket vesszük be, melyek minden csoportra egyértékűek lesznek. Az utasítás alakja:

SELECT tip, AVG(ar) FROM auto GROUP BY tip;

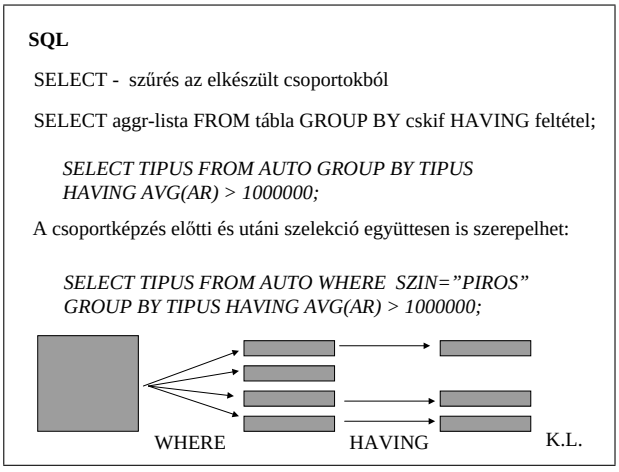
Az eredménytábla:

TIP	AVG(AR)
Opel	725000
Golf	350000
Trabant	100000
Lada	275000

A csoportképzés esetén felléphet olyan igény is, hogy a felhasználó nem minden csoportra kíváncsi, hanem csak egyes, bizonyos feltételeknek eleget tévő csoportok adatait kéri le. Ezzel úgymond szeretnénk a kapott csoport táblázat rekordjai között szelektálni. Erre a már megismert WHERE szelekciós opció viszont nem alkalmas, mert az alaptáblázat rekordjain értelmezett szelekcióra vonatkozik, tehát azt jelöli ki, hogy mely rekordokból képezzük a csoportokat, és nem azt szabályozza, hogy mely csoportok kerüljenek be az eredménytáblázatba.

Ezért egy újabb opcióra lesz szükség, ami a *csoportok szelekciójára* vonatkozik. Ezen opció kulcsszava a *HAVING*, melyet egy feltétel követ. Az opció hatására azon csoportok kerülnek be az eredménytáblázatba, melyek kielégítik a HAVING után megadott feltételt. Az opció általános alakja:

SELECT... FROM... [WHERE...] GROUP BY... HAVING feltétel [ORDER BY...];



5.13. ábra. Szűrés megadása csoportokra

A feltételben az eredménytáblázatban szereplő mezők szerepelhetnek, azaz olyan kifejezést adhatunk meg, amely minden csoportra egyértelműen kiértékelhető. Példaként vehetjük azt az esetet, amikor csak a 300000 Ft alatti átlagárral rendelkező típusokat listázzuk ki, melyhez a következő SQL parancsot kell kiadni:

SELECT tip, AVG(ar) FROM auto GROUP BY tip HAVING AVG(ar) < 300000;

Az eredménytábla:

TIP	AVG(AR)
Trabant	100000
Lada	275000

A csoportképzés és aggregáció gyakorlására vegyük az alábbi példákat.

– Írassuk ki az autók átlagárát.

SELECT AVG(ar) FROM auto;

Az eredmény:

AVG(AR)
435000

– Írassuk ki a minimum évjáratot (a legidősebb autót).

SELECT MIN(evj) FROM auto;

Az eredmény:

MIN(EVJ)
1981

– Írassuk ki az autók összértékét.

SELECT SUM(ar) FROM auto;

Az eredmény:

SUM(AR)
2175000

– Írassuk ki, hogy milyen autó típusok léteznek és azt, hogy hány darab van belőlük.

SELECT tip, COUNT(tip) FROM auto GROUP BY tip ORDER BY tip;

Az eredménylista:

TIP	COUNT(TIP)
Golf	1
Lada	1
Opel	2
Trabant	1

A COUNT aggregációs függvény a nem üres mezők darabszámát adja vissza. Ha a rekordok számára vagyunk kíváncsiak, akkor a COUNT(*) kifejezést használjuk.

– Írassuk ki az autók típusait, és a típusok átlagárait.

SELECT tip, AVG(ar) FROM auto GROUP BY tip ORDER BY tip;

Az eredménylista:

TIP	AVG(AR)
Golf	350000
Lada	275000
Opel	725000
Trabant	100000

– Írassuk ki azon autók típusait és átlagéletkorát, amelyek átlagévjárata 1986 utáni.

SELECT tip,AVG(evj) FROM auto GROUP BY tip HAVING AVG(evj)>1986;

Az eredménylista:

TIP	AVG(EVJ)
Opel	1991
Lada	1989

– Mely kék színű autótípusok átlagévjárata nagyobb 1990-nél.

SELECT tip FROM auto WHERE szin='kek' GROUP BY tip HAVING AVG(evj)>1990;

Az eredménylista:

TIP
Opel

– Írassuk ki az autó táblában szereplő összes autótípust, a típushoz tartozó darabszám szerint rendezve.

SELECT tip FROM auto GROUP BY tip ORDER BY COUNT(*);

Az eredménylista:

TIP
Golf
Trabant
Lada
Opel

A relációs algebra és az SQL leggyakrabban használt műveletei közé mindenképpen be kell venni az egyesítés, a join műveletét is. E művelet fontosságának egyik legfőbb oka az, hogy az adatbázis tervezése során, a normalizálással az információkat szétbontjuk táblázatokra, azaz egy összetettebb lekérdezéshez szükséges információk több táblázatban szétszórva helyezkednek el, így a lekérdezés során össze kell gyűjteni ezen adatokat a különböző táblázatokból, ahol az összetartozás bizonyos mezők érékeinek kapcsolatán alapszik. Azt a folyamatot, amikor több táblázatból származó adatokból állítunk elő egy újabb eredménytáblázatot, egyesítésnek vagy join-nak nevezzük.

Az SQL-ben két táblázat egyesítésének a legegyszerűbb formája az, amikor a két táblázat Descartes-szorzatát képezzük, amely során az eredménytáblázat egy rekordja úgy áll elő, hogy az egyik táblázat rekordjához hozzáfűzzük a másik táblázat egy rekordját, ahol az eredménytáblázat minden lehetséges párosítást tartalmaz. Ha tehát az egyik, a példában A-val jelölt táblázat az M1, M2, M3 mezőket tartalmazza és a táblázat előfordulás 3 rekordból áll, a másik, a B-vel jelölt táblázat az N1, N2 mezőket tartalmazza és az előfordulása 4 rekordból épül fel, akkor az eredménytáblázat az M1, M2, M3, N1, N2 mezőket fogja tartalmazni. Az eredménytáblázat előfordulása pedig 12 rekordot foglal magába, hiszen ennyi lehetséges párosítása lehet az A-beli és a B-beli rekordoknak.

A

M1	M2	M3
1	2	3
4	5	6

B

N1	N2
7	8
9	10
11	12

A*B

M1	M2	M3	N1	N2
1	2	3	7	8
1	2	3	9	10
1	2	3	11	12
4	5	6	7	8
4	5	6	9	10
4	5	6	11	12

Két táblázat Descartes-szorzatának előállításához a következő SQL utasítást kell kiadni:

*SELECT * FROM táblázatnév1, táblázatnév2;*

A példában megadott lekérdezéshez az alábbi utasítás tartozik:

*SELECT * FROM A, B;*

Az így előálló egyesített táblázatot ezután tetszőlegesen tovább lehet alakítani a már megismert opciókkal. Erre rendszerint szükség is van, hiszen csak

nagyon ritkán van szükség két táblázat rekordjainak teljes Descartes-szorzatára, legtöbbször csak a Descartes-szorzat bizonyos részhalmazára van szükségünk. A szorzat táblázat szelekciójával valósítható meg például a táblázatok összekapcsolására szolgáló kulcs és kapcsolókulcs szerkezet alapján előálló rekord párok kijelzése.

A példánkban az ember és az autó rekordok között tulajdonosi kapcsolat áll fenn, melyet az id-tul kulcs-kapcsolókulcs szerkezettel tartunk nyilván. A korábban megadott két alaptáblázat felhasználásával az autók rendszámainak és tulajdonosaik nevének, címének kiíratása a következő paranccsal állítható elő:

SELECT rsz, nev, cim FROM auto, ember WHERE tul = id;

Az eredménytáblázatot tehát úgy kapjuk meg, hogy előbb képezzük az összes lehetséges autó és ember rekordpárosítást, majd ebből kiválasztjuk azokat, melyekben az autó rekordrész tulajdonos mezőkódja megegyezik az ember rekordrész azonosító kódszámával, hiszen a kapcsolat e két mező értékazonosságán keresztül valósul meg. A példánkban az RDBMS egyértelműen meg tudja határozni, hogy milyen információkat kell kiemelni, hiszen minden hivatkozási név is egyértelmű, hiszen nincsenek azonos nevű mezők a két táblázatban. Viszont más példákban előfordulhat, hogy mindkét táblázatban szerepel ugyanolyan nevű mező. Ebben az esetben már nem egyértelmű, hogy mely mezőre is gondolunk az SQL parancsban. Ekkor pontosítani kell a megjelölést, nem elegendő csupán a mezőnév megadása, a táblázat nevét is megadjuk. Az ilyen együttes hivatkozás formája a következő:

táblázatnév.mezőnév

Ez alapján az előző példa a következő alakot öltené:

SELECT auto.rsz, ember.nev, ember.cim FROM auto, ember WHERE auto.tul = ember.id;

SQL

Szelekciós join:

*SELECT * FROM AUTO, EMBER WHERE*
TUL = ID;

Ha több táblában is azonos elnevezésű mező van, akkor a kibővitett mezőnevet használjuk: tábla.mező

*SELECT * FROM AUTO, EMBER WHERE*
AUTO.TUL = EMBER.ID;

Ha hosszú a táblanév alias nevet használhatunk: tábla alias

*SELECT * FROM AUTO A, EMBER E WHERE*
A.TUL = E.ID;

K.L.

5.14. ábra. Szelekciós join

A pontosított, együttes hivatkozás mindig egyértelmű, így viszont a felhasználónak hosszabb parancsokat kell kiadni, hiszen mindannyiszor le kell írni a táblázat nevét is, valahányszor a mezőre hivatkozunk. A tömörség végett az SQL lehetőséget ad egy rövidebb írásmódra, melyben minden táblázatnévhez egy egyértelmű rövidítést rendelhetünk és a mezőnevekben elegendő csak ezen rövidített névre hivatkozni. A rövidítést a FROM opcióban adjuk meg a teljes táblázatnév után. Alakja: *táblázatnév rövidítés*. Az előző példánál maradva, az utasítás a következő formában írható le:

SELECT a.rsz, e.nev, e.cim FROM auto a, ember e WHERE a.tul = e.id;

Az SQL utasításokban viszonylag szabadon használhatjuk az előbb említett mezőhivatkozásokat, tetszőlegesen választhatunk a megadott lehetőségek között, csak akkor kell a legpontosabb, táblázatnevet is tartalmazó megadást választani, ha a mezőnév önmagában nem egyértelmű, azaz több táblázatban is előfordul ugyanaz a mezőnév.

A join műveletének szemléltetésére néhány példát mutatunk be.

– Írassuk ki az összetartozó autó-ember párosokat.

SELECT a.rsz, e.nev FROM auto a, ember e WHERE a.tul = e.id;

Az eredménylista:

RSZ	NEV
bkx720	Bela
cmt111	Bela
aaa156	Geza
lui999	Feri
kjs234	Bela

– Írassuk ki az 1991 előtti évjáratú autók rendszámát és a tulajdonos nevét.

SELECT a.rsz, e.nev FROM auto a, ember e WHERE a.evj < 1991 AND a.tul = e.id;

Az eredménylista:

RSZ	NEV
cmt111	Bela
aaa156	Geza
kjs234	Bela

– Írassuk ki az emberek neveit az autóik darabszámával együtt.

SELECT e.nev, COUNT(*) FROM ember e, auto a WHERE a.tul = e.id GROUP BY e.nev;

Az eredménylista:

NEV	COUNT(*)
Bela	3
Geza	1
Feri	1

– Írassuk ki azon emberek nevét, akiknek 1-nél több autójuk van.

SELECT e.nev FROM ember e, auto a WHERE a.tul = e.id GROUP BY e.nev HAVING COUNT(*)>1;

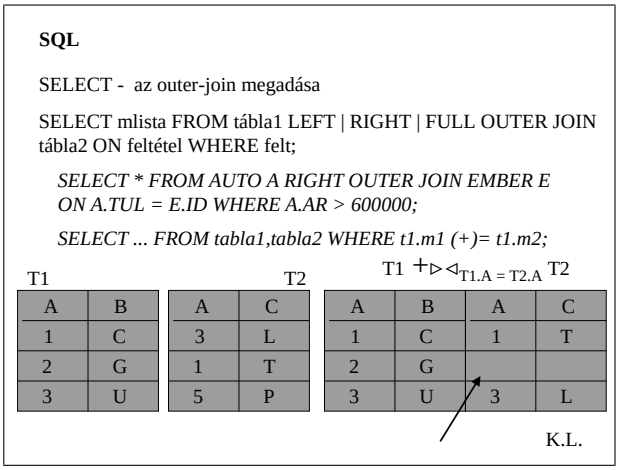
Az eredménylista:

NEV
Bela

Mint már ismert, az összekapcsolás (join) műveletének egyik fontos fajtája *a külső összekapcsolás* (outer join), amelybe az illeszkedés nélküli rekordok is bekerülnek az eredménybe NULL értékű párral kiegészülve. E műveletnek három altípusa is van, attól függően, hogy mely oldalon szereplő rekordokat veszi be illeszkedés nélkül is az eredménybe. Az SQL szabvány e művelet megadására a vessző operátor helyett szöveges join operátorokat használ:

tábla1 LEFT OUTER JOIN tábla2 ON feltétel : bal oldali outer join,
tábla1 RIGHT OUTER JOIN tábla2 ON feltétel : jobb oldali outer join,
tábla1 FULL OUTER JOIN tábla2 ON feltétel : kétoldali outer join.

A fenti kulcsszavakat a FROM utáni részben kell megadni, ahol az alap join jele is előfordul. Az operátorhoz tartozó ON kulcsszóval az illesztési feltételt lehet



5.15. ábra. Külső join megadása

megadni. Ezen formalizmusban tehát külön válik a táblák összekapcsolásának feltétele a normál szelekciós feltételektől. E szeparáció fő előnye, hogy a parancsot értelmező személy könnyebben meg tudja határozni az összekapcsolás logikáját, könnyebben érthető lesz a parancs jelentése. A következő példában az emberek neve és autóik darabszáma lesz a lekérdezendő információ úgy, hogy az autóval nem rendelkező emberek is szerepeljenek az eredmény listában:

SELECT nev, COUNT(*) FROM auto RIGHT OUTER JOIN ember ON tul = id GROUP BY nev;

Külső join nélkül a fenti lekérdezésben az illesztés után csak az autóval rendelkező személyek rekordjai maradtak volna meg, így az eredmény nem tartalmazná azon személyek adatait, akik nem autótulajdonosok.

Az SQL lekérdezési (query) komponensének, a SELECT utasításnak már az eddigiekben is számos olyan opcióját láttuk, mely kellő rugalmasságot adott a lekérdezések megfogalmazásában. Maradtak viszont olyan lekérdezések, melyek az eddigi opciókkal nem oldhatók meg, például tekinthetjük azt a kérést, hogyan listázzuk ki azon emberek nevét, akiknek nincs autója. Ehhez sem a sima szelekció, sem a két tábla egyesítése nem elegendő, hiszen az ott szereplő feltételek rekordszintűek, a mi esetünkben pedig arra lenne szükség, hogy csak azon rekordok kerüljenek az ember táblázatból az eredménytáblázatba, amelyeknél az id mező értéke nem szerepel egyetlen egy rekord esetén sem az autó táblázatban a tulajdonos mezőben. Itt tehát a feltételben egy egész táblázatra vonatkozó kifejezés szerepel. A példában a megoldás az lenne, ha előbb lekérdeznénk az összes tulajdonos mezőértéket az autó táblázatból, és utána ezen listával hasonlítanánk össze az emberek id mezőjének értékeit. Vagyis a szelekciós feltételben egy másik lekérdezés eredményére hivatkoznánk.

Az SQL szerencsére támogatja a fenti jellegű szelekciós feltétel megadást is. Ezt az opciót nevezik *al-lekérdezésnek* (subquery), ami tehát annyit jelent, hogy az egyik lekérdezés szelekciós feltételében hivatkozunk egy másik lekérdezés eredményére. Az al-lekérdezést mindig zárójelben kell megadni, hogy elemei elkülönüljenek, elválaszthatók legyenek a fő lekérdezés opcióitól. Az al-lekérdezés formailag megegyezik a normál SELECT utasítás alakjával, azzal a különbséggel, hogy az al-lekérdezésben nem használható a rendezett kiírást előíró ORDER BY opció, tehát az al-lekérdezés eredményét rendezettség nélküli halmaznak kell tekinteni.

Az al-lekérdezést az eredményétől függően más és más típusú operátorokhoz kapcsolhatjuk. Ha az eredmény egyetlen egy érték, akkor a skalárokhoz kötődő operátorokat használhatjuk, mint például a relációs operátorokat. Ha viszont a lekérdezés eredménye több rekordot is tartalmaz, akkor csak a halmazokat kezelő operátorok jöhetnek szóba. Eddig egy ilyen operátort vettünk, az IN operátort. Emellett az al-lekérdezéseknél használható egy másik, hasonló jellegű operátor is, mely azt vizsgálja meg, hogy az al-lekérdezés eredménytáblázata *üres-e vagy sem.* Ezen operátor alakja:

EXISTS (al-lekérdezés).

Ha az al-lekérdezés eredménytáblázata nem üres, akkor a fenti kifejezés igaz értéket

ad vissza, ha pedig üres az eredménytáblázat, akkor hamis értéket szolgáltat. Az EXISTS operátor tagadásának alakja:

NOT EXISTS (al-lekérdezés).

A példaként említett lekérdezés tehát, amikor azon emberek neveit kérjük le, akiknek nincs autója, az alábbi SQL utasítás formájában fogalmazható meg:

*SELECT e.nev FROM ember e WHERE NOT EXISTS (SELECT * FROM auto a, ember e WHERE a.tul=e.id);*

Az eredménytáblát más módon is megkaphatjuk, ha előbb előállítjuk az autóval rendelkező személyek azonosítóinak listáját:

SELECT e.nev FROM ember e WHERE e.id NOT IN (SELECT a.tul FROM auto a);

A példában bemutatott kétfajta megközelítés legalapvetőbb különbsége, hogy a második esetben a szelekciós feltétel minden rekordnál ugyanarra az al-lekérdezésre vonatkozik, míg az első esetben minden al-lekérdezés más és más eredményt szolgáltathat. Emiatt egy jó optimalizáló esetén a második esetben csak egyszer kell az al-lekérdezést végrehajtani, míg az első esetben többször is lefut az al-lekérdezés.

Az SQL lehetőséget ad a minden és a létezik logikai kvantorok megvalósítására az al-lekérdezések esetében. A normál, a skalár értékek összehasonlítására alkalmazott reláció operátorokat ki lehet terjeszteni a több értéket visszaadó szelekciókra is. A *létezik operátora* az *ANY*, a *minden kvantor* operátora pedig az *ALL*. Ezen operátorokat a halmazt visszaadó lekérdezés előtt, de az alkalmazott skalár relációs operátor után kell szerepeltetni az SQL utasításon belül. A

kif reláció_ operátor ANY (halmaz)

akkor igaz értékű, ha a halmaznak legalább egy elemére igaz értékű a *kif reláció_ operátor halmazelem* kifejezés. A

kif reláció_ operátor ALL (halmaz)

pedig akkor ad igaz értéket, ha a halmaznak minden elemére teljesül a *kif reláció_ operátor halmazelem* kifejezés.

Az al-szelekciók bemutatására vegyük a következő mintapéldákat.

– Írassuk ki a kék színű autók tulajdonosainak a neveit és címeit (vagyis írassuk ki azon emberek adatait, akiknek azonosító száma benne van a kék autók tulajdonosainak a kód halmazában).

SELECT nev, cim FROM ember WHERE id IN (SELECT tul FROM auto WHERE szin = 'kek');

Az eredménylista:

NEV	CIM
Bela	Budapest
Feri	Pecs

– Írassuk ki azon embereket, akiknek nincs autója.

SELECT nev, cim FROM ember WHERE id NOT IN (SELECT tul FROM auto);

Az eredménylista üres lesz:

NEV	CIM

– Írassuk ki azokat az autókat, amelyek ára az átlagár alatt van.

*SELECT * FROM auto WHERE ar < (SELECT AVG(ar) FROM auto);*

Az eredménylista:

TUL	RSZ	TIP	SZIN	EVJ	AR
1	cmt111	Golf	piros	1981	350000
2	aaa156	Trabant	feher	1985	100000
1	kjs234	Lada	kek	1989	275000

– Írassuk ki azokat az autókat, melyek ára nagyobb valamely másik autó áránál.

*SELECT * FROM auto a WHERE a.ar > ANY (SELECT b.ar FROM auto b);*

Az eredménylista:

TUL	RSZ	TIP	SZIN	EVJ	AR
1	bkx720	Opel	bordo	1991	1000000
3	lui999	Opel	kek	1991	450000
1	cmt111	Golf	piros	1981	350000
1	kjs234	Lada	kek	1989	275000

Az SQL tartalmaz még két, ritkábban használt lehetőséget is, a táblázatok *uniójának* illetve *metszetének* a műveletét. Két azonos felépítésű táblázat fűzhető egymáshoz az alábbi SQL utasításokkal:

SELECT... UNION SELECT...;
SELECT... INTERSECT SELECT...;

A fenti UNION művelet hatására a két reláció halmaz egyesítésére kerül sor. Ennek hatására minden elem csak egyszer lesz benne az eredményben, tehát a duplikálások eliminálódnak. Ennek végrehajtása az ismétlődések ellenőrzése miatt

viszonylag lassabban fut le. Az egyesítésnek van egy gyorsabb megvalósítása is, amikor a rendszer nem ellenőrzi a duplikálások meglétét, egyszerűen egymásután fűzi a két tábla rekordjait. Ezen utasítás operátora az UNION ALL:

SELECT... UNION ALL SELECT...;

Az eddigiekben bemutatott opciók, lehetőségek bizonyítják, hogy milyen rugalmasan felhasználható az SQL SELECT utasítása, és kellő kreativitással szinte minden lehetséges lekérdezési igény kielégíthető. Az SQL prescriptív jellege miatt ismernünk kell az elvégzendő műveleteket, mielőtt kiadnánk a megfelelő SQL utasítást. Összetettebb esetekben több különböző úton is el lehet jutni az eredménytáblázatokhoz, melyek esetleg a végrehajtás hatékonyságában különbözhetnek egymástól.

A korábbi példa lekérdezés - azok az emberek akiknek nincsen autójuk - megfogalmazható például, a fent megadott utasítások mellett különbségképzéssel is.

SELECT nev FROM ember MINUS SELECT nev FROM ember e, auto a WHERE e.id = a.tul;

Azaz az összes embert tartalmazó halmazból kivonva az autótulajdonosokat, megkapjuk azokat az embereket, akiknek nincs autójuk. Természetesen ennél a műveletnél is ügyelni kell a két operandus táblázat szerkezeti azonosságára.

5.5. Az SQL DCL utasításai

Az SQL komponenseiből már csak egy maradt hátra, a DCL rész. Az SQL ezen a területen a *felhasználók jogosultságainak* nyilvántartási parancsaira és a tranzakció kezelés utasításaira korlátozódik. Az adatbáziskezelés egyik alapfeltétele ugyanis, hogy az adatbázist több felhasználó is használhatja, mindenki letárolhatja benne a saját adatait. Mivel az adatbázis több különböző felhasználó információit együttesen tárolja, az RDBMS-nek kell gondoskodnia arról, hogy mindenki csak a jogosult műveleteket végezhesse el, csak a jogosult adatokhoz férhessen hozzá. Az SQL rendszerekben, hasonlóan az operációs rendszerekhez, minden felhasználónak van egy azonosító neve és egy jelszava. Minden felhasználói névhez egy jogosultsági kör rendelhető. A SQL rendszerekben minden objektumnak, melyekből mi most a táblázatokat vettük, van egy tulajdonosa, mégpedig az a felhasználó, aki létrehozta. A táblázatokon elvégezhető műveleteket a jogosultság ellenőrzése szempontjából az alábbi csoportokra bontjuk:

SELECT (m1 [,m2...,mi])	: táblázat lekérdezése (olvasása),
INSERT	: táblázat bővítése,
DELETE	: táblázat rekordjainak törlése,
UPDATE (m1 [,m2...,mi])	: táblázat rekordjainak módosítása.

A fenti fontosabb műveleti kategóriák mellett más kategóriák is megadhatók, melyek többek között az indexelésre, az idegen kulcs hivatkozásra, vagy szerkezet módosításra vonatkozhatnak. Ha minden lehetséges jogot át kívánunk adni, akkor az

ALL

kategóriát kell kijelölni. Az engedélyezés során táblázatonként megadhatjuk, hogy az egyes felhasználóknak mely műveletek végrehajtását engedjük meg. Az UPDATE és SELECT esetén az engedély csak a felsorolt mezőkre vonatkozik, illetve ha nincs megadva mezőnév, akkor a teljes rekord módosítható vagy olvasható. Alapesetben csak a tulajdonosnak van joga a műveletek végrehajtására, és csak a tulajdonos adhat át jogokat a táblázatra vonatkozólag. Az operációs rendszerhez hasonlóan itt is léteznek privilégiumok, melyek lehetővé teszik, hogy bizonyos kitüntetett felhasználók, mint például a rendszergazdák, explicit jogosultság nélkül is elérhessenek bármely adatot az adatbázisban.

A *jogosultságok megadása* a *GRANT* utasítással történik, melynél meg kell adni, hogy mely táblázatra, mely műveleteket és kinek engedélyezünk. Az utasítás alakja:

GRANT művelet ON tábla TO felhasználó [WITH GRANT OPTION];

Az utasításban a művelet alatt egyrészt az előbb említett négy műveletet értjük, mely kibővül egy új kulcsszóval az ALL kulcsszóval, ami azt jelenti, hogy mind a négy műveletet engedélyezzük a megadott felhasználónak. Ha az opcionális *WITH GRANT OPTION* kifejezést is megadjuk, akkor a megadott felhasználó jogosulttá válik arra, hogy a most kapott jogokat továbbadhassa más felhasználóknak is. Azaz ezután már bárki megszerezheti a kiadott jogokat. A felhasználó neve helyett a PUBLIC kulcsszót használva, ugyanazt a jogot egyetlen utasítással kiadhatjuk az összes felhasználónak. A gyakorlat során előfordulhatnak olyan esetek, amikor a megadott jogokat vissza kell vonni valamilyen ok miatt, például az illető beosztása megváltozik, és ekkor a korábban kiadott jogok már nem alkalmazhatók.

A kiadott *jogosultságok megszüntetése* a *REVOKE* utasítással lehetséges, melyben meg kell adni a felhasználót, a visszavont műveletet és a vonatkozó táblázatot is. Az utasítás alakja:

REVOKE művelet ON táblázatnév FROM felhasználó;

Az utasítás hatására a kijelölt felhasználó többé már nem végezheti el a megadott műveleteket a megadott táblázaton. Az a felhasználó vonhatja vissza a jogosultságokat, akinek van joga adományozni is, azaz a tulajdonosnak.

Példaként vegyük az autó táblában való olvasási jog engedélyezését a gi476 azonosítójú felhasználónak. Itt az azonosítás az RDBMS-en belüli azonosítást jelenti, ami sok esetben lényegesen különbözhet az operációs rendszerben alkalmazott azonosítótól:

GRANT SELECT ON auto TO gi476;

Vonjunk vissza minden jogot az autó táblára vonatkozólag ugyanettől a személytől:

REVOKE ALL ON auto FROM gi476;

Egyes adatbázis kezelőkben az engedélyezés és visszavonás művelete mellett egy harmadik utasítás is él, amely az egyes műveletek szigorú letiltására szolgál. Ennek első hallásra nem sok haszna van, hiszen ha valamely műveletet nem kívánjuk engedélyezni, akkor vagy nem adjuk meg, vagy visszavonjuk a megfelelő jogosultságot. A gyakorlatban azonban a jogokat nem minden esetben egyenkénti művelet engedélyezésekkel vagy tiltásokkal adminisztrálják, mivel ez egy idő után igen nehezen kezelhetővé, áttekinthetetlenné válna a sok különböző jogosultsági lehetőség miatt. Emiatt az adatbáziskezelők lehetővé teszik, hogy a jogosultságokat nagyobb egységbe fogjuk össze, és ezen jogosultság halmazokat egységként adjuk át a felhasználóknak. Természetesen több ilyen jogosultsági csoport létezhet, ezért egy idő után már ezek tartalmát sem láthatjuk át egyszerűen. A másik probléma az, hogy egy későbbi, esetleg más DBA (Database Administrator, rendszergazda jogosultságokkal rendelkező adatbázis felhasználó) által adott csomag is tartalmazhatja a nem kívánt jogosultságot. Ezen esetekre gondolva a szigorú tiltás hatására a felhasználó semmilyen esetben sem tud élni a kijelölt jogosultsággal. A szigorú tiltás utasítása:

DENY művelet ON táblázat TO felhasználó;

A művelet a GRANT utasításnál használatos műveletekkel egyezhet meg. A DENY paranccsal megadott tiltás a legerősebb prioritású, tehát ha ezt követően egy másik GRANT műveletben engedélyeznénk a jogot, a DBMS nem engedné a végrehajtást, mert a DENY felülbírálja a GRANT utasítást. A DENY által kiadott tiltás szintén a

REVOKE művelet ON táblázat FROM felhasználó;

utasítással szüntethető meg.

A DBMS rendszerekben természetesen a táblák mellett más objektumokra is él az engedélyezés mechanizmusa, és jogosultság csoportok is kezelhetők. A védelmi elemek részletesebb bemutatása a könyv második kötetében szerepel.

A vezérlő utasításokhoz szokás sorolni a műveletvégrehajtást szabályozó, úgynevezett *tranzakció kezelő* utasításokat is. Az SQL szabványban két, a tranzakció végét jelző utasítást szokás definiálni. Az egyik utasítás a tranzakció sikeres befejezését jelenti, míg a másik a korábban végrehajtott tevékenységek visszagörgetését, azaz megsemmisítését írja elő. A tranzakció *sikeres befejezésének* utasítása:

COMMIT;

Ennek hatására a korábban végrehajtott tevékenységek véglegesítődnek, megőrződnek az adatbázisban. Ezzel szemben a

ROLLBACK;

hatására a korábbi tevékenységek *érvénytelenítődnek*, mintha ki sem adtuk volna őket. Ezzel, valamilyen hibával félbeszakadt műveletsort lehet törölni, visszavonni.

Nyilvánvalóan, csak a tranzakció hatáskörben végrehajtásra kerülő, adatbázist módosító SQL utasítások esetén van értelme kiadni ezeket a tranzakció kezelő utasításokat. Tehát a Query utasítás – ami csak olvassa az adatbázist, nem módosítja – és a DDL utasítások – amelyek nem tranzakció hatáskörben hajtódnak végre – esetében nem szoktuk alkalmazni ezeket, mert hatásuk nem érzékelhető.

Az 5.16. ábráról leolvasható, hogy csak a DML (adatkezelő) utasításokra teljesül e két feltétel. Ezek módosítják az adatbázist, de nem közvetlenül az adatbázis adatain manipulálnak, hanem egy úgynevezett adat-cache (munkaterület) adatain, és csak a tranzakció végén dől el, a tranzakció lezárásának módjától függően, hogy a módosított adatok átíródnak az adatbázisba (véglegesítődnek) vagy visszagörgetésre kerülnek.

5.6. A VIEW használata

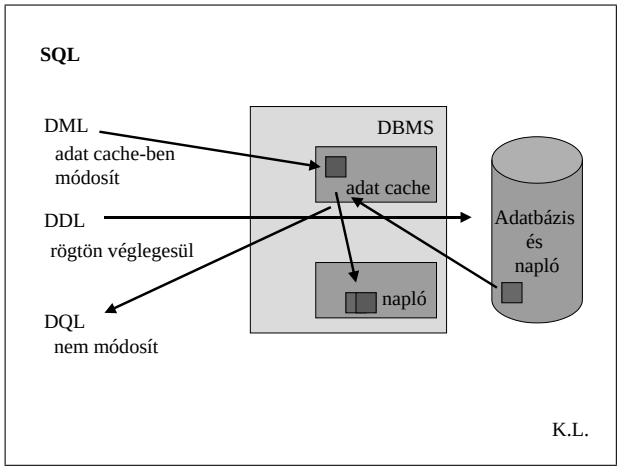
Az SQL alapok lezárásaként pillantsunk még vissza egy fontos utasításra, a VIEW-k létrehozására. Most már ismert egy adott műveletsor pontos megadásának szintaktikája. Az elkészült nézeti tábla a későbbi lekérdezési műveletekben ugyanúgy használható, mint az alaptáblák, így például összekapcsolható más táblákkal is. Nézzünk néhány mintautasítást a VIEW-k használatának gyakorlására.

– Hozzunk létre embaut néven olyan VIEW-t, amely az autók és tulajdonosuk adatait tartalmazza rekordonként. A VIEW az autó rendszámát, típusát és a tulajdonos nevét tartalmazza.

  CREATE VIEW embaut AS SELECT a.rsz, a.tip, e.nev FROM auto a, ember e WHERE a.tul = e.id;

– Kérdezzük le az embaut nézeti tábla tartalmát.

  *SELECT * FROM embaut;*



5.16. ábra. SQL utasítások hatása

Az eredménylista:

RSZ	TIP	NEV
bkx720	Opel	Bela
cmt111	Golf	Bela
aaa156	Trabant	Geza
lui999	Opel	Feri
kjs234	Lada	Bela

– Kérdezzük le az O betűvel kezdődő autótípusok tulajdonosainak nevét a nézeti táblából.

SELECT nev FROM embaut WHERE tip LIKE 'O%';

Az eredménylista:

NEV
Bela
Feri

– Szüntessük meg az elkészült nézeti táblát.

DROP VIEW embaut;

A fenti lekérdezést természetesen view használata nélkül, egyetlen egy SELECT utasítással is meg lehetett volna oldani. Viszont azt például, hogy melyik az az autótípus, amelyiknek a legmagasabb az átlagára, nem tudjuk egyetlen utasítással lekérdezni. Ennek az az oka, hogy először le kellene kérdezni, hogy melyik autótípusnak mennyi az átlagára (csoportképzéssel), majd ennek a lekérdezésnek az eredményét felhasználva meg kellene állapítani a legmagasabb átlagárral rendelkező autótípust. Tehát az utasítás összeállításakor alaptáblaként (FROM után) egy lekérdezés eredményét szeretnénk megadni, amit azonban az SQL nem támogat, ugyanis al-SELECT csak a szelekciós részben (WHERE után) szerepelhet. Vagyis szét kell bontanunk az utasítást két részre, és a csoportképzést tartalmazó lekérdezés eredményét el kell tárolnunk egy view-ban, ami már lehet egy másik lekérdezés alaptáblája. A megoldás tehát:

CREATE VIEW v AS SELECT tip, AVG(ar) atlagar FROM auto GROUP BY tip;
SELECT tip FROM v WHERE atlagar = (SELECT MAX(atlagar) FROM v);

Ezzel átvettük az SQL alapok szabványát, amely mint említettük, a létező RDBMS kezelő felületek magját fogja csak át. A későbbiekben áttekintjük az SQL néhány további elemét is, amely már átöleli és bizonyos esetekben túl is mutat a mai RDBMS rendszerek lehetőségein, és bepillantunk egy konkrét RDBMS-be, az Oracle SQL műveleteibe is.

Elméleti kérdések

1. Ismertesse az adatdefiníciós utasításokat (DDL).
2. Mutassa be a csoportképzés kijelölését és működését az SQL-ben.
3. Az SQL nyelv utasításcsoportjai.
4. Ismertesse az SQL al-SELECT elemét és a hozzá kapcsolódó operátorokat.
5. Adja meg az adatvédelem utasításait SQL-ben.
6. Mutassa be az SQL SELECT utasítás lehetőségeit, elemeit.
7. Ismertesse az SQL adatkezelő (DML) utasításokat.
8. Ismertesse a relációs modellben definiált integritási feltételeket, és adja meg SQL-beli megadásuk formátumát is.
9. Ismertesse a szelekciós feltételben alkalmazható operátorokat, relációjeleket.
10. Mutassa be a TABLE és a VIEW fogalmát, és létrehozásukat SQL-ben.
11. Ismertesse a join művelet definícióját, típusait, és megvalósításukat SQL-ben.
12. Adja meg az alábbi SELECT utasítás relációs kalkulusbeli megfelelőjét:

    SELECT A.B, Z.D FROM A, Z WHERE A.B < 3 AND A.K = Z.H;

13. A relációs algebra csoportképzés műveletének végrehajtása és az ide kapcsolódó parancs elemek megadása SQL-ben.
14. Készítsen SQL parancsállományt, mely az osztás műveletét végzi el.

Feladatok

*1. Adott **DOLGOZÓ** [KÓD, NÉV, BEOSZTÁS, FIZETÉS] és **PROJEKT** [NÉV, VÁROS, DOLGOZÓ] sémához adja meg az alábbi műveletek SQL megfelelőjét:
   - PROJEKT tábla létrehozása.
   - Kiss nevű dolgozók törlése.
   - Operátor beosztású dolgozók fizetésének növelése 10%-al.
   - Operátor beosztású dolgozók neve ABC sorrendben.
   - 120000 forintnál többet keresők darabszáma.
   - Miskolc városhoz tartozó projektek és azon résztvevő dolgozók neve.
   - Mely beosztásokban nagyobb az átlagfizetés 100000 forintnál.
   - Mennyi a projektekben nem dolgozók összfizetése.

2. Adott **DOLGOZÓ** [KÓD, NÉV, KOR, FIZETÉS, BEOSZTÁS, FŐNÖK, OSZTÁLY] és **OSZTÁLY** [KÓD, NÉV, ÉPÜLET] séma esetén végezze el az alábbi SQL műveleteket:
   - DOLGOZÓ tábla létrehozása.
   - Új osztály rekord felvétele.
   - Az operátor beosztású dolgozók fizetésének növelése 5%-al.
   - Dolgozók és osztályuk neve együtt.
   - A 33 évnél idősebb dolgozók és osztályuk neve együtt.
   - Mely beosztásokban nagyobb az átlagfizetés 100000 forintnál.
   - Az átlagnál többet keresők kódja.
   - Mely osztályon dolgozik Bonyolító beosztású ember.
   - A DOLGOZÓ tábla rekordok módosítási jogának átadása Zoltán felhasználónak.

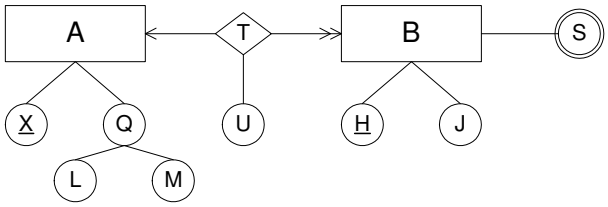
3. Adott az **EB** [NÉV, FAJTA, GAZDA, KOR] és a **GAZDA** [SZEMSZÁM, NÉV, VÁROS, UTCA] séma. Végezze el az alábbi műveleteket:
   - Miskolci kutyák nevei és fajtái.
   - Hány olyan kutya van, amelyiknek nem ismert a gazdája.
   - Hány 6 évnél fiatalabb kutya van az egyes fajtákból.
   - Kinek van a legtöbb kutyája.
   - Állítsuk párba azon gazdákat, akiknek azonos fajtájú kutyája van (képezzünk minden lehetséges párt).

4. Adott a **PILÓTA** [PKÓD, NÉV, KOR, BEOSZTÁS] és a **JÁRAT** [JKÓD, CÉL, DÁTUM, PILÓTA1, PILÓTA2] séma. Adja meg az alábbi lekérdezések SQL megfelelőjét:
   - Azon pilóták nevei akik utaznak x és y dátum között.
   - Mennyi azon pilóták átlagfizetése akik 40 évnél idősebbek.
   - Hány olyan pilóta van aki az átlagfizetés felénél többet keres.

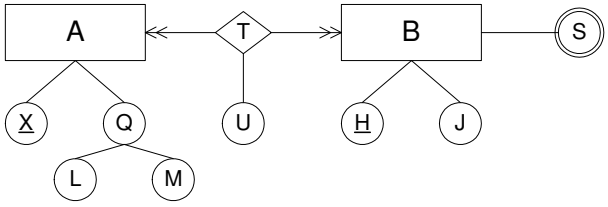
*5. Adottak az alábbi SQL adattáblák: **CSAPAT** [CSAPATNÉV, CSAPATKÓD, PONTSZÁM] és **JÁTÉKOS** [NÉV, CSAPATKÓD, KOR]. Végezze el az alábbi SQL utasításokat:
   - Szüntesse meg a JÁTÉKOS táblát.
   - Növelje a K-val kezdődő játékosok életkorát eggyel.

- Írassa ki mely csapatban hány játékos játszik. Ahol nincs játékos, ott 0 álljon.
- Listázza ki azon csapatokat, ahol nincs 20 évnél idősebb játékos.
- Azon játékosok, akik idősebbek csapatuk átlagéletkoránál.

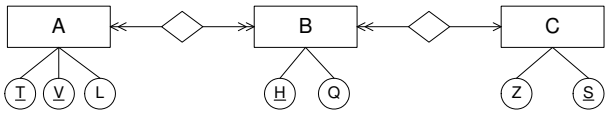
6. Adja meg az alábbi ER sémához a relációkat létrehozó SQL utasításokat:



*7. Hozza létre a relációkat, táblákat SQL paranccsal az alábbi ER sémához:



8. Hozza létre a relációkat, táblákat SQL paranccsal az alábbi ER sémához:



*9. Adottak az alábbi sémák: **OKTATÓ** [ID, NÉV, FOKOZAT] és **TÁRGY** [TID, CÍM, OKTATÓ, ÓRA]. Adja meg a megfelelő SQL utasításokat:

- Vigyen fel egy új tárgyat.
- Növelje a 3-as kódú oktatóhoz tartozó tárgyak óraszámát eggyel.
- Engedélyezze a TÁRGY tábla olvasását Péternek.
- A legtöbb tárgyat tanító oktató neve.

10. Adja meg a lekérdezések SQL parancsát, ahol az alábbi három tábla létezik: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

- Az Adatbázisok című könyv szerzőjének neve és lakcíme.
- A kiadók neve és a kiadott könyvek átlagára.
- Azon szerzők nevei, akiknek 5-nél több könyve van.

11. Adja meg az alábbi lekérdezések SQL utasítását, ahol a sémák: **TANFOLYAM** [TID, NÉV, TÍPUS, FELELŐS], **TÁRGY** [ID, CÍM, ÓRASZÁM, TANFOLYAMID].
    - Tárgyak átlagos óraszáma.
    - Tárgyak címét tanfolyamuk nevével együtt kilistázni.
    - Tanfolyamtípusok, és hány tanfolyam tartozik oda.
    - A 20 óránál rövidebb összóraszámú tanfolyamok.
    - A legtöbb tárgyat magába foglaló tanfolyam neve.
12. Adja meg az alábbi lekérdezések SQL parancsát, ahol a sémák: **TELEFON** [SZÁM, NÉV, VÁROS] és **HÍVÁS** [HÍVÓ, HÍVOTT, IDŐ, HOSSZ].
    - Telefon tulajdonosok nevei.
    - Hívások időpontja és a hívó neve.
    - A leghosszabb hívás időtartama.
    - Előfizető neve és hívásainak összidőtartama.
    - A legtöbb hívást kezdeményező ember neve.
13. Adja meg az alábbi lekérdezések SQL kifejezését, ahol a sémák: **TAG** [ID, NÉV, KOR] és **BEFIZETÉS** [ID, DÁTUM, ÖSSZEG].
    - A befizetések dátumai.
    - A 3500 Ft összegű befizetéssel rendelkező tagok nevei.
    - A befizetések összege adott x dátum után.
    - Azon napok, amikor a napi befizetés összege nagyobb, mint 5000 Ft.
    - A tagok átlagos befizetése (egyes tagok befizetéseinek átlaga).
14. Adottak az alábbi sémák: **OKTATÓ** [ID, NÉV, FOKOZAT], **TÁRGY** [TID, CÍM, OKTATÓ, ÓRA]. Adja meg a megfelelő SQL SELECT utasításokat:
    - Tárgyak címei.
    - A 8 órás tárgyak címe(i) és oktatójuk neve(i).
    - Milyen fokozatok vannak és hány oktató tartozik oda.
    - Azon oktatók, akiknek 4-nél több tárgyuk van.
    - A 'H' fokozatú oktatók száma, akiknek összóraszáma kisebb mint 20.
15. Adottak az alábbi sémák: **TELEFON** [SZÁM, NÉV, VÁROS] és **HÍVÁS** [HÍVÓ, HÍVOTT, IDŐ, HOSSZ]. A végrehajtandó SQL utasítások:
    - Szüntesse meg a TELEFON táblát.
    - Írja át a '412123' telefonszámhoz tartozó előfizető címét Miskolcra.
    - Vonja vissza a bővítési jogot a HÍVÁS táblára a Zoltán nevű felhasználótól.

*16. Adottak az alábbi sémák: **DOLGOZÓ** [KÓD, NÉV, FIZETÉS, BEOSZTÁS, ÜZEMKÓD] és **ÜZEM** [KÓD, NÉV]. A megadandó SQL parancsok:
    - Dolgozók létszáma.
    - 100000 forintnál többet keresők neve.
    - Dolgozók neve és üzemük neve.
    - Mennyi az egyes beosztásokban az átlagfizetés.
    - Az átlagnál többet keresők neve.

17. Adottak az alábbi sémák: **TAG** [ID, NÉV, KOR] és **BEFIZETÉS** [ID, DÁTUM, ÖSSZEG]. A megadandó SQL parancsok:
    - Törölje ki a TAG tábla összes rekordját.
    - Növelje a 'K' betűvel kezdődő tagok életkorát eggyel.
    - Hozzon létre VIEW-t a 30 évesnél fiatalabb tagok tárolására.
18. Adottak az alábbi sémák: **OKTATÓ** [ID, NÉV, FOKOZAT] és **TÁRGY** [TID, CÍM, OKTATÓ, ÓRA]. Adja meg a megfelelő SQL SELECT utasításokat:
    - Oktatók nevei ABC sorrendben.
    - Az 'egyetemi adjunktus' fokozatú oktatók tárgyai és az oktatók neve.
    - Azon oktatók akiknek nincs tárgya.
19. Adottak az alábbi sémák: **TELEFON** [SZÁM, NÉV, CÍM, KOR] és **HIVÁS** [HÍVÓ, HÍVOTT, IDŐ, HOSSZ]. Adja meg a megfelelő SQL SELECT utasításokat:
    - A miskolci telefonelőfizetők adatai.
    - Hívások kilistázása: hívó neve és a hívás hossza.
    - Az átlagos hívási időtartam.
    - Városonként a kezdeményezett hívások száma.
    - A hívást nem kezdeményezett személyek átlagéletkora.

*20. Végezze el az alábbi műveleteket SQL-ben, ha a felhasználói azonosítója JOE:
    - A DOLGOZÓ tábla olvasási jogának átadása a PETER felhasználónak.
    - PETER lekérdezi az engedélyezett táblát.
    - A DOLGOZÓ tábla rekord bővítési jogának megvonása minden felhasználótól.
    - A DOLGOZÓ táblára a rekord törlési jog átadása JOHN felhasználónak úgy, hogy ő továbbadhatja ezt a jogot.

*21. Végezze el az alábbi műveleteket SQL-ben, ha a felhasználói azonosítója JOE:
    - A DOLGOZÓ táblára minden műveleti jog kiadása JAMES felhasználónak.
    - A DOLGOZÓ tábla tartalom módosítási jogának engedélyezése minden felhasználó számára.
    - A DOLGOZÓ tábla NÉV mezőjére vonatkozó olvasási jogosultság átadása JOHN felhasználónak.
    - A DOLGOZÓ tábla rekord törlési jogának szigorú letiltása PETER felhasználó esetén.
22. Adott az **ÜGYFÉL** [UID, NÉV, CÍM, KOR, ÜGYNÖK] és az **ÜGYNÖK** [ID, NÉV, BEOSZTÁS] tábla. Végezze el az alábbi SQL utasításokat:
    - Új ügyfél rekord felvitele.
    - Az ÜGYFÉL tábla módosításának engedélyezése Péternek.
    - Az egri lakosú ügyfelek legidősebbikének életkora.
    - Azon ügynökök listája, akiknek 5-nél több ügyfele van.
    - Az 5-nél kevesebb ügyféllel rendelkező ügynökök ügyfelei.
    - Melyik az a város ahol a legalacsonyabb az ügyfelek átlagéletkora.

*23. Adottak az alábbi sémák: **DOLGOZÓ** [KÓD, NÉV, BEOSZTÁS, FIZETÉS, FŐNÖK, ÜZEM] és **ÜZEM** [NÉV, KÓD]. Adja meg a megfelelő SQL SELECT utasításokat:

- Azoknak a dolgozóknak a neve és főnökük neve, akiknek a fizetése 100000 forint.
- Üzemenként az átlagfizetés.
- Kik nem főnökök.
- Mely beosztásban dolgoznak 3-nál kevesebben.
- Melyik üzemben dolgozik Nagy nevű ember.
- Melyik az az üzem, ahol a legmagasabb az átlagfizetés.

*24. Az alábbi sémákhoz adja meg a megfelelő SQL parancsokat: **ZENESZÁM** [KÓD, CÍM, TÍPUS, ELŐADÓ] és **ELŐADÓ** [KÓD, NÉV, LAKCÍM].

- A népdal típusú számok törlése.
- Az ELŐADÓ tábla módosítási jogának engedélyezése Péternek.
- A miskolci előadók számainak címe.
- Hány zeneszám van az egyes típusokból.
- Mely városbeli előadóknak nincs népdal típusú száma.
- Azok az előadók, akiknek az átlagnál több zeneszámuk van.

6. fejezet

A relációs adatstruktúra helyességének vizsgálata

A relációs adatmodell eddigi ismertetésében a modell formai követelményeit, a modell felépítését adtuk meg. A megadott struktúra elemekkel, mint láttuk, egyazon problémakörre számos, egymástól lényegesen vagy árnyalatokban különböző modell készíthető. Ezek a modellek nem lesznek egyformán hatékonyak, egyes elkészült modellek tartalmazhatnak bizonyos szépséghibákat, amelyek a működés hatékonyságát is csökkenthetik.

Mint már említettük, a relációs séma megalkotásánál a fizikai megvalósítás és a kezelés hatékonyságára is gondolni kell. Most elsősorban a kezelés hatékonyságára koncentrálunk, és ebből a szempontból vizsgáljuk meg az elkészült sémákat. Mielőtt egy séma átalakításának lépéseit megnéznénk, mindenképpen meg kell említeni az adatbázis kialakítása közben fellépő legalapvetőbb hibákat.

A tervezés során sajnos több hibalehetőség is felmerülhet. Ezen veszélyforrások ismerete igen fontos a hibák kikerüléséhez, a hibamentes séma eléréséhez. Mivel az elkészült séma több különböző részből áll, ezek mindegyike egy-egy potenciális hibaforrást jelent. Így hibát véthetünk azzal, ha

- nem megfelelő relációkat hozunk létre,
- nem megfelelő mezőket alkotunk meg,
- nem megfelelő a mezők elnevezése,
- nem a megfelelő mezők kerülnek egy relációba,
- nem megfelelő a relációk kapcsolatának megvalósítása,
- nem megfelelő a mezők adattípusa,
- nem megfelelő a megadott integritási feltételrendszer.

Ezen hibák egy része, mint például a kapcsolatok ábrázolásának kérdése elkerülhető azzal, ha a szabályoknak megfelelően végezzük a szemantikai modell relációs modellre történő *konverzióját*. Többször is ellenőrizzük, hogy minden információelem, amit a *követelmény specifikáció* tartalmaz, benne van-e a relációs modellben.

A hibák másik része viszont olyan, hogy azok felderítésére még nem elegendő az

eddigiekben átvett ismeretanyag. E fejezet fő célja tehát az, hogy egy részletesebb áttekintést adjon a tervezés jellemző buktatóiról, a felmerült hibák felismeréséről és kijavításáról.

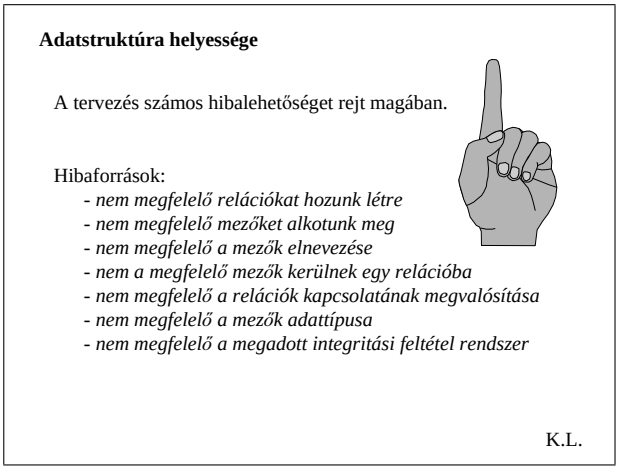
6.1. Mező elnevezési hibák

Az eddigiekben nem sokat törődtünk a mezők elnevezésének kérdésével. Feltettük ugyanis, hogy a mezőket elnevezni nem lehet probléma. Valóban nem jelenthet problémát nevet adni a mezőknek, ha nincs túl sok mezőnk. Mindjárt megváltozik a helyzet, ha a modellezett problémakör terebélyes méretű, több száz elemből álló. Az elemzésre váró feladat *mennyiségi növekedése új minőségi problémákat* hoz elő.

A probléma ebben az esetben az, hogy már nem lehet egyszerre fejben tartani az összes eddigi mező definíciót, ezért ha nem figyelünk a részletekre, akkor egyes mezők megadása ütközhet más mezők definíciójával, főleg az elnevezés területén. Nézzük meg előbb általánosságban, miféle veszélyek léphetnek fel az elnevezések területén:

- *Homonima*: azonos néven léteznek különböző jelentésű adatelemek.
- *Szinonima*: különböző néven léteznek azonos jelentésű adatelemek.
- *Nyílt logikai átfedés*: egy adatelem redundánsan kapcsolódik több különböző egyedhez, rekordhoz.
- *Név inkonzisztencia*: az elnevezés nem felel meg a tartalomnak vagy az elfogadott név kijelölési szabálynak.

Nézzük meg ezen hibalehetőségeket egy kicsit közelebbről, egy konkrét példán is. A példa adatbázisunk legyen egy vállalat osztályait, dolgozóit és projektjeit nyilvántartó rendszer, amelyben a következő táblákat találjuk:



6.1. ábra. Adatbázis-tervezési hibák

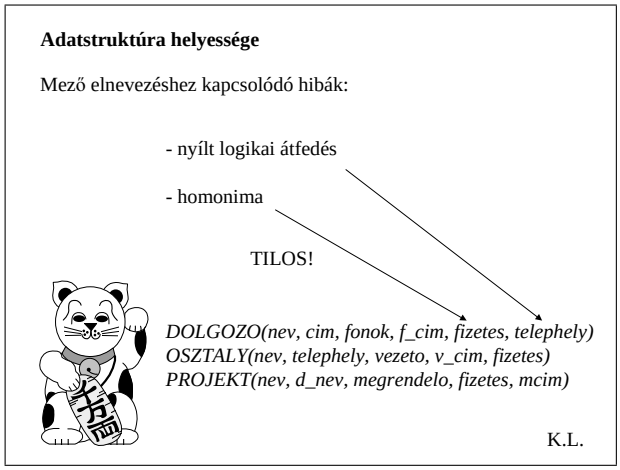
DOLGOZÓ					
név	*cím*	*főnök*	*f_ cím*	*fizetés*	*telephely*
C. D.	Rózsa u.	A. B.	Pipacs u.	500	Budapest

OSZTÁLY				
név	*telephely*	*vezető*	*v_ cím*	*fizetés*
Tervezés	Budapest	A. B.	Pipacs u.	500.000

PROJEKT				
megnevezés	*d_ név*	*megrendelő*	*fizetés*	*mcím*
Szoftverfejlesztés	C. D.	Nagy Kft.	50	Szolnok

Azonnal feltűnik a fizetés tulajdonság, amely mind a három egyedet jellemzi. Az OSZTÁLY és a DOLGOZÓ egyedben az osztály vezetőjének (aki egyben az osztályon dolgozó alkalmazott főnöke is) fizetését jelenti, míg a PROJEKT egyedben azt az összeget, amit a projektben dolgozó alkalmazott kapott. Az OSZTÁLY és a DOLGOZÓ egyedben a telephely tulajdonság is ismétlődik. (Ez a tulajdonság az osztály telephelyét jelenti.)

Vizsgáljuk meg ez utóbbi ismétlődést. A telephely mindkét egyedben ugyanazt jelenti és a tartalma is ugyanaz. Nem kapcsolódás miatt kell ismételni, hiszen egyik egyedben sem azonosító tulajdonság. Az ilyen esetben, ha egy tulajdonság több egyedtípushoz is kapcsolódik, azonos tartalommal és egyikben sem azonosító, akkor *nyílt logikai átfedés*ről beszélünk. Azaz redundancia lép fel. A vizsgált tulajdonság feleslegesen szerepel a DOLGOZÓ egyedben. A dolgozó adatai között



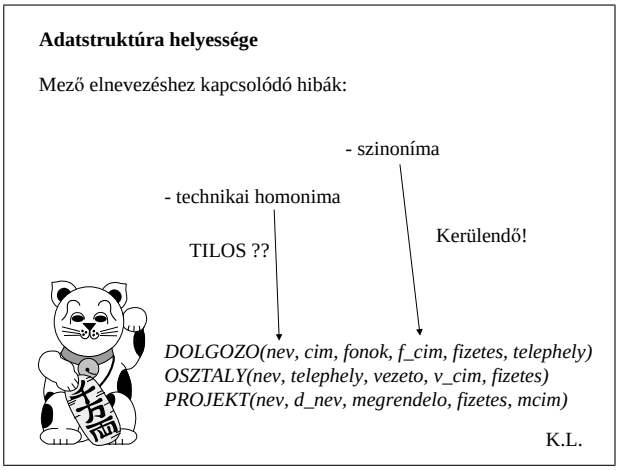
6.2. ábra. Mezőelnevezéshez kapcsolódó hibák I.

inkább az osztály nevét kellene szerepeltetni, hiszen az alapján kikereshető az osztály címe. Ez utóbbi megoldás azért is kívánatos lenne, mert a mostani táblában a dolgozó azonosítása nem megoldott, de a továbbiakban azzal a feltételezéssel élünk, hogy egy osztályon nem dolgozik két azonos nevű ember.

A másik probléma a fizetés tulajdonság volt. Mint említettük a PROJEKT táblában mást jelent, mint a másik kettőben. Ez az ún. *homonima*. Ha egy tulajdonság azonos névvel, de eltérő tartalommal kapcsolódik két különböző egyedhez, akkor *látszólagos logikai átfedés*ről beszélünk. A homonima igen nagy zavart okozhat a tervezés és alkalmazásfejlesztés későbbi fázisaiban, ezért igen fontos szabály, amire ügyelni kell a tervezés során, hogy *az adatmodellben nem lehet homonima*.

Nyílt logikai átfedések előfordulhatnak, ha például a mértékegység tulajdonság több helyen is előfordul azonos jelentésben. Homonimákat viszont semmiképpen sem engedhetünk meg. Akkor veszélyes különösen, ha az egyik egyedben azonosító, mert nem létező kapcsolatokat vélhetünk felfedezni. A megoldás egyszerű: például a PROJEKT-ben lévő fizetést nevezzük el mondjuk megbízási díjnak. A név mező esetében ugyanez a probléma. Használjunk helyette dolgozónév és osztálynév tulajdonságokat. Ha a kapcsolat és az azonosítás miatt a DOLGOZÓ egyedbe beépítjük az osztálynév mezőt, az már nem lesz nyílt logikai átfedés, mert az az OSZTÁLY-ban azonosító.

A következő probléma abból adódik, hogy egy tulajdonságot különböző nevekkel is illethetünk. A dolgozó főnöke és az osztály vezetője ugyanaz a személy. Ezek az úgynevezett *szinonimák*. Ha azonos tartalmú tulajdonság eltérő nevekkel kapcsolódik egyedekhez, akkor *rejtett logikai átfedés*ről beszélünk. Különösen akkor veszélyes ez, ha az egyik helyen azonosító, mert egy meglévő kapcsolat esetleg észrevétlen marad. Ezért az adatmodell elsődleges változatából a *szinonimákat ki*



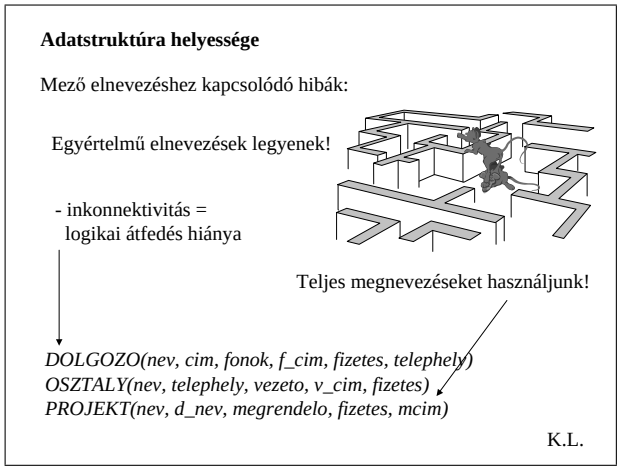
6.3. ábra. Mezőelnevezéshez kapcsolódó hibák II.

kell küszöbölni. A szinonimák használatát nem minden esetben kell kerülni. Ha a dolgozónak az osztály vezetője tényleg a főnöke, akkor miért ne lássa főnökként. Ebben az esetben le kell rögzíteni, hogy melyik az elsődleges név és melyik a szinonima. Egy név nem lehet több másiknak a szinonimája, mert akkor homonima lenne.

A DOLGOZÓ és az OSZTÁLY fizetés mezője nem nyílt logikai átfedés a definíció értelmében, mert a tartalmuk nem egyezik meg. A különbség mindössze annyi, hogy az egyik forintban, a másik pedig ezer forintban tárolja a főnök bérét. A definíció értelmében ezek homonimák. Az ilyen eseteket, amikor az elvi azonosság ellenére a technikai megvalósítás miatt (például más kódok, jelölések, osztályozások használata miatt) jön létre homonima, *technikai homonimának* nevezzük. Az elvi tartalom egyezősége ellenére a különböző értékkészlet okozza a problémát. A *név* valódi homonima, mert ott az elvi tartalom is különbözik. A modell nem tartalmazhat technikai homonimákat sem.

A v_cím és az f_cím szinonimák, mert a tartalmuk ugyanaz – az osztályvezető címe – azonban a nevük más. Igaz, hasonló. Ha két tényező tartalma és értékkészlete megegyezik, viszont a nevükben apró eltérés van (hasonlítanak egymásra), akkor *technikai szinonimák*ról beszélünk. Ezeket is ki kell küszöbölni, mert később senki sem fog emlékezni, hogy mit jelentett a rövidítés. Így kapcsolatokat is elveszíthetünk. Fontos szabály tehát, hogy *a megnevezéseknél mindig természetes és teljes megnevezéseket célszerű alkalmazni.*

Ékezetes és teljes hosszúságú neveket használva elkerülhetjük a félreértéseket. Például a *tul* jelenthet tulajdonságot és tulajdonost is. A tervezés fázisában még nem kell az alkalmazandó eszköz névkorlátait figyelembe venni. A neveknek egyértelműeknek kell lenniük, ez elengedhetetlen a későbbiekben tárgyalandó normalizáláshoz is.



6.4. ábra. Mezőelnevezéshez kapcsolódó hibák III.

A példaként felhozott adatbázisban a projekthez nem tudjuk a dolgozókat hozzákapcsolni, mert a dolgozó neve nem azonosítja egyértelműen a dolgozót. Valamilyen módon tehát egyértelműen kell azonosítani a dolgozót és az erre szolgáló mező(ke)t kell a PROJEKT táblába beépíteni. Hasonló a helyzet a DOLGOZÓ táblával, ahol a főnök neve nem azonosítja egyértelműen az osztályt, mert két osztálynak is lehet azonos nevű főnöke. Az ilyen adatkapcsolati hiány az *inkonnektivitás*. Az adatbázisból azért nem tudunk adatokat kinyerni, mert hiányoznak a megfelelő kapcsolatok. Ez a *logikai átfedés hiánya*. Azért kellemetlen, mert a redundáns elemeket könnyebb felderíteni, mint a hiányzókat.

Komoly hibaforrás lehet a nevek mögötti tartalmak tisztázatlansága. Ennek elkerülésére *beszélő nevek*et és rövid *értelmező magyarázatok*at kell használni. A tartalmak definiálásánál *korlátok*at lehet kikötni, amelyekben a felvehető értéktartományt határozhatjuk meg.

Mint említettük, a fenti hibák az egyes adatelemek jelentésének bizonytalanságaiból, a rendszer nem teljes körű átlátásából erednek, melyek elkerülésére különösen figyelni kell akkor, ha

- az adatbázist több különböző, helyi adatbázisból próbáljuk azok egymás mellé helyezésével felépíteni. Az adatbázis *egy*. Egyetlen adatbázis létezik, különben a redundanciák, homonimák és szinonimák kiküszöbölhetetlenek.
- a fejlesztőeszközt tartjuk szem előtt már ezen a szinten. Az adatmodell *fogalmi*. Nem a megvalósítással kell törődni, hanem a valóság helyes leírásával.
- a felhasználó és a fejlesztő nem értik meg egymást. A felhasználó zavaros fogalmakat használ és nem tudja igazán mi kellene neki, a fejlesztő pedig nem hallgatja meg a felhasználót és a saját kényelme, tudása szerint cselekszik. Az adatmodellt a felhasználónak és a fejlesztőnek *együtt* kell kidolgoznia.
- például bizonylatok vagy űrlapok és az azokban lévő adatok szolgálnak az adatbázis alapjául. Az adatmodell mind a bemenettől, mind a kimenettől *független*.
- menet közben módosulnak az adatmodell elemei, a korábban felépített elemek jelentése és kapcsolata már nem pontosan él a fejlesztők fejében, ezért ellentmondó, hibás elnevezések születnek.
- nem megfelelő, nem a feladat nagyságához méretezett a tervezési módszertan. Az alkalmazott eszköz nem képes a megnőtt méretek kezelésére.

Ha az adatmodell felépítésének buktatóin túljutottunk, és sikerült egy olyan sémát felépítenünk, ahol nincsenek homonimák, szinonimák, redundanciák, akkor elkezdhetjük a sémák vizsgálatát. A sémákat, mint már említettük, hatékonysági szempontokból vizsgáljuk.

6.2. Redundanciából eredő hibák

A helyes elnevezések kiválasztása után is maradhatnak olyan hibaforrások, amelyek az egyes mezők nem megfelelő társításából, a hibás rekord szerkezet kialakításából fakadnak. A hibás reláció séma tervezés egyik lényeges és káros következménye a felesleges, nem szabályozott *redundancia*. A redundancia az egyes adatelemek többszörös letárolását, megismétlését jelenti. A redundancia hatását a következő példán keresztül szemléltetjük.

Képzeljünk el egy olyan adatbázist, ahol egy vállalat minden osztályáról nyilvántartjuk az osztály nevét, vezetőjét, az osztály címét, vezetőjének címét, a beosztottak nevét és címét, valamint a beosztottak végzettségeit. Az osztályok fontosabb adatait magába foglaló tábla egy részlete:

VÁLLALAT						
osztály	*osztcím*	*osztvez*	*osztvcím*	*beosztott*	*beosztcím*	*végzettség*
Tervezés	Budapest	A. B.	Pipacs u.	C. D.	Rózsa u.	Informatikus
Tervezés	Budapest	A. B.	Pipacs u.	C. D.	Rózsa u.	Közgazdász
Tervezés	Budapest	A. B.	Pipacs u.	E. F.	Tulipán u.	Gépész
Értékesítés	Miskolc	G. H.	Kankalin u.	I. J.	Rózsa u.	Közgazdász
Értékesítés	Miskolc	G. H.	Kankalin u.	K. L.	Hóvirág u.	Jogász

Mint látható, a VÁLLALAT reláció tartalmazza azon osztály nevét (osztály), címét (osztcím) és az osztályvezető nevét (osztvez) illetve címét (osztvcím), ahol a dolgozó (beosztott) éppen most dolgozik. Ezen kívül még nyilvántartjuk a dolgozó címét (beosztcím) és végzettségét (végzettség). Ez a megvalósítás viszont azt is jelenti, hogy annyiszor kell letárolni az osztály adatait, ahány dolgozója van az osztálynak, sőt, ahány végzettsége van a dolgozóknak. Mindez felesleges adatletárolást jelent, hiszen ugyanaz az információ sokszorozva foglal helyet. A rossz relációséma tehát redundanciához vezethet.

A redundanciát kerülni kell, mert amellett, hogy feleslegesen több helyet foglal, inkonzisztenciát is okozhat. Így például egy többszörösen tárolt adatérték mellett a mögötte álló információelem módosítása minden egyes adatelőfordulás módosítását igényli. Ha azonban nem mindegyik előfordulásnál hajtódik végre ugyanaz a módosítás, akkor inkonzisztensek, ellentmondásosak lesznek az egyes példányok. A többszörözés mindig inkonzisztencia veszéllyel jár, mint ahogy mondani szokták: akinek egy órája van, az tudja az időt, de akinek több van, az nem tudja a pontos időt.

A redundancia mellett számos egyéb műveleti nehézséget is okozhatnak a modell hiányosságai. A nem megfelelő relációs sémából eredő problémákat szokás *anomáliák*nak nevezni. Az anomáliák legfontosabb típusai:

- *Beszúrási anomália*: amikor egy rekord felvitelekor felesleges, már letárolt információkat is újra fel kell vinni. Például minden új dolgozó felvitelénél újra fel kell vinni az osztály adatait is.
- *Módosítási anomália*: amikor egy információegység módosításához több helyen is kell módosítani az adatbázisban. Ha például megváltozik az

osztály címe, akkor minden egyes, azon az osztályon dolgozó alkalmazott rekordjában el kell végezni a módosítást. Ez nem csak többletmunkát jelent, de megnöveli az inkonzisztens állapot valószínűségét, ha valahol elmaradna a módosítás.

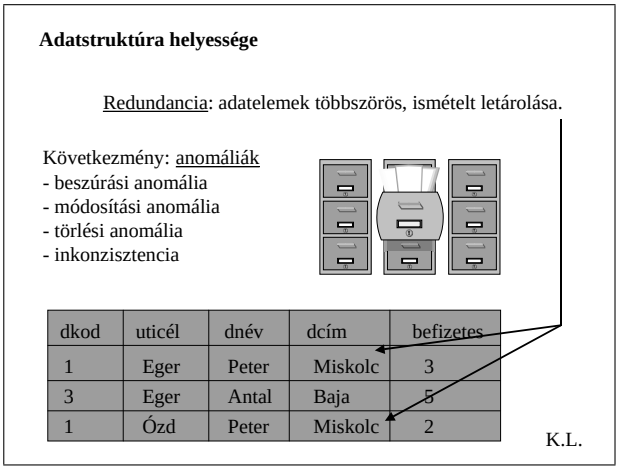
– *Törlési anomália*: a törlési anomália azt jelenti, hogy egy információelem megszűnésekor más, hozzá nem tartozó információk is elvesznek. Ha például máshol nem lenne letárolva az osztály címe, akkor az adott osztályon dolgozók kitörlésével az osztályra vonatkozó adatok is törlődnek.

A felsorolt anomáliák láthatóan mind abból származtak, hogy nem az igazán összetartozó adatokat vettük be egy relációba. Bizonyos adatok feleslegesen, de szükségszerűen ismétlődtek. Ez abban nyilvánult meg, hogy bizonyos összetartozó mezők többször is előfordultak. Például az osztály neve, az osztályvezető neve, címe és az osztály címe mindig együttesen fordul elő, hiszen ezek az adatok szorosan összetartoznak. Ezeket érdemesebb egy külön relációban összefogni, ahol nem keverednek olyan hozzájuk nem tartozó adatokkal, mint például a beosztott címe.

Hogy mely mezők tartoznak igazán egy relációba, azt a mezők közötti összetartozási viszony, a mezők közötti *függőség*ek határozzák meg. A legfontosabb függőségi típus a funkcionális függőség (*Functional Dependency, FD*). Az FD, amely két mezőcsoport között léphet fel, az alábbi jelentéssel bír:

> *A B attribútum csoport funkcionálisan függ az A attribútum csoporttól, azaz A meghatározza B-t (jele: A → B), ha az A minden értékéhez a B maximum egy értéke kapcsolható.*

A mezőcsoportok lehetnek elemiek, azaz a mezők is állhatnak egymással funkcionális függőségben. A functional kifejezés arra utal, hogy ez egy függvényszerű függőség, hiszen a függvény definíciójában szerepel az a megkötés, hogy minden elemhez maximum egy egyértelmű hozzárendelés létezik. A funkcionális függőséget formálisan is megadhatjuk:



6.5. ábra. Redundancia → anomáliák

Ha r(R), és X, Y ⊆ R, akkor r kielégíti az X → Y funkcionális függőséget ⇔ $|\pi_Y(\sigma_{X=x}(r))| \leq 1$ ∀x, X tulajdonságértékre.

Más oldalról vett megközelítést ad a következő megfogalmazás:

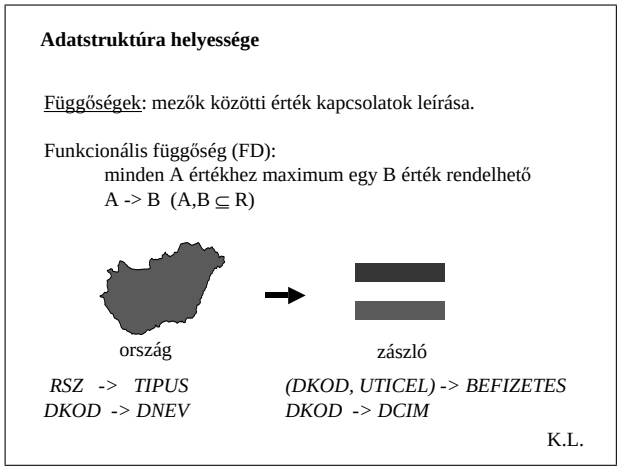
Egy adott R sémában az A, B ⊂ R attribútumokra, ha ∀ R feletti r relációban ∀ t_1, t_2 rekordra $t_1(A) = t_2(A) \Rightarrow t_1(B) = t_2(B)$, akkor $A \rightarrow B$.

Szavakkal megfogalmazva az előző definíciókat: egy sémában egy B attribútum halmaz funkcionálisan függ az A-tól, ha bármely két sor, amely az A értékeiben megegyezik, szükségszerűen a B értékeiben is megegyezik.

Példaként a személyi_szám és a név mezőket nézve, az ember egyednél minden személyi szám értékhez egyetlen egy név rendelhető csak. Ilyenkor azt mondjuk, hogy a név mező funkcionálisan függ a személyi_szám mezőtől. Két adatelem között a funkcionális függőséget egy nyíllal jelöljük, és azt mondjuk, hogy a név funkcionálisan függ a személyi_számtól, illetve a személyi_szám funkcionálisan meghatározza a nevet. A név és személyi_szám között fennálló funkcionális függőség jelölése: *személyi_szám → név*. Ezzel szemben a *személyi_szám ← név* függőség nem teljesül, hiszen a *név* mező egy értékéhez több különböző személy és személyi szám is rendelhető.

Egy relációs séma mezőhalmazán nagyon sok funkcionális függőséget értelmezhetünk. A VÁLLALAT sémánkban például az alábbi függőségek fedezhetők fel:

osztály → osztcím
osztály → osztvez
osztvez → osztvcím (az osztályvezető azonosítására csak a nevét használtuk, feltételezve, hogy nincs két azonos nevű osztályvezető)



6.6. ábra. Funkcionális függőség

osztály → *osztvcím* (!)
(*beosztott*, *osztály*) → *beosztcím* (tételezzük fel, hogy egy osztályon nem dolgozik két azonos nevű ember, azaz a vállalaton belüli azonosításra a dolgozó nevét és munkahelyét együtt használjuk)
(*osztály*, *osztvez*) → *osztvcím*
(*beosztott*, *osztály*, *osztcím*) → *beosztcím*
(*osztvez*, *beosztott*) → *beosztott*

Még sok hasonló függőséget fedezhetünk fel, azonban nyilvánvaló, hogy bizonyos függőségek kevésbé lényegesek, míg mások elengedhetetlenül fontosak. Vegyük például az (*osztály*, *osztvez*) → *osztvcím* és az *osztvez* → *osztvcím* függőségeket. A két függőséget elemezve látható, hogy az elsőnek biztosan teljesülnie kell, ha a második függőség teljesül. Ezért a második függőség teljesülése magával hozza az első függőség teljesülését. Emiatt a különböző függőségek között is létezik kapcsolat, amire támaszkodva bizonyos függőségekből újabb függőségek vezethetők le. Ebben a példában tehát az *osztvez* → *osztvcím* függőség lényegesebb szerepet játszik mint az első.

A függőségek fontosságát elemezve látható, hogy az (*osztvez*, *beosztott*) → *beosztott* FD sem jelent igazából új ismeretet a számunkra. Ugyanis triviális, hogy egy adatelem funkcionálisan meghatározza önmagát (vagy mezőben gondolkodva egy mezőt kijelölve egy adott értékhez mindig csak önmaga rendelődik a táblázaton belül, azaz A → A mindig teljesül). Tehát ez a fajta függőség sem tekinthető lényegesnek.

Ezek az úgynevezett *triviális függőségek*, hiszen minden esetben teljesülnek az A és B megválasztásától függetlenül. Nyilvánvaló, hogy egy osztályvezető és beosztott pároshoz csak egy beosztott tartozhat. Minden ilyen triviális függőség érvényes minden sémában.

A nem lényeges függőségek azok, melyek más függőségekből következnek vagy triviálisak. Az egyes függőségek egymás közötti levezethetőségének szabályait az *Armstrong-axiómák* fogalmazzák meg. A következő Armstrong-axiómák léteznek:

1. Ha $A \supset B \Rightarrow A \rightarrow B$. Ez azt jelenti, hogy egy halmaz részhalmaza mindig funkcionálisan függ a halmaztól (egész meghatározza a részét). Ezek az úgynevezett *triviális függőség*ek.

2. Ha $A \rightarrow B \Rightarrow CA \rightarrow CB$. Ez azt jelenti, hogy egy funkcionális függőség mindkét oldalát kibővíthetjük egy új halmazzal, és a kapott két új oldal között fennmarad a funkcionális függőség. Ez a *bővítési* szabály.

3. Ha $A \rightarrow B$, $B \rightarrow C \Rightarrow A \rightarrow C$. Ez a funkcionális függőségek *tranzitivitás*át jelenti. Szavakkal: ha C olyan halmaztól függ funkcionálisan, amelyet egy másik halmaz, nevezetesen A funkcionálisan meghatároz, akkor a C is funkcionálisan függ az A-tól.

Ezekből az axiómákból különböző tételek vezethetők le. Vegyük például a szétvághatósági szabályt:

A → BC ⇒ A → B, A → C

Ez azt mondja ki, hogy az A-tól funkcionális függő attribútum halmaz elemei külön-külön is funkcionálisan függenek az A-tól.

Bizonyítás: az 1. axiómából BC → B triviális függőség; a 3. axiómából A → BC, BC → B ⇒ A → B. A szabály másik része hasonlóképpen bizonyítható.

Második példánk legyen az összevonhatósági szabály:

$$A \to B,\ A \to C \Rightarrow A \to BC$$

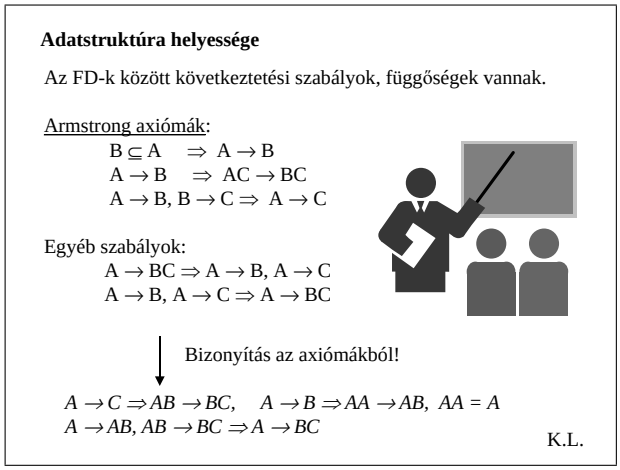
azaz az A-ból kiinduló függőséghalmaz helyettesíthető egyetlen függőséggel. Ez a következő módon bizonyítható.

Bizonyítás: a 2. axiómából AA → AB; ahonnan A → AB, hiszen az A saját magával bővítve ismét csak A, illetve AB → BC. Ebből az utóbbi két eredményből a 3. axióma alapján A → BC.

Végül a példánkban talált függőségek közül bizonyítsuk be a *(beosztott, osztály, osztcím)* → *beosztcím* függőséget.

Bizonyítás: az 1. axióma szerint (beosztott, osztály, osztcím) → (beosztott, osztály). A 3. axióma segítségével (beosztott, osztály, osztcím) → (beosztott, osztály), (beosztott, osztály) → beosztcím ⇒ (beosztott, osztály, osztcím) → beosztcím.

A triviális függőségek ellentétei a nem triviális függőségek. Ekkor $A \to B$ és $B \not\subset A$, vagyis a B halmaz legalább egyik eleme nincs benne az A halmazban. Ezeket nevezik *nem triviális* FD-knek. Ha a B egyik eleme sincs benne az A halmazban, akkor *teljesen nem triviális* funkcionális függőségekről beszélünk.



6.7. ábra. Armstrong-axiómák

Látható, hogy a kezdetben adott függőségekből az Armstrong-axiómák segítségével számos további függőséget vezethetünk le. Egy adott függőségi halmazból sok más függőség vezethető le, de természetesen nem minden függőség. A levezethető függőségek köre függ a függőségi alaphalmaztól. A következő definíciók ezen halmazra vonatkoznak.

Adott egy F funkcionális függőségi halmaz. Az F elemeiből az Armstrong-axiómák alkalmazásával képezhető összes függőség halmazát az F lezártjának nevezzük. Jele: F^+. A lezárt fogalom segítségével megállapíthatjuk két eltérő elemszámú FD halmaz ekvivalenciáját. Az F_1 és F_2 FD halmazok ekvivalensek, ha $F_1^+ = F_2^+$. Ez azt jelenti, hogy a két FD halmaz közül bármelyik reprezentálhatja az adott séma függőségi viszonyait. Ilyen esetben célszerű a kisebbet választani. Az FD halmaz lezártja véges, és úgy képezhető, hogy az alap FD halmaz minden elemét kipróbáljuk minden Armstrong-axiómával. Azaz az Armstrong-axiómákkal, mint operátorokkal új függőségeket alkotunk. Ha a kapott függőség még nincs benne a lezártban, akkor a lezártat kibővítjük vele. Ha a képzett függőség az eredeti FD halmazban is benne van, akkor a kiindulási halmazunk nem a legkisebb lehetséges halmaz, mert nem független FD-ket is tartalmaz.

A funkcionális függőségek legtömörebb halmazát *irreducibilis FD halmaz*nak nevezzük. Ez az a legkisebb FD halmaz, amely segítségével előállítható a sémában előforduló összes funkcionális függőség. Ebben a halmazban a következő tulajdonságok teljesülnek:

- minden függőség jobboldala elemi;
- a baloldal nem csökkenthető, azaz tovább nem egyszerűsíthető;
- ebből a halmazból nem hagyható el egyetlen egy függőség sem az ekvivalencia megsértése nélkül.

Más szavakkal, ha az irreducibilis FD halmazból akár egy függőséget hagynánk el, akkor a séma függőségi viszonyai már nem lennének előállíthatók. A VÁLLALAT sémában egy irreducibilis FD halmaz a következő:

osztály → *osztcím*
osztály → *osztvez*
osztvez → *osztvcím*
(beosztott, osztály) → *beosztcím*

Az FD halmazban minden jobboldal elemi, a baloldal pedig csak az utolsó esetben összetett, azonban az sem egyszerűsíthető, mert a beosztottat a neve és munkahelye azonosítja. Ennek a halmaznak a segítségével bármelyik, már felírt függőség előállítható, mint azt az Armstrong-axiómáknál láttuk. Ha azonban például az utolsó függőséget elhagynánk, akkor ez már nem lenne ekvivalens a fejezet elején bemutatott FD halmazzal, hiszen az előbb említett bizonyítást sem tudnánk véghezvinni.

A funkcionális függőségek felderítése nagy segítséget nyújt a helyes séma kialakításában. Ugyanis a funkcionális függőségekkel magyarázható a redundancia is,

mert ha egy ismétlődő értékű mezőből FD indul ki, akkor az FD definíciója alapján a függő mezőnek is ismétlődnie kell. Tehát a redundancia oka a nem megfelelő, felesleges FD a reláció sémán belül. A tétel pontos megfogalmazása:

> *Adott $R(A_1, A_2, \ldots, A_n)$ séma és a benne lévő $A_i \rightarrow A_j$ funkcionális függőség; ezután ha A_i ismétlődik $\Rightarrow$ A_j is ismétlődik. Ez okozza a redundanciát.*

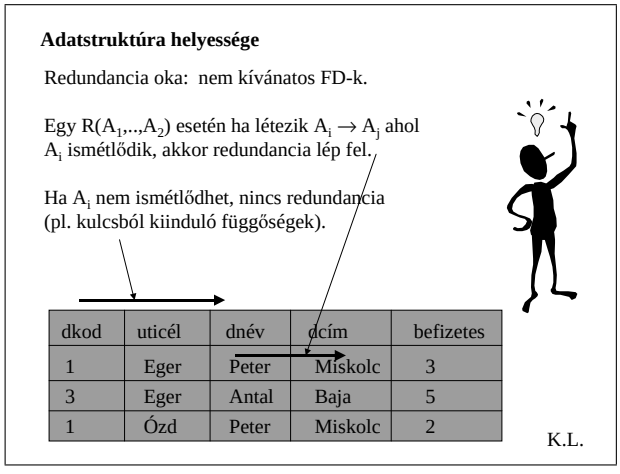
A helyes séma megtervezésére irányuló irányelvek és módszertan az irodalomban *normalizálás* néven vált közismertté.

6.3. Normalizálási lépések

A *normalizálás* egy tervezési metodika, amely segítséget nyújt a helyes, anomália-mentes, relációs sémák és adatbázis sémák kialakításában. A normalizálás célja az olyan $A_i \rightarrow A_j$ függőségek megszüntetése, ahol az A_i ismétlődhet. Ezt két módon érhetjük el. Vagy ne legyen függőség, vagy az A_i ne ismétlődhessen. A cél az, hogy ha kiindul FD az A_i-ből, akkor értéke ne ismétlődhessen. Mivel a nem ismétlődő érték elsősorban a kulcs mezőcsoportra jellemző, ezért általánosságban azt mondhatjuk, hogy FD csak kulcsból induljon ki.

A nem kívánt függőségek megszüntetését egy eljárás sorozattal adják meg, méghozzá több, egymásra épülő követelmény alakjában. Az egyes követelményeket szokás *normálformáknak* is nevezni. A normalizálás erről az oldalról nézve nem más, mint a megadott normálformák teljesülésének ellenőrzése, illetve az adatsémák átalakítása olyan alakra, hogy azok már kielégítsék a megadott normálformákat.

A normalizálás néhány rögzített irányelven alapszik, amelyek iránymutatást adnak a tervezéshez, helyes mederbe terelve a modellezés menetét. A *dekompo-*



6.8. ábra. A redundancia oka

zíció során az induló séma felbontásával emeljük ki a nem kívánt FD-ket külön relációkba, a felesleges FD-t tartalmazó sémát pedig dekompozícióval hozzuk normalizált alakra. A dekompozíciós módszerben felállított tervezési irányelveket követelmények formájában adják meg.

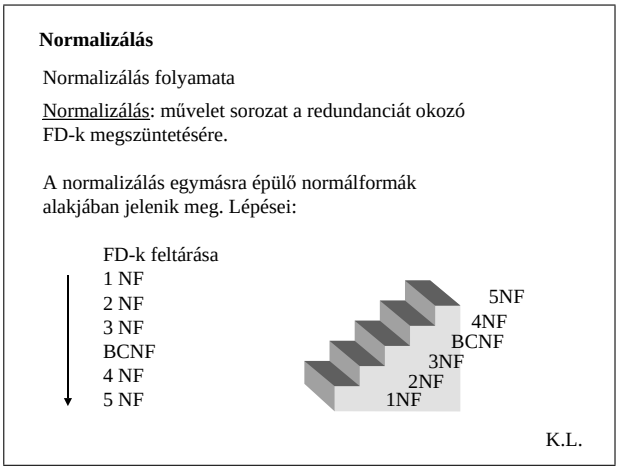
A normalizálás a sémák helyességét az adatsémában fennálló függőségeken keresztül méri, ellenőrzi. A normalizáció elengedhetetlen előfeltétele tehát, hogy pontosan ismerjük a séma elemei között fennálló funkcionális függőségi viszonyokat. Az egyes normalizációs lépések célja a függőségi viszonyok különböző szempontok szerinti ellenőrzése, vizsgálata. Az egyes normálformáknak megfelelő ellenőrzéseket, vizsgálatokat és módosítási lépéseket *normalizációs lépéseknek* nevezzük. Minden normálforma egy követelményrendszer alakjában jelenik meg. E normálformák egymásra épülése azt jelenti, hogy egyes normálformák rendszerint megkövetelik az alsóbb normálformák teljesülését. Az egyes normálformák egy egyre szigorodó feltételrendszert jelentenek. A feltételek egymásra épülése alapján az egyes normálformákat rangsorba lehet állítani.

A rangsor alján elhelyezkedő, leglazább feltételt nevezzük első normálformának. Az őt követő feltételt, amely csak az első normálforma teljesülését követeli meg, második normálformának nevezik. Ezt további, egyre növekvő sorszámú normálformák követhetik.

Szerencsére azért nem annyira terebélyes ez a rendszer, az irodalomban öthat szintet szoktak megemlíteni, a gyakorlat szempontjából azonban még ettől is kevesebb, csak az első három normálformának van jelentősége.

> *Első normálformában van a relációs séma, ha minden mezője funkcionálisan függ a kulcsmező csoporttól.*

Eszerint a függőségi rendszerben *léteznie kell egy kulcsnak*, és minden más mezőnek ettől kell függnie. Ez a megkötés láthatóan több feltételt is magába fog-



6.9. ábra. A normalizálás folyamata

lal. Egyrészt megkívánja az egyed integritási feltétel teljesülését, azaz hogy legyen kulcsa a relációnak. Másrészt megköveteli, hogy *minden mezője atomi értéket hordozzon.* Ez a megkötés egyben a relációs modell egyik alappillére is.

Formálisan: $\exists$ K $\subseteq$ R, hogy $\forall$ A $\subseteq$ R esetén K $\rightarrow$ A teljesül.

Példánkban a végzettség mező nem teljesíti az atomiságra vonatkozó feltételt, így ki kell emelnünk a sémából. Azonban valamiképpen meg szeretnénk tartani azt az információt, hogy kinek mi a végzettsége. A dolgozó azonosítására viszont a dolgozó nevét és munkahelyét használjuk. Ezért a létrejövő új sémába mind a beosztott, mind az osztály, mind a végzettség mezőnek be kell kerülnie. Azaz a többértékű mezővel együtt kiemeljük az(oka)t a mező(ke)t is, amely(ek)hez kapcsolódik. Itt jegyezzük meg jól azt, hogy minden átalakításnak úgy kell megtörténnie, hogy ne veszítsünk közben információt!

A felbontás célja a redundancia csökkentése információvesztés nélkül. Ezt hívjuk *veszteségmentes dekompozíció*nak. Más szavakkal ez azt jelenti, hogy a létrejövő új táblákból újra elő kell tudnunk állítani az eredeti táblát. Vagyis az új relációk join-ja az eredeti relációt adja vissza. A mi esetünkben két új reláció lesz. Nevezetesen

V1 := {osztály, osztcím, osztvez, osztvcím, beosztott, beosztcím} és
VÉGZETTSÉG := {beosztott, osztály, végzettség}.

Láthatjuk, hogy a redundancia valóban csökkent, mert az eredeti táblához képest eltűnt a második sor, amely csupán egy új információt hordozott. Azonban ez az információ sem veszett el, mert a két tábla join-jával előállítható az eredeti reláció. A két új reláció:

V1					
osztály	*osztcím*	*osztvez*	*osztvcím*	*beosztott*	*beosztcím*
Tervezés	Budapest	A. B.	Pipacs u.	C. D.	Rózsa u.
Tervezés	Budapest	A. B.	Pipacs u.	E. F.	Tulipán u.
Értékesítés	Miskolc	G. H.	Kankalin u.	I. J.	Rózsa u.
Értékesítés	Miskolc	G. H.	Kankalin u.	K. L.	Hóvirág u.

VÉGZETTSÉG		
osztály	*beosztott*	*végzettség*
Tervezés	C. D.	Informatikus
Tervezés	C. D.	Közgazdász
Tervezés	E. F.	Gépész
Értékesítés	I. J.	Közgazdász
Értékesítés	K. L.	Jogász

Láthatjuk, hogy a kulcsmezők lényeges szerepet játszanak a relációs sémákban, hiszen ki kell tudnunk választani, hogy mely attribútum(ok) lesz(nek) a kulcsmező(k). A kulcsmező az, amely egyedi és egyértelműen azonosítja a teljes rekordot. Pontosabban minden kulcsértékhez maximum egy érték tartozik a

többi attribútumnál, azaz a többi mező funkcionálisan függ a kulcstól: *kulcs* → *többi_mező*. Ez feltételezi, hogy minden mezőbe irányul funkcionális függőség.

A kulcs pontosabb definíciója érdekében vezessük be a következő fogalmat: legyen L ⊂ R, és legyen K egy FD halmaz, ekkor az $L(K)^+ \subset R$, *L-nek a K-ra vonatkozó lezártja*, azaz azon attribútumok halmaza, melyek függnek L-től a K-beli függőségek alapján. A példánk alapján legyen

L := {*osztály*},
K := {*osztály* → *osztvez*}.

Ekkor

$L(K)^+ = \{osztály, osztvez\}$.

Ha a K-ban szerepel még az

osztvez → *osztvcím*

függőség is, akkor

$L(K)^+ = \{osztály, osztvez, osztvcím\}$.

A *szuperkulcs* olyan részséma, attribútum csoport, amelynek a séma FD halmaza feletti lezártja a teljes séma. Ha

L = {*osztály*, *beosztott*, *osztvez*} és
K = {*osztály* → *osztcím*, *osztály* → *osztvez*, *osztvez* → *osztvcím*, *beosztott* → *beosztcím*},

akkor

$L(K)^+ = V1$.

Próbáljunk kisebb szuperkulcsot találni a V1 sémában. Először nézzük az {*osztály*, *beosztott*} párost. Azt találjuk, hogy ennek a K-ra vonatkozó lezártja maga a séma, azaz ez a két mező együtt szuperkulcs. Ha viszont az {*osztály*, *osztvez*} párost próbáljuk ki, rájövünk, hogy a lezártjában nincs benne a *beosztcím* mező. Ez a két mező együtt tehát nem szuperkulcs. Hasonlóképpen belátható, hogy több szuperkulcsot is találhatunk, de mindegyikben benne lesz az {*osztály*, *beosztott*} páros. A VÉGZETTSÉG sémában a szuperkulcs az {*osztály*, *beosztott*, *végzettség*}.

A *jelölt kulcs* olyan szuperkulcs, amely nem tartalmaz más szuperkulcsot, azaz minimális. A példánkban, mint az előzőekben láttuk, ez az {*osztály*, *beosztott*} attribútumpár a V1-ben. Ennél kisebb nincs, mert bármelyiket hagyjuk is el, a másik lezártja nem egyezik meg a teljes sémával. Egy sémában több jelölt kulcs is lehet. Ha a *személyi_szám* és *név* mezőket tartalmazó EMBER sémát kibővítjük a *személyi_igazolvány_szám* mezővel, akkor már két jelölt kulcsunk van: a *személyi_szám* és a *személyi_igazolvány_szám*. Miért hívjuk ezeket jelölt kulcsnak? Mert belőlük lehet kulcs, azaz a kulcs betöltésére ők a jelöltek.

*Elsődleges kulcs*nak nevezzük azt a jelölt kulcsot, amelyet azonosításra választottunk ki. Bármely jelölt kulcs lehet elsődleges kulcs. Például az EMBER sémában mind a *személyi_igazolvány_szám* mind a *személyi_szám* lehet elsődleges kulcs. Az elsődleges kulcsot aláhúzással jelöljük. A V1 sémában csak egy jelölt kulcsunk van, az {*osztály, beosztott*}, így ez egyben az elsődleges kulcs is. A VÉGZETTSÉG sémában csak egy szuperkulcsunk van, így az lesz a jelölt és egyben az elsődleges kulcs is: {*osztály, beosztott, végzettség*}.

> *Második normálformában van a reláció, ha az első normálformát teljesíti és ezen felül minden nem kulcs mező a teljes kulcstól függ, de nem függ a kulcs bármely valódi részhalmazától.*

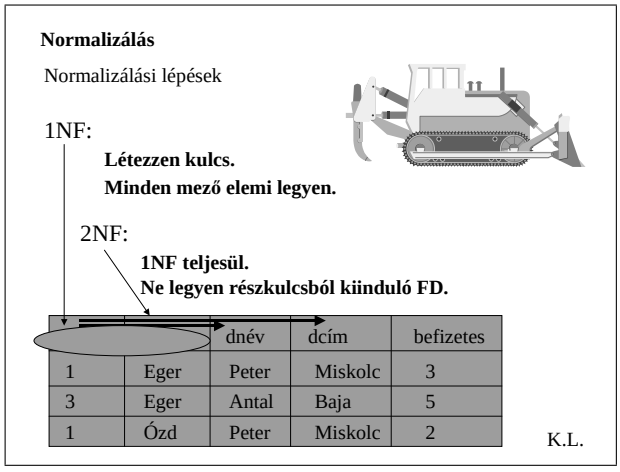
Ezzel azt fejezzük ki, hogy a kulcs központi szerepet játszik a relációban, minden mezőnek a teljes kulcstól, és nem annak egy részétől kell függnie. Ha rátekintünk a V1 sémára, látható, hogy megsérti a második normálformát, mivel az *osztvez*, az *osztvcím*, valamint az *osztcím* egy részkulcstól, az osztály mezőtől függ csak.

Formálisan: $\forall$ $K_1 \subseteq K$ esetén, ha $\exists$ $A \subseteq R$, hogy $K_1 \rightarrow A \Leftrightarrow$ ha $K_1 = K$, K jelölt kulcs az R-ben.

A második normálformát ekkor a reláció feldarabolásával lehet elérni. A példában célszerű egy V2 relációt létrehozni, és ebbe letárolni a csak az osztálytól függő adatokat. Az eredményül kapott relációk a következők:

V2 := {osztály, osztvez, osztvcím, osztcím} és a
BEOSZTOTT := {osztály, beosztott, beosztcím}.

Ekkor a VÉGZETTSÉG sémával együtt már három táblára bontottuk szét az eredeti sémánkat. Az új táblák:



6.10. ábra. Normalizálási lépések: 1NF, 2NF

V2			
osztály	*osztcím*	*osztvez*	*osztvcím*
Tervezés	Budapest	A. B.	Pipacs u.
Értékesítés	Miskolc	G. H.	Kankalin u.

BEOSZTOTT		
osztály	*beosztott*	*beosztcím*
Tervezés	C. D.	Rózsa u.
Tervezés	E. F.	Tulipán u.
Értékesítés	I. J.	Rózsa u.
Értékesítés	K. L.	Hóvirág u.

Immár három reláció van és mindegyiknél teljesül a második normálforma. Ha nem végezzük el ezt a felbontást, akkor újra szembe találjuk magunkat a módosítási, beszúrási anomáliákkal. Ha csupán csak egy mezőből áll a kulcs, azaz nincs összetett kulcs, akkor az adott séma automatikusan teljesíti a második normálformát.

A reláció felbontásakor úgy hajtottuk végre a dekompozíciót, hogy az eredeti séma visszaállítható belőle. A célunk most is a veszteségmentes felbontás elérése, azaz az információ vesztés elkerülése. Az új relációk join-jával ismét előállítható a kiindulási séma. Heath tétele arról szól, hogy egy felbontást mikor tekinthetünk biztosan veszteségmentesnek.

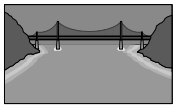
Heath tétele: ha R(A, B, C) adott és teljesül, hogy $A \rightarrow B$*, akkor a* Π_{AB} *és* Π_{AC} *veszteségmentes felbontás.*

Normalizálás

Veszteségmentesség: a résztáblákból előállítható az alaptábla.

Heath tétele:

R(A,B,C) adott és $A \rightarrow B$ akkor
(Π_{AB} , Π_{AC}) veszteségmentes.



Elegendőség és nem szükségesség kifejezése.

UTAK(DKOD, UTICEL, DNEV, DCIM, BEFIZETES)

DKOD -> (DNEV, DCIM)
($\Pi_{DKOD, DNEV, DCIM}$, $\Pi_{DKOD,UTICEL, BEFIZETES}$)

K.L.

6.11. ábra. Heath tétele

A tétel nem mondja ki, hogy csak mikor lehet veszteségmentes a felbontás, szükséges kritériumot nem ad, csak elegendőséget fogalmaz meg.

Alkalmazzuk Heath tételét egy példán. Legyen $A :=$ osztály, $B :=$ osztvez és $C :=$ beosztott, ahol *osztály* $\rightarrow$ *osztvez* és a *beosztott* mező független az előző kettőtől; és legyenek ezek egy DOLGOZÓ tábla mezői.

DOLGOZÓ		
osztály	*osztvez*	*beosztott*
Tervezés	A. B.	C. D.
Tervezés	A. B.	E. F.
Értékesítés	G. H.	I. J.
Értékesítés	G. H.	K. L.

Ezt a sémát szét tudjuk bontani a

D1 := {*osztály*, *osztvez*} és a
D2 := {*osztály, beosztott*} sémákra

D1	
osztály	*osztvez*
Tervezés	A. B.
Értékesítés	G. H.

D2	
osztály	*beosztott*
Tervezés	C. D.
Tervezés	E. F.
Értékesítés	I. J.
Értékesítés	K. L.

A két tábla egyesítéséből előállítható az eredeti DOLGOZÓ reláció. A D1 tábla esetén azonnal szembetűnőek a dekompozíció előnyei. Kevesebb helyet foglal és csökkenti az anomáliák előfordulásának lehetőségét is. A D2 reláció a beosztott mező függetlensége miatt nem tartalmaz kevesebb rekordot, azonban azok kisebbek lettek.

> *Harmadik normálformában van a reláció, ha teljesíti a második normálformát és ezenkívül igaz, hogy nem áll fenn tranzitív függőség.*

Ekkor ugyanis a kulcs a köztes mezőn keresztül, tranzitíven határozza meg a másik mező értékét. Ezért a köztes mező egyfajta kulcs, meghatározó szerepet játszik a másik mezőnél. Erre látunk példát a V2 relációban az *osztvez* mező esetében, melyből függőség indul az *osztvcím* felé. A harmadik normálforma pontos megfogalmazása: a nem kulcsmezők közvetlenül, ne tranzitíven függjenek a teljes kulcstól. Ez két dolgot jelent: először is azt, hogy két nem kulcsmező között nem

lehet függőség, másodszor pedig, hogy részkulcsból nem indulhat ki függőség. Ez utóbbit könnyen beláthatjuk, ha figyelembe vesszük, hogy a részkulcs mindig funkcionálisan függ a teljes kulcstól. Tehát a nem kulcs mező a részkulcson keresztül függene tranzitíven a teljes kulcstól. Ez nem megengedett. Így látható, hogy a harmadik normálforma magába foglalja a második normálformát.

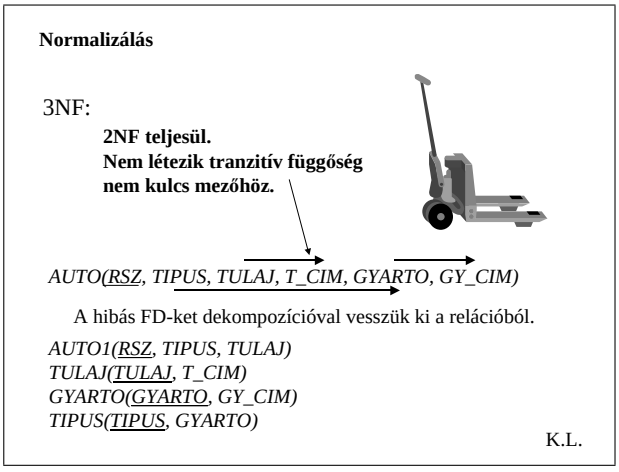
Formálisan: $\forall$ A $\rightarrow$ B $\Leftrightarrow$ ha vagy A = K, vagy B $\subseteq$ K, ahol K $\subseteq$ R jelölt kulcs.

A tranzitív függőség feloldása szintén a reláció feldarabolásával történik. Ehhez kiemeljük a tranzitív függést egy külön relációba. Példánkban az *osztvez* mezőt kell kiemelni a hozzá tartozó adatokkal együtt. Az eredeti táblában csak a kapcsolatot biztosító *osztvez* mező marad meg. Az eredményül kapott relációk:

OSZTÁLY		
osztály	*osztcím*	*osztvez*
Tervezés	Budapest	A. B.
Értékesítés	Miskolc	G. H.

VEZETŐ	
osztvez	*osztvcím*
A. B.	Pipacs u.
G. H.	Kankalin u.

Látható, hogy az eredeti VÁLLALAT séma négy sémára esett szét: OSZTÁLY, VEZETŐ, BEOSZTOTT és VÉGZETTSÉG. A vezető és az osztály adatainak szétválasztása feleslegesnek tűnhet, azonban egy ember több osztálynak is lehet a vezetője. Minden tábla a nevének megfelelő, összetartozó adatokat tárol. Ez alól egyetlen kivétel a végzettségeket tároló reláció. Ez a mező többértékű, így külön



6.12. ábra. Normalizálási lépések: 3NF

kellett választani a beosztott többi adatától, hogy ne kelljen azokat mindig megismételni, ha egy dolgozónak több végzettsége is van. Vegyük észre azt is, hogy a harmadik normálformában lévő eredménytábláink kielégítik az első két normálformát is.

A normalizálási folyamat áttekintéseként vizsgáljuk meg az alábbi példát. Van egy autószerelő műhely, ahova valamilyen hibával érkeznek az autók. Az autókat a rendszámuk segítségével azonosítjuk és a típusukat, valamint a gyártójukat tartjuk nyilván. Egy hiba nyílván több autónál is előfordulhat, illetve különböző időpontokban egy autónál is többször felmerülhet. Egy adott hibát egy bizonyos autónál egy időpontban csak egyetlenegy szerelő javíthat. Természetesen máskor más is javíthatja ugyanazt az autót ugyanazzal a hibával. A szerelőről nyilvántartjuk a címét is. A SZERVIZ séma mezői: {*rendsz, típus, gyártó, hiba, dátum, szerelő, szercím*}. A fennálló függőségi viszonyok:

rendsz → *típus*
típus → *gyártó*
szerelő → *szercím*
(rendsz, hiba, dátum) → *szerelő*

Ez egyben az irreducibilis FD halmaz.

A séma megfelel az első normálformának, mert minden mező atomi értékeket hordoz, és létezik kulcs, melytől minden mező függ. Könnyen belátható, hogy az elsődleges kulcs a {*rendsz, hiba, dátum*} attribútum halmaz.

A séma sérti a második normálformát, mert a *rendsz* részkulcsból FD indul ki a típus, illetve a gyártó felé. Bontsuk szét a sémát:

SZ1 := {*rendsz, típus, gyártó*} és
SZ2 := {*rendsz, hiba, dátum, szerelő, szercím*}.

Most már mind a két részséma második normálformában van, hiszen az SZ1-ben a *rendsz* mező egyszerű kulcs, az SZ2-ben pedig a szerelő és a címe a teljes kulcstól függ.

A harmadik normálformát sérti mind a két részséma. Az SZ1-ben a típustól függ a gyártó tranzitíven, míg az SZ2-ben a szerelő címét határozza meg funkcionálisan a szerelő. Mindkét sémát szét kell tehát bontanunk. A végeredményül kapott relációk, amelyek megfelelnek a harmadik normálformának:

TÍPUS := {*típus, gyártó*}
AUTÓ := {*rendsz, típus*}
JAVÍTÁS := {*rendsz, hiba, dátum, szerelő*}
SZERELŐ := {*szerelő, szercím*}

A felbontás során követhetünk el olyan hibát, hogy a dekompozíció megfelel Heath tételének, veszteségmentes, mégsem a legjobb megoldás. Ha például az SZ1 sémánkat a

{*rendsz, típus*} és a
{*rendsz, gyártó*}

relációkra bontjuk fel, akkor ez is természetesen veszteségmentes felbontás. Viszont ekkor a {*rendsz*, *gyártó*} táblát nem lehet tetszőlegesen feltölteni, mert a rendszám meghatározza a típust, az pedig a gyártót.

A bemutatott felbontásban a két reláció nem független egymástól. A nem független felbontások eredményeként kapott relációk kezelése sem lehet független, ami ellentmond a relációs modell alapelvének. Legyen Heath tételében az $A :=$ rendsz, $B :=$ típus és $C :=$ gyártó. Az $A \rightarrow B$ természetesen létezik és az eredeti reláció is visszaállítható a két új tábla join-jával, mégis érezzük, hogy ez nem olyan jó, mint az eredeti. Ellenben a TÍPUS és az AUTÓ felbontás egy *független felbontás*, ekkor a keletkezett relációk függetlenek egymástól. Ilyen független felbontások megvalósítása a cél. A felbontás függetlenségének eldöntésében segít Rissanen tétele.

Rissanen tétele: ha R(A, B, C) akkor Π_{AB}, Π_{AC} *független, ha*

- *R(A, B, C) minden FD-je származtatható R(A, B) és R(A, C) FD-ből, és*
- *A legalább az egyik felbontásban jelölt kulcs.*

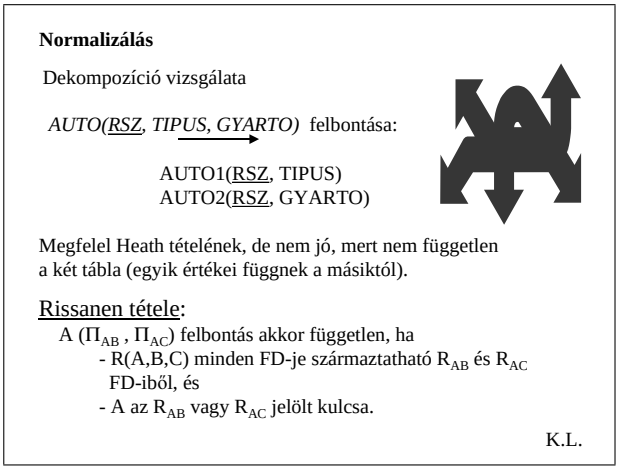
Vannak bizonyos sémák, melyeknek nincs független felbontása, ezeket nevezik *atomi sémák*nak.

Vizsgáljuk meg függetlenség szempontjából az AUTÓ, TÍPUS felbontást. A következő függőségeket találjuk:

rendsz $\rightarrow$ *típus* és
típus $\rightarrow$ *gyártó*.

A kiindulási sémában még egy függőség szerepel, a

rendsz $\rightarrow$ *gyártó*



6.13. ábra. Rissanen tétele

ez azonban azonnal származtatható az előző kettőből. Ha a típus a tételben szereplő A mező, akkor a második feltétel is teljesül, hiszen a TÍPUSban elsődleges kulcs. A másik felbontásnál az A mező szerepét a *rendsz* játsza. Az A mindkét felbontásban elsődleges kulcs, azonban a jelenlévő

rendsz → *típus* és
rendsz → *gyártó*

függőségekből nem származtatható a *típus* → *gyártó* függőség. Tehát ez utóbbi felbontás nem független.

Ez volt tehát az első három normálforma. A további normálformákat szintén konkrét példán keresztül vizsgáljuk. Tekintsük a következő vizsganyilvántartó sémát. Minden diák több tantárgyból vizsgázik. A diákok neveit használjuk azonosításra, emellett eltároljuk a címüket (dcím) is. A tantárgyak nevei egyediek, egy tárgyat több tanár is tarthat, azonban egy tanár csak egy tantárgyból adhat elő. Minden diák maga választhatja meg, hogy kinél akarja a tárgyat hallgatni. A tanár neve legyen egyedi és tároljuk a tanár címét (tcím) is. Tároljuk el még ezeken kívül a tantárgyhoz ajánlott irodalmakat (irodalom) valamint a tárgy felvételéhez kötelező korábbi tárgyakat (előzmény). A kialakult VIZSGA séma a következő:

VIZSGA						
tcím	*tanár*	*tantárgy*	*diák*	*dcím*	*irodalom*	*előzmény*
Kék u.	A.	Infó 3.	Jani	Kakukk u.	Unix	Infó 1.
Kék u.	A.	Infó 3.	Jani	Kakukk u.	Unix	Infó 2.
Kék u.	A.	Infó 3.	Jani	Kakukk u.	Win.	Infó 1.
Kék u.	A.	Infó 3.	Jani	Kakukk u.	Win.	Infó 2.
Piros u.	B.	Infó 3.	Ödön	Rigó u.	Unix	Infó 1.

A tábla így folytatódik tovább is. Látható, hogy annak letárolása, hogy Jani informatikát tanul Aladár tanár úrnál, négy sort vett igénybe. Lássuk ezek után a sémában előforduló függőségeket, hogy segítségükkel normalizálhassunk:

(tantárgy, diák) → *tanár*
diák → *dcím*
tanár → *tcím*
tanár → *tantárgy* (!)

ezen kívül a tantárgy meghatározza az irodalom és az előzmény mezőket is, azonban ezek nem funkcionális függőségek.

Az első normálforma eléréséhez az irodalom és az előzmény mezőket el kell távolítani. Mivel a tantárgyhoz kapcsolódnak, így a következő sémákhoz jutunk:

TANTÁRGY := {*tantárgy, irodalom, előzmény*},
V1 := {*tcím, tanár, tantárgy, diák, dcím*}.

A függőségekből belátható, hogy a {*tantárgy, diák*} páros lesz a kulcs.

A második normálforma szerint részkulcsból nem indulhat ki függőség. Márpedig a *dcím* a diák mezőtől függ. Így ezt a két mezőt ki kell emelnünk. A V1-ből két új séma lesz:

V2 := {*tcím, tanár, tantárgy, diák*} és
DIÁK := {*diák, dcím*}.

A harmadik normálforma szerint a sémákban nem lehet tranzitív függőség. Ennek minden séma eleget tesz, kivéve a V2, mert ebben a *tcím* a tanár mezőn keresztül függ a kulcstól. Ha jól megnézzük, a tanár mezőből indul függőség a tantárgy felé is, ez azonban egy jelölt kulcs része, azaz nem sérti a harmadik normálformát. A sémát ismét szétszedjük:

TANÁR := {*tanár, tcím*} és
V3 := {*tanár, tantárgy, diák*}.

Most már négy sémánk van: TANTÁRGY, DIÁK, TANÁR és V3. A DIÁK és a TANÁR 3NF sémák, ezekkel minden rendben. A TANTÁRGY egy kicsit furcsa, erről még lesz szó. Vizsgáljuk most meg a V3 sémát.

A V3 sémában az alábbi függőségek találhatók:

tanár → *tantárgy* és
(tantárgy, diák) → *tanár.*

Az az érdekes helyzet állt elő, hogy egy nem kulcs mező funkcionálisan meghatározza a kulcs egy részét. (Ha az egész kulcsot határozná meg, ő maga is jelölt kulcs lenne.) Ennek ellenére ez a séma eleget tesz a harmadik normálformának.

Ennél a sémánál törlési anomália léphet fel, hiszen ha kitöröljük egy tanár összes diákját, akkor elvész az az információ is, hogy milyen tárgyat tanít a tanár. Tehát a 3NF normálforma a megadott formájában nem biztosít tökéletes felbontást, ezért egy javított alakra van szükség ezen a szinten. A harmadik normálforma *helyett* bevezetett normálformát, amely nem engedi meg, hogy nem jelölt kulcs mezőből függőség induljon ki, Boyce-Codd normálformának nevezik.

> *Boyce-Codd normálformában van a reláció, ha minden függőség csak jelölt kulcsból indul ki.*

Ez tulajdonképpen a harmadik normálforma általánosítása azzal, hogy egy nem kulcsmezőből, valamely kulcsmezőbe sem indul függőség. A harmadik normálformát célszerű ezzel helyettesíteni.

Formálisan: az R séma BCNF-ben van, ha ∀ A, B ⊆ R-re A → B ⇔ ha A = K, ahol K az R szuperkulcsa.

A BCNF egyébként más tekintetben is kijavítja a hagyományos 3NF definícióját. Vegyünk ugyanis egy *DOLGOZÓ(kód, tajszám, név, beosztás)* relációt. Ebben a sémában a kód lesz a kulcs, mint belső azonosítója a dolgozó egyedeknek. A 3NF értelmezése szerint függőség nem indulhat ki nem kulcs mezőből egy nem kulcs mezőbe. A példánkban viszont láthatóan teljesül egy *tajszám* → *név* függőség, mivel a *tajszám* is egyedi a dolgozókra nézve. Így a fenti séma nem teljesíti a 3NF kritériumot. Ezért dekompozícióval fel kellene bontani kisebb relációkra. Viszont a hagyományos felbontási szabályt követve nem minden mezőt kellene átvinni, hiszen minden más mező függ a nem kulcs *tajszám*-tól is. Ezért nincs

értelme a felbontásnak annak ellenére, hogy a 3NF szabály ezt megkövetelné. Ha viszont belegondolunk, hogy a normalizálás igazi célja az ismétlődő értékű mezőkből adódó függőségek megszüntetése, akkor a *tajszám* nem igényelne felbontást, mert a belőle kiinduló függőségek nem okoznak redundanciát, hiszen a *tajszám* értéke nem ismétlődik. Tehát a séma a 3NF-nek nem felel meg, de az általános elveknek megfelel. Ez a példa is mutatja, hogy szükség van a 3NF egy javított értelmezésére. A BCNF formula esetén a sémánk már megengedett lesz, hiszen a *tajszám* mező egy jelölt kulcs, ezért a belőle induló függőségek nem károsak.

Most már tudjuk, hogy a V3 séma sérti a BCNF-et, hiszen a *tanár* nem szuperkulcs. Megpróbálhatjuk szétbontani. Természetesen az egyik új séma a

$$TANÍT := \{\underline{tanár},\ tantárgy\}$$

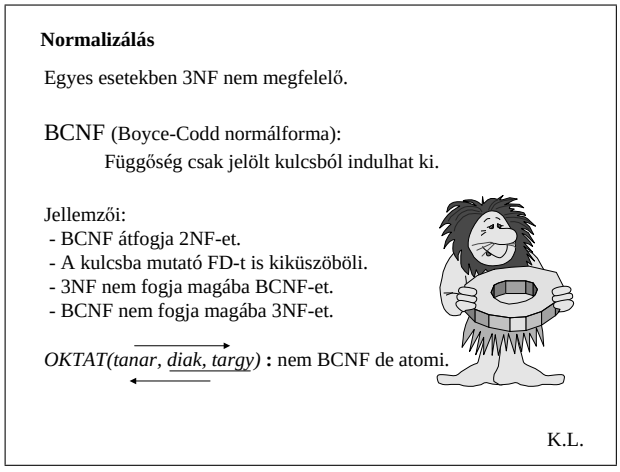
De mi legyen a másik? Az autós példában az SZ1 séma felbontása kapcsán láttuk, hogy nem mindegy mely mezők kerülnek a részsémákba. A cél a veszteségmentesség mellett a független felbontás. Ha a másik séma, a

$$HALLGAT := \{\underline{tantárgy,\ diák}\}$$

akkor a felbontás nem veszteségmentes. Ezt Heath tételével könnyen ellenőrizhetjük, de azonnal látszik, hiszen nem tudjuk, hogy kinél hallgatja a tárgyat a diák. Ha

$$HALLGAT := \{\underline{tanár,\ diák}\}$$

akkor ez már veszteségmentes. Azonban a felbontás nem független! Nem tudjuk az eredeti függőségi viszonyokat visszaállítani. Lehet próbálkozni! Azonnal látható a függőség, ha belegondolunk, hogy a HALLGAT relációban egy diák mellett nem



6.14. ábra. Normalizációs lépések: BCNF

szerepelhet akármelyik tanár. Olyan tanár nem lehet mellette, aki olyan tantárgyat tanít, amit ő már egy másik tanárnál hallgat. A V3 séma tehát atomi, de nem felel meg a Boyce-Codd normálformának.

A BCNF és a 3NF nem ágyazódnak egymásba, hiszen láttunk olyan sémát, mely 3NF de nem BCNF és olyat is, amely BCNF de nem 3NF.

Haladjunk tovább a példánkkal, vizsgáljuk meg a TANTÁRGY sémát:

TANTÁRGY		
tantárgy	*irodalom*	*előzmény*
Infó 3.	Unix	Infó 1.
Infó 3.	Unix	Infó 2.
Infó 3.	Win.	Infó 1.
Infó 3.	Win.	Infó 2.

Ez a reláció redundáns, annak minden hátrányával. Azt az információt, hogy milyen könyvek és milyen előző tárgyak szükségesek, már két sorból is látjuk. Ha csak két sorban akarnánk ezt az információt tárolni, akkor melyik két sor legyen? Az első és a negyedik, vagy a második és a harmadik? És ha az egyik könyvet törölni szeretnénk, akkor elvész vele együtt az egyik kötelező előzmény is! Érezzük, hogy itt ismét a nem összetartozó adatok egy táblába helyezése okozza a problémát.

Ebben a relációban nincs funkcionális függőség, ennek ellenére azonnal látható a veszteségmentes felbontás

{*tantárgy, irodalom*} és
{*tantárgy, előzmény*}

részekre. Ekkor az eredeti séma a két rész join-jával megkapható. Az ilyen jellegű függőségeket *többértékű függőség*nek hívják (Multivalued Dependency, MVD) és kettős nyíllal jelölik : $\rightarrow\!\!\rightarrow$.

> *Az R(A, B, C) sémán az A* $\rightarrow\!\!\rightarrow$ *C többértékű függőség teljesül, ha* $\forall$ *(a_i, b_i)-hez tartozó {c_i} halmaz csak a_i-től függ b_i-től nem. Minden a_i-hez egy {b_i} és egy {c_i} halmaz rendelhető. Az R{A, B, C}-ben A* $\rightarrow\!\!\rightarrow$ *C MVD igaz* $\Leftrightarrow$ *ha A* $\rightarrow\!\!\rightarrow$ *B is igaz.*

Így a továbbiakban az A $\rightarrow\!\!\rightarrow$ B|C jelölést használjuk az MVD megadására. A definíció megengedi, hogy a {b_i} és a {c_i} halmaz egyértékű legyen, ekkor A $\rightarrow$ B és A $\rightarrow$ C, azaz az MVD szélső esete az FD.

A többértékű függőség formális definíciója: r(R) és X, Y $\subseteq$ R és Z = R - (X $\cup$ Y), X $\rightarrow\!\!\rightarrow$ Y $\Leftrightarrow$ $\forall$x, X tulajdonság értékre, $\pi_{YZ}(\sigma_{X\,=\,x}(\mathrm{r})) = \pi_Y(\sigma_{X\,=\,x}(\mathrm{r}))$? $\pi_Z(\sigma_{X\,=\,x}(\mathrm{r}))$.

Az MVD-ket ki kell küszöbölni, mert, mint láttuk, redundanciát okoznak. Az a séma, amelyben nincs nem triviális MVD, negyedik normálformában van.

> *Negyedik normálformában van a reláció, ha minden nem triviális MVD egyben FD is, azaz ha A* $\rightarrow\!\!\rightarrow$ *B/C, akkor A* $\rightarrow$ *B és A* $\rightarrow$ *C is teljesül, ahol A szuperkulcs.*

Ekkor a sémában nincs többértékű függőség.

Formálisan: R 4NF $\Leftrightarrow$ ha $\forall$ A $\rightarrow\!\!\rightarrow$ B nem triviális MVD esetén A szuperkulcs R-ben, azaz A $\rightarrow$ B, ahol A szuperkulcs.

Kérdéses a 4NF és a BCNF viszonya. Ha egy séma 4NF, akkor automatikusan BCNF is? A kérdést megfordítva: létezik-e olyan nem BCNF séma, ami a 4NF-nek megfelel? Tekintsük a következő, nem BCNF sémát:

$R\{X,\ Y,\ Z\}$, ahol $X \rightarrow Y$; $Y \rightarrow Z$.

Ekkor azonban fennáll a sémában az $Y \rightarrow\!\!\rightarrow X|Z$ MVD, ami nem kulcsból indul ki és az $Y \rightarrow X$ sem teljesül. Láthatjuk, hogy nincs olyan séma, ami nem BCNF, de 4NF. Azaz 4NF már csak az lehet, ami BCNF is, tehát a *4NF sémák teljesítik a BCNF-et* is.

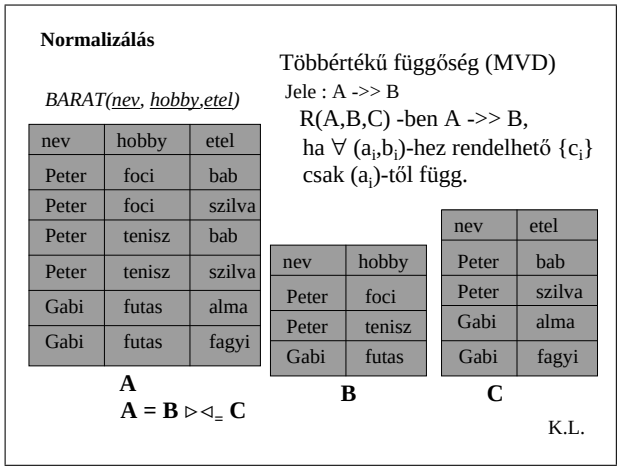
Most már csak azt kell eldönteni, hogy hogyan bontsuk fel a TANTÁRGY sémát veszteségmentesen. Mint már említettük, a felbontás természetesen adódik és Fagin tétele kimondja, hogy ez a felbontás veszteségmentes.

> *Fagin tétele: a veszteségmentes felbontás szükséges kritériuma. Ha adott R(A, B, C) séma, amelyben A $\rightarrow\!\!\rightarrow$ B|C, akkor* Π_{AB} *és* Π_{AC} *veszteségmentes felbontás.*

Tehát a TANTÁRGY táblát felbonthatjuk az

IRODALOM := {*tantárgy, irodalom*} és az
ELŐZMÉNY := {*tantárgy, előzmény*}

relációkra. Ezek már redundancia nélkül tárolják a szükséges információkat és join-jukból visszakaphatjuk az eredeti táblát. A VIZSGA séma tehát öt részre



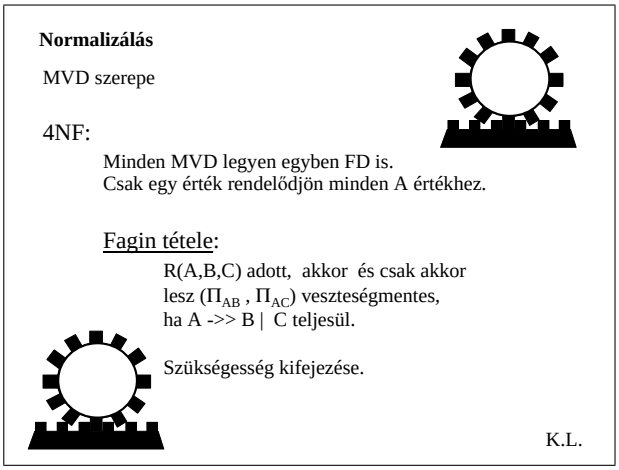
6.15. ábra. Többértékű függőség

esett szét: TANÁR, DIÁK, ELŐZMÉNY, IRODALOM és V3.

Az eddigi példáink olyanok voltak, hogy a sémát egy lépésben két részre bontottuk fel. Léteznek azonban olyan sémák is, melyeket nem lehet veszteségmentesen két részre bontani, csak több részre. Ennek illusztrálásra tekintsük az alábbi példát. Tartsuk nyílván, hogy a diákok mely órákra járnak és melyik tanárokhoz. A diákok tantárgyat és tanárt választanak. A diákok több tantárgyat és órát is választhatnak. A tanárok több tantárgyat is taníthatnak. Tegyük fel, hogy a diákok minden olyan órára járnak, amelyet kiválasztottak, bármelyik általuk választott tanár tartja. Ha több, általuk választott tanár is tart olyan órát, akkor mindre bejárnak. Például ha *A* tart Infó 3-at és Jani választotta ezt a tárgyat, valamint Jani választotta *A*-t (lehet, hogy más tárgy miatt), akkor Jani bejár *A* Infó 3 óráira. Persze ez a valóságban nem így van. Lássuk a táblát:

ÓRALÁTOGATÁS		
tantárgy	*tanár*	*diák*
Infó 3.	A.	Jani
Progi 3.	A.	Jani
Progi 3.	C.	Jani
Infó 3.	A.	Ubul
Infó 3.	B.	Ubul
Adatb 3.	B.	Ubul

Ilyen függőséggel még nem találkoztunk. Nem FD, mert nincs egyik értékhez sem egyértelmű hozzárendelés, viszont nem is MVD, mert egyik mező sem független valamelyik másiktól. Ha az lenne, akkor szerepelnie kellene az (Infó 3.; C.; Jani) és az (Adatb 3.; A.; Ubul) rekordoknak. Azonban a szóban forgó tanárok ilyen órákat nem tanítanak. Viszont a relációban anomáliák vannak, ezért át kell



6.16. ábra. Normalizációs lépések: 4NF, Fagin tétele

alakítani. Ilyen anomália például az, hogy ha beszúrjuk a (Progi 3.; C.; Ubul) rekordot, akkor be kell szúrnunk a (Progi 3.; A.; Ubul) rekordot is, mert Ubul tanul *A*-nál és *A* tanít Progi 3-at. Fordítva viszont nem igaz. Ha beszúrjuk a (Progi 3.; A.; Ubul) rekordot, akkor a (Progi 3.; C.; Ubul) rekordot nem szúrhatjuk be! Másik anomália az az eset, amikor kitöröljük a (Progi 3.; A.; Jani) rekordot. Ekkor ki kell törölnünk még vagy a (Progi 3.; C.; Jani) rekordot, mert már nem tanul Progi 3-at, vagy az (Infó 3.; A.; Jani) rekordot, mert már nem tanul *A*-nál. De melyiket töröljük?

A függőség megértéséhez nézzük meg külön-külön, hogy melyik diák mit tanul, kinél tanul, és melyik tanár mit tanít.

TANÍT	
tantárgy	*tanár*
Infó 3.	A.
Progi 3.	A.
Progi 3.	C.
Adatb 3.	B.
Infó 3.	B.

FELVESZ	
tantárgy	*diák*
Infó 3.	Jani
Progi 3.	Jani
Infó 3.	Ubul
Adatb 3.	Ubul

HALLGATÓ	
tanár	*diák*
A.	Jani
C.	Jani
A.	Ubul
B.	Ubul

Ha most az utóbbi két tábla join-ját vesszük, akkor megkapjuk az MVD-nek megfelelő esetet:

ÓRA		
tantárgy	*tanár*	*diák*
Infó 3.	A.	Jani
Progi 3.	A.	Jani
Progi 3.	C.	Jani
* Infó 3.	C.	Jani *
* Adatb 3.	A.	Ubul *
Infó 3.	A.	Ubul
Infó 3.	B.	Ubul
Adatb 3.	B.	Ubul

Ez a tábla már majdnem a kiindulási reláció, mindössze a *-al megjelölt sorokat kell eltüntetni belőle. Ezt megtehetjük ha vesszük az ÓRA és a TANÍT join-ját. Ekkor visszakapjuk az eredeti táblát. Tehát ez egy olyan függőség, amely a részek join-jával egyezik meg. Az ilyen függőségeket nevezik *join függőség*nek. Jele: *JD(A, B, ..., Z)*, illetve **(A, B, ... Z)*. Közben megtaláltuk annak a módját is, hogy hogyan lehet veszteségmentesen felbontani az ilyen függőségeket.

> *Join függőség (Join Dependency, JD): az R séma JD(F_1, F_2, ... F_n) ha a séma megegyezik az F_1, ..., F_n projekciók joinjával.*

Azaz a séma veszteségmentesen felbontható *n* részre. (Ez csak azt jelenti, hogy felbontható, de nem kell mindenképpen felbontani.) Tehát a példánkban:

JD({tantárgy, tanár}, {tanár, diák}, {tantárgy, diák}).

Fagin tétele alapján R(A, B, C) kielégíti a

JD(AB, AC) JD-t $\Leftrightarrow$ *ha* $A \rightarrow\!\!\rightarrow B|C$.

Ebből látható, hogy a *JD speciális esete az MVD.*

> *A funkcionális függőség általánosítása a többértékű függőség és a többértékű függőség általánosítása a join függőség. Hasonlóképpen az MVD a JD speciális esete és az FD az MVD speciális esete.*

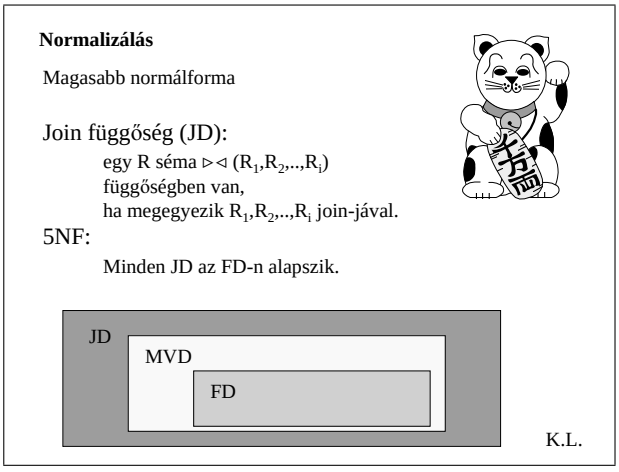
Ahogy az MVD is redundanciát okozott, úgy a JD megléte is sok felesleges adatelemet jelent a táblában, ezért célszerű ezen függőségi típus eliminálása is. Az ötödik normálforma a JD korlátozását mondja ki.

> *Ötödik normálformában van a reláció, ha minden JD FD-ken alapul. Más szavakkal minden JD a jelölt kulcsra épül, a jelölt kulcs implikálja a felbontást.*

Ez azt jelenti, hogy egy R{$\underline{\text{A}}$, B, C} sémában, ahol A $\rightarrow$ B és A $\rightarrow$ C, teljesül a JD(AB, AC), tehát a felbontást, illetve a JD-t, az $\underline{\text{A}}$ jelölt kulcs implikálta. Az ÓRALÁTOGATÁS nem 5NF, mert felbontható három részre de ez a felbonthatóság nem abból következik, hogy a tantárgy, tanár, diák mezők együtt alkotják a kulcsot. Viszont a felbontás után már nincs JD a sémában, tehát az már 5NF.

Az ötödik normálforma után már csak tiszta funkcionális függőségek lehetnek a sémában. Az ötödik a legmagasabb normálforma.

Az ötödik normálforma teljesülése estén teljesül a negyedik is. A negyedik normálforma pedig magába foglalja a BCNF-et is. Tehát egy ötödik normálformájú séma elérésekor a többi normálformát is teljesítettük. Léteznek még más normálformák is, azonban ezekre itt nem térünk ki.



6.17. ábra. Normalizációs lépések: 5NF, Join függőség

A normalizálás eredményeképpen tehát olyan függőségi rendszert kaptunk, amely tiszta és egyértelmű. Minden táblánál kell léteznie egy függőségi centrumnak, a kulcsnak, és minden más mezőhöz léteznie kell függőségi kapcsolatnak a kulcsból. Ezen függőségeken kívül a relációséma nem tartalmazhat más függőségeket.

6.4. Kiegészítő megjegyzések

A tervezés során a normalizálás mellett sokszor lehet olvasni a *de-normalizálás* folyamatáról is. Ez nem jelent mást, mint a normalizált táblák visszafelé alakítását, azaz a szétszedett adatok újra közös relációba kerülnek. Vajon mi lehet ennek az oka? Miért rontják el a kialakult normalizált sémát? A választ a rendszer hatékonysági problémájánál kell keresni. Ugyanis a normalizált sémák nagyon jók az adatok kezelésében, de nem a leghatékonyabb megoldást jelentik az adatok lekérdezésében, hiszen ekkor nagyon sok join műveletre lehet szükség a kívánt adatok egybeolvasztására az eredménytáblázaton belül, ami igen időigényes tehát magas költségű művelet.

> *A de-normalizáció célja a leggyakrabban igényelt join műveletek eliminálása a szétszedett táblák újbóli egyesítésével.*

Természetesen a de-normalizálás lépéseit csak azután szabad megtenni, ha előbb előállítottuk a normalizált alakot, és meghatároztuk a rendszer teljesítő képességét, feltártuk a műveletvégrehajtás gyenge pontjait. Ezen információk birtokában lehet megnézni, mely átalakítások biztosítják az igényelt hatékonyságnövekedést.

Tehát a de-normalizálás nem azt jelenti, hogy nem kell normalizálni, hogy el lehetne felejteni a normalizálási lépéseket. Ellenkezőleg, a de-normalizáláshoz ismerni kell és el kell végezni a normalizálási lépéseket, majd meg kell tudni határozni a hatékonysági problémák okait, és megoldást kell találni a válaszidők lerövidítésére.

A normalizáció során az eddigiekben a dekompozíciót használtuk a helyes séma kialakítására, de emellett az irodalomban ismert az úgynevezett *szintézis* módszere is, ami a normálformáknak megfelelő sémák előállítását jelenti.

A dekompozíció felülről lefelé (top-down) módszer, amelyben egy kezdeti relációs sémából alakítjuk ki több lépésben a végleges sémákat. A szintézis ezzel szemben alulról felfelé (bottom-up) építkező módszer, amelyben a kezdeti függőséghalmazt alakítjuk és az utolsó lépésben hozzuk létre a relációs sémákat. A dekompozíció az egyedtípusokat tekinti elsődlegesnek, a Bernstein által kidolgozott szintézis viszont a tulajdonságtípusokat helyezi előtérbe.

A szintézis során a funkcionális függőségek halmazát ekvivalens módon átalakítjuk úgy, hogy a kulcsoktól való függőségek halmazát kapjuk, amelyből a sémák közvetlenül kialakíthatók. A szintézis legnagyobb hátránya, hogy csak a funkcionális függőségek kezelésére alkalmas.

> *A szintézis menete: vegyük az összes tulajdonságtípust és határozzuk meg a köztük lévő függéseket. Szüntessük meg a redundáns (más függő-*

ségekből leszármaztatható) függőségeket. Egyesítsük azokat a függőségeket, amelyeknek a baloldala azonos, vagy ekvivalens. Az egyedtípusok kulcsa, illetve tulajdonságai ezeknek az ekvivalencia-osztályoknak a baloldala, illetve a jobboldala lesz.

A módszer hátránya, hogy két tulajdonság között csak egyféle függési viszony határozható meg, azaz két tulajdonság nem fordulhat elő több egyedben, ha van olyan egyed, ahol az egyik meghatározza a másikat. Emellett a szintézissel nem lehet kapcsolóegyedeket sem meghatározni.

Az elkészült, úgymond normalizált modell már mentes azon alapvető tervezési hibáktól, amelyek az anomáliákat okozzák. Ezzel sikerülhet egy, a szemantikai tartalmat a lehetőségek szerint megőrző és a hatékonysági szempontokat is figyelembe vevő, relációs adatmodellt alkotni, melyet felhasználhatunk az adatbázis megvalósításához. A teljesség kedvéért megemlíthető, hogy bizonyos műveletek hatékonyabb végrehajtása érdekében egyes esetekben a tervezők inkább lemondanak a tisztaságról, áttekinthetőségről, és összevonnak egy relációba olyan adatokat is, amelyeknek a normalizálás elmélete szerint külön relációkban kellene helyet foglalniuk.

Elméleti kérdések

1. Mi a normalizálás célja, adja meg főbb lépéseit.
2. Adja meg az FD fogalmát, és mutassa be fontosságát a normalizálásban.
3. Hogyan értelmezett az FD halmazok ekvivalenciája?
4. Mit jelent és miért fontos a BCNF normálforma?
5. Mi okozza a redundanciát a relációs sémában?
6. Miért kerülendő a redundancia?
7. Miért lehet hasznos a redundancia?
8. Ismertesse az MVD fogalmát, és mutasson be példát a létezésére.
9. Mit jelent az atomi séma és a független felbontás fogalma?
10. Milyen szabályokból állnak az Armstrong-axiómák, és mire szolgálnak?
11. Ismertesse a veszteségmentes felbontás fogalmát és tételeit.
12. Ismertesse a független felbontás (dekompozíció) fogalmát és tételét.
13. Hogyan fogalmazható meg Fagin tétele, adjon példát a teljesülésére.
14. Adja meg Rissanen tételét, és mutasson be példát, amikor nem teljesül a tétel.
15. Sorolja fel az alap normálformákat.
16. Ismertesse a magasabb normálformákat.
17. Vezesse le az $A \rightarrow B$, $C \rightarrow D \Rightarrow AC \rightarrow BD$ szabályt.
18. Adja meg az FD, MVD fogalmakat példával is illusztrálva.
19. Adja meg a függőségek típusait és kapcsolatukat.
20. Mik a normalizálás lépései?
21. Adja meg a Heath és Fagin tételeket, mi a kapcsolat a két tétel között?
22. Ismertesse a mezők közötti függőségek típusait példával is bemutatva.
23. Milyen tételek vonatkoznak a reláció dekompozíciójának megvalósíthatóságára?
24. Az anomáliák típusai; a normalizálás célja, lépései.
25. MVD formális felírása, és bemutatása egy példán keresztül; magasabb normálformák.
26. Adjon meg nem független dekompozíciót, és ismertesse Rissanen tételét.
27. Ismertesse az FD halmazok ekvivalenciájának fogalmát és az ekvivalencia ellenőrzésének lépéseit, elvét.
28. Az MVD fogalma. Igazolja, hogy minden FD MVD is egyben.
29. Adja meg R(A,B,C) egy veszteséges dekompozícióját, ahol $A \rightarrow B$, $A \rightarrow C$, és $B \rightarrow C$.

Feladatok

1. Normalizálja az alábbi sémát 3NF-ig:
   T(A,B,C,D,E,F) ahol B → E , C → F, (B,C) → A, E → D.
2. Normalizálja az alábbi sémát 3NF-ig:
   VIZSGA(indexszám, dátum, hallgatónév, tankör, tárgynév, előadó, jegy).

*3. Normalizálja az alábbi sémát 3NF-ig:
   R(A,B,C,D,E,F) ahol A → C, C → E , (A,B) → F , B → D.

4. Normalizálja az alábbi sémát 3NF-ig:
   T(A,B,C,D,E,F) ahol (A,B) → C, E → F, C → D, F → B.

*5. Normalizálja az alábbi sémát 3NF-ig:
   R(X,Y,Z,Q,W) ahol Y → W, X → (Q,Z), Z → Y.

6. Normalizálja az alábbi sémát 3NF-ig:
   T(X,Y,Z,V,W) ahol X → (V,W), Y → Z, W → V.

*7. Normalizálja az alábbi sémát BCNF-ig:
   R(A,B,C,D,E) ahol C → E, A → D, E → B, (A,E) → A.

8. Normalizálja az alábbi sémát BCNF-ig:
   RENDELÉS(vevőkód, dátum, menny., vevőnév, szül.év, kor, árukód, árunév).

9. Normalizálja az alábbi sémát BCNF-ig:
   SZERVIZ(rsz, típus, tulajnév, tulajcím, dátum, ár, hiba) ha egy szervíznél több hiba is lehet.

10. Normalizálja az alábbi sémát 4NF-ig:
    PROJEKT(projektszám, dátum, résztvevő, képesítés, feladatkör) ahol egy projektnek több résztvevője van, és egy résztvevő több képesítéssel rendelkezhet.

11. Normalizálja az alábbi sémát BCNF-ig:
    KÖLCSÖNZŐ(kazetta_ID, kazetta_CIM, kazetta_MŰFAJ, tag_ID, tag_NÉV, dátum, ár) ahol azonos műfaj azonos árat jelent.

12. Normalizálja az alábbi sémát BCNF-ig:
    X(A,B,C,D,E) ahol E → D, A → E, B → (A,C).

*13. Normalizálja az alábbi sémát BCNF-ig:
    R(X,Y,Z,Q,R,S) ahol (Y,Q) → Y , Q → Z, Y → S, (Y,Q) → R, S → X.

*14. Normalizálja az alábbi sémát BCNF-ig:
    R(A,B,C,D,E,F) ahol A → C, E → B, C → (F,C), (A,E) → D.

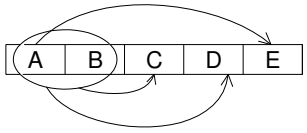
15. Normalizálja az alábbi sémát BCNF-ig:
    R(A,B,C,D,E,F) ahol A → D, E → B, D → FD, (A,E) → C.

*16. Normalizálja az alábbi sémát BCNF-ig:
    R(A,B,C,D,E) ahol A → B, A → C, B → A, B → C, C → D, D → E.

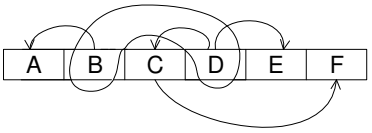
17. Mutassa be a normalizálást az FD-k kijelölése után az alábbi sémára:
    RÉSZVÉTEL(tanf_id, hallgatóid, H.neve, H.lakcím, jegy, tárgycím, tanár).

18. Normalizálja az alábbi sémát, írja fel a normálformák jelentését.
    X(A,B,C,D,E) és (A,B) → C, A → E, C → D, A → B.
    Vezesse le, hogy A → C.

19. Normalizálja az alábbi sémát:



20. Normalizálja az alábbi sémát:



(B,D) ⟶ E

(B,D) ⟶ C

7. fejezet

Gazdanyelvbe ágyazott SQL felületek

7.1. Általános áttekintés a gazdanyelvbe ágyazott SQL nyelvről

Az előzőekben megismert SQL nyelv egyik fő jellemzője, hogy hiányoznak belőle a procedurális elemek. Ennek következtében nem lehet pusztán SQL utasításokra építve komplett alkalmazásokat készíteni, hiszen az SQL nem tartalmaz elágazási, ciklus vezérlési vagy éppen terminál felület működését leíró nyelvi elemeket. Így az SQL nem tekinthető alkalmazásfejlesztő nyelvnek. Az SQL kizárólagos célja az adatbázissal történő adatforgalom biztosítása. Ebből az is következik, hogy az alkalmazások elkészítéséhez más jellegű procedurális nyelvre lesz szükségünk. A piacon számos ilyen fejlesztő nyelv áll rendelkezésre. Többek között megemlíthetjük a C, Pascal vagy Fortran nyelveket, mint a legelterjedtebb általános célú programfejlesztő rendszereket.

E nyelvekben igen rugalmas eszközkészlet áll rendelkezésre a különböző vezérlési és IO műveletek ellátására, viszont nem tartalmaznak semmilyen lehetőséget, hogy a programból az adatbázisban tárolt adatokat elérhessük. Az SQL éppen e funkciók elvégzésére szolgál, de ott meg a proceduráis elemeket kell nélkülöznünk. Egy adatbáziskezelést megvalósító alkalmazásnál viszont mindkét elemre szükségünk lenne. Így felmerül a kérdés, hogyan lehetne olyan fejlesztő környezetet létrehozni, amely mindkét elemet tartalmazza, lehetőleg úgy, hogy könnyen illeszthető legyen az eddigi rendszerekhez, és viszonylag kevés ráfordítással meg lehessen tanulni.

A felvetett problémára többféle megoldás is létezik, mint ahogy azt a piacon megjelenő termékek is mutatják. Az egyik megoldás, hogy egy teljesen új, független nyelvet hozunk létre. Ennek számos hátránya van éppen a szabványosság és a megtanulhatóság vonatkozásában, viszont előnyös lehet a hatékonyság szempontjából, hiszen levethetők a korábbi korlátok, megkötöttségek. Egy más jellegű megközelítést jelent, amikor vagy az SQL alapokat megőrzik vagy a megismert

programozási nyelv elemeit hagyják meg. A választás ebben az esetben már arra irányul, hogy melyik irányból közelítsünk a másik komponens felé. Összefoglalóan tehát az alábbi utak állnak rendelkezésre:

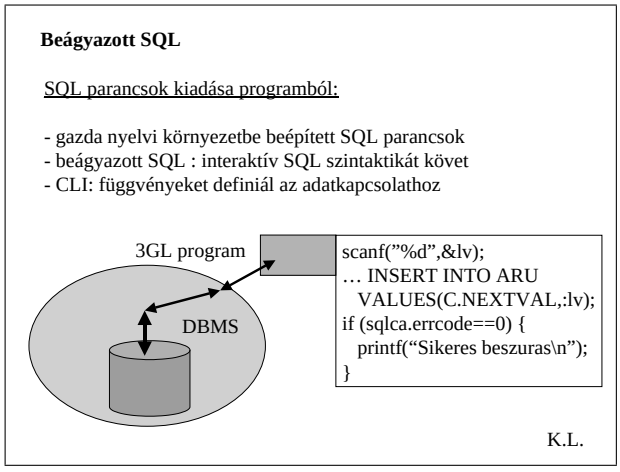
- új fejlesztő rendszer kidolgozása;
- az SQL nyelv kibővítése procedurális elemekkel;
- a programozási nyelvek kibővítése SQL elemekkel.

A piacon megtalálható mindhárom megközelítés megvalósítása szinte minden RDBMS rendszernél. Mi a továbbiakban az Oracle rendszerén keresztül mutatjuk be az egyes változatok megvalósításait, alkalmazását.

Elsőként, e fejezet keretében a programozási nyelvek kibővítésének lehetőségeit vesszük át. A fejezet tanulmányozásánál feltesszük, hogy az eddigiekben már találkoztunk a C procedurális nyelvvel, így a példákat erre a nyelvre vonatkozóan fogjuk tárgyalni. Mivel a fejezet fő célja nem a C nyelv speciális rutinjainak ismertetése, hanem az adatbázisoknak e nyelvekből történő elérésének bemutatása, így az itt megadott ismeretek viszonylag könnyen általánosíthatók más programozási nyelv esetére is.

A programozási nyelvek kiegészítésénél a magas szintű nyelvek procedurális elemei közé beilleszthetjük az adatbázis kezelésére szolgáló SQL utasításokat. A kiválasztott procedurális nyelvet *gazdanyelvnek* szokás nevezni. A legelterjedtebb gazdanyelvek közé tartozik többek között a C, Ada, Pascal, COBOL, FORTRAN is, így szinte bármilyen programozói előképzettséggel készíthetünk beépített SQL utasításokat tartalmazó adatbáziskezelő alkalmazásokat.

Az SQL alapú adatkezelő funkciók bevonására kétféle formalizmus terjedt el a gyakorlatban. Az első változat esetén a gazdanyelv utasításai közé *beszúrjuk* a már ismert SQL utasításokat, méghozzá olyan szintaktikával, mint ahogy azt az SQL



7.1. ábra. SQL parancsok kiadása programból

szabvány előírja. Így az alkalmazás forrásprogramjában a gazdanyelvi és az SQL kifejezések együttesen, egymást váltogatva fordulnak elő. E kétféle utasításelemek szemmel láthatóan, jól elkülönülnek egymástól, hiszen lényeges különbség van egy C-beli kifejezés, például

c = (x > 6) ? 3 : 2;

és egy SQL kifejezés, mint az

INSERT INTO auto VALUES ('fdg345', 'Opel', 4);

szintaktikája között. Az ezen az elven működő kibővítéseket nevezik az SQL *gazdanyelvi beágyazásának*, ami arra utal, hogy a gazdanyelv utasításai közé beültetjük, beillesztjük az SQL utasításokat.

A másik fajta megközelítés közelebb áll a hagyományos 3GL programozási nyelvekhez, mivel itt a gazdanyelv szintaktikájának megfelelő formalizmussal lehet az SQL utasításokat végrehajtatni. Ekkor a gazdanyelv utasításai közé *eljárások, függvények formájában* szúrjuk be az adatkezelő tevékenységeket. Az előző INSERT utasítás például a következő függvényhívással valósítható meg az Oracle OCI rendszerében:

osql3(&cursor,"INSERT INTO auto VALUES('fdg345','Opel',4);",-1);

Ebben a formalizmusban az SQL utasítások szervesen beilleszkednek a C nyelv utasításai, kifejezései közé, és csak a függvények elnevezései, a paraméterek értékei utalnak arra, hogy itt egy adatbázis kapcsolat kerül megvalósításra. Ez a fajta mechanizmus a *CLI* (Call Library Interface) elnevezést kapta, utalva arra, hogy itt a kapcsolat könyvtári függvények hívásán keresztül valósul meg.

A kétféle megközelítés ugyan formálisan igen eltér egymástól, de funkcionálisan igen közeliek, hiszen mindkettő ugyanazt a célt szolgálja. Az egyenértékűséget mutatja, hogy mindkét formalizmussal végrehajtható az SQL-hez tartozó utasítások teljes készlete, tehát mindkettőt lehet nyugodtan választani egy adott feladat megoldására. A kétféle megközelítésnél azonban a formai különbségek mellett bizonyos megközelítési, szemléletbeli eltérés is tapasztalható.

A beágyazott SQL előnye, hogy
- az SQL szabványokhoz közelebb álló formalizmus,
- az egyszerűbb adatkezelési környezet könnyebben megvalósítható;

míg a CLI a beágyazott SQL-el szemben
- részletesebb előkészítést igényel,
- bonyolultabb formalizmussal rendelkezik.

A CLI előnye viszont, hogy
- természetesebb módon tudja kezelni az összetettebb, dinamikus adatkezelési funkciókat is,
- a felhasználó könnyebben kézben tudja tartani a végrehajtást,
- közvetlenebb végrehajtást tesz lehetővé, és
- jobban kapcsolható más rendszerekhez.

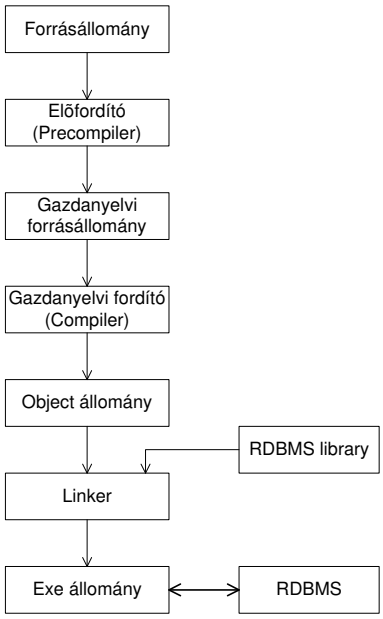
Általános irányelvként az mondható, hogy az egyszerűbb adatkezelési funkciókat megvalósító alkalmazások esetében célszerű a beágyazott SQL bevonása, míg az összetettebb, dinamikus rendszereknél a CLI lesz a megfelelő eszköz.

7.2. Beágyazott SQL utasítások használata

A beágyazott SQL utasítások fejlesztésének a menete ugyanúgy kezdődik, mint egy hagyományos gazdanyelvi program készítése. Első lépcsőfokként egy szövegszerkesztővel előállítjuk a beépített SQL utasításokat tartalmazó gazdanyelvi forrásállományt. Ebben a gazdanyelv és az SQL utasítások együttesen szerepelhetnek.

Ezt követően egy előfordító segítségével a beépített SQL utasítások átkonvertálódnak a gazdanyelvi szintaktikának megfelelő eljáráshívásokká, ugyanis a forrásállományban a megszokott SQL szintaktikát használhatjuk, ami lényegesen eltér a programozási nyelvekben megszokott formáktól. E konverzió eredménye egy szabályos gazdanyelvi forrásszöveg lesz.

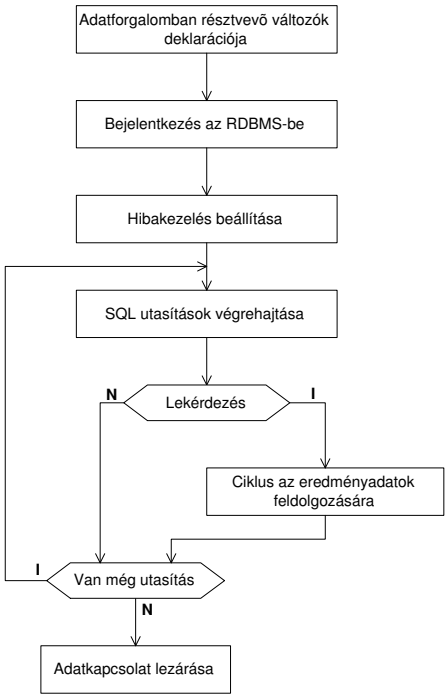
Ezután előbb a gazdanyelvi fordító, compiler átalakítja a forrásszöveget gépi kódú, object állománnyá, majd a szerkesztő, a linker összegyűjti a hivatkozott szimbólumok kódjait. Ehhez természetesen felhasználja az SQL eljárásokat tartalmazó tárgykönyvtárakat, library-kat is. A fenti lépések eredményeként előáll egy végrehajtható, futtatható program, melyet a többi futtatható programhoz hasonlóan indíthatunk például a nevének a megadásával vagy éppen a RUN parancs segítségével. A *fejlesztés általános menetét* mutatja a következő folyamatábra.



7.2. ábra. Programfejlesztés menete beágyazott SQL-el

A beépített SQL legnagyobb előnye, hogy egyesíti a hatékony adatbáziskezelő nyelvet a hatékony algoritmus leíró, felhasználói kezelőfelület készítő programozási nyelvekkel. Mivel mindkettő viszonylag független életet élhet, nincsen összekötve a fejlődésük, így a két nyelv függetlenül és gyorsabban is fejlődhet, és az egyik komponens fejlesztésénél elért eredmények közvetlenül felhasználhatók a beépített SQL nyelvet tartalmazó programokba is. Mivel az SQL több gazdanyelvbe is beépíthető, ezért lehetőség van a feladathoz és a fejlesztőgárdához igazított programozási nyelv kiválasztására, sokkal rugalmasabbá téve a fejlesztést, mintha csak egyetlen egy fejlesztőeszköz állna rendelkezésre. Az adatbáziskezelési, rugalmassági előnyök mellett azonban azt is észre kell venni, hogy az SQL utasítások eljárásainak beépítésével természetszerűleg megnő a programok mérete, és egy adatbáziskezelő utasítás végrehajtása is több időt vesz igénybe, mint egy normál állománykezelő utasítás elvégzése.

Egy beágyazott SQL-t tartalmazó programnak az adatbázissal történő adatforgalom megvalósítása miatt tartalmaznia kell bizonyos tevékenységi elemeket. E tevékenységi elemek kitérnek az adatkapcsolat felépítésére, az adatforgalom megadására és végrehajtására és a kapcsolat lebontására. Mivel e tevékenységeket a fenti sorrendben kell végrehajtani, ezért a beágyazott SQL programokban egyfajta szekvenciát kell megvalósítani az adatkezeléshez kapcsolódó utasításoknál, mint ahogy azt a következő folyamatábra is mutatja.



7.3. ábra. Az adatkezelés folyamata beágyazott SQL-ben

A beépített SQL nyelv használatával kapcsolatban három fő kérdés merülhet fel. Egyrészt, hogyan lehet az SQL utasításokat beépíteni a gazdanyelvi programba. Másrészt az is látható, hogy az alkalmazás azért fordul az adatbáziskezelőhöz, hogy onnan adatokat kérjen le vagy éppen adatokat helyezzen el oda. Tehát igen lényeges kérdés, hogy hogyan történik az adatcsere a gazdanyelvi program és az adatbáziskezelő között. Harmadsorban azt is tapasztaltuk már, hogy utasításainkat bizonyos esetekben nem tudta végrehajtani az RDBMS, tehát a programnak is értesülnie kell a kiadott utasítás végrehajtásának eredményességéről, azaz meg kell oldani a hibakezelés kérdését is. A továbbiakban ezen kérdésekre összpontosítva mutatjuk be a beépített SQL használatát, mely során az Oracle Pro*C fejlesztőeszközét használjuk majd szemléltetésként, tehát gazdanyelvként a C nyelv fog szerepelni.

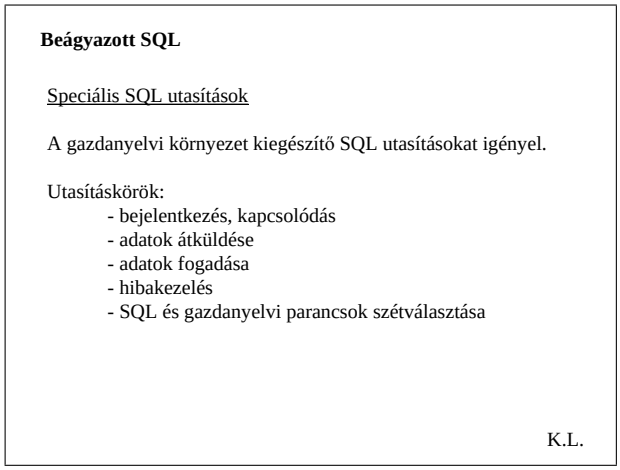
A gazdanyelvbe beépített *SQL utasítások*at nagyon egyszerű megkülönböztetni a gazdanyelvi parancsoktól, ugyanis minden beépített SQL utasítás az

EXEC SQL

kulcsszóval kezdődik. Ha például az autó táblázatban az összes ár értéket megnöveljük 12 százalékkal a C gazdanyelvi programban két értékadás között, akkor a következőképpen néz ki a megfelelő forrásszöveg részlet:

```
a = 3;
EXEC SQL UPDATE auto SET ar = ar * 1.12;
c = f++;
```

A forrásszövegben is több soron keresztül folytatódhat egy SQL utasítás, az utasítás végét ugyanis a pontosvessző, és nem a sorvég karakter jelzi. A gazdanyelve beépített SQL utasítások azonban bizonyos mértékben különböznek az



7.4. ábra. Speciális SQL utasítások

eddig megismert SQL utasításoktól, vannak ugyanis olyan utasítások, melyeket a korábbiakban még nem említettünk, habár az SQL szabvány tartalmazza őket. Ezen utasításokra ugyanis csak itt, a beépített SQL esetén van szükség, és csak itt használhatók. Ezen utasítások a korábban említett problémák, mint a hibakezelés vagy adatcsere, kezelésére szolgálnak majd. A most következő utasítások tehát ismét az SQL92 részei, és csak jellegük miatt tárgyaljuk őket az interaktív SQL szabványutasításoktól elkülönülten.

Az *adatcsere* kérdését érintve először, vegyünk egy olyan példát, amikor a FIAT126 típusú autó árát módosítani kell, ahol az új értéket nem tudjuk még a forrásszöveg megírásakor, a program fejlesztésekor, hanem csak a program futása során fog majd kiderülni. Azaz a felhasználó által a futás során megadott értéket kell bevinni az adatbázisba. Ez a tevékenység a beépített SQL használata esetén két lépésben hajtódhat végre. Elsőként a gazdanyelvi eszközöket felhasználva be kell kérni egy számértéket a felhasználótól, ahol az érték egy gazdanyelvi változóba fog letárolásra kerülni. A második lépésben kiadjuk az UPDATE utasítást, melyben hivatkozunk az értéket tároló gazdanyelvi változóra. Mivel ezen adatcserére szolgáló változókat a többi normál gazdanyelvi változóktól eltérően kell kezelni az előfordítás és a futás során, ezért a beépített SQL nyelv megköveteli, hogy az adatcserére szolgáló gazdanyelvi változókat az előfordító számára is ismertté tegyük, úgymond deklaráljuk őket. A változók *deklarálása* egy deklarációs blokkban történik, melynek kezdetét a

BEGIN DECLARE SECTION;

SQL utasítás és a lezárását, a végét az

END DECLARE SECTION;

SQL utasítás jelzi. Mivel ezek is SQL utasítások, ugyanúgy az EXEC SQL kulcs-

Beágyazott SQL

Adatkapcsolat

A gazdanyelvi változók beépíthetők az SQL parancsba.

```
BEGIN DECLARE SECTION;
  int v;
END DECLARE SECTION;

hivatkozás: :v
```

```
EXEC SQL BEGIN DECLARE SECTION;
 int lv;
EXEC SQL END DECLARE SECTION;
main() {
 scanf("%d",&lv);
 EXEC SQL INSERT INTO ARU VALUES(C.NEXTVAL,:lv);
}
```

K.L.

7.5. ábra. Gazdanyelvi változók

szóval kell bevezetni őket, mint a már ismert SELECT vagy UPDATE utasításokat. A deklarációs blokkban minden olyan gazdanyelvi változónak szerepelni kell, amit egy SQL utasításban adatcserére használunk fel, ahol az adatcsere jelentheti adat fogadását és elküldését is. A szokásos terminológiában *input gazdanyelvi változónak* nevezik azon változókat, melyek értékeit bevisszük az adatbázisba, és *output gazdanyelvi változónak* nevezik azon változókat, melyek az adatbázisból kapnak értéket. A deklarációs blokkban meg kell adni a változó nevét és adattípusát is. Az itt deklarált változókat nem szabad a blokkon kívül még egyszer deklarálni. A blokkban megadott változódeklarációknak, adattípusoknak követniük kell a gazdanyelv szabályait. Azonban csak olyan adattípusok használhatók, amelyek kompatibilisek a hivatkozott adatbázismező típusával. Egyes adattípusok használata azonban sohasem megengedett, így például a Pro*C esetében a *struct* adattípus nem adható meg a deklarációs blokkon belül.

Példaként vegyük azt az esetet, amikor az autó táblázat típus és ár mezőjének lekérdezésére, módosítására hozunk létre két gazdanyelvi változót:

```
EXEC SQL BEGIN DECLARE SECTION;
        int auar;           /* autóár */
        char tipus[26];
EXEC SQL END DECLARE SECTION;
```

Az így megadott *auar* változót felhasználhatjuk az ár mező módosítására. Mint látható nem kell megegyezni a gazdanyelvi változó és a hivatkozott mező nevének egymással. A szabályok azonban nem is tiltják, hogy a változó neve megegyezzen egy mezőnévvel. Ebben az esetben a név alapján viszont már nem tudnánk eldönteni, hogy az SQL utasításban szereplő név a mezőt, vagy a gazdanyelvi változót jelöli-e. A kétértelműségek elkerülése végett a beépített SQL megkívánja, hogy az SQL utasításokban felhasznált gazdanyelvi változók azonosítói elé egy *kettőspontot tegyünk.* A példaként felvetett feladat ezek alapján következőképpen oldható meg:

```
scanf ("%d", &auar);
EXEC SQL UPDATE auto SET ar = :auar WHERE tip LIKE 'FIAT126%';
```

Ha a gazdanyelvi változót nem input céllal, hanem output célra kívánjuk használni, akkor a lekérdező, azaz a SELECT utasításhoz kell kapcsolni. Mivel az eddig megismert SELECT utasítás nem definiálja, hogy hova kerüljön az eredmény, viszont a beépített SQL esetén szükség van a fogadó változó kijelölésére, ezért az SQL szabvány egy módosított SELECT utasítást is tartalmaz erre a célra.

Ez a *kibővített SELECT* utasítás tartalmaz egy új opciót, amit az *INTO* kulcsszó vezet be, melynek segítségével kijelölhető, hogy az eredmény mely változókba kerüljön le. Az INTO kulcsszó közvetlenül a FROM kulcsszó előtt foglal helyet. Ha egyszerre több mező értékét kérdezzük le, akkor több gazdanyelvi változónak is kell szerepelni az INTO opcióban, egymástól vesszővel elválasztva, és a mezőértékek a megadott sorrend alapján kerülnek át a gazdanyelvi változókba. Az INTO opció használata során ügyelni kell a típusok kompatibilitására is. A legnagyobb és legkisebb autóár kiírása a következő utasítással lehetséges, feltéve hogy deklaráltuk mind az *auar1* mind az *auar2* gazdanyelvi változókat a deklarációs blokkban:

```
EXEC SQL SELECT MAX(ar), MIN(ar)
        INTO :auar1, :auar2 FROM auto;
printf ("Max auto ar: %d\n", auar1);
printf ("Min auto ar: %d\n", auar2);
```

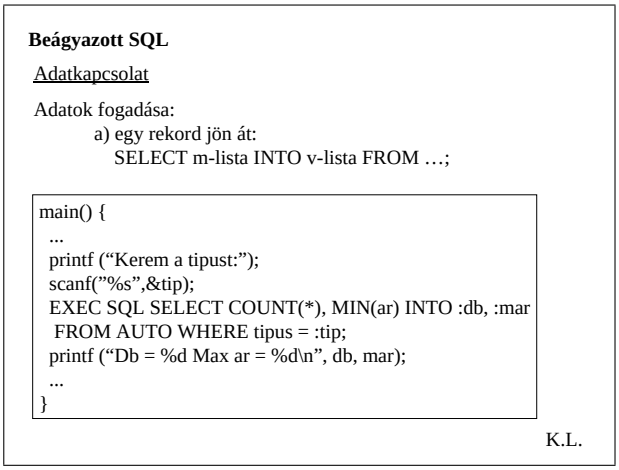
A SELECT többi része megegyezik a korábban megismert SELECT utasítással, azaz tartalmazhat például ORDER BY, GROUP BY, HAVING vagy join művelet opciókat is.

A SELECT utasítás esetén azonban mindenképpen felmerül egy fontos probléma. A SELECT utasítás ugyanis egy eredménytáblázatot szolgáltat, mely rendszerint nemcsak egyetlen egy rekordból áll. Ha például az autók típusait kívánjuk kiíratni, akkor több nevet is fog tartalmazni az eredménytáblázat. Ebben az esetben az INTO kapcsoló után nem állhat egyetlen egy skalár változó, mert abba nem helyezhető el az egész táblázat. Ugyan megoldásként kínálkozik, hogy tömböt adjunk meg az INTO kulcsszó után, azonban ennek a megoldásnak is van hátránya. Ugyanis a tömböket rögzített méretűre kell deklarálni a gazdanyelvi programban, így előre kellene ismerni a SELECT által szolgáltatott eredménytáblázat méretét, hogy annál nagyobb méretűre válasszuk a tömböt.

Ez a megoldás azonban több szempontból is kifogásolható. Egyrészt nem lehet mindig előre megbecsülni az eredménytáblázat méretét, így nagy valószínűséggel rossz tömbméretet fogunk megadni. Másrészről az sem szerencsés, hogy mindegyik eredményrekordnak helyet foglalunk, hiszen sok alkalmazásnál nincs szükségünk mindegyik rekordra egyidejűleg, hanem mindig csak egyet íratunk ki belőle. Ekkor felesleges mindnek helyet foglalni a memóriában, elég lenne egyenként átnézni őket.

A beépített SQL a több rekordból álló eredménytáblák rekordjainak biztonságos lekérdezésére fejlesztette ki a *kurzor (cursor) szerkezetet.*

A kurzor olyan adatstruktúra, mellyel egy több rekordot tartalmazó eredmény-



7.6. ábra. Egy rekord lekérdezése

táblázat rekordjai egymás után beolvashatók. Használata több lépésben történik. Elsőként a *kurzor deklarálását* végezzük el, mely során ismertté tesszük a szerkezet. Ennek során kell megadni azt is, hogy mely SELECT utasítás eredménytáblázatának a lekérdezésére fog szolgálni a kurzor. A második lépés a *kurzor megnyitása*, amikor is a megadott SELECT végrehajtódik, azaz létrejön az eredménytáblázatot tartalmazó struktúra, és ezt követően egy belső mutató rááll az eredménytáblázat első rekordjára. A harmadik fázisban a *rekordok lekérdezése* történik. Az SQL89 értelmezésében, és a legtöbb létező RDBMS esetében a belső mutató egyesével halad előre. Minden rekordolvasás automatikusan eggyel előre lépteti a mutatót. Visszafelé azonban nem mozgatható a mutató, tehát csak szekvenciálisan dolgozhatjuk fel a rekordokat. Az SQL92 szabvány azonban már megengedi a mutató tetszőleges előre vagy hátra irányú mozgatását is. Az utolsó lépés a *kurzor lezárása*, amikor felszabadul az eredménytáblázatnak lefoglalt hely, megszűnik a mutató is. A kurzorhoz tartozó SELECT utasítás nem tartalmazhatja az INTO opciót hiszen ennek elkerülésére készült.

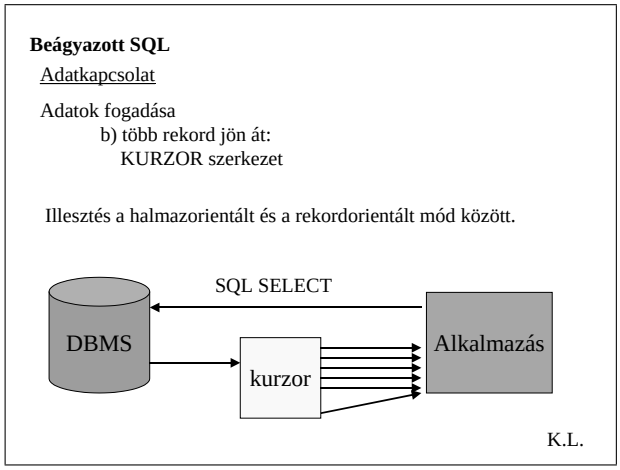
Az SQL89 esetében ha menetközben szeretnénk a mutatót az első eredményrelációra visszaállítani, akkor azt csak a kurzor lezárásával és újbóli megnyitásával tehetjük meg. Ekkor azonban már nem biztos hogy ugyanazt az eredménytáblázatot kapjuk vissza, hiszen minden megnyitás a lekérdezés végrehajtását is jelenti, és ugyanazon SELECT parancs két egymást követő végrehajtásnál más és más eredményt szolgáltathat, hiszen közben megváltozhatott az adatbázis is. A kurzor kezeléséhez a következő utasítások kapcsolódnak:

1. Kurzor definiálása:

   DECLARE kurzornév CURSOR FOR SELECT...;

2. Kurzor megnyitása:

   OPEN kurzornév;



7.7. ábra. Több rekord lekérdezése

3. Kurzor lekérdezése, eredményrekord beolvasása:

   FETCH [NEXT/PREV/FIRST/LAST] kurzornév INTO változólista;

4. Kurzor lezárása:

   CLOSE kurzornév;

Egy FETCH utasítás egy rekordot olvas be az eredménytáblázatból és hatására az aktuális eredményrekord mezőinek értékei sorba elhelyeződnek az INTO kulcsszó után megadott gazdanyelvi változókba. A változólista több gazdanyelvi változó azonosítót tartalmazhat, egymástól vesszővel elválasztva. Az első mező értéke az első változóba, a második mező értéke a második változóba kerül, és így tovább. Itt is ügyelni kell a típusok konvertibilitására. Az utasításban megadott irányjel kulcsszavak jelentése a következő:

NEXT	mozgás előre eggyel;
PREV	mozgás hátra eggyel;
FIRST	mozgás az első rekordra;
LAST	mozgás az utolsó rekordra.

Az elmondottakból következik, hogy az eredménytáblázat lekérdezésére több FETCH utasítást kell kiadni, mégpedig annyit ahány rekordot az eredménytáblázat tartalmaz. A programban mindez egy ciklus szervezését igényli. A ciklus leállási feltételének ellenőrzéséhez pedig azt szükséges tudnunk, hogy mikor olvastuk be már az összes rekordot. Erre a beépített SQL a hibakezelő lehetőségeit használja fel: hibajelzés generálódik, ha a FETCH utasítás elérte az eredménytáblázat végét. Így a programban figyelni kell a hiba jelentkezését, és ha fellépne, ki kell lépni a beolvasó ciklusból.

Ezzel át is léptünk a harmadik nagy területre, a *hibakezelés* területére. A gazdanyelvi programok alapesetben nem értesülnek az RDBMS műveletek során

Beágyazott SQL

Kurzor használata:

- deklaráció
- nyitás
- lekérdezési ciklus
- lezárás

Deklaráció:

DECLARE cnev CURSOR FOR sql-select
FOR UPDATE OF mezo;

Kurzor nyitása = SELECT művelet elvégzése:

OPEN cnev;

```
EXEC SQL DECLARE CURSOR k1 FOR SELECT * FROM auto;
EXEC SQL DECLARE CURSOR k1 FOR SELECT rsz, ar FROM
auto FOR UPDATE OF ar;
…
EXEC SQL OPEN k1;
```

K.L.

7.8. ábra. Kurzor szerkezet: deklarálása, megnyitása

fellépő hibákról, így a végrehajtott SQL utasítás sikerességétől vagy sikertelenségétől függetlenül fut tovább a gazdanyelvi program. Mivel normál esetben egészen más utasításokat kell végrehajtani, ha az SQL utasítás lefutott vagy ha nem futott le, ezért a programozónak expliciten be kell építenie a gazdanyelvi programba az SQL hibák kezelésére vonatkozó elemeket. A hibakezelés alapvetően két módon végezhető el a beépített SQL programok esetén, lehet ugyanis

- közvetlen vagy
- közvetett

a hibakezelés módja. Mindkét esetben az alkalmazásban deklarálni kell egy megadott szerkezetű struktúrát, amelyhez az RDBMS is hozzáférhet. Ez a struktúra arra szolgál, hogy az RDBMS elhelyezze benne a legutoljára elvégzett SQL utasításhoz tartozó hibakódot, azaz hogy ezen keresztül jelezze az alkalmazás felé, hogy sikerült-e végrehajtani a parancsot vagy sem, és ha nem akkor mi volt a hiba oka. Az alkalmazásban az

INCLUDE sqlca;

SQL utasítással lehet a hibakezelésre szolgáló struktúrát deklarálni. Az SQLCA egy *kommunikációs területként* is felfogható, igaz igen egyoldalú kommunikáció folyik ezen a téren az alkalmazás és az RDBMS között, ugyanis az RDBMS csak információküldésre használja ezt a szerkezetet. Az SQLCA struktúra több mezőből épül fel, melyek közül az alábbi mezők tekinthetők a legfontosabbnak:

- *sqlcode*: a kiadott utasítás végrehajtása során felmerült hiba kódszáma; a mező 0 értéket tartalmaz, ha sikeres volt a művelet.
- *sqlerrm*: a felmerült hiba szöveges megadása, ez valójában egy összetett mező, mely két almezőt tartalmaz:
  - *sqlerrml*: a hibaüzenet hossza,

Beágyazott SQL

Kurzor lekérdezése ciklusban:

```
while () {
        FETCH cnev INTO vlista;
}

FETCH NEXT | PREV | LAST | FIRST
```

```
EXEC SQL DECLARE CURSOR k1 FOR SELECT rsz, ar
FROM  auto FOR UPDATE OF ar;
…
EXEC SQL OPEN k1;
while () {
        EXEC SQL FETCH k1 INTO :r1, :a1;
        printf (”rsz=%s ar=%d”,r1,a1);
}
```

K.L.

7.9. ábra. Kurzor lekérdezése

  - *sqlerrmc*: a hibaüzenet szövege.
- *sqlerrd*: egy hatelemű egésztípusú tömb a státuszkódok jelzésére; az egyik státuszkód például a művelet során feldolgozott rekordok darabszámát tartalmazza (azaz például hány rekordot módosított az UPDATE utasítás).

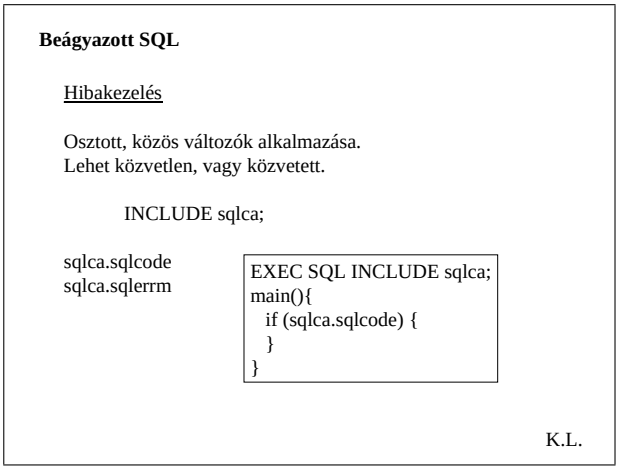
Ebből látható, hogy a szöveg hosszát nem egy lezáró karakter jelzi, hanem egy külön változó tartalmazza.

Közvetett hibakezelésnél az alkalmazás az SQLCA struktúra ellenőrzésével, olvasásával szerez tudomást a felmerült hibákról. Tehát az alkalmazás feladatai közé tartozik az SQLCA rendszeres ellenőrzése és az esetlegesen fellépő hibák lekezelésének, elhárításának a megoldása. Az SQLCA struktúrát az alkalmazásban is gazdanyelvi struktúraként kell kezelni, így például a hiba észlelése és a hibaüzenet kiírása az alábbi C nyelvi utasításokkal lehetséges:

```
if (sqlca.sqlcode) {
        sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrl] = '\0';
        printf ("Hiba: %s\n", sqlca.sqlerrm.sqlerrmc);
        ...
}
```

A példa második C utasítása elhelyezi a hibaüzenet végére a C sztringeket lezáró null karaktert.

Közvetlen hibakezelés esetén az alkalmazásnak nem kell direkt módon lekérdezni az SQLCA mezőinek értéket, egy automatikus lekérdezési opciót lehet beállítani, amely automatikusan és folytonosan ellenőrzi az SQLCA mezőinek értékeit, és a megadott hiba esetén elvégez egy megadott tevékenységet. Az automatikus hibakövetés elindításának SQL parancsa:



7.10. ábra. Hibakezelés

WHENEVER hiba tevékenység;

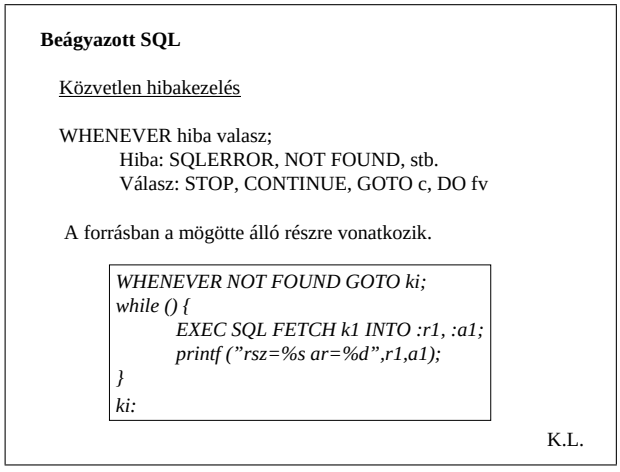
A parancsban az alábbi hibatípusok figyelését jelölhetjük ki:

- *SQLERROR*: hiba lépett fel a végrehajtás során, az SQL utasítás nem hajtódott végre.
- *SQLWARNING*: az SQL utasítás végrehajtódott, de figyelmeztető üzenetet küldött az RDBMS, ugyanis valamilyen nem várt esemény következett be.
- *NOT FOUND*: az RDBMS nem talált a feltételnek megfelelő rekordot. Ez a hiba két esetben következhet be. Egyrészt ha a SELECT utasítás üres eredménytáblázatot adna vissza, másrészt akkor ha a FETCH utasítás elérkezett a lista végére, és nem tud további rekordot visszaadni.

A parancsban a tevékenység helyére az alábbi kulcsszavak adhatók meg:

- *CONTINUE*: a program folytatása;
- *STOP*: a program leállítása;
- *GOTO* címke: az vezérlés átadása az adott címkére;
- *DO* fv(): az adott függvény meghívása.

A WHENEVER utasítás igen kényelmes és egyszerűen használható eszközt ad a kezünkbe az SQL hibák kezelésére, azonban érdemes a használata során néhány momentumra, jellegzetességre odafigyelnünk. Az első észrevétel, hogy a WHENEVER utasítás hatásköre nem a gazdanyelvi program blokkjaihoz kötött, hanem az utasításnak a forrásszövegbeli pozíciója dönti el, hogy mely SQL utasításokra vonatkozik. Ezt úgy képzelhetjük el, hogy az előfordító köti a hibakezelést az SQL utasításokhoz, és ahogy halad előre a forrásszövegben, mindig az utoljára megtalált WHENEVER utasítást tekinti érvényesnek az aktuális SQL utasításra

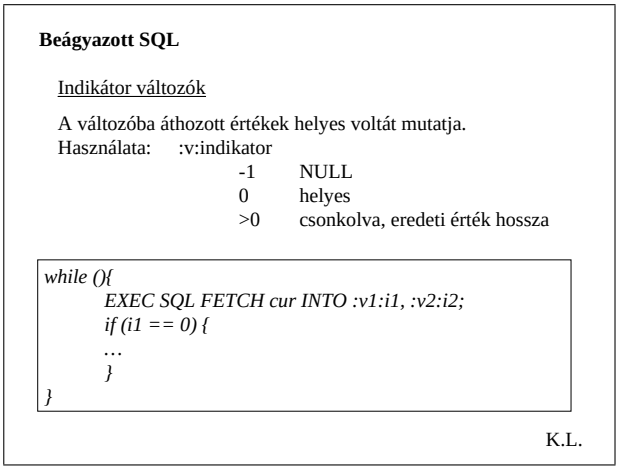


7.11. ábra. Közvetlen hibakezelés

vonatkozólag. Így egy elől elhelyezett WHENEVER utasítás, ha másik WHENEVER utasítást nem adunk meg, egészen a forrásszöveg végéig érvényben marad. Ez egyedül a GOTO tevékenység esetében igényel nagyobb odafigyelést, ugyanis ekkor a megadott címke rendszerint csak egy gazdanyelvi programegységben, például egy függvényben látható, és a többi függvényben, ami még a forrásszövegben utána található, a WHENEVER utasításban szereplő hivatkozások már ismeretlen címkére fognak mutatni, ami a fordítás során fog jelentkezni hibaként.

A másik észrevétel a WHENEVER utasítással kapcsolatban, hogy ha figyelmetlenek vagyunk, könnyen végtelen ciklust idézhetünk elő vele. Ha ugyanis egy WHENEVER utasítással egy megadott címkére küldöm a vezérlést, ahol például visszagörgetem a tranzakciót, és az előző WHENEVER itt is érvényben marad, akkor egy olyan komolyabb hiba esetén, amikor már a tranzakció-lezárás is hibához vezet, végtelen ciklust kapunk. A vezérlés ugyanis egy örökös ciklusban oda-vissza ugrik a címke és a tranzakció-lezárás között, mivel a WHENEVER a címkére küldi a vezérlést a tranzakció-lezárástól, a címkétől pedig a tranzakció-lezáráshoz vezet az út a megadott programkódban.

A hibakezeléshez kapcsolódóan meg kell még említeni az *indikátorváltozók* fogalmát is. Az indikátorváltozók segítségével tulajdonképpen azt ellenőrizhetjük, hogy helyes adatokat kaptunk-e vissza a gazdanyelvi változóinkba. Ha ugyanis egy mező értéke a lekérdezett rekordban üres, vagyis NULL érték van benne, akkor a gazdanyelvi változóban megtalálható érték hamis eredményt fog mutatni, hiszen a hagyományos programozási nyelveknek nincs eszközük a NULL érték kezelésére. Ezért a beépített SQL bevezette az indikátorváltozók használatát, melyek típusukra nézve rövid egészként deklarálhatók. Az indikátorváltozókat a deklarációs blokkban kell deklarálni, és az SQL utasításokban közvetlenül a hozzá kapcsolódó gazdanyelvi változó után szerepel, szintén kettősponttal bevezetve. Egy indikátorváltozó egy SQL utasításban egy gazdanyelvi változóhoz köthető. Ugyanaz az



7.12. ábra. Indikátorváltozók

indikátorváltozó az egymás után következő különböző SQL utasításokban más-más gazdanyelvi változóhoz kapcsolható. Ha egy output gazdanyelvi változóhoz kötjük, a lekérdezés után az alábbi értékek szerepelhetnek az indikátorváltozóban:

- a *nulla* (0) érték azt jelzi, hogy helyes adat van a kapcsolódó gazdanyelvi változóban;
- a *mínusz egy* (-1) érték azt jelzi, hogy az adatbázismező NULL értéket tárolt, és a gazdanyelvi változóban helytelen adat van;
- a *pozitív érték* azt jelzi, hogy a mezőben tárolt érték nem fért el a gazdanyelvi változóban, így ott egy csonkított értéket találunk, az indikátorváltozó ilyenkor a mezőben tárolt adat eredeti hosszát adja meg.

Az indikátorváltozót az input gazdanyelvi változók esetében is használhatjuk, és ilyenkor a NULL érték bevitelének egyik eszköze lehet. Ha ugyanis a -1 értéket rakjuk az indikátorváltozóba, akkor a kapcsolódó gazdanyelvi változóból a NULL érték kerül át az adatbázismezőbe.

A hibakezelés főbb vonásainak áttekintése után egy pillanatra megint visszakanyarodunk a kurzor szerkezethez. Az előzőekből most már látható, hogy az eredménytáblázat rekordjai lekérdezési ciklusának ellenőrzésére vagy az SQLCA közvetlen lekérdezése vagy a WHENEVER utasítás használata szolgáltat megoldást. A könnyebben kezelhető WHENEVER lehetőséget kiválasztva a ciklus megkezdése előtt egy

WHENEVER NOT FOUND GOTO kilepes;

SQL utasítással a vezérlés a ciklus utáni, '*kilepes*' címkével ellátott sorra adható át, ha a FETCH utasítás elérte az eredménytáblázat végét.

A kurzor segítségével tehát sikerült az SQL halmazorientált szemlélete és a hagyományos programozási nyelvek rekordorientált szemlélete közötti konverziót, illetve hidat megvalósítani. A lekérdezett rekord adatai a FETCH révén átkerülnek a gazdanyelvi változókba, és ezután már a program szabadon gazdálkodhat a kiolvasott értékekkel. A FETCH azonban nem teszi lehetővé, hogy az aktuális rekord adatait módosítsuk, így úgy tűnik, hogy továbbra is csak a halmazorientált UPDATE utasítás marad számunkra.

Van azonban egy közvetlenebb megoldás erre a problémára is a beépített SQL esetén, mely rekordok egyenkénti, rekordorientált módosítását vagy törlését is lehetővé teszi bizonyos értelemszerű korlátozások között egy *módosítható kurzor* alkalmazásával. Ennek első lépcsőjeként a kurzor deklarációjánál meg kell adnunk, hogy a kiválasztott rekordokat módosítani vagy éppen törölni kívánjuk. Ezt a kurzor deklarációjának a végén megadott

FOR UPDATE [OF (m_1 [,m_2,...,m_i])]

opcióval tehetjük meg. Ha a kifejezésben mezőlistát is megadunk, akkor csak az adott mezők lesznek módosíthatók. Ha nem adunk meg mezőlistát, akkor minden mező módosítható és a rekord törölhető is. A rekordorientált módosítás második

lépésében a rekordfeldolgozó cikluson belül kiadott UPDATE vagy DELETE utasítást úgy szűkíthetjük le az éppen feldolgozott rekordra, hogy az SQL utasítások WHERE szelekciós feltételében a

CURRENT OF kurzornév

feltételt adjuk meg. Ekkor a műveletet csak a megadott kurzor aktuális rekordjára fog vonatkozni. Az előbb azt is említettük még, hogy nem minden esetben módosíthatók a kurzor rekordjai. Vannak ugyanis olyan esetek, amikor a kurzor SELECT utasítás eredménytáblázata igen összetett módon származik le a bázistáblázatokból, amikor is nem egyértelmű, hogy mely helyen kell módosítani vagy éppen törölni. Az RDBMS-ek rendszerint csak az egy táblázatra vonatkozó SELECT utasítások esetében engedik meg a módosítást, feltéve hogy azok nem tartalmaznak csoportképzési vagy DISTINCT opciókat.

A kurzor szerkezetének bemutatására egy komplettebb példát adunk a most következőkben. A feladatunk az, hogy a FIAT autók árát 15%-kal, a LADA autók árát 12%-kal növeljük meg.

```
EXEC SQL DECLARE autokurz CURSOR FOR
        SELECT tip, ar FROM auto WHERE tip LIKE 'FIAT%' OR
        tip LIKE 'LADA%' FOR UPDATE OF ar;

EXEC SQL OPEN autokurz;

EXEC SQL WHENVER NOT FOUND GOTO vege;

while (1) {
        EXEC SQL FETCH autokurz INTO :atip, :aar:aai;
        if (aai < 0) continue;
        if (tip[0] == 'F') {
             EXEC SQL UPDATE auto SET ar = ar*1.15
                        WHERE CURRENT OF autokurz;
        } else {
             EXEC SQL UPDATE auto SET ar = ar*1.12
                        WHERE CURRENT OF autokurz;
        }
}
vege:
EXEC SQL CLOSE autokurz;
```

A folyamat minden lépése már ismerős számunkra, előbb deklaráljuk a kurzort, melyben a FIAT és LADA típusú rekordokból a típust és árat kérdezzük le. Mivel módosítani is kívánjuk a rekordokat, megadjuk a FOR UPDATE opciót. Ezután megnyitjuk a kurzort, azaz végrehajtódik a lekérdezés. A lekérdező ciklust a WHENEVER utasítással készítjük elő, mely megadja, hogy ha elérnénk az eredménytáblázat végét, a vezérlés kerüljön a '*vege*' címkével jelzett sorra. A ciklusban beolvassuk a típus és ár mező értékeit a megfelelő gazdanyelvi változókba. Az ár

mezőhöz egy *aai* azonosítású indikátorváltozót is tettünk, hogy ellenőrizhessük, van-e ára az autónak vagy még kitöltetlen ez a mező. Ha nincs, továbblépünk a következő rekordra. Ha van, akkor a típustól függően módosítjuk az aktuális rekordot.

Az SQL utasításokban eddig tudatosan csupán skalár értékeket használtunk, a gazdanyelvi tömbváltozók ugyanis csak igen korlátozottan használhatók az SQL utasításokban. A legtöbb RDBMS csak egydimenziós gazdanyelvi tömbök deklarálását engedélyezi. A tömbök használata különben nagyon hasonlít a skalár változók használatához. A tömböket megadhatjuk output és input változóként is. Output változóként használva azonban ügyelni kell arra, hogy az eredménytáblázat beleférjen a tömbbe, különben hibajelzést kapunk. Például az

```
EXEC SQL BEGIN DECLARE SECTION;
int tar [100];
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT ar INTO :tar FROM auto;
```

utasítások esetén, ha a SELECT száznál több rekordot adna vissza, SQL hiba lépne fel, ha pedig kevesebb a rekordok száma, akkor normálisan lefut az utasítás, és az első rekord bekerül a tömb első elemébe, a második bekerül a második tömbelembe, és így tovább. Sokkal kényelmesebb viszont a tömböknek, mint input változóknak a használata. Ekkor ugyanis az

```
EXEC SQL BEGIN DECLARE SECTION;
int tar [100];
char nev[100][20];
EXEC SQL END DECLARE SECTION;
```

Beágyazott SQL

Kurzor lezárása:

```
CLOSE cnev;
```

Módosítási kurzor:

```
UPDATE tabla … WHERE CURRENT OF cnev;
```

```
EXEC SQL DECLARE CURSOR k1 FOR SELECT rsz, ar
FROM  auto FOR UPDATE OF ar;
…
EXEC SQL OPEN k1;
while () {
    EXEC SQL FETCH k1 INTO :r1, :a1;
    EXEC SQL UPDATE auto SET ar=:ua
          WHERE CURRENT OF k1;
}
EXEC SQL CLOSE k1;
```

K.L.

7.13. ábra. Módosítási kurzor

EXEC SQL INSERT INTO tabla VALUES (:tar, :nev);

utasítás hatására mind a 100 rekord bekerül a táblába. A kiadott INSERT utasítás ugyanis sorba veszi a tömb elemeket és mindegyikre végrehajtja a kijelölt utasítást. Így az utasítás hatása megegyezik a következő gondolati ciklussal:

```
for (i=0; i<100; i++) {
        EXEC SQL INSERT INTO tabla VALUES (:tar[i], :nev[i]);
}
```

Az előbb azért említettünk gondolati ciklust, mert a megadott ciklus nem felel meg a beépített SQL szabályainak, mivel az SQL utasításban nem lehet tömbelemekre hivatkozni. A tömböket az INSERT utasításhoz hasonlóan használhatjuk a DELETE vagy UPDATE utasításokban is.

Az eddigiekben ismertetett utasítások áttekintésére vegyünk egy egyszerű példát, amelyben az autó táblázat sorait kérdezzük le. Mint már korábban említettük az Oracle esetén minden felhasználónak azonosítania kell önmagát mielőtt az adatbázishoz hozzáférhetne, és ez az alkalmazások esetén is így van. Ezért a gazdanyelvi programunk is tartalmaz egy bejelentkezési utasítást, melyben megadunk egy azonosító nevet és egy hozzátartozó jelszót. Sikeres bejelentkezés esetén férhetünk csak hozzá a megadott adatbázis objektumokhoz. A bejelentkezés utasítása:

CONNECT :nev IDENTIFIED BY :jelszo;

A bejelentkezés sajátossága, hogy a nevet és a jelszót csak input változókon keresztül lehet megadni, szöveg konstans nem szerepelhet benne.

A példában még két, eddig nem tárgyalt utasítás is szerepel, melyek az RDBMS tranzakció szervezését vezérlik. A tranzakció egy logikailag összefüggő műveletsor, mely egy egységes egészként elfogadtatható vagy elvethető. Ez utóbbi esetben minden eddig benne elvégzett művelet meg nem történtté válik. A tranzakció elfogadásának utasítása a *COMMIT*, elvetésének parancsa pedig a *ROLLBACK*.

Nézzük ezután a példát, melyben egy új, a Pro*C-ben használható adattípust is megismerhetünk. Ez a VARCHAR adattípus változó hosszúságú szövegek kezelésére alkalmas, és tulajdonképpen egy struktúrát takar, melynek két mezője van, egyik a szöveget tartalmazza, másik az aktuális hosszat jelzi. A deklarációban megadott hossz a maximális szöveghosszat jelenti.

```
#include <stdio.h>
                        /* output és input változók deklarálása */
EXEC SQL BEGIN DECLARE SECTION;
    int auar;           /* ar mező */
    char tipus[30];     /* típus mező */
    char rsz[6];        /* rendszám */
    short iar;          /* indikátor */
    VARCHAR nev[40];    /* felhasználó azonosító neve */
    VARCHAR jelszo[40]; /* felhasználó jelszava */
EXEC SQL END DECLARE SECTION;
                        /* hibakezelési kommunikációs terület */
EXEC SQL INCLUDE sqlca;

main()                  /* főprogram */
{
                        /* hibakezelés definiálása */
    EXEC SQL WHENEVER SQLERROR DO hiba();
                        /* bejelentkezés a rendszerbe */
    strcpy (nev.arr,"SCOTT");
    nev.len = strlen (nev.arr);
    strcpy (jelszo.arr,"TIGER");
    jelszo.len = strlen (jelszo.arr);
    EXEC SQL CONNECT :nev IDENTIFIED BY :jelszo;
                        /* lekérdezési ciklus deklarációja */
    EXEC SQL DECLARE kurz CURSOR FOR
        SELECT rsz, ar FROM auto WHERE tip = :tipus;
                        /* típus lekérdezése */
    printf ("tipus = ");
    scanf ("%s", tipus);
                        /* lekérdezés elindítása */
    EXEC SQL OPEN kurz;
                        /* lekérdező ciklus */
    EXEC SQL WHENEVER NOT FOUND GOTO veg;
    while (1) {
        EXEC SQL FETCH kurz INTO :rsz, :auar:iar;
        if (iar == 0) {
            printf ("rendszam=%s ar=%d\n", rsz, auar);
        }
    }
                        /* kilépés a ciklusból */
    veg:
                        /* kurzor lezárása */
    EXEC SQL CLOSE kurz;
                        /* tranzakció lezárása */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}
```

```
hiba()                          /* hibakezelő rutin */
{
                                /* végtelen ciklus elkerülése */
    EXEC SQL WHENEVER SQLERROR CONTINUE;
                                /* hibaüzenet kiírása */
    printf ("hiba: %s\n", sqlca.sqlerrm.sqlerrmc);
                                /* tranzakció visszagörgetés */
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

7.3. Speciális lehetőségek

A beépített SQL utasításoknál a tartalmi különbségek ellenére bizonyos formai azonosságokat is felfedezhetünk. Így az utasításokat a kiválasztott szempont szerinti viselkedésük alapján különböző csoportokba válogathatjuk szét. Az elkövetkezőkben kétféle szempontot vizsgálunk meg, melyből az első azt emeli ki, hogy az SQL utasítás kinek is szól. Ha ugyanis végigtekintünk az eddig vett utasításokon és kiemelünk belőle két utasítást, mondjuk az UPDATE és a WHENEVER utasításokat, akkor láthatjuk, hogy az UPDATE hatására az RDBMS fog bizonyos műveletsort elvégezni, tehát úgy is mondhatjuk, hogy az UPDATE az RDBMS-nek szólt. Ezzel szemben a WHENEVER utasítás nem fog semmilyen RDBMS tevékenységet kiváltani, szerepe az, hogy az előfordítóval közölje, milyen hibákra figyeljen, és hogyan kezelje le az észlelt hibákat. Így a WHENEVER az előfordítónak szóló utasítást jelent.

Ha végignézzük a többi utasítást is, akkor megállapíthatjuk, hogy az összes többi utasítás is e két csoport valamelyikébe sorolható be. Az egyik csoportba tartoznak azok az utasítások, melyek az RDBMS-nek szólnak, azaz amelyek RDBMS műveletsort eredményeznek. Ezen utasításokat *végrehajtható utasításoknak* nevezik. E csoportba tartozik többek között a

SELECT
INSERT
UPDATE
DELETE

utasítás is. A másik csoport az előfordítónak szóló utasításokat foglalja magába, melyeket összefoglalóan *deklarációs utasításoknak* szoktak nevezni. Ezen csoport tagjai a

BEGIN DECLARE SECTION
END DECLARE SECTION
DECLARE
WHENEVER
INCLUDE

utasítások. Az említett szempont szerint képzett csoportok között ugyan némi különbséget találhatunk az utasítások kiadása, használata között, de igazán lényeges, formai eltérést nem tapasztalhatunk. Ezzel szemben a következőként vizsgált

szempont szerinti osztályozásnál már jelentősebb eltéréseket kapunk a kialakult csoportok között.

A második szempontot röviden úgy fogalmazhatnánk meg, hogy az az SQL utasítások *dinamikusságát* vizsgálja. Részletesebben ez alatt a következőket értjük. A gazdanyelvi program tartalmazhat olyan SQL utasításokat, melyek minden eleme már a forrásszöveg megírásakor ismert. Egy ilyen utasítás lehet, például a következő:

*EXEC SQL UPDATE auto SET ar=ar*1.34 WHERE tip LIKE 'FIAT%';*

Ennél az utasításnál már a forrásszöveg írásakor tudjuk, hogy milyen utasítást, műveletet kell végrehajtani, tudjuk milyen objektumokkal és értékekkel kell elvégezni a műveletet, azaz az elvégzendő utasítás minden részletét ismerjük a forrásszöveg megírásakor. Az ilyen jellegű utasításokat *statikus utasításoknak* nevezzük.

Vettünk azonban már olyan utasításokat is a korábbiakban, melyekben bizonyos adatértékeket nem ismertünk a forrásszöveg megírásakor, csak a futás során derült ki az értékük. Ezt a feladatot a gazdanyelvi változók használatával oldhattuk meg. A következő utasítás erre ad példát:

EXEC SQL UPDATE auto SET ar=ar:nov WHERE tip LIKE 'FIAT%';*

Ebben az esetben továbbra is ismert, hogy milyen utasításokat kell kiadni, ismertek az utasításokban szereplő objektumok és az utasítás szerkezete is adott, csupán a benne szereplő egyes adatértékek ismeretlenek a forrásszöveg megírásakor. Az ilyen jellegű utasításokat *szemi statikus utasításoknak* nevezhetjük, hiszen használata nagyban hasonlít a statikus utasítások használatához.

Az eddigiekben ugyan nem találkozhattunk olyan megoldásokkal, habár a beépített SQL majd ezt is megengedi, hogy az utasítás szerkezete is ismeretlen a forrásszöveg megírásakor, és csupán a futás során dől el, hogy végülis milyen szerkezetű, milyen jellegű SQL utasítást kell végrehajtani. A következőkben az ilyen, *dinamikus SQL* parancsok használatát ismertetjük. Természetesen csak a végrehajtható utasításokat adhatjuk meg ilyen módon, illetve még ezek között is vannak megszorítások, így az

OPEN
FETCH
PREPARE
CLOSE
EXECUTE

parancsok nem használhatók dinamikus módon. A dinamikus SQL segítségével jelentősen megnövelhető az alkalmazás rugalmassága, hiszen minden futás alkalmával más és más utasítás hajtható végre. E rugalmasság ára azonban a bonyolultabb programozási technikában és a valamivel lassúbb végrehajtásban fizetendő meg. A dinamikus SQL-t ezért akkor célszerű használni, amikor feltétlenül szükség van rá, azaz akkor, ha nem ismerjük a programírás során, hogy

– milyen utasítást kell végrehajtani,
– milyen feltételeket kell figyelembe venni, vagy
– mely adatbázis-objektumokon kell a műveletet végrehajtani.

Mivel az utasítások szövege a futás során, dinamikusan jön létre, ezért a futás ideje alatt kell elvégezni egy sor olyan előkészítő tevékenységet, melyek normál körülmények között már az előfordítás és a fordítás során lefutnak. A dinamikus SQL parancsok használatának általános lépései az alábbiakban foglalhatók össze:

1. Egy szöveges változóba *letároljuk a végrehajtandó SQL utasítás szövegét.*
2. Az utasítás szövegét *elküldjük az RDBMS-hez ellenőrzés végett (parsing).* Az ellenőrzés során az RDBMS megvizsgálja, hogy szintaktikailag helyes-e az utasítás, illetve megnézi, hogy a hivatkozott objektumok léteznek-e és elérhetők-e.
3. A következő lépésben a felhasznált *gazdanyelvi változókat jelöljük ki (binding).* A létrehozott utasítássorban ugyanis nem a felhasznált gazdanyelvi változók azonosító nevei szerepelnek (a változók azonossága lényegtelen az utasítás szintaktikai, jogosultsági ellenőrzése során), hanem csak úgynevezett *helyettesítő szimbólumokat* (placeholder) tartalmaznak. Ezen lépés feladata az egyes helyettesítő szimbólumokhoz a kiválasztott gazdanyelvi változók hozzárendelése. E megoldás révén ugyanaz a parancssor több különböző változó helyettesítéssel is végrehajtható egymás után.
4. Az utolsó fázisban az így összeállított és ellenőrzött utasítás *tényleges végrehajtása* történik meg, azaz az utasítást átadjuk az RDBMS-nek végrehajtásra.

Az egyes lépések bonyolultságát tekintve a dinamikus SQL utasításokat az alábbi csoportokba sorolhatjuk be:

A: azon nem Query utasítások, melyekben nem szerepelnek gazdanyelvi változók.
B: azon nem Query utasítások, melyekben ismert darabszámú és típusú gazdanyelvi változó szerepel.
C: azon Query utasítások, melyekben ismert darabszámú és típusú gazdanyelvi változó szerepel.
D: azon Query utasítások melyekben ismeretlen darabszámú és típusú gazdanyelvi változó szerepel.

Az *A* esetben a binding fázis elmarad, és az ellenőrzés és a végrehajtás fázisai is egyetlen egy utasítással hívhatók meg. Így ebben az esetben elegendő először összeállítani a parancsot tartalmazó szöveget, és utána rögtön elküldhetjük végrehajtásra az

EXECUTE IMMEDIATE szöveg;

utasítással, melyben a szöveg jelenthet sztring típusú konstanst és változót is. A szöveg szimbólum a parancsot tartalmazó szöveget jelenti. Egy egyszerű példa lehet az *A* típusra a következő utasítássor:

```
EXEC SQL BEGIN DECLARE SECTION;
        char szoveg[40];
EXEC SQL END DECLARE SECTION;

strcpy (szoveg,"UPDATE auto SET ar=ar*1.34");
EXEC SQL EXECUTE IMMEDIATE szoveg;
```

Mint a fenti példából is látható, az SQL utasítást tartalmazó szövegben nem kell megadni sem az EXEC SQL előtagot, sem az utasítást lezáró ';' karaktert. A *szoveg* változó tartalmát nemcsak értékadással, hanem beolvasással is meghatározhattuk volna.

A *B* típusú dinamikus SQL parancsok esetén már három lépés szükséges a végrehajtáshoz. Elsőként előállítjuk a parancsot tartalmazó szöveges változót, majd ellenőrizzük annak helyességét, és legvégül egy utasítással elvégezzük a gazdanyelvi változók hozzárendelését és a tulajdonképpeni végrehajtást is. A *szintaktikai ellenőrzés* parancsa:

PREPARE parnev FROM szoveg;

Az utasításban a *szoveg* az utasítást tartalmazó változó vagy konstans, és a *parnev* a parancshoz rendelt egyedi azonosító nevet jelöl, melynek célja, hogy a további parancsokban egyértelműen azonosítsa a végrehajtandó dinamikus utasítást. A *végrehajtás és a változó hozzárendelés* együttes parancsa:

EXECUTE parnev USING valtlist;

ahol a *valtlist* a parancsban felhasználandó gazdanyelvi változók listáját jelöli, a listában vesszővel elválasztva az egyes gazdanyelvi változókat, melyben minden változóhoz köthető egy indikátorváltozó is. Mind a gazdanyelvi változók, mind az

Beágyazott SQL

Dinamikus SQL

Futás során állítjuk össze az SQL parancsot. Lépései:
- parancs sztring összeállítása
- parancs elküldése ellenőrzésre
- változók kötése
- végrehajtás

Különböző módozatok léteznek a dinamikus jelleg erősségétől függően.

A) nem SELECT utasítások, melyekben nincsenek gazdanyelvi változók
B) nem SELECT utasítások, melyekben adott típusú és számú gazdanyelvi változó szerepel
C) SELECT utasítások, melyekben adott típusú és számú gazdanyelvi változó szerepel
D) SELECT utasítások, melyekben ismeretlen számú és típusú gazdanyelvi változó szerepel

K.L.

7.14. ábra. Dinamikus SQL utasítások

indikátorváltozók neveit kettőspont előzi meg. Példaként tekintsük az előző feladat azon módosítását, amikor az ár mező változásának mértékét egy input változóval adjuk meg:

```
EXEC SQL BEGIN DECLARE SECTION;
        char szoveg[40];
        float arany;
EXEC SQL END DECLARE SECTION;

strcpy (szoveg,"UPDATE auto SET ar=ar*:arany");
EXEC SQL PREPARE parancs FROM :szoveg;
printf ("arany:");
scanf ("%f", &arany);
EXEC SQL EXECUTE szoveg USING :arany;
```

Ebben az esetben a parancsszöveg már tartalmaz helyfoglaló szimbólumokat a későbbi input változók számára. A helyfoglaló szimbólum neve lehet tetszőleges, független a később hozzákötött input változó nevétől. A USING opcióban annyi input változót kell megadni, ahány helyfoglaló szimbólum található a parancsszövegben. Az összerendelés az előfordulási sorrend alapján történik.

Ha egy lekérdezést, azaz egy SELECT utasítást szeretnénk használni dinamikus módon, akkor vagy a *C* vagy a *D* módszer áll rendelkezésünkre. A dinamikus SELECT utasítás egyik jellegzetessége, hogy csak a kurzor szerkezet segítségével hajtható végre. A lekérdezésnél, ha a program írásakor már ismerjük az eredménytáblázat mezőinek és a lekérdezés input változóinak a darabszámát és típusát, akkor a következő lépéseket kell végrehajtanunk:

1. A SELECT utasítás szövegét elhelyezzük egy szöveges változóba.
2. Ellenőriztetjük az utasítás helyességét az előbb megismert

   PREPARE parancs FROM szoveg;

   utasítással.
3. Deklarálunk egy kurzort a lekérdezéshez:

   DECLARE kurnev CURSOR FOR parancs;

   ahol most nem egy SELECT utasítást kell megadni a FOR utáni részben, hanem a létrehozott és ellenőrzött parancs azonosítóját.
4. Hozzárendeljük az input változókat a helyfoglaló szimbólumokhoz és egyúttal végre is hajtjuk a lekérdezést, előállítva az igényelt eredménytáblázatot. A tevékenység parancsa:

   OPEN kurnev USING valtlist;

   A hozzárendelendő változók listájának megadása megegyezik az előbb említett EXECUTE parancsban használt USING opcióval.

5. Lekérdezzük az eredménytáblázat rekordjait egymás után a már ismert

   FETCH kurnev INTO valtlist;

   utasítással, ahol a *valtlist* most az output változók listáját tartalmazza.

6. Lezárjuk a kurzort a

   CLOSE kurzor;

   utasítással.

A *C* típusú dinamikus SQL parancsok használatának bemutatására visszanyúlunk a korábban ismertetett mintaprogramhoz, úgy módosítva azt, hogy a lekérdezés valamilyen, előre nem ismert feltételt tartalmaz a típusra nézve.

```
#include <stdio.h>
                              /* output és input változók deklarálása */
EXEC SQL BEGIN DECLARE SECTION;
     int auar;                /* ar mező */
     char tipus[30];          /* típus mező */
     char rsz[6];             /* rendszám */
     short iar;               /* indikátor */
     VARCHAR nev[40];         /* felhasználó azonosító neve */
     VARCHAR jelszo[40];      /* felhasználó jelszava */
     VARCHAR szoveg[50];      /* parancsszöveg */
EXEC SQL END DECLARE SECTION;
                              /* hibakezelési kommunikációs terület */
EXEC SQL INCLUDE sqlca;
                              /* főprogram */
main()
{
     char feltetel[20];
                              /* hibakezelés definiálása */
     EXEC SQL WHENEVER SQLERROR DO hiba();

                              /* bejelentkezés a rendszerbe */
     strcpy (nev.arr,"SCOTT");
     nev.len = strlen (nev.arr);
     strcpy (jelszo.arr,"TIGER");
     jelszo.len = strlen (jelszo.arr);
     EXEC SQL CONNECT :nev IDENTIFIED BY :jelszo;

                              /* parancssor előállítása */
     printf ("Kérem az ár mezőre vonatkozó feltételt:");
     scanf ("%s", feltetel);
                              /* rr az ár mező helyfoglaló szimbóluma */
     strcpy (szoveg.arr,"SELECT rsz, ar FROM auto WHERE :rr ");
     strcat (szoveg.arr, feltetel);
     szoveg.len = strlen (szoveg.arr);
```

```
                        /* parancs ellenőrzése */
    EXEC SQL PREPARE parancs FROM :szoveg;

                        /* lekérdezési ciklus deklarációja */
    EXEC SQL DECLARE kurz CURSOR FOR parancs;

                        /* a lekérdezés elindítása */
    EXEC SQL OPEN kurz USING :ar;

                        /* lekérdező ciklus */
    EXEC SQL WHENEVER NOT FOUND GOTO veg;
    while (1) {
        EXEC SQL FETCH kurz INTO :rsz, :auar:iar;
        if (iar == 0) {
            printf ("rendszam=%s ar=%d\n", rsz, auar);
        }
    }
                        /* kilépés a ciklusból */
    veg:
                        /* kurzor lezárása */
    EXEC SQL CLOSE kurz;
                        /* tranzakció lezárása */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}
                        /* hibakezelő rutin */
hiba()
{
                        /* végtelen ciklus elkerülése */
    EXEC SQL WHENEVER SQLERROR CONTINUE;
                        /* hibaüzenet kiírása */
    printf ("hiba: %s\n", sqlca.sqlerrm.sqlerrmc);
                        /* tranzakció visszagörgetés */
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

A *C* típusú dinamikus parancsokhoz rendelt megkötések, mint látható, azért kellettek, hogy a forrásszövegben meg tudjuk adni mind az input mind az output változók listáját. Ha viszont nem ismertek az input változók vagy a helyfoglaló szimbólumok, akkor a megadott formátumú OPEN és FETCH utasítások nem használhatóak, és ekkor jutunk el a *D* típusú dinamikus SQL parancstípusokhoz.

A *D* esetben külön kell közölni az RDBMS-sel a végrehajtás előtt, hogy milyen változók és hány darab szerepel a kiadott utasításban. Ehhez a rendszer egy leíró struktúrát (*sqlda*) bocsát rendelkezésre. E struktúrák feltöltésében nyújtanak segítséget a DESCRIBE utasítások. Az *sqlda* egyes mezőit viszont csak közvetlenül

lehet feltölteni, és ezen utasítások megvalósítása rendszerfüggő és ráadásul igen összetett feladat is, ezért ennek részletes bemutatását most mellőzzük. A végrehajtandó tevékenységek alapvonalaiban megegyeznek az előbb említettekkel, azzal a különbséggel, hogy az

EXECUTE parancs USING valtlist;
OPEN kurzor USING valtlist;
FETCH kurzor INTO valtlist;

utasításoknál a változólista helyett mindenütt a létrehozott és feltöltött descriptor struktúrákat kell megadni, így az alábbi utasításokat kell használni:

EXECUTE parancs USING DESCRIPTOR leiro;
OPEN kurzor USING DESCRIPTOR leiro;
FETCH kurzor USING DESCRIPTOR leiro;

A rendszerben két különböző típusú descriptor létezik, az egyik az input változók, a másik az output változók leírására alkalmas. A descriptorok feltöltése igen hosszú folyamat és sok specifikus ismeretet igényel, ezért nem részletezzük mélyebben.

7.4. A CLI program interface

Mint már korábban is említettük, az SQL alapú adatbáziskezelést megvalósító alkalmazások egyik lehetséges csoportját a *CLI alapú* alkalmazások alkotják, melyek fő jellemzője, hogy a gazdanyelvi programba a gazdanyelv szabályait teljesítő formalizmussal RDBMS-t kezelő függvényhívásokat illeszthetünk be. A ProC mechanizmustól eltérően itt nem szabvány SQL utasítások formájában hajtódik végre az adatkezelés, hanem a gazdanyelv függvényhívásain alapszik. Természetesen az RDBMS felé kiadott relációs algebrán alapuló utasítások továbbra is SQL utasítások lesznek, csak más formalizmusban, más előkészítési lépéseken keresztül hajtódnak végre.

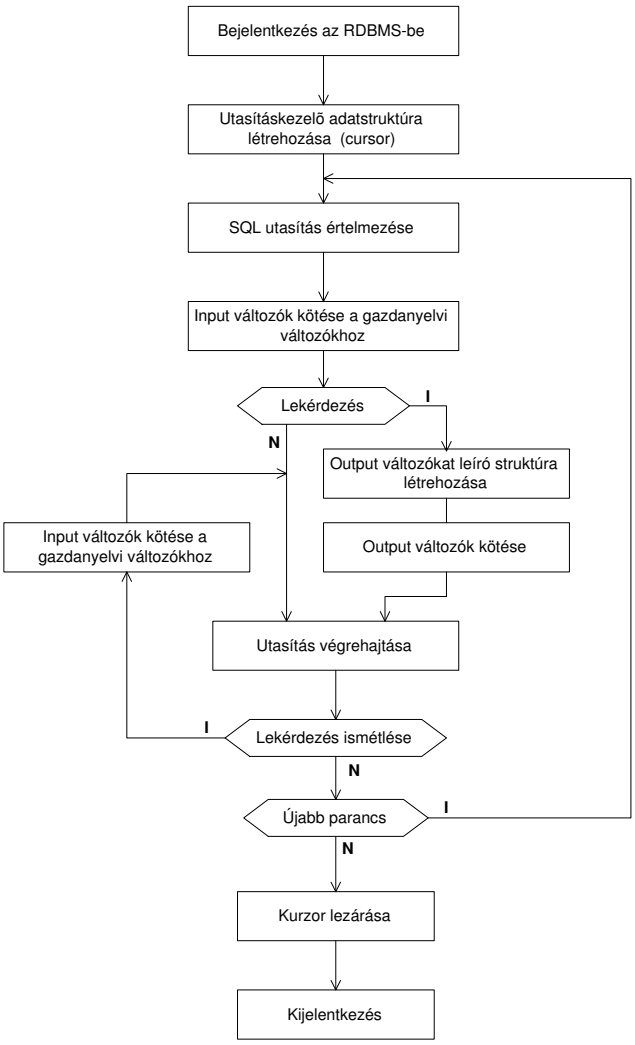
A CLI lehetőségek tárgyalásánál az Oracle rendszer OCI, azaz Oracle Call Interface rendszerét fogjuk bemutatni.

Az OCI formalizmus egy alacsonyabb szinten valósítja meg az adatkapcsolatot, mint a beágyazott SQL programok, hiszen itt a felhasználó felelőssége és feladata az adatkezelő utasítások előkészítése, a gazdanyelvi változók hozzákötése az adatkezelő utasításokhoz. Az OCI szemléletmódja sokkal közelebb áll a hagyományos programozói szemlélethez, hiszen nem tartalmaz a gazdanyelvtől idegen kifejezéseket. A meghívandó rutinok a gazdanyelv normál rendszer-rutinjaihoz hasonlóan egy könyvtárban foglalnak helyet. A program szerkesztése (link) során e könyvtárból kerülnek át a végrehajtási kódok a futtatandó állományba. Az OCI programok esetében nincs tehát szükség előfordítási fázisra, hiszen a felhasználó rögtön az adatkezelő rutinokat adja meg a forrásszövegben.

Az RDBMS rendszerek rendszerint több 3GL gazdanyelvhez is tartalmaznak CLI könyvtárat, melyekből most mi itt is a C nyelvhez kapcsolódó OCI könyvtárra fogunk támaszkodni az ismertetésnél.

A C OCI rendszer mintegy 25 függvényt tartalmaz az adatkezelő SQL utasítások kiadására. Hasonlóan a ProC rendszerhez, az egyes adatkezelő tevékenységek viszonylag kötöttebb sorrendben követik egymást. Ez a kötöttség itt még jobban érezhető, hiszen sokkal részletesebben kell előkészíteni az egyes SQL utasításokat. A részletesség pozitív oldala viszont, hogy így egy általánosabb végrehajtási mechanizmust kapunk, amely viszonylag könnyen kiterjeszthető a dinamikus utasítások területére is.

Az OCI programok adatkezelő utasításainak a végrehajtási kapcsolatát mutatja be a következő folyamatábra.



7.15. ábra. Az adatkezelés folyamata OCI-ban

Az ábra alapján látható, hogy az SQL parancsok kiadása előtt sokkal több előkészítő lépést kell tenni: létre kell hozni egy *bejelentkezési adatterületet*, melyen a kapcsolat főbb jellemzői kerülnek letárolásra. Ezt követően egy *kurzor struktúrát* kell létrehozni, amely egy SQL utasítás végrehajtására szolgál. Egy kurzor szerkezet egyidejűleg csak egyetlen egy SQL utasítást foglalhat magába, de egyszerre több kurzor is nyitva lehet. A kurzor táblába fel kell vinni az *SQL utasítás szövegét*, amit az RDBMS-sel *ellenőriztetni* kell. Az elemzés során előáll az utasítás optimális *végrehajtási terve* is. Az utasítás hivatkozhat az adatbázis adatok mellett gazdanyelvi változókra is, melyek az SQL utasításokban adatbázis-idegen elemekként, hivatkozásokként jelennek meg.

A program elején, az adatforgalom megkezdése előtt *be kell jelentkezni* az Oracle-hez a megadott név és jelszó segítségével. A sikeres bejelentkezés után létrejön a kapcsolat az Oracle adatbázissal. A bejelentkezés is egy függvényhíváson keresztül valósítható meg. A bejelentkezés függvénye az

```
int olon (struct ldadef *lda, char *user_id, int usr_id_len, char *passwd,
          int passwd_len, int audit_flag);
```

Az *olon()* függvény első argumentuma egy struktúra, amely *LDA* (Logon Data Area), azaz bejelentkezési adatterületként ismert. E struktúra mintegy közvetítő szerepet tölt be a program és az RDBMS között, melynek fő funkciója a bejelentkezésnél előforduló esetleges hiba okának felderítése. A bejelentkezés sikerességének ellenőrzésére egyszerű és jól kezelhető módszert kínál az OCI. Az *olon()* függvény ugyanis zérus értékkel tér vissza, ha sikeres volt a végrehajtás, és a nem-zérus érték pedig hiba felléptére utal. A sikeresség ellenőrzésének ez a módja azonban nemcsak az *olon()* függvényre, hanem az összes többi OCI függvényre is vonatkozik. Az LDA struktúra mezőit mutatja be a struktúra C nyelvbeli definíciója:

```
struct ldadef {
        short lda_v2_rc;            /* v2 return kód, már nem használt */
        unsigned char fill1[10];    /* nem használt */
        short lda_rc;               /* visszatérési kód */
        unsigned char fill2[19];    /* nem használt */
        unsigned int lda_ose;       /* OS függő hibakód */
        unsigned char lda_chk;      /* check byte */
        unsigned char lda_sysparam[26];     /* rendszer paraméterek */
};
```

Az *olon()* függvény második paramétere a bejelentkező Oracle felhasználó azonosítására szolgál. Itt egy sztringet kell megadni, melynek tartalmaznia kell a felhasználó nevét. A név mellett a jelszó is megadható *név/jelszó* alakban, és ebben az esetben nincs szükség a *passwd* jelszó mező kitöltésére sem. Ha az átadott sztringek végét a C-ben szokásos '\0' karakter zárja le, akkor a sztringek hosszát jelző *usr_id_len* és *passwd_len* mezők üresen maradhatnak. Az utolsó paraméter egy naplózás jelző.

A következő példa egy tipikus bejelentkezési programrészletet mutat, melyben a felhasználó neve *scott* és jelszava *tiger*.

```
struct ldadef {
        short lda_v2_rc;            /* v2 return kód, már nem használt */
        unsigned char fill1[10];    /* nem használt */
        short lda_rc;               /* visszatérési kód */
        unsigned char fill2[19];    /* nem használt */
        unsigned int lda_ose;       /* OS függő hibakód */
        unsigned char lda_chk;      /* check byte */
        unsigned char lda_sysparam[26];     /* rendszer paraméterek */
};
struct ldadef lda;
char uid[20];

...

strcpy (uid, "scott/tiger");
if (olon(&lda, uid, -1, (char *) -1, -1, 0)) {
        printf ("Hiba a bejelentkezésnél\n");
        exit(-1);
}
else {
        printf ("Sikeres bejelentkezés\n");
}
```

Sikeres bejelentkezés után SQL parancsokon keresztül lehet az adatbázisban tárolt adatokhoz hozzáférni. Az SQL utasítások végrehajtásához azonban definiálni kell a programban egy *CDA* (Cursor Data Area), azaz egy kurzor adatterületet, mely a program és az Oracle rendszer közötti kapcsolattartásra szolgál az SQL parancs végrehajtásához kapcsolódóan. E struktúrán keresztül lehet a fellépő hibákról és a parancs végrehajtásának állapotáról értesülni. A CDA terület C-beli definíciója:

```
struct crsdef {
        short crs_v2_rc;            /* v2 hibakód, nem használt */
        short crs_sql_fc;           /* SQL művelet kódja */
        unsigned long crs_rpc;      /* feldolgozott rekordok száma */
        short crs_peo;     /* szintaktikai hiba helye az SQL parancsban */
        unsigned char crs_oci_fc;   /* meghívott OCI függvény kódja */
        unsigned char crs_fil;      /* nem használt */
        unsigned short crs_rc;      /* hibakód */
        unsigned char crs_wrn_flg1; /* figyelmeztető jelzések jelzője */
        unsigned char crs_wrn_flg2; /* figyelmeztető jelzések jelzője */
        unsigned int crs_cn;        /* kurzor azonosító */
        unsigned char crs_rowid[13]; /* utoljára érintett rekord
                                        pozíciója */
        unsigned int crs_ose;       /* OS hibakód */
        unsigned char crs_chk;      /* check byte*/
        unsigned char crs_fill[26]; /* foglalt terület */
};
```

A programban nem elég csak létrehozni a *crsdef* típusú változót, azt hozzá is kell kötni egy aktív bejelentkezéshez, vagyis *meg kell nyitni* a kurzor területet. A megnyitást végző függvény szintaktikája:

```
int oopen (struct crsdef *cursor, struct ldadef *lda, char *dbn,
           int dbn_len, int area_size, char *user_id, int user_id_len);
```

A megnyitásnál az első paraméter a kurzor terület címét, a második pedig a bejelentkezési adatterületet adja meg. A többi paraméter értéke szokásos esetben elhagyható. A következő példa a kurzor létrehozását mutatja általános esetre vonatkoztatva.

```
struct crsdef {
        short crs_v2_rc;            /* v2 hibakód, nem használt */
        short crs_és ql_fc;         /* SQL művelet kódja */
        unsigned long crs_rpc;      /* feldolgozott rekordok száma */
        short crs_peo;              /* szintaktikai hiba helye
                                       az SQL parancson belül */
        unsigned char crs_oci_fc;   /* meghívott OCI függvény kódja */
        unsigned char crs_fil;      /* nem használt */
        unsigned short crs_rc;      /* hibakód */
        unsigned char crs_wrn_flg1; /* figyelmeztető jelzések jelzője */
        unsigned char crs_wrn_flg2; /* figyelmeztető jelzések jelzője */
        unsigned int crs_cn;        /* kurzor azonosító */
        unsigned char crs_rowid[13]; /* utoljára érintett rekord
                                        pozíciója */
        unsigned int crs_ose;       /* OS hibakód */
        unsigned char crs_chk;      /* check byte*/
        unsigned char crs_fill[26]; /* foglalt terület */
};

struct crsdef cursor;
if (oopen (&cursor, &lda, (char *) -1, -1, -1, (char *) -1, -1) ) {
        hiba();
}
else {
        sikeres();
}
```

A CDA terület létrehozása után elindítható az SQL utasítás *végrehajtása*. Ennek első lépéseként az SQL utasítást el kell küldeni *szintaktikai ellenőrzésre*, és a *végrehajtási terv elkészítésére*. Az előkészítést végrehajtó függvény az *osql3()* függvény, melynek paraméterezése:

```
int osql3 (struct crsdef *cursor, char *sql_stmt, int sql_len);
```

A függvény első paramétere egy létező CDA területre mutat. A második paraméter a végrehajtandó SQL utasítás szövegét tartalmazza. A harmadik paraméterre csak akkor van szükség, ha az SQL szöveg végét nem a '\0' karakter határolja.

A következő programrészlet egy példát mutat be adatbevitelre vonatkozó SQL utasítás esetén:

```
struct crsdef cursor;
struct ldadef lda;
...
olon (&lda, uid, -1, (char *) -1, -1, 0);
oopen (&cursor, &lda, (char *) -1, -1, -1, (char *) -1, -1);
osql3 (&cursor,"INSERT INTO auto VALUES('fgd765','Fiat',4)",-1);
```

Azon SQL utasításokban, melyek nem vonatkozhatnak gazdanyelvi változókra, azaz a DDL és DCL utasítások esetén, az *osql3()* függvényhívás nemcsak előkészíti, hanem végre is hajtja a megadott SQL parancssort. A DML és Query utasítások esetén azonban még további lépésekre van szükség a végrehajtáshoz.

Ennek oka, hogy az alkalmazások rendszerint több egymást követő adatkezelő utasítást is kiadnak ugyanarra a táblára vonatkozóan. Ugyanis az egyes módosítási vagy új értékek nem ismertek a program megírásakor, mivel azok csak a program futása során határozódnak meg, ezért a program forrásszövegében sem lehet előre megadni az aktuális értékeket. Így a DML utasításokban az értékeket nem konstansokként adjuk meg, hanem egy programváltozót jelölünk ki, amelynek aktuális értéke fogja meghatározni az átadandó értéket. Így ugyanazzal az SQL utasítással több különböző értéket lehet felvinni vagy módosítani a parancs egymást követő meghívásaival.

Ha a DML vagy query utasítás egy programváltozót tartalmaz, amelynek aktuális értéke kerül felhasználásra a végrehajtás során, akkor az OCI mechanizmusban közölni kell az Oracle RDBMS-sel, hogy pontosan mely és milyen típusú gazdanyelvi változókhoz kötődnek az SQL utasításban szereplő változó hivatkozások. Vagyis az OCI esetében az SQL utasítás szövegében nem szükségszerű, hogy létező gazdanyelvi változóra hivatkozzunk, mivel egy külön lépés keretében lehet a szereplő hivatkozásokat és a gazdanyelvi változókat összerendelni. Az SQL utasításokban szereplő, változókat reprezentáló szimbólumokat *helyfoglaló szimbólumoknak*, azaz placeholder-eknek nevezik. E szimbólumok lehetnek azonosító névvel és azonosító számmal is megadva. Ez utóbbira ad példát a következő query utasítás:

```
SELECT nev, kor FROM dolgozok WHERE oszt = :1 AND fiz > :2;
```

A példában két szimbólum szerepel, az 1 és 2, melyek értékei a végrehajtáskor határozódnak meg. Természetesen a végrehajtáshoz az 1 és 2 szimbólumoknak létező gazdanyelvi változókhoz kell kötődniük. A helyfoglaló szimbólumok és a gazdanyelvi változók egymáshoz kapcsolódását az

```
int obndrn ( struct crsdef *cursor, int sqlvarnum, void *progvar,
    int progvl, int ftype, int scale, short *indp, char *fmt, int fmtl, int fmtt);
```

illetve az

```
int obndrv ( struct crsdef *cursor, char *sqlvar, int sqlvl, void *progvar,
    int progvl, int ftype, int scale, short *indp, char *fmt, int fmtl, int fmtt);
```

függvényhívások végzik el. Az *obndrn()* függvény a számmal megadott, míg az *obndrv()* függvény az azonosító névvel megadott helyfoglaló szimbólumok hozzárendelésére szolgál.

Az *obndrn()* esetében az *sqlvarnum* a szimbólum számértéke, a *progvar*, a gazdanyelvi változó címe, a *progvl* a változó tárolási hossza, *ftype* a változó típusa, a *scale* nem használt paraméter, és az *indp* jelzi hogy NULL értéket kell-e átadni a programváltozó értéke helyett. Ha az érték negatív a NULL érték fog átkerülni, különben a gazdanyelvi változó értéke kerül át. A további paraméterek nem használtak a C nyelvben.

Az *obndrv()* hívásnál az *sqlvar* a változó azonosító nevét, míg az *sqlvl* a név hosszát tartalmazza. Ha a nevet '\0' határolja, értéke elhagyható, vagyis -1 érték szerepelhet a helyén.

A következőkben vegyünk két példát a változók hozzárendelésére. Az első SQL utasítás legyen

```
osql3(&cursor,"SELECT id, nev FROM dolgozo WHERE fiz > :1",-1);
```

Tegyük fel, hogy a *fizlim* gazdanyelvi változót kívánjuk hozzákötni az 1 szimbólumhoz. Ekkor a hozzárendelést végző függvényhívás:

```
obndrn(&cursor, 1, &fizlim, (int) sizeof (int), INT, -1, (short *) -1,
        (char *) -1, -1, -1);
```

Ha a végrehajtandó SQL utasításban a szimbólum szöveges azonosítású, például

```
osql3(&cursor,"SELECT id, nev FROM dolgozo WHERE fiz > :x",-1);
```

akkor az

```
obndrv(&cursor, ":x", -1, &fizlim, (int) sizeof (int), INT, -1,
        (short *) -1, (char *) -1, -1, -1);
```

lesz az összekötést végző függvényhívás. A példában INT, azaz egész volt a gazdanyelvi változó típusa. Az egész típus mellett használhatók például még a NULSTR (szöveges), FLOAT (valós) és DATE (dátum) adat típusok is.

A query utasítások esetében nemcsak az input értékek kötődhetnek gazdanyelvi változókhoz, hanem a lekérdezések eredményét tároló output értékek is. Ahogy a ProC programokban, így itt sem engedhető meg, hogy egy

```
SELECT * FROM auto;
```

lekérdezés eredménye mindennemű kontroll nélkül kikerüljön az alkalmazás képernyőjére. Így itt is gazdanyelvi változókba kell átirányítani a lekérdezés eredményeit. Úgy ahogy a bemenő szimbólumoknál, itt is *hozzá kell kötni* az eredmény minden mezőjét egy-egy gazdanyelvi változóhoz az alábbi függvénnyel:

```
int odefin (struct crsdef *cursor, int position, void *buffer,
            int buffl, int ftype, int scale, short *indp, char *fmt, int fmtl,
            int fmtt, short retl, short rcode);
```

Egy *odefin()* hívás az eredménytábla egyetlen egy mezőjét fogja hozzákötni valamelyik gazdanyelvi változóhoz. A függvényben a *position* az output szimbólum sorszáma az eredmény rekordon belül, a *buffer* a gazdanyelvi változó címe, a *bufl* a változó tárolási hossza, *ftype* a változó típusa, a *scale* nem használt paraméter, az *indp* jelzi hogy NULL érték került-e be a gazdanyelvi változóba. Ha az érték negatív a NULL érték került át, különben a gazdanyelvi változó értéke kerül át. A *retl* az átadott érték hossza. A további paraméterek nem használtak a C nyelvben.

A következő példa az *odefin()* használatát mutatja be. A végrehajtandó SQL lekérdezés:

SELECT ar, tip FROM auto;

Az eredménylista értékeinek hozzárendelése gazdanyelvi változókhoz, ha az *ar* mező értékét az *auar*, a *tip* mező értékét pedig az *autip* változóban kívánjuk elhelyezni:

```
odefin (&cursor, 1, &auar, (int) sizeof (int), INT, -1, (short *) -1,
        (char *) -1, -1, -1, &ret[0], &ret_ codes[0]);
odefin (&cursor, 2, &autip, 20, NULSTR, -1, (short *) -1,
        (char *) -1, -1, -1, &ret[1], &ret_ codes[1]);
```

A példában azért kellett kétszer is meghívni az *odefin()* függvényt, mert az eredménylistában két mező szerepel.

Az SQL lekérdezések végrehajtása előtt tehát ismerni kell, hogy milyen adatokat fog küldeni az RDBMS válaszként. Az alkalmazások azonban biztosítani tudnak olyan lehetőséget is, hogy a felhasználó a futás során határozza meg a végrehajtandó lekérdezést. Ebben az esetben nem lehet előre tudni a válaszrekordok szerkezetét sem. Hogyan lehet ezt a problémát megoldani?

A megoldást az OCI egy újabb függvénye adja, amely arra szolgál, hogy egy elküldött lekérdezésről meghatározza a válaszként adandó rekordok szerkezetét. A *szerkezetet lekérdező* függvény alakja a következő:

```
int odsc (struct crsdef *cursor, int position, short *dbsize, short fsize,
          short *rcode, short dbtype, char *cbuf, short cbufl, short *dsize);
```

Egy *odsc()* hívás az eredményrekord egy mezőjét adja vissza, így a teljes válaszrekord-szerkezet feltérképezéséhez többször egymásután meg kell hívni ezt a függvényt. A függvény visszatérési értéke mutatja, hogy van-e még újabb eleme a válaszrekordnak.

Az *odsc()* függvény paraméterei a következő jelentéssel bírnak. A *position* a lekérdezett mező sorszáma a válaszrekordban, a *dbsize* a mező maximális mérete, az *fsize* a mező legutolsó beolvasott értékének a mérete, *rcode* a mező legutoljára beolvasott értékéhez tartozó return kód, *dbtype* a mező adattípusának a kódja, *cbuf* a mező adatbázisbeli azonosító neve, *cbufl* a név hossza, és *dsize* a mező értékének maximális kijelzési hossza.

Ha mind az input, mind az output szimbólumokhoz elkészült a gazdanyelvi változókhoz történő hozzárendelés, akkor lehet elvégezni az SQL utasítás tényleges végrehajtását. A tényleges végrehajtás is egy függvényhíváson keresztül valósítható meg, melynek formátuma:

*int oexec (struct crsdef *cursor);*

Látható, hogy ennek a függvénynek a paraméterezése jóval egyszerűbb, mint az előző függvényeknek, ugyanis a végrehajtáskor már minden paramétert beállítottunk az előkészítő függvények segítségével. E paraméterek letárolásra kerültek a kurzor struktúrán keresztül, így elegendő a parancs végrehajtásához a megfelelő kurzor szerkezetre hivatkozni.

Mint korábban említettük, a DDL és DCL utasítások esetében az *osql3()* függvény egyben a végrehajtást is magába foglalja, így az *oexec()* csak a DML és query utasítások esetén válik szükségessé.

A query utasítások esetében a ProC rendszerben külön problémát jelentett, hogy a válaszként megkapott adatok több rekordból álló táblázatok is lehettek. A probléma forrása az volt, hogy a gazdanyelvi változók nem alkalmasak ismeretlen méretű rekordhalmaz közvetlen feldolgozására, hiszen szemléletük a rekordorientált megközelítésen alapul, azaz egyidejűleg egy rekord feldolgozására készültek fel. A ProC rendszerben a kurzor struktúra alkalmazásával tudtuk az RDBMS relációorientált és a gazdanyelvi program rekordorientált adatkezelését összekapcsolni. Mivel az adatkezelés ezen megközelítésbeli különbsége az OCI alkalmazások esetében is fennáll, így itt is szükség lesz egy hasonló megoldásra a több rekordból álló eredménytáblák feldolgozásához.

Az OCI rendszerekben azonban nincs különbség az előkészítés tekintetében az egy rekordot, illetve a több rekordot eredményező lekérdezések között. Mindkét esetben ugyanazon lépések sorozatán keresztül kell eljutni az *oexec()* függvényhívásig. Az eredmény elérése tekintetében is csupán abban van különbség a kétféle lekérdezés között, hogy hányszor kell elvégezni a rekordbeolvasást az eredménytáblából. Az OCI tehát minden lekérdezést úgy kezel, mint több eredményrekordot visszaadó lekérdezést. Így az egy rekordot származtató query egy speciális esetként értelmezhető. Az eredménytáblából az

*int ofetch (struct crsdef *cursor);*

függvényhívással lehet a következő *eredményrekord adatait átvinni* a korábban kijelölt gazdanyelvi változókba. Az output adatokat tartalmazó gazdanyelvi változókat az *odefin()* függvénnyel rendelhettük az egyes eredménymezőkhöz. Az *ofetch()* függvény zérus értékkel tér vissza, ha még van rekord az eredménytáblában. Mivel előre nem tudható az eredményrekordok darabszáma, így itt is egy ciklussal oldják meg a lekérdezést. Az *ofetch()* visszatérési értéke 1403 ha már nincs több rekord az eredménytáblában. Egyéb hiba esetén más lesz a visszatérési érték.

A következő minta egy lekérdezési ciklust mutat be a C gazdanyelv felhasználásával.

osql3 (&cursor, "SELECT DISTINCT kor FROM DOLGOZO", -1);

```
odefin (&cursor, 1, &dkor, sizeof (int), INT, -1, (short *)-1, (char *) -1,
        -1, -1, (short *) -1, (short *) -1);
oecex (&cursor);
for (; ;) {
        if (rv = ofetch (&cursor)) break;
        printf ("kor = %d\n", dkor);
}
```

A parancs végrehajtása és az eredményrekordok feldolgozása után, a program befejezése előtt célszerű a *kurzor terület felszabadítása*. Ehhez a kurzor területet le kell zárni az

```
int oclose (struct crsdef *cursor);
```

függvényhívással. A program végleges leállítása előtt az adatbázissal való kapcsolatot is le kell zárni, azaz ki kell jelentkezni az adatbázisból. Az adatbázisból való *kilépésre* szolgáló függvény az

```
int ologof (struct ldadef *lda);
```

függvény. Meghívásával kijelentkezünk az LDA által kijelölt adatbázisból és felszabadítja az LDA által lefoglalt területet is.

Az előzőekben említett függvények átölelik az OCI legfontosabb adatkezelő funkcióit. Az OCI rendszer e függvények mellett még számos egyéb funkciót ellátó komponenssel rendelkezik, melyek magukba foglalják többek között az itt nem részletezendő tranzakció kezelés és hibakezelés tevékenységeit is. A megadott függvények azonban önmagukban is jól szemléltetik a CLI felület jellegét, legfontosabb tulajdonságait. A későbbiekben egyéb CLI alapú felületeket is meg fogunk ismerni, például az ODBC vagy a JDBC felületet.

Elméleti kérdések

1. Hogyan integrálhatók az SQL utasítások a gazdanyelvi programokba?
2. Hasonlítsa össze a CLI és a beágyazott SQL programozási felületeit.
3. Milyen lépésekből áll egy beágyazott SQL felületű program fejlesztése?
4. Miért tekinthetjük a beágyazott SQL felületet korlátozottabbnak a CLI felülettel való összehasonlítás során?
5. Mire szolgálnak a gazdanyelvi változók és hogyan kell használni őket?
6. Mutassa be az adatok lekérdezésének módozatait a beágyazott SQL felületen.
7. Ismertesse a kurzor fogalmát és kezelő parancsait a beágyazott SQL felületen.
8. Melyek a hibakezelés módozatai és parancsai a beágyazott SQL felületen?
9. Miért és mikor lehet fontos a WHENEVER utasítás CONTINUE típusa?
10. Mutassa be az indikátorváltozó szerepét és használatát a beágyazott SQL felületen.
11. Ismertesse a módosításra kijelölt kurzor jelentését és használatát a beágyazott SQL felületen.
12. Mutassa be a dinamikus SQL utasítások típusait a beágyazott SQL felületen.
13. Melyek a dinamikus lekérdezési művelet végrehajtásának lépései?
14. Melyek a CLI felület legfontosabb parancsai?
15. Hogyan lehet gazdanyelvi változó értékét felhasználni az UPDATE parancsban az OCI felület esetén?
16. Mutassa be a kurzorkezelés menetét az OCI felületen.

Feladatok

1. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely felvisz egy új dolgozó rekordot a DOLGOZÓ[KÓD, NÉV, BEOSZTÁS] táblába. A kód egyediségét nem kell ellenőrizni.
2. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely felvisz egy új dolgozó rekordot a DOLGOZÓ[KÓD, NÉV, BEOSZTÁS] táblába. A kód egyediségét ellenőrizni kell.

*3. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely bekér egy beosztás értéket és kiírja az ilyen beosztású dolgozók névsorát ABC sorrendben. A tábla szerkezete: DOLGOZÓ[KÓD, NÉV, BEOSZTÁS].

*4. Készítsen C gazdanyelv esetén OCI felületű programot, mely felvisz egy új dolgozó rekordot a DOLGOZÓ[KÓD, NÉV, BEOSZTÁS] táblába. A kód egyediségét nem kell ellenőrizni.

5. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely bekér egy kód értéket és kiírja a megadott kódhoz tartozó dolgozó beosztását a DOLGOZÓ[KÓD, NÉV, BEOSZTÁS] táblából. Az átvett értékek helyességét ellenőrizze többek között az indikátorváltozó használatával.
6. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely kiszámolja a dolgozók adóját a fizetés mező alapján egy *ADO(fizetes)* C függvény felhasználásával.
7. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely olyan listát készít, melyben minden kiadónál szerepel egy megadott év után kiadott könyveinek darabszáma. Az évet a program kéri be a felhasználótól. Az adattáblák: KIADÓ[NÉV, VÁROS, KÓD], KÖNYV[ISBN, CÍM, KIADÁSÉV, KIADÓ].
8. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely dinamikusan állítja össze a lekérdező utasítást. A programnak a DOLGOZÓ táblát kell kezelnie és a név, beosztás adatokra lehet szűrési feltételt kijelölni. A tábla szerkezete: DOLGOZÓ[KÓD, NÉV, BEOSZTÁS].
9. Készítsen C gazdanyelv esetén egy beágyazott SQL felületű programot, mely egy üzenet szöveges állományt állít elő. Ebben az állományban a lejárt kölcsönzésekre vonatkozó figyelmeztetések szerepelnek olvasónként. A megfelelő adattáblák: OLVASÓ[KÓD, NÉV, CÍM], KÖLCSÖNZÉS[DÁTUM, LEJÁRAT, KÖNYV, OLVASÓ].
10. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely adatmentést vagy adatbeolvasást végez egy szöveges mentési állománnyal dolgozva. A kezelendő adattábla a DOLGOZÓ[KÓD, NÉV, BEOSZTÁS].
11. Készítsen olyan C programot, mely egy C nyelvbe beágyazott SQL programot elemez és kigyűjti, hogy mely táblák kerülnek feldolgozásra és milyen műveletet hajtanak végre rajtuk.
12. Készítsen C gazdanyelv esetén OCI felületű programot, mely adatokat visz át az egyik adatbázis DOLGOZÓ[KÓD, NÉV, BEOSZTÁS] táblájából egy másik adatbázis ugyanezen szerkezetű és nevű táblájába. A teljes tábla átvitelre kerül.

13. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely egy paraméterként megadott terméknévre kilistázza a termék havi összesített forgalmi adatait, havi sorrendben. A felhasználandó adattáblák: TERMÉK[KÓD, NÉV, TÍPUS], FORGALOM[ÁRU, HÓ, NAP, ÉRTÉK, BOLT].
14. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely termékenként kilistázza a termék nevét és azt, hogy mennyi volt a legnagyobb havi forgalma. A felhasználandó adattáblák: TERMÉK[KÓD, NÉV, TÍPUS], FORGALOM[ÁRU, HÓ, NAP, ÉRTÉK, BOLT].

8. fejezet

Az SQL nyelv további elemei

A korábbi fejezetekben megismerhettük az alap SQL utasításokat, mind az interaktív mind a gazdanyelvbe ágyazott környezetre vonatkozólag. Ezen parancsok ismeretében már meg lehet élni az RDBMS világában, alkalmazásokat is készíthetünk az alapvető adatbáziskezelési műveletekre támaszkodva. Azonban az SQL világa korán sem merül ki a megismert lehetőségekben. Még számos lehetőség és problematikus elem létezik ebben a témakörben, melyek ismerete nagyban segíthet minket a hatékony adatbáziskezelő alkalmazások fejlesztésében, vagy az RDBMS karbantartásában. Ez a fejezet az SQL további lehetőségei közül csak az alkalmazói szinthez szükséges lényegesebb elemeket öleli fel, az RDBMS adminisztrálásához szükséges parancsokra nem térünk most ki. Ezen utóbbi elemek már adatbáziskezelő rendszer specifikusak, hiszen nagyon sok fizikai szintű paraméterezési lehetőséget is tartalmaznak, ezért e terület külön tárgykörbe tartozik.

Az ismertetésbe bevont SQL elemek köre elsősorban az SQL92 szabvány új elemeire vonatkozik, melyek jelentősen kibővítik az alap SQL funkciókörét. A korábbi verziók parancskészletével szemben az SQL92 szabvány még bővebb tábla típus készletet bocsát rendelkezésre, megengedve többek között az ideiglenes táblák használatát is. Emellett az adatkezelő műveletek opcióinak a köre is bővült számos, az értékadás lehetőségeit bővítő elemmel. A funkcionális elemek bővítéséből a lekérdezési rész sem maradt ki, számos új függvényt építettek be az SQL ezen változatába.

Az SQL92 szabvány ismertetése után kitérünk néhány olyan elemre, melyek már a korábbi változatokban is ismertek voltak, azonban hatásuk csak bizonyos szélső esetekben volt érezhető, ezért nem kerültek tárgyalásra az alap SQL keretében, de most érdemes kitérni a bemutatásukra, hogy egy teljesebb képet kapjunk az SQL működéséről.

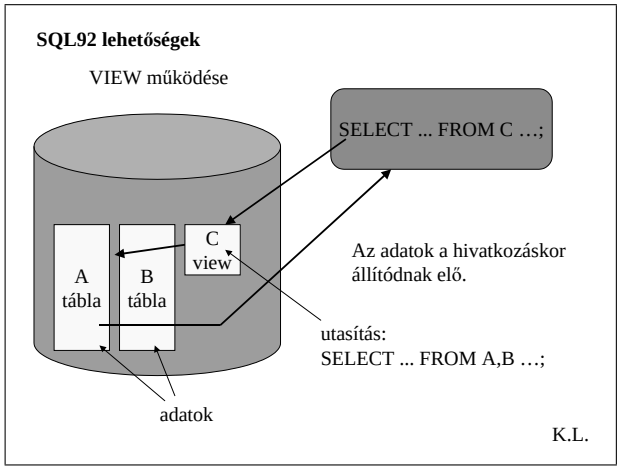
8.1. Az SQL92 adatdefiníciós elemei

A korábbiakban már említettük, hogy az alap SQL-ben az adatok tárolhatók alaptáblában és származtatott táblákban is. A TABLE mint alaptábla szerepelt, azaz minden adateleme ténylegesen is letárolásra kerül az adatbázisban. A VIEW

nézeti tábla esetén csak az azt előállító SQL SELECT utasítás őrződik meg perzisztens módon az adatbázisban, és hivatkozás igény esetén fog a hozzá tartozó műveletsor lefutni, melynek eredményét kapja meg a felhasználó feldolgozásra.

Az SQL92-ben jelentősen kibővült az adattáblák típusainak köre. Elsőként vegyünk egy olyan tábla típust, mely igazából már korábban is létezett egyes RDBMS implementációkban, és a származtatott táblák körébe tartozó új objektum fajtát határoz meg. A VIEW, mint származtatott tábla fő előnye volt, hogy egyszerű hivatkozással lehetett egy komplex eredménytábla adatait elérni, és nem kellett a műveletsort újból végrehajtatni. Az egyszerűség, átláthatóság mellett e megoldás pozitívuma, hogy hivatkozáskor mindig lefut a mögötte álló lekérdezés, így minden esetben a legfrissebb adathalmazt kapjuk meg. Sajnos ezen előnyös vonás megvalósítása egy negatív következményt is hoz magával, nevezetesen azt, hogy minden hivatkozáskor le kell futnia a lekérdezésnek. Ezzel pedig az a gond, hogy bizony időt vesz igénybe, meg kell várni míg előáll a VIEW mögötti eredménytábla. Ez a végrehajtási idő egyes esetekben igen jelentős is lehet. Egy korábbi feladatunknál, sokszázezres rekordot tartalmazó relációkat kellett összekapcsolni a jelentések végrehajtásához, ezért a lekérdezés műveleti ideje órás nagyságrendbe esett. Vannak olyan alkalmazások is, melyekben a válasz generálása nap nagyságrendű időt igényel. Ennyi időt azonban senki nem fog kivárni a számítógépe előtt ülve a jelentés megtekintésére. A VIEW jellegű megközelítés, vagy a vele hasonló időszükségletű közvetlen lekérdezés futtatás tehát nem alkalmas ilyen jellegű feladatokra. Egy más, új megközelítésre van szükség a hatékonyság biztosítása érdekében.

Ez az újfajta megközelítés abban áll, hogy a lekérdezéshez tartozó eredményhalmazt nem az igénylés időpontjában kezdjük előállítani, hanem sokkal korábban, viszont megőrzésre kerül az előállított eredménytábla, és hivatkozáskor ezen eredménytábla kerül felhasználásra. Ebben a megoldásban azt is mondhatjuk, hogy



8.1. ábra. A VIEW működése

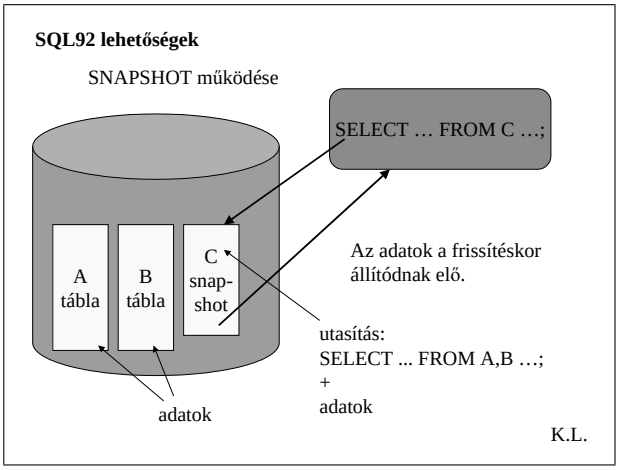
egy pillanatkép készül az eredményről, és ez a felvétel épül be a későbbi lekérdezési műveletekbe. Erre a működési módra utalva, az így előálló objektumot *SNAPSHOT*-nak, azaz pillanatfelvételnek nevezik. A SNAPSHOT is egy származtatott tábla, mely mögött azonban két dolog is letárolásra kerül:

– a származtatás módja, a tartalmat előállító SELECT utasítás és
– a lekérdezés eredményeként előálló rekordhalmaz.

A SNAPSHOT tábla legfontosabb előnye, hogy tartalma gyorsan elérhető, nem kell várni a művelet végrehajtására. Sajnos azonban ezzel a módszerrel elveszítettünk egy korábbi előnyt, az adatok aktualitását. Mivel egy korábban végrehajtott lekérdezés eredményét használjuk, a feldolgozott rekordhalmaz különbözhet attól, amit az adott időpontban való végrehajtás révén kapnánk. A válaszidő lecsökkentésével elvesztettük tehát az adatok aktualitását.

Természetes tehát, hogy a SNAPSHOT-ban tárolt adatok az idő múlásával elévülnek, a kapott eredmény egyre jobban eltér a tényleges adatoktól. Emiatt szükség van tehát az adatok frissítésére. Ezt azt jelenti, hogy a rendszer bizonyos időközönként újra lefuttatja a tábla mögött álló SELECT műveletet, és a kapott eredményt őrzi meg a legközelebbi frissítés időpontjáig. A frissítés gyakorisága természetesen az adatok jellegétől, tartalmától függ, hiszen például egy vállalatnál a dolgozók gyerekeinek a száma nagyon ritkán változik, ezért nem is kell sűrűn frissíteni. Ezzel szemben, ha a kivett szabadságokat nézzük, ez már gyakrabban módosulhat, ezért itt egy sűrűbb frissítési gyakoriságra van szükség. Emiatt a SNAPSHOT tábla egy további paramétere a frissítés gyakorisága.

A SNAPSHOT és a VIEW összevetésével látható, hogy a VIEW akkor célszerű eszköz, ha lényeges az aktualitás, ha gyorsan lefut a lekérdezési művelet. A SNAPSHOT ezzel szemben a ritkábban változó, nagy műveleti időt igénylő lekérdezések kezelésére ad megfelelő megoldást. A SNAPSHOT tábla létrehozásának



8.2. ábra. A SNAPSHOT működése

utasítása a fontosabb paraméterek megadásával:

CREATE SNAPSHOT snév
REFRESH START kezdőidő
NEXT következő frissítési időpont
AS SELECT művelet;

A létrehozott tábla a

DROP SNAPSHOT snév;

utasítással szüntethető meg. Ha például van egy *DOLGOZÓ(kód, név, beosztás, gyerekszám, szülidő)* alaptáblánk, és a több gyerekes 40 év alatti dolgozók kódja és neve szükséges a jelentéshez, akkor a kívánt adatokat egy SNAPSHOT táblába téve, a következő utasítással hozható létre az igényelt SNAPSHOT tábla:

CREATE SNAPSHOT gyerekesek
REFRESH START SYSDATE NEXT 'SYSDATE + 1'
AS SELECT kod, nev FROM dolgozo
WHERE gyerekszam > 1 AND YEAR(SYSDATE)- szulido < 40;

A példában a NEXT paraméter határozza meg a következő frissítés időpontját, méghozzá úgy, hogy minden végrehajtás után kiértékeli a kifejezést, és a kapott dátum típusú kifejezés lesz a következő frissítés időpontja.

A pillanatkép jellegű származtatott tábla mellett az SQL92 rendszer *ideiglenes táblákat* is bevezetett. Az ideiglenes tábla arra utal, hogy a tábla ugyan fizikailag tárolódik, önálló, független tartalommal is rendelkezhet, viszont létezése csak egy megadott tevékenységi egység időtartamára korlátozott. Vagyis a táblát nem szükséges a DROP utasítással megszüntetni, mert az automatikusan megsemmisül a műveleti egység végén.

Az ideiglenes táblák tehát nem nélkülözhetetlen elemek az adatbázisban olyan értelemben, hogy normál alaptáblák alkalmazásával is meg lehetne oldani az általuk ellátott feladatokat, viszont létezésükkel megkönnyítik az alkalmazás fejlesztők dolgát. Az ideiglenes táblák alkalmazásának tipikus esetei:

- jelentések készítéséhez a háttér táblázat összeállítása normál táblákból;
- összetett számítások részeredményeinek tárolása;
- más forrásból átvett adatok ideiglenes tárolási helye.

A szabvány három különböző ideiglenes táblát is értelmez, melyek a létrehozásuk módjában, és az elérhetőségükben, hatáskörükben különböznek egymástól:

- globális ideiglenes tábla,
- létrehozott lokális ideiglenes tábla,
- deklarált lokális ideiglenes tábla.

A *globális ideiglenes tábla* esetében egy CREATE utasítással hozható létre a tábla, és a létrejött tábla a neve ellenére sem látható mindenhol, csak egy ülésen,

azaz session-ön belül. Tehát a felhasználó kijelentkezéséig él a tábla, addig viszont a felhasználó által meghívott minden modulban, rutinban érvényes ez az objektum.

A *létrehozott lokális tábla* létrejötte hasonló az előző globális tábláéhoz, tehát a CREATE utasítás hatására jön létre, viszont láthatósága szűkebb, mint a globális tábláé volt. Egy lokális ideiglenes tábla csak egy modul vagy rutin erejéig, idejéig él. Tehát, ha egy bejelentkezés alatt több modult is futtat a felhasználó egymás után, akkor minden modulban egy önálló példánya jön létre a táblának. A modul a DBMS környezetben egy egységként kezelt SQL utasításcsoportot jelent. Szerepe leginkább a programozási nyelvekben megismert eljárásokhoz, függvényekhez hasonlítható.

A *deklarált lokális ideiglenes tábla* láthatósági köre megegyezik a létrehozott lokális ideiglenes táblával, vagyis csak egy modulon, rutinon belül él. Ez a tábla viszont nem a megfelelő CREATE utasítás kiadásakor jön létre, hanem a rutin meghívásakor, elindulásakor, mivel ez a tábla a lokális változókhoz hasonlóan deklarált a rutin elején, így nincs szükség a CREATE paranccsal történő létrehozására.

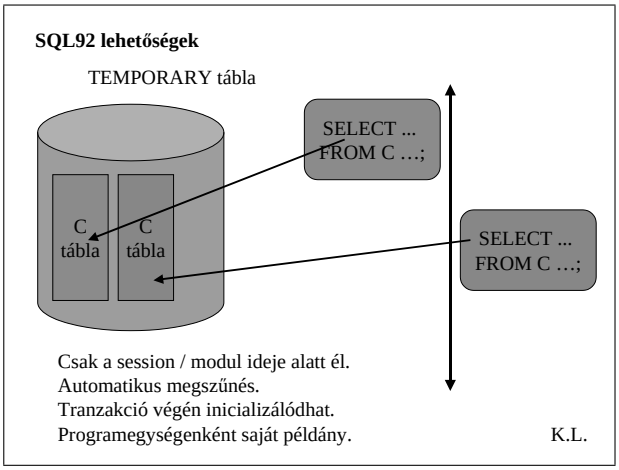
Az ideiglenes táblákat létrehozó SQL utasítások annyiban különböznek a normál tábla létrehozó utasításoktól, hogy szerepel bennük az ideiglenes jellegre utaló TEMPORARY kulcsszó is. Ezek után tehát a lokális ideiglenes tábla létrehozásának utasítása:

CREATE LOCAL TEMPORARY TABLE táblanév (szerkezet) ON COMMIT DELETE|PRESERVE ROWS;

Globális ideiglenes tábla létrehozása:

CREATE GLOBAL TEMPORARY TABLE táblanév (szerkezet) ON COMMIT DELETE|PRESERVE ROWS;

Az ideiglenes táblák egyik fő tulajdonsága az automatikus megszűnés mellett az, hogy a párhuzamosan futó session-ökben, modulokban lehet ugyanazon elneve-



8.3. ábra. Ideiglenes táblák

zésű ideiglenes táblákra is hivatkozni, mégis a felhasználó különböző példányukat fogja látni a tábláknak. Ebben az értelemben a viselkedése nagyban hasonlít a lokális változókéhoz, hiszen ott is különböző rutinokban lehetett egyazon elnevezésű változóra hivatkozni, melyek mögött viszont különböző változó előfordulások voltak. Ebből következően az ideiglenes táblákkal végzett műveletek nem akadályozzák egymást, hiszen amikor mindkét alkalmazás ír egy ugyanolyan elnevezésű ideiglenes táblába, akkor mindkettő egy saját táblába ír, melyet a másik fél nem is lát. Tehát egy adott elnevezésű ideiglenes táblának *több példánya* is lehet a különböző, párhuzamosan futó végrehajtási egységekben.

A táblalétrehozó CREATE utasítások végén álló ON COMMIT tag az ideiglenes tábla viselkedését írja elő a tranzakció lezárások esetére (a COMMIT parancs esetében, hiszen a ROLLBACK visszagörget minden elvégzett műveletet). Ha a PRESERVE ROWS opció van megadva, akkor a COMMIT után megmarad, megőrződik a tábla minden rekordja a következő tranzakcióban való felhasználásra. A DELETE ROWS opció választásával előírjuk, hogy a tranzakció végén minden esetben törlődjön a tábla összes rekordja, így minden tranzakció üres táblával kezdi meg a működését. Ez utóbbi esetben a rendszer erőforrásaival is takarékosabban bánunk, ugyanis az RDBMS ekkor elhagyhatja a táblán végzett műveletek naplózását, mivel úgyis törölni kell a tábla tartalmát a tranzakció végén.

Az előzőekben említettük, hogy az ideiglenes tábla esetében a különböző modulokban egyazon táblanév alatt más-más konkrét ideiglenes táblát lát a felhasználó. A normál táblák esetében, mint ismert, a felhasználó egy adott név alatt a különböző bejelentkezései során mindig ugyanazt a táblát látja. Tehát egy AUTÓ tábla egyértelmű tábla a felhasználó részére. Felmerülhet a kérdés, hogy ha egy *A* felhasználó létrehozott egy AUTÓ táblát, a *B* felhasználó használhat-e szintén AUTÓ táblát? Vajon ekkor ez ugyanaz a tábla lesz vagy két különböző tábla jön létre?

Nos a válasz az, hogy minden felhasználó létrehozhat saját AUTÓ táblát, ugyanis az RDBMS az objektumokat nem egy nagy közös fazékban tárolja, hanem minden felhasználónak megvan a saját munkaterülete. Az SQL89-es verzióban ezt a munkaterületet *sémának* nevezték el. Ebben a megközelítésben minden felhasználónak van pontosan egy sémája, melynek elnevezése megegyezik a felhasználó azonosító nevével. A felhasználók a bejelentkezés után bekerülnek a saját sémájukba, és az ott lévő objektumokra a normál objektumazonosító névre, mint például AUTÓ, hivatkozhatnak. Az AUTÓ hivatkozás tehát a saját sémában lévő AUTÓ objektumot jelenti.

Ha más felhasználó sémájában lévő objektumokat kell elérni, akkor az objektum név önmagában nem elegendő, elé kell tenni, hogy mely sémán belül kell keresni. Ezért ekkor az objektum hivatkozás alakja *sémanév.objektumnév* lesz.

Az SQL92-ben, elsősorban az osztott adatbázis kezelő rendszerek elterjedésére felkészülve, az egyszintű séma-objektum csoportosítás mellett egy többszintű objektum hierarchia is létezik. A szabvány az alábbi szinteket különbözteti meg:

- mezők,
- táblák,

– séma,
– katalógus,
– adatbázis.

A *séma* újszerű értelmezése abban különbözik a korábbi használattól, hogy most egy felhasználó több sémát is birtokolhat, ezért elnevezése nem feltétlenül egyezik meg a felhasználó nevével. Egy új séma létrehozásának utasítása:

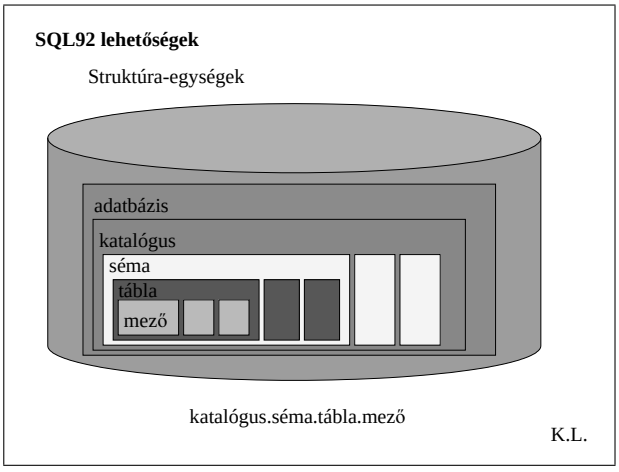
CREATE SCHEMA sémanév AUTHORIZATION felhasználó;

A parancsban az AUTHORIZATION kulcsszóval a séma tulajdonosát lehet kijelölni.

A *katalógus* a sémák bizonyos logikai együttesét jelenti. A katalógus segítségével az adatbázist fel lehet darabolni több különböző logikai szegmensre, melyek egy-egy összetartozó séma halmazt reprezentálnak. A katalógus létrehozásának utasítását nem definiálták az SQL92 szabványon belül, így az RDBMS-től függően hozható létre. A legtöbb adatbázisban csak egy katalógus él. Az objektumok kiterjesztett hivatkozása tehát *katalógusnév.sémanév.objektumnév* lesz az SQL92 szabvány szerint. Az adatbázis rész az elérhető katalógusok összességét jelenti, ami lehet normál vagy osztott adatbázis.

8.2. Az SQL92 speciális függvényei, operátorai

A korábbiakban nem említettük, de bizonyára érzékelhető, hogy az adatok hatékony lekérdezéséhez nem elegendő csupán a mezők értékeit visszaadni, sokszor szükség lehet a mezők adatain végzett konverziós, átalakítási műveletekre is. Ha például a táblában a *szülidő* mező van elhelyezve, akkor az e héten születésnapjukat ünneplők listájához szükség van a mai dátumra, a dátumból a hónap és nap



8.4. ábra. DBMS struktúraegységek

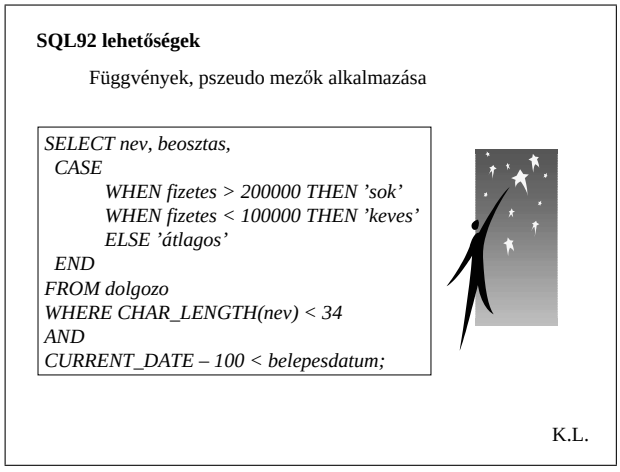
adatokra, valamint a dátumból a hét kijelölésére.

Ezért a gyakorlati RDBMS rendszerek mindig tartalmaznak több különböző függvényt is, melyek kiterjednek a szokásos szöveg, szám, dátumkezelési funkciókra. A következőkben néhány, az SQL92-ben definiált függvényt adunk meg a szokásos funkciók áttekintése végett:

POSITION (mi IN miben)	részszöveg keresése egy szövegben
CHAR_LENGTH(szöveg)	szöveg hossza
UPPER(szöveg)	nagy betűs alakra konvertál
TRIM(mód FROM szöveg)	határoló szóközök eltávolítása
SUBSTRING (rszöveg FROM tsz FOR hossz)	részszöveg kiemelése
CURRENT_DATE	aktuális dátum
||	szövegek összefűzésének operátora
CASE	feltételes érték kiíratás
WHEN feltétel1 THEN eredmény1	
WHEN feltétel2 THEN eredmény2	
...	
ELSE eredmény	
END	
NULLIF(kif, ért)	NULL értéket ad vissza, ha kif=ért

A CASE szerkezet megvizsgálja a megadott feltételeket, és megáll az első olyannál, ahol a feltétel igaz, és annak eredményét adja vissza. Például a lekérdezésben a fizetés érték helyett a kevés, átlagos vagy sok elnevezések szerepelnek, ha a következő parancsot adjuk ki:

```
SELECT nev, beosztas
CASE WHEN fizetes > 200000 THEN 'sok'
```



8.5. ábra. Az SQL92 függvényei

WHEN fizetes < 100000 THEN 'keves'
ELSE 'átlagos' END
FROM dolgozo;

A példában a harmadik mezőnél a *fizetés* mező helyett egy CASE szerkezetet alkalmaztunk.

8.3. SQL92 globális integritási feltétele

A korábbi SQL verziókban az integritási feltételek mind egy táblához kötötten definiálható feltételek voltak, beleértve az idegen kulcs feltételt is, hiszen azt a hivatkozó táblában adtuk meg. Az integritási elemek a tábla létrehozásakor kiadott *CREATE TABLE* utasításban szerepeltek. Az SQL92 lehetővé teszi, hogy egy újfajta globális, nem egy táblához kötött integritási feltételt is kiadhassunk.

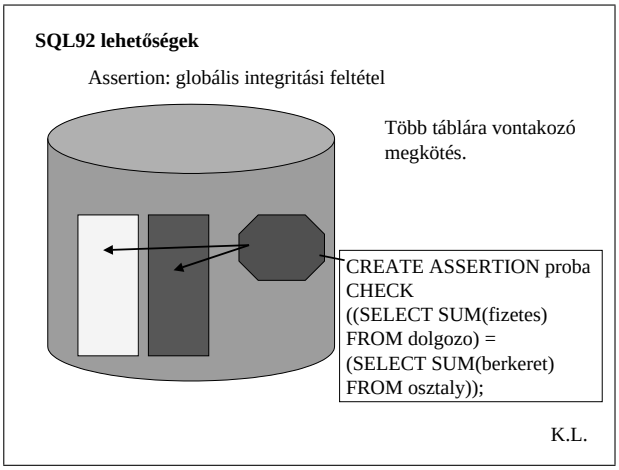
Egy ilyen, nem egy táblához kötött feltételre van szükség, ha például azt akarjuk ellenőriztetni, hogy egy vállalatnál a dolgozók fizetéseinek összege megegyezik-e az osztályokhoz rendelt bérkeret összegével. A példánkban két táblát használunk:

DOLGOZÓ(kód, név, beosztás, osztály, fizetés)
OSZTÁLY(okód, megnev, bérkeret, főnök)

A megadott feltételt az SQL eszközeivel két lekérdezés formájában lehet ellenőrizni, összehasonlítva a két táblában kapott összeget. Ha a két összeg egyenlő, akkor teljesül a megadott integritási feltétel.

A több táblára vonatkozó, önálló integritási feltétel az ASSERTION elnevezést kapta. Az ASSERTION feltételt önálló utasítással lehet létrehozni, melynek alakja:

CREATE ASSERTION a-név CHECK (feltétel);



8.6. ábra. Globális integritási feltétel

A parancsban az *a-név* az önálló feltétel azonosító neve, melyen keresztül lehet később megszüntetni vagy módosítani. A *feltétel* az ellenőrizendő logikai értékű kifejezés, melynek igaz értéke mellett teljesül az integritási feltétel.

Példánkra visszatérve, a most alkalmazandó önálló integritási feltétel a következő paranccsal adható meg:

CREATE ASSERTION proba CHECK ((SELECT SUM(fizetes) FROM dolgozo) = (SELECT SUM(berkeret) FROM osztaly));

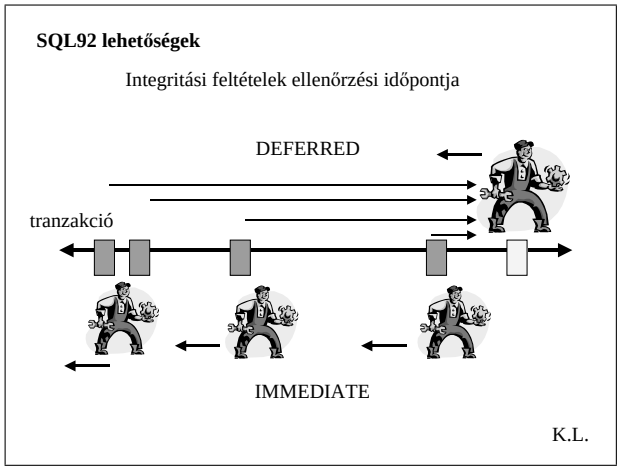
A két SELECT eredményének összegét hasonlítjuk össze az integritási elem feltétel részében.

A könyv második kötetében, amikor a tranzakció kezelésről esik majd szó, látni fogjuk, hogy a tranzakciók, mint adatbázis műveleti egységek egyik fontos jellemzője, hogy konzisztens állapotból kiindulva az adatbázist újra konzisztens állapotba hozzák a tranzakció végére. Menetközben azonban nem feltétlenül kell és lehet minden szabály teljesülését megkövetelni.

Vegyük példaként az előző integritási szabályt. Ha minden pillanatban megkövetelnénk a feltétel teljesülését, akkor az adatbázisunkban nem is módosíthatnánk a fizetés mezőt. Mert ha például az Y osztályon dolgozó X kódú dolgozó fizetését megnövelik Z értékkel, akkor ezen módosításnak az adatbázisba való átvezetéséhez minimum két adatkezelő műveletre van szükség:

UPDATE dolgozo SET fizetes = fizetes + Z WHERE kod = X;
UPDATE osztaly SET berkeret = berkeret + Z WHERE okod = Y;

Ha ebben a sorrendben hajtódik végre a két művelet, akkor az első művelet után, a második előtt, a DOLGOZÓ táblában több lesz a fizetések összege, mint az OSZTÁLY táblában. Tehát nem teljesül az integritási feltétel. Ha a fenti két utasítást



8.7. ábra. Integritási feltételek ellenőrzési időpontja

fordított sorrendben hajtanánk végre, akkor pedig az OSZTÁLY táblában lenne nagyobb az összeg a köztes állapotban. Tehát van egy olyan időpont a tranzakció végrehajtása során, amikor nem teljesül az integritási feltétel.

Az SQL92 az integritási feltételek ellenőrzési idejének a szabályozására is lehetőséget ad, ahol a felhasználó kijelölheti, hogy a megadott szabályt rögtön ellenőriztetni kívánja-e vagy majd csak a tranzakció végén kéri a vizsgálatát. Ezen opció megadása az integritási szabály definíciós részének paraméterei között az

INITIALLY IMMEDIATE | DEFERRED

kulcsszóval történik. Az

INITIALLY IMMEDIATE

hatására rögtön, az SQL paranccsal együtt ellenőrzésre kerül az integritási feltétel. Ez a mód választható például akkor, ha csak egy adatmezőre vonatkozik a feltétel, más objektumtól nem függ a szabály teljesülése. Az

INITIALLY DEFERRED

kiadásával a tranzakció végére toljuk el az integritási szabályok ellenőrzését.

8.4. Az SQL92 adatkezelő műveletek

Az adatkezelő utasítások terén az SQL92 abban hozott bővülést, hogy lehetővé teszi a mező értékek megadásánál a konstans vagy mező értékek helyett al-SELECT utasítások használatát is. Vagyis más táblák adataiból előállított értékek is felvihetők egy-egy adattábla mezőbe. Ez az új funkció az INSERT és UPDATE esetében alkalmazható.

Az INSERT esetében a VALUES listában szerepeltethetünk egy al-lekérdezést, így a parancs alakja a következő formátumra módosul:

INSERT INTO tnev VALUES (. . . ,(SELECT kif FROM . . .),. . .);

Ha például az új osztály rekordban az *okód* az eddigiekben letárolt legnagyobb értéknél eggyel nagyobb értéket vesz fel, akkor a rekordfelvivő utasítás alakja:

INSERT INTO osztaly VALUES ((SELECT max(okod) + 1 FROM osztaly),'iktato',. . .);

A fenti adatfelvivő utasításnak azon alakja is alkalmazható, amikor nevükkel jelöljük ki az értéket felvevő mezőket:

INSERT INTO osztaly VALUES (okod = (SELECT max(okod) + 1 FROM osztaly), megnev = 'iktato',. . .);

A másik, módosítási parancs esetében a SET kulcsszót követően lehet megadni az új értékeket. Ebben az esetben is alkalmazható az al-SELECT lehetőség, melynek formátuma:

UPDATE tnev SET mezo = (SELECT. . .) . . . WHERE . . . ;

Ha például az osztály *bérkeret* értékét kívánjuk aktualizálni a DOLGOZÓ tábla fizetés mezői alapján, akkor az alábbi utasítást kell kiadnunk:

UPDATE osztaly o SET berkeret = (SELECT SUM(fizetes) FROM dolgozo d WHERE d.osztaly = o.okod);

A bevezetett lehetőséggel egyszerűbben meg lehet fogalmazni a létező adatokra építő adatkezelő műveleteket.

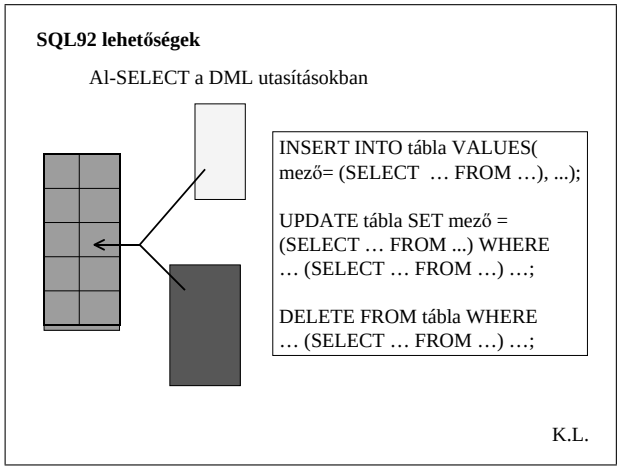
8.5. A NULL érték kezelése

Az adatbázisok egyik lényeges és a többi adatrendszertől eltérő tulajdonsága, hogy a mezők úgymond üres értékűek is lehetnek. Ha például egy normál programozási nyelvet, mint a C nyelv, veszünk alapul, akkor a változók kezelésével kapcsolatban ismert, hogy egy változó a deklarációja után normál módon használható, mindig tartalmaz valamilyen értéket. Ha még nem is adtunk neki értéket, akkor is van mögötte értékes adat, aminek nagysága a programozási nyelv inicializálási mechanizmusától függ. Vegyük példaként az alábbi C nyelvben íródott műveletsort:

```
int c;
printf ("%d",c);
```

Ha a C rendszer a numerikus változókat zérus értékűre inicializálja, akkor a válaszként megjelenő érték is 0 lesz. Ugyanilyen értéket kapunk akkor is, ha az

```
int c;
c = 0;
printf ("%d",c);
```



8.8. ábra. Al-select a DML utasításokban

utasítások hajtódnak végre, sőt akkor is, ha a példa az alábbi elemeket tartalmazza

int c;
scanf(„%d”,&c);
printf (”%d”,c);

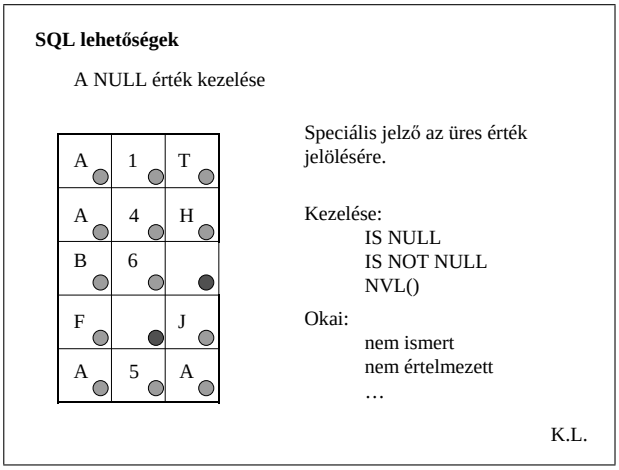
és a felhasználó a zérus értéket adja meg adatfelvitelkor. A példa alapján látható, hogy ugyanazt az értéket látjuk a c változóban akkor is, amikor még ténylegesen nem kapott értéket, mint akkor, amikor már kapott értéket a programban. Vagyis az érték alapján nem tudjuk megkülönböztetni, hogy kapott-e már értéket a változó, vagy sem.

Természetesen, kerülő úton itt is megoldható ez a probléma, hiszen elegendő egy másik segédváltozót bevezetni, melynek értéke mutatja majd, hogy kapott-e már normál módon értéket az alapváltozó, vagy sem. Mivel az adatbázisokban különösen fontos, hogy a valóság adatainak pontos értékei legyenek a mezőkben, ezért beépítették az adatbázisok értékkezelésébe az üres, értéket még nem kapott állapot figyelését, kezelését. Az adatbázisokban tehát közvetlenül láthatjuk, hogy egy mező kapott-e már értéket, vagy még nem.

Az üres érték ismerete igen fontos lehet például egy meteorológiai mérésgyűjtő rendszernél, ahol többek között letárolásra kerül a napi átlaghőmérséklet több állomásra vonatkozólag is. Ha például nem lenne külön üres érték figyelés, akkor egy olyan állomáshoz, amelytől a hőmérséklet még nem érkezett meg, de a többi adat igen, a hőmérséklet mező nulla értékű lenne. Ez az érték viszont lehetne tényleges érték is, ezért ez a megoldás az alábbi problémákat hozza magával:

– az inicializációs érték jelenne meg, mint normál érték;
– nemcsak az egyedi érték torzulna, hanem az aggregált értékek is.

Mindezen problémák miatt előnyös, hogy van külön üres érték kezelés. Az adatbázisokban a mezőhöz automatikusan hozzákapcsolt jelzőbit mutatja meg, hogy



8.9. ábra. A NULL érték kezelése

kapott-e már értéket a mező, vagy még üres. SQL szinten az üres érték úgy jelenik meg, hogy a mező értéke nem egy domain-beli érték, hanem a *NULL* érték. Az SQL-ben több operátor is van az üres érték ellenőrzésére, lekérdezésére:

IS NULL	igaz, ha a mező üres;
IS NOT NULL	igaz, ha a mező nem üres;
NVL(kif1, kif2)	ha kif1 üres értékű, akkor kif2-t adja vissza, különben kif1 a visszatérési érték.

Például a

SELECT nev, NVL(lakcim,"nem ismert lakcim") FROM dolgozo WHERE szuldatum IS NULL;

utasításban azon dolgozók neve és lakcíme kerül kiíratásra, akiknek születési dátuma nem ismert. Ha a lakcím sem ismert, akkor egy üzenet kerül kiírásra.

A rendszer nem veszi figyelembe az üres értéket az aggregációs értékek kiszámítása során sem. Ha például kiadunk egy

SELECT AVG(szuldatum) FROM dolgozo;

lekérdezést, akkor csak a kitöltött, ismert születési dátumú rekordok kerülnek bevonásra.

Az üres érték explicit felvitele szintén a NULL szimbólum segítségével történik:

INSERT INTO dolgozo VALUES (2234,"Peter",NULL,1968);

Az üres értékek mögött többféle ok is húzódhat, ezért többen azt javasolják, hogy érdemesebb lenne több különböző üres érték jelzőt használni. Így például az alábbi különböző okokat lehetne megkülönböztetni:

- nem kitöltött;
- nem értelmezett, nem fog soha értéket kapni;
- ideiglenesen nem értelmezett;
- nem lényeges;
- nem ismert, hogy nem kitöltött vagy nem értelmezett;

  ...

Valójában lehetne a listát még hosszan sorolni, azonban amiatt, hogy a gyakorlatban nem lehetséges mindig ismerni az okokat, és nagyon elbonyolítaná az adatkezelést a több különböző üres érték jelzés, ezért csak egy NULL jelző maradt meg az SQL szabványban.

Tehát csak egy NULL érték van a nyelvben, mégis ezzel is adódhatnak érdekes szituációk. Tegyük fel, hogy adott a következő minta táblázat:

DOLGOZÓ			
kód	*név*	*szüldátum*	*fizetés*
1	Lajos	1967.02.14	NULL
3	Laszlo	NULL	45666
2	Peter	1971.06.09	98111

Ha például keressük az 50000-nél többet keresőket a

*SELECT * FROM dolgozo WHERE fizetes > 50000;*

utasítással, akkor az eredményünk, mint ahogy várjuk is a

kód	*név*	*szüldátum*	*fizetés*
2	Peter	1971.06.09	98111

táblázat lesz. Ha az 50000-nél többet nem keresőket válogatjuk le a

*SELECT * FROM dolgozo WHERE fizetes <= 50000;*

paranccsal, akkor az eredmény a

kód	*név*	*szüldátum*	*fizetés*
3	Laszlo	NULL	45666

tartalmú tábla lesz. A példa alapján jogosan felmerülhet a kérdés, milyen értékű lesz vajon az 1-es kódú rekordnál az összehasonlítás eredménye. Úgy tűnik, mindkét esetben hamis értéket kaptunk. Ha azonban most a

*SELECT * FROM dolgozo WHERE NOT fizetes > 50000;*

lekérdezést hajtjuk végre, akkor az eredmény újra csak az alábbi lesz:

kód	*név*	*szüldátum*	*fizetés*
3	Laszlo	NULL	45666

Vagyis, az üres értékű mezőnél az összehasonlítás eredménye se nem igaz, se nem hamis.

Tehát az adatbázis kezelők az üres értékű mezőknél nem mondhatják biztosan azt hogy igaz, sem azt hogy hamis. Így például az első rekordnál, nem mondhatjuk azt, hogy a fizetés nagyobb mint 50000, mert nem ismert a fizetés érték. Hasonlóan azt sem mondhatjuk, hogy nem igaz, hogy nagyobb mint 50000. Ezen megfontolások alapján olyan megoldást kellett választani, melyben a rendszer mondhatja azt az összehasonlítás eredményére, hogy nem tudom. Vagyis egy logikai kifejezés értéke ennek alapján lehet

- igaz,
- hamis,
- nem ismert.

Az ilyen, három logikai értéket megengedő logikát nevezik *háromértékű logikának*. A 3VL rendszerben természetesen megváltozik a logikai műveletek igazság táblázata is, hiszen be kell venni a nem ismert értéket (U) is. Példaként nézzük az AND művelethez tartozó műveleti táblát:

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

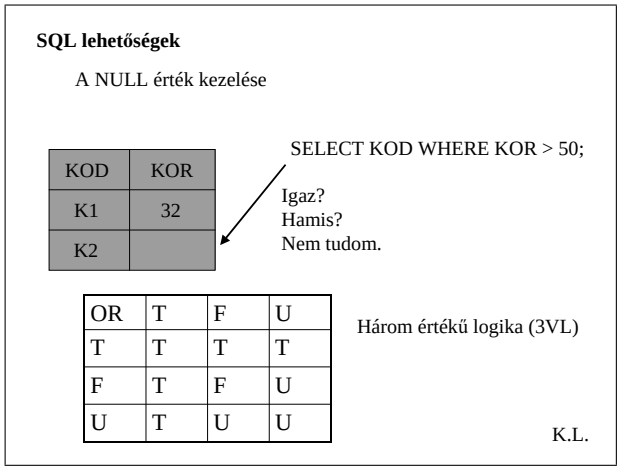
Az SQL rendszerek a szelekciós feltételek kiértékelésekor a 3VL szabályai szerint dolgoznak, azzal a kiegészítéssel, hogy a megjelenítésnél csak a T értékű rekordokat veszik be az eredménytáblázatba, az U és F értékűek nem szerepelnek.

A fenti működési mód következménye, hogy néhány hagyományos logikai szabály itt nem a megszokott módon működik. Így például a *harmadik kizárt elve* nem teljesül, mely szerint egy állítás vagy igaz vagy hamis, harmadik út nincs. Vegyük például az alábbi lekérdezést:

*SELECT * FROM dolgozo WHERE fizetes > 50000 OR NOT fizetes > 50000;*

Vagyis lekérdezzük, kinek nagyobb a fizetése mint 50000, vagy nem nagyobb a fizetése mint 50000. Mivel egy embernek vagy nagyobb, vagy nem nagyobb a fizetése mint 50000 és minden embernek van fizetése, minden dolgozó rekordnak meg kellene jelennie az eredmény relációban. A 3VL szabályai szerint viszont az üres fizetés mezővel rendelkező rekordnál a feltétel mindkét tagja U típusú így az OR eredménye is U lesz. Az eredménybe viszont csak a T értékűek kerülnek be, ezért ez a dolgozó nem jelenik meg, holott a hagyományos gondolkodási mód szerint meg kellene jelennie.

Ehhez a problémához kapcsolódóan további bővítést is el lehetne végezni az SQL szabványon belül. Itt most csak két kérdésre világítanánk rá, az egyik az



8.10. ábra. Három értékű logika

eredményhalmaz értelmezése és a másik az EXIST operátor működése.

Az SQL SELECT eredményhalmazában csak a T értékűek jelennek meg a működési szabály alapján. Emiatt nem lehet különbséget tenni a hamis (F) vagy nem ismert (U) állapotok között közvetlenül. A nem ismert értékű rekordok halmazát ezért csak kerülő úton lehet elérni, amikor is a teljes halmazból kivonjuk az igaz és hamis értékűek unióját. Az egyszerűbb kezelés végett jó lenne, ha közvetlenül is le lehetne kérdezni az U értékűre kiértékelt rekordok halmazát.

A másik megemlítendő probléma az EXIST operátor működése. Mint ismert az EXIST után egy al-SELECT áll, és ha ez az eredményhalmaz nem üres, akkor igaz értékű a kifejezés, ha pedig üres az eredményhalmaz, akkor hamis értékű a kifejezés. Az ide vonatkozó probléma az, hogy az EXIST operátor nem tud U logikai értéket visszaadni, csak T vagy F értéket. Így viszont, ha az al-SELECT-ben olyan rekord van, melyben a vizsgált mező üres értékű, akkor lesznek U értékű rekordok is, melyek itt F-re konvertálódnak. Emiatt a tényleges eredmény halmaz nem tükrözi az U értékek létezését, eltakarja azokat a felhasználó elől.

Ha például vesszük az alábbi módosított DOLGOZÓ táblát, amely a megfelelő OSZTÁLY táblára tartalmaz hivatkozást

DOLGOZÓ				
kód	*név*	*osztály*	*szüldátum*	*fizetés*
1	Lajos	1	1967.02.14	NULL
3	Laszlo	2	NULL	45666
2	Peter	2	1971.06.09	98111

és lekérdezzük, mely osztályokon van 50000-nél többet kereső dolgozó, akkor az 1-es dolgozó esetében hiába U értékű lesz a szelekciós feltétel, az eredménybe nem kerül be, így úgy veszi a rendszer, mintha biztosan kisebb fizetése lenne az 1-es dolgozónak 50000-nél. A lekérdezés parancsa:

*SELECT * FROM osztaly o WHERE EXISTS (SELECT nev FROM dolgozo d WHERE d.osztaly = o.kod AND d.fizetes > 50000);*

Mindez tehát azt mutatja, hogy célszerűbb lenne az eredményben is szétválasztani, megadni, hogy T vagy U logikai érték adódott a kiértékelés során.

8.6. A hierarchikus SELECT művelete

A korábbiakban sokat foglalkoztunk a relációs algebra jellemzésével, megemlítve többek között az algebrában rejlő nagyfokú rugalmasságot és kifejező erőt. A különböző algebrai műveletek megfelelő kombinálásával igen összetett feladatokat is meg tudunk oldani. Úgy tűnik, hogy az algebra eszközeivel minden, a relációs adatbázisra vonatkozó információ igény kielégíthető. Ugyan valóban igen hatalmas a relációs algebra kifejező ereje, mégsem teljesen univerzális eszköz, és mint látni fogjuk, lehetnek olyan lekérdezések, információ igények, melyek megválaszolásához a relációs algebra eszközrendszere önmagában nem elegendő.

A relációs algebra tárgyalása során láthattuk, hogy az algebra halmazorientált alapokon nyugvó műveleteket tartalmaz, és nem a rekordorientált megközelítésen alapszik. Ebből kiindulva talán már érezhető, hogy valahol a rekordorientált megközelítést igénylő, procedurális elemeket is tartalmazó műveletek körében kell keresni azon fehér foltokat, melyekre nem tud választ adni a relációs algebra. Mi most a hierarchikus SELECT problémáját emeljük ki, mivel ez a feladat a gyakorlatban nem elhanyagolható műveletet jelent, melynek fontosságát az RDBMS készítők is felismerték, és néhányan implementálták is ezt a funkciót SQL kiegészítésként a termékükben. A tárgyalásunk során mi az Oracle cég megoldását fogjuk részletesebben bemutatni.

Példaként vegyünk egy vállalati dolgozó nyilvántartási rendszert, melyben a dolgozók közötti főnök és beosztott kapcsolat is letárolásra kerül. Az alkalmazott minta tábla szerkezete:

DOLGOZÓ (kód), név, beosztás, főnök)

A pontosabb táblasémát leíró CREATE utasítás felépítése a következő:

CREATE TABLE dolgozo (kod NUMBER(5) PRIMARY KEY, nev CHAR(35), beosztas CHAR(25), fonok REFERENCES dolgozo);

A sémából látható, hogy a főnök kijelölése egy, a táblára önmagára mutató idegen kulccsal történik. Azaz minden dolgozónak egy közvetlen főnöke van, de lehet több közvetlen beosztottja. Így egy dolgozó hierarchia alakul ki.

A táblára vonatkozó információ lekérdezések egyike lehet az az eset, amikor nemcsak a dolgozók adatai kellenek önmagukban, hanem a főnökre vonatkozó információknak is szerepe van. Egy ilyen lekérdezést jelent az, ha például egy megadott személy összes közvetlen beosztottját kérdezzük le. Ha a személyt mondjuk most az egyszerűség végett a kódjával azonosítjuk, akkor a SELECT utasítás alakja a következő lesz:

SELECT nev FROM dolgozo WHERE fonok = X;

ahol X jelöli a megadott személy kódját.

Ez a lekérdezés egy egyszerű szelekciót jelent, és nem okoz problémát a felírása. Ha azonban most egy lépéssel tovább megyünk, és a kérdést úgy tesszük fel, hogy kérjük az X kódú személy összes közvetlen beosztottját illetve azok közvetlen beosztottjait. Mintha a családfában a gyerekek és unokák listájára lenne szükség. Ekkor már egy összetettebb lekérdezés lesz, hiszen a második szinten álló beosztottakat külön a

SELECT nev FROM dolgozo WHERE fonok IN (SELECT kod FROM dolgozo WHERE fonok = X);

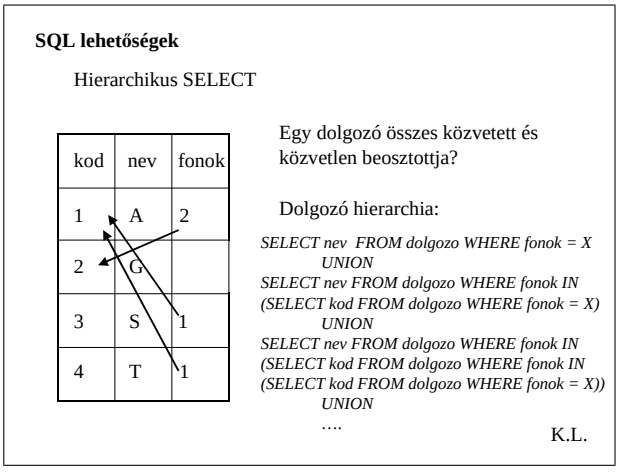
utasítással kérdezhetjük le, ahol az al-SELECT-ben a beosztottak közötti főnököket keressük meg. Mindkét szint lekérdezéséhez a két szint eredményeinek unióját kell venni:

SELECT nev FROM dolgozo WHERE fonok = X
UNION
SELECT nev FROM dolgozo WHERE fonok IN (SELECT kod FROM dolgozo WHERE fonok = X);

Ha most folytatjuk a szinteken való előre haladást, és a kérdést úgy tesszük fel, hogy adjuk meg az X kódú személy összes, bármely szinten álló beosztottját, akkor már egy megoldhatatlan problémába ütközünk. Ugyanis a *k.* szinten álló beosztottakhoz a *(k-1).* szinten álló beosztottak ismeretében tudunk csak eljutni. Ezért a *k.* szintig *k* különböző rész-SELECT utasítást kell szerepeltetni, melyek eredményeinek uniója adja meg a teljes választ. Ennek során látható, hogy a *k.* szinten álló rész-SELECT tag már *(k-1)* al-SELECT utasítást ágyaz be a szelekciós részben. Példaként megadjuk az utasítás első néhány tagját:

```
SELECTnev FROM dolgozo WHERE fonok = X
UNION
SELECT nev FROM dolgozo WHERE fonok IN
    (SELECT kod FROM dolgozo WHERE fonok = X)
UNION
SELECT nev FROM dolgozo WHERE fonok IN
    (SELECT kod FROM dolgozo WHERE fonok IN
        (SELECT kod FROM dolgozo WHERE fonok = X))
UNION
...
```

Látható tehát, hogy minél mélyebb szintre megyünk, annál terjedelmesebb lesz a lekérdezési utasítás. S mivel véges hosszú SELECT utasítást tudunk csak készíteni, ezért csak véges mélységbe tudunk lemenni a beosztottak felkutatásában. Tetszőleges mélység elérése nem lehetséges a relációs algebra, és az arra épülő SQL alkalmazásával.



8.11. ábra. A hierarchikus SELECT problémája

A probléma megoldása egy rekurzív hierarchia bejárás lenne, melyben minden csomópontnak vesszük a gyerekeit egy megadott bejárási elvet követve. A relációs algebra viszont nem tartalmaz semmilyen rekurzív végrehajtási elemet, csak valamilyen procedurális környezetben oldható meg ez a feladat. Mivel a feladat az adatbázis alkalmazásokban is előfordul, az RDBMS fejlesztők kibővítették a SELECT utasítás funkció körét úgy, hogy alkalmas legyen az ilyen jellegű, hierarchia bejárásra is. Az így módon kiterjesztett SELECT utasítást nevezik hierarchikus SELECT-nek.

A hierarchikus SELECT mögött tehát egy rekurzív fa bejárási algoritmus van, mely nem része a relációs algebrának, ezért az SQL szabványban sem kapott még helyet ez a művelet. A hierarchikus SELECT alkalmazására az Oracle rendszerben megvalósított szintaktikát vesszük alapul. A fa bejárás megvalósításához a következő paraméterekre van mindenképpen szükség:

- induló csomópontok kijelölése,
- kapcsolódási feltétel megadása a szülőtől a gyerekek felé.

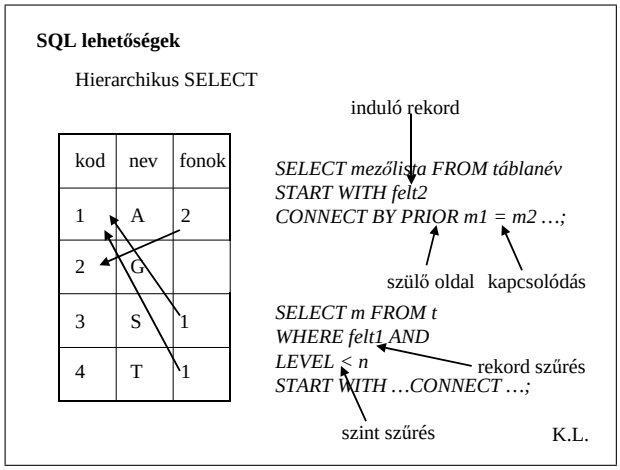
Emellett még további paraméterként felhasználhatók az alábbi elemek:

- bejárásba bevont elemek körének kijelölése,
- maximális mélység megadása.

A fenti paraméterek az alábbi alakban adhatók meg:

SELECT mezőlista FROM táblanév WHERE felt1 START WITH felt2 CONNECT BY PRIOR m1 = m2 . . . ;

A fenti parancsban az alábbi speciális paraméterek fordulnak elő:



8.12. ábra. A hierarchikus SELECT megoldása

START WITH felt2	a bejárás induló elemét kijelölő feltétel megadása.
CONNECT BY PRIOR m1 = m2	a szülő-gyerek kapcsolatot kijelölő kapcsolódási feltétel. A kapcsolatleíró kifejezésben a szülő szerepkörhöz tartozó oldal előtt meg kell adni a PRIOR kulcsszót.

A bejárásba bevont rekordok körét a WHERE részben lehet szűkíteni. Ebben a szelekciós részben lehet a bejárási mélységet is korlátozni, mégpedig egy *LEVEL* pszeudó mező segítségével. Ez a változó minden rekord esetén megadja az illető rekordnak a bejáráskor elfoglalt szintmélységét. Ez az érték nem része a fizikai alaptábla rekordnak, csak a fenti művelet végrehajtása során kapcsolódik a rekordhoz ideiglenesen.

A fenti parancs alapján az eredetileg feltett kérdésünk most már egy véges SELECT paranccsal megfogalmazható:

SELECT nev FROM dolgozo START WITH kod = X CONNECT BY PRIOR kod = fonok;

A parancsban a PRIOR szó áthelyezésével megfordíthatjuk a hierarchia bejárásának az irányát.

8.7. Adatbázis objektumok

Az eddigiekben megismert adatbázis objektumok köre elsősorban a táblákat öleli fel, megadva az alaptáblák és a származtatott táblák több különböző módozatait is. Egy igazi adatbázis azonban a táblákon túlmenően nagyon sokféle különböző objektum típust is tárolhat, kezdve a táblákhoz kapcsolódó kiegészítő elemektől, egészen a procedurális elemeket tartalmazó eljárásokig vagy függvényekig. A következőkben ezen objektum típusok alap vonásait foglaljuk össze, megadva jelentésüket és a fontosabb alkalmazási területeket. Jelen leírásban nem szerepelnek viszont a létrehozáshoz és karbantartáshoz szükséges parancselemek szintaktikai leírása; ezekre nem terjed ki a fejezet tartalma.

Index

Az index szerkezet, mint talán már ismert az adattáblában tárolt rekordok hatékony, gyors elérését szolgálja. Az index struktúrában a keresés alapjául szolgáló indexkulcs és a hozzá tartozó rekord pozíciójának megadása található. Az indexbejegyzések az indexkulcs szerint rendezetten foglalnak helyet a struktúrában. Az indexek létrehozása kétféle módon történhet:

- explicit létrehozás a CREATE INDEX paranccsal,
- implicit létrehozás integritási elem létrehozásakor.

Az *explicit* esetben a felhasználó adja meg az index nevét és az indexkulcsot. Az indexet létrehozó SQL utasítás alakja:

CREATE INDEX inev ON tabla (mezo1 ASC | DESC, mezo2 ...);

ahol *inev* az index elnevezése és *tabla* a vonatkozó táblázat neve. Zárójelben az indexkulcsot megadó mezőket kell felsorolni. Az első mező adja az elsődleges rendezési szempontot. A másodikként megadott mezőt csak akkor veszi figyelembe a rendszer, ha az elsődleges mező alapján nem tud dönteni a rekordok sorrendiségét illetően.

Az *implicit* létrehozási mód esetében olyan integritási feltétel megadására kerül sor, melynek hatékony ellenőrzése igényli egy megfelelő index létezését, használatát. Ilyen jellegű megkötések azok, melyek a mezők értékeinek egyediségére vonatkoznak, mint a

- *PRIMARY KEY*,
- *UNIQE*.

Ha tehát létrehozunk egy táblát, melyben elsődleges kulcs is szerepel, akkor automatikusan létre fog jönni egy index a kulcsként kijelölt mezőhöz.

Szekvencia

Bizonyos adatbáziskezelő rendszerekben az adatbázisban létrehozhatók globális számlálók (sequence-ek), melyek felhasználhatók mesterséges sorszám típusú értékek, egyedi azonosítók generálására. A szekvencia, mint minden más adatbázis objektum, védelemmel ellátható, tehát szabályozható, hogy kik férhetnek hozzá. A szekvencia létrehozásakor többek között megadható, hogy milyen értéktől kezdődjön a sorszámok generálása, mi legyen a legnagyobb érték, lehet-e ciklikusan haladni a sorszámok generálásában, vagy sem. A szekvencia létrehozásának utasítása:

CREATE SEQUENCE snev CYCLE | NOCYCLE MINVALUE er1. . . ;

A szekvenciák felhasználhatók a rekordok felvitelekor és módosításakor közvetlenül is. Ebben az esetben mezőértékként a szekvencia NEXTVAL tulajdonságát kell szerepeltetni:

INSERT INTO dolgozo VALUES (kod=snev.NEXTVAL, nev="Peter",. . .);

A szekvencia segítségével a DOLGOZÓ tábla kód mezője automatikus egyedi értéket kapott, ugyanis a NEXTVAL hivatkozás nyomán a szekvencia számláló állása nemcsak lekérdeződik, hanem előre is lép a megadott lépésköz értékkel. Ezzel a mechanizmussal egy AUTOINCREMENT jellegű mező hozható létre, melynek értéke automatikusan beállítódik új mezők felvitele esetén a soron következő szabad értékre.

Eljárás

Az adatbázisok ugyan elsődlegesen az adatok tárolására szolgálnak, de az adatkezelő alkalmazások hatékony végrehajtása érdekében lehetővé tették, hogy végrehajtandó kód modulokat is megőrizzünk az adatbázisban. Ezen mechanizmus előnye, hogy egy megadott műveletsor elvégzéséhez nem kell a tevékenységet leíró SQL és procedurális elemeket minden alkalmazásban külön-külön megvalósítani,

elég csak egyszer, egy helyen letárolni az adatbázisban. A különböző alkalmazások ekkor ezen központi eljárást (procedure) hívhatják meg a műveletsor végrehajtására. E megoldás előnyei:

- elég egy helyen karbantartani a program modult,
- az adatbázis védelmi rendszere alkalmazható a modulra is,
- hatékonyabb végrehajtást eredményez,
- kisebb hálózati forgalmat jelent.

Az eljárások létrehozása is SQL paranccsal történik:

CREATE PROCEDURE enev (paraméterlista) IS modultörzs;

A modul törzsében kell megadni, hogy milyen elemi lépéseket kell végrehajtani az eljárás működése során. A törzsben a műveleti lépések megadása rendszerint az SQL valamely procedurális kiegészítésével történik. Ez a fajta kiegészítés azonban már RDBMS specifikus, ezért azt az egyes RDBMS rendszerek megismerése során szokás elsajátítani. Az Orcale rendszer esetében a PL/SQL (Procedural Language SQL) nyelv az, amely ötvözi az SQL utasításokat a procedurális vezérlési elemekkel, a változó- és hibakezelési mechanizmusokkal.

Függvény

A tárolt eljárásokhoz hasonlóan egy program modult valósít meg a függvény (function) is. A fő eltérés az eljárással összevetve az, hogy a függvénynél szerepel visszatérési érték is. A függvény létrehozásának SQL utasítása:

CREATE FUNCTION fnev (paraméterlista) RETURN tipus IS modultörzs;

A parancsban a RETURN után álló adattípus adja meg a visszatérési érték típusát.

Trigger

Az adatbázisok egyik fő előnye az integritás, a konzisztencia magas szintű biztosítása. Ennek érdekében az adatbázisok lehetőségeit is állandóan fejlesztik, hogy minél jobban megfeleljenek ennek az elvárásnak. A fő irányelv az adatbázis motorok fejlesztése során, hogy amit csak lehet, azt tegyük át az alkalmazási oldalról az adatbázis oldalra, biztosítva azt, hogy az objektum tárolása redundancia mentesen megvalósítható legyen. A másik nagy előnyük, hogy így központi felügyelet, a DBMS ellenőrzése alatt hajtódnak végre az utasítások, nem függve az egyes alkalmazások helyességétől, hatékonyságától.

Mint a bevezető részben említettük, az adatbázis alkalmazásoknál a műveletek egyik fontos tulajdonsága, hogy a tevékenységek nagyobb egységekbe, tranzakciókba szervezetten hajtódnak végre. A tranzakció összes művelete biztosítja a helyes adatbázis állapot átmenetet. A gyakorlati alkalmazások során még azt is megfigyelhetjük, hogy ezen műveletsorokat rendszerint egy-egy megadott, jól definiálható esemény, tevékenység indítja el. Ha például egy új dolgozó kerül a vállalathoz, akkor egy jól meghatározott tevékenységi láncot kell végrehajtani a felvétel

teljes adminisztrálására. Mivel a tevékenységsor elemei jól definiáltak, meg lehet adni e műveletsort előre, és amikor szükséges el lehet indítani.

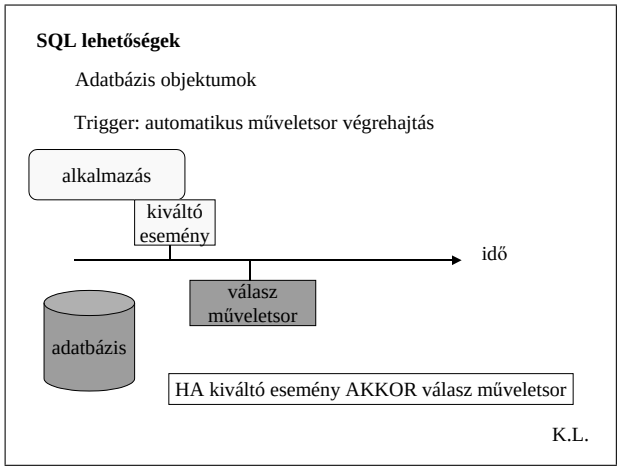
Az egyes DBMS rendszerek lehetőséget adnak arra, hogy ne kézzel kelljen elindítani az előre összeállított műveletsort, hanem hozzá lehessen kötni azt egy megadott eseményhez. Vagyis a DBMS figyeli az egyes események bekövetkezését, és ha szükséges ő maga automatikusan elindítja a műveletsort. Így nem kell félni, hogy egyes alkalmazások esetleg elfelejtik végrehajtani az igényelt csatlakozó műveleteket. A mechanizmust megvalósító adatbázis objektumot nevezik triggernek. A trigger létrehozása:

CREATE TRIGGER tnev esemény választevékenység;

ahol *esemény* a kiváltó esemény és a *választevékenység* a meghívandó, csatlakozó műveletsor. Az adatbázis kezelő rendszerekben több különböző típusú triggert értelmezhetnek. Egyik speciális fajtája az úgynevezett helyettesítő trigger, amikor a kiváltó művelet egy SELECT utasítás, és a kiváltott esemény egy másik SELECT utasítás, amely tehát az eredeti lekérdezés helyett fog lefutni. Így tehát a háttérben lehetnek olyan szabályok, melyek hatására az eredetileg kiadott lekérdezés helyett egy másik lekérdezés fog lefutni a DBMS-ben.

Munkaköteg

Míg a triggernél a kijelölt műveletsor valamely esemény hatására indul el, addig a munkakötegnél (job) az eseménysor indítását egy időponthoz köthetjük. Így a munkaköteg nem más, mint egy műveletsor és egy időzítési paraméter. A DBMS a háttérben egy listában összegyűjti milyen munkakötegek élnek és mikor kell indítani őket. Ha a kijelölt időpont elérkezik, akkor a megadott műveletsor aktivizálódik, lefut. Ennek során meghatározásra kerül a következő futtatás időpontja is, és ezzel az új bejegyzéssel újra bekerül a munkaköteg a várakozó listába. A



8.13. ábra. Trigger

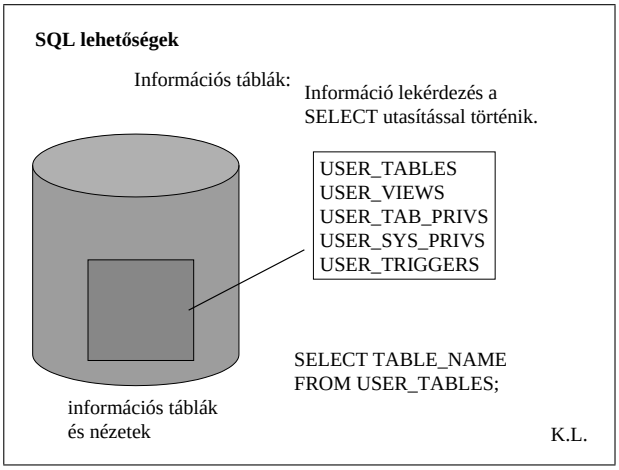
munkaköteg létrehozása DBMS függő, például az Oracle esetében tárolt eljárással lehet új munkaköteget definiálni.

Információs táblák

A korábbiakban igen részletesen áttekintettük az adatbázisban a táblák létrehozását, az adatok kezelését illetve lekérdezését. Ezen SQL műveletek során az egyik állandó paraméter az érintett tábla neve volt. E paraméter nélkül nem tudunk megfelelő SQL műveletet elindítani. Ezen elnevezések és paraméterek ismerete nélkül nem tudunk dolgozni az adatbázisban. Tehát úgy néz ki, hogy minden ilyen jellegű információt fejben kell tartanunk, vagy legalábbis egy biztos helyre le kell írnunk, hogy szükség esetén újra ismertek legyenek számunkra ezen adatok. Ez azonban nem tűnik ésszerű dolognak, hiszen ott az adatbázis, és abban minden ilyen jellegű információ benne van valamely információs táblában a metaadatok között.

Valóban van ilyen megoldás, melyben nem kell minden séma leíró adatot fejben tartani, elég csak azt tudni, hogy hogyan és honnan lehet az igényelt információkat az adatbázisból előszedni. Az általános elv az, hogy a leíró adatok is táblákban tárolódnak, ezért lekérdezésük a már ismert SELECT alkalmazásával lehetséges, és nincs külön erre a célra szolgáló SQL utasítás. Sajnos ezen jó hír mellett van egy rosszabb hír is, nevezetesen az, hogy az információkat leíró rendszertáblák neve és szerkezete nem szabványosított, RDBMS függő, ezért itt nincs más lehetőség, meg kell jegyezni a használt RDBMS-ben alkalmazott rendszertáblák nevét.

Példaként ismét az Oracle RDBMS-t vesszük, és megnézünk néhány, hasznos információt tartalmazó rendszertáblát. A táblanév ismeretében a szerkezetüket már le tudjuk kérdezni.



8.14. ábra. Információs táblák

USER_ TABLES:	a felhasználó által birtokolt adattáblák séma adatait tartalmazó tábla.
USER_ VIEWS:	a felhasználó által birtokolt nézeti táblák adatai.
USER_ TRIGGERS:	a felhasználó által definiált triggerek adatai.
USER_ OBJECTS:	a felhasználó által létrehozott összes adatbázis objektum adatai.
USER_ TAB_ PRIVS:	a felhasználóhoz kapcsolódó tábla jogosultságok.
USER_ SYS_ PRIVS:	a felhasználóhoz kapcsolódó rendszer jogok.

A fenti táblák alapján, ha például arra vagyunk kíváncsiak milyen tábláink vannak, akkor a *SELECT * FROM USER_ TABLES;* parancsot kell kiadnunk.

8.8. Codd szabályai

A következő részben a relációs adatmodell megalkotójának nevéhez fűződő szabályrendszert ismertetjük. Nos, mire is vonatkozik ez a szabályrendszer. E szabályok célja, hogy pontos kritériumokat fogalmazzanak meg arra vonatkozóan, mikor tekinthető egy adatbáziskezelő rendszer relációs adatbázis kezelőnek, és mikor nem.

A fenti szabályok megalkotásának indítéka az volt, hogy a 80-as években igen hamar és széles körben elterjedt a relációs adatmodell fogalma, ezáltal igen népszerű lett ez az adatmodell, amit sokan igyekeztek nem igazán tisztességes módon ki is használni, és visszaélni az elnevezéshez fűződő mítosszal. Ebben az időben a nagyobb piaci haszon reményében, sok DBMS fejlesztő saját termékét relációs adatbázis kelezőként tűntette fel, holott a termék adatkezelési motorja távolról sem úgy működött, mint azt egy adatbázistól elvárnánk. Több olyan esettel is találkozhattunk abban az időben, amikor a normál állománykezelést (egy állomány esetén is) relációs adatbázisnak nevezték. Természetesen ezek a rendszerek nem biztosították az elvárt szolgáltatásokat és előnyöket. Ezen esetek az igazi relációs adatbáziskezelés területére sem vetettek jó fényt, ezért mintegy védekezésként, az egyértelműség végett Codd pontokba összefoglalta, mikor tekinthetünk egy adatbázist relációs adatbáziskezelőnek.

A megadott pontok között van fontosabb és kevésbé fontos kritérium. A szabályok, mint látni fogjuk, nem egészen homogének, de ez nem vesz el semmit fő érdemükből, hogy megkísérlik megadni a relációs adatbáziskezelők alap kritériumait. E szabályrendszer másik fő erénye, hogy biztosítani tudja a termék, a gyártó függetlenséget, ami sajnos nem mondható el több későbbi hasonló jellegű munkáról. Így a szabályrendszer nem preferál egyetlen egy gyártót sem, nincs benne termék specifikus szolgáltatás vagy funkció kiemelve. A következőkben egyenként áttekintjük a Codd-féle szabályrendszer kritériumait.

1. Az adatbáziskezelésnek relációs alapokon kell nyugodnia.

   > Ez a követelmény alapkövetelménynek tekinthető, mely kimondja, hogy az adatok relációkban tárolódjanak, a kapcsolatok az idegen kulcs és kulcs mezők értékeinek azonosságán alapuljanak, és az

adatok kezelése a relációs algebrán alapuljon. Ezzel a kritériummal elveti a korábbi adatmodellekre épülő rendszerek és a hagyományos állománykezelésen alapuló adatrendszereket is. A későbbi pontok a relációs adatbáziskezelés legfontosabb, az akkori időben leginkább újszerű tulajdonságait emelik ki.

2. Minden adat táblákban tárolt, beleértve a metaadatokat is.

   A modell egyik érdekes vonása, hogy nincs külön tárolási struktúra a normál adatok és a szerkezetleíró adatok részére. A korábbi modellekben a séma leíró adatok egészen más struktúrában tárolódtak, a felhasználók elől rejtve. A relációs modellben homogénné vált a különböző funkciójú adatok kezelése, logikailag egyszerűbbé téve az adatok tárolását, a leíró és normál adatok közötti átmenetet.

3. Minden adatelem eléréséhez elegendő a táblanév, a rekord kulcsérték és a mezőnév megadása.

   A fenti tétel azt mondja ki, hogy a relációs modellben az adatok legkisebb elérhető egysége a mező, mely a tábla, a rekord és a mezőnév hármassal egyértelműen beazonosítható. A mezőn belül további strukturális egységet nem értelmez a relációs adatbáziskezelő rendszer. A kritérium egy másik következménye, hogy az adatbázisban adat nem létezhet csak táblában egy mezőbe letárolva. Adat nem lehet táblán kívül. Ez a követelmény tehát megint az adatok egységes kezelése irányába mutat.

4. A rendszer támogatja a NULL érték kezelését.

   Az adatbáziskezelő rendszerekben kívánatos, hogy a felhasználók mindig a helyes adatértéket lássák, ezért biztosítani kell, hogy a felhasználó és a kezelő rendszer a még értéket nem kapott adatelemeket is felismerje, és az üres adatelemeket ne vegye figyelembe az aggregációs számításoknál. Azaz az adatelemből kapott értéket ne tekintse értékes, valós értéknek. Az üres állapot nyilvántartására a rendszer minden mező mellett egy állapot jelzőt is letárol.

5. A meta-adatokat is ugyanúgy kezelhetjük, mint a normál adatokat.

   Mivel minden adat, függetlenül attól, hogy normál adat vagy metaadat egységesen tárolódik a táblázat, rekord és mező struktúraegységekben. Az egységes tárolási formátum lényeges következménye, hogy a kétféle adattípus adatainak módosítása és lekérdezése egységes módon történhet, hiszen mindkettő azonos alakban kerül elhelyezésre az adatbázisban. Így a felhasználó ugyanazt a parancsnyelvet használhatja, függetlenül attól, hogy normál adatokat vagy séma leíró információkat kíván elérni. A normál felhasználó és az adatbázis adminisztrátor is egyazon parancsnyelvet használja a relációs adatbázis kezelésben.

6. Létezik hozzá parancsorientált relációs adatkezelő nyelv.

   Ez a követelmény azt a lényeges újítást emeli ki a korábbi adatmodellekhez viszonyítva, hogy az adatlekérdezés, adatkezelés vagy séma karbantartás minden feladata megoldható egy szöveges, magas szintű parancsnyelv segítségével. Így például egy adat lekérdezési feladathoz nem szükséges többé programot írni, a lekérdezés elvégezhető a megfelelő parancs kiadásával. Ezáltal sokkal áttekinthetőbben lehet megfogalmazni összetett lekérdezési műveleteket is. A relációs modellen alapuló parancsnyelv szabványává az SQL nyelv vált.

7. A rendszer biztosítja a nézeti táblák (VIEW) alkalmazását.

   Az alaptáblák mellett célszerű a származtatott táblák alkalmazása is, hiszen ezáltal összetettebb lekérdezés esetén nem szükséges a komplex SQL műveletsort megadni és elküldeni, ehelyett elég egy származtatott táblát létrehozni a megadott művelethez, és ettől kezdve elég csak hivatkozni erre a nézeti táblára, a rendszer automatikusan végrehajtja a mögötte álló műveletsort. A VIEW alkalmazása megnöveli a rugalmasságot, áttekinthetőséget.

8. Az elméletileg módosítható VIEW-k a gyakorlatban is módosíthatók.

   Igen érdekes kérdés a nézeti táblák módosíthatóságának kérdése. Az egyik fő irányelv a VIEW alkalmazásánál, hogy a felhasználó lehetőleg ne érezzen különbséget a VIEW és a normál tábla használata között. Ennek egyik eleme a módosíthatóság kérdése. Mivel a normál táblák módosíthatók, ezért biztosítani kellene a VIEW-k módosíthatóságát is. Azonban ez már nem olyan egyszerű feladat, hiszen sok VIEW csak virtuálisan létezik, ezért a módosításokat az alaptáblákra kell visszavezetni. A visszavezetés azonban nem mindig egyértelmű feladat. Ha például aggregációt tartalmaz a VIEW, akkor egy összesített érték módosítása nem vezethető vissza egyértelműen az alapértékek módosítására. Ezért nem mindegyik VIEW módosítható, csak azok, melyekben a VIEW egy táblán alapszik és nem tartalmaz a származtatás aggregációs lépéseket.

9. A rendszer biztosítja az integritási feltételek megvalósítását.

   Az adatok helyességének biztosításánál fontos feladat, hogy az integritási szabályokat az adatbázisban le lehessen tárolni, és azok ellenőrzését az RDBMS minden hozzáférés esetén ellenőrizze, tehát ne lehessen e szabályokat kikerülni.

10. Fizikai függetlenségi szint biztosítása.

    A fizikai függetlenség azt biztosítja, hogy megváltoztatható az adatbázis fizikai tárolási szerkezete anélkül, hogy módosítani kellene az adatok relációs modellbeli szerkezetét vagy kezelő utasításait. Így egy alkalmazásban továbbra is használhatjuk ugyanazt az

SQL utasítást, függetlenül attól, hogy megváltozott az adattáblák indexelésének mechanizmusa.

11. Logikai függetlenségi szint biztosítása.

    A logikai függetlenség arra szolgál, hogy a táblákban módosíthassuk a sémát, és ne kelljen ennek hatására minden táblakezelő programot újraírni. A logikai függetlenség az által biztosított, hogy az alkalmazások mező szinten érhetik el az adatokat, így a lekérdezés, elérés módja független a séma többi mezőjétől.

12. Hálózati függetlenségi szint biztosítása.

    A hálózati függetlenség igénye az osztott adatbázisok esetében jelenik meg, és arra szolgál, hogy ne kelljen ismerni az adatbázis egyes komponenseinek címét, az egyes adatelemek fizikai címét. A hálózati függetlenség eredményeként a felhasználó egy nagy és logikailag egységes adatbázist lát, melynek elemeit úgy érheti el, mintha egy lokális adatbázissal dolgozna. Eközben természetesen az egyes elemek más és más csomóponton, más és más fizikai címen helyezkedhetnek el, mely címek menetközben változhatnak is. Ezért fontos annyira, hogy a felhasználónak ne kelljen az aktuális címet ismernie, maga a rendszer veszi ki az aktuális elérési címet egy nyilvántartási listából.

13. Nincs olyan kezelő felület, mely kikerülné a relációs DBMS modult.

    Ez a pont arra szolgál, hogy az adatkezelést csak a relációs felületen keresztül lehessen elérni, biztosítva azt, hogy ne tudjon senki ellenőrzés nélküli adatkezelést végezni, hiszen akkor inkonzisztens állapotba kerülhet az adatbázis, ezáltal semmissé válhat az RDBMS minden ellenőrző funkciója is.

A fenti kritériumok együttesen a relációs adatbáziskezelők legfontosabb tulajdonságait foglalják össze, és sokáig e kritériumok vezették a fejlesztőket az RDBMS rendszerek kidolgozása során. A mai vezető RDBMS rendszerek mindegyike teljesíti a felsorolt kritériumokat.

Elméleti kérdések

1. Adja meg a TABLE és VIEW objektumok hasonlóságát és különbözőségét.
2. Mi a SNAPSHOT objektum és mikor célszerű használni?
3. Melyik táblatípusnál lehet megadni frissítési paramétereket?
4. Lehet-e SNAPSHOT-ra építeni VIEW-t?
5. Lehet-e VIEW-t VIEW-ra építeni?
6. Ismertesse az ideiglenes táblák létrehozását és előnyeit.
7. Mutassa be a TABLE, VIEW és a SNAPSHOT fogalmát és létrehozásukat SQL-ben.
8. Mikor szűnik meg egy *CREATE GLOBAL TEMPORARY TABLE* paranccsal létrehozott tábla?
9. Mit jelentenek a DELETE ROWS és PRESERVE ROWS kapcsolók az ideiglenes tábláknál? Melyik megvalósítása igényel több DBMS erőforrást?
10. Melyik objektumegység fogja össze egy felhasználó különböző objektumait?
11. Hogyan lehet séma objektumot létrehozni és a benne tárolt táblákra hivatkozni?
12. A SELECT utasítás mely részében szerepelhet a CASE szerkezet és hogyan működik?
13. Mire szolgál az ASSERTION objektum?
14. Mely parancsokban adhatunk meg értékellenőrzésre szolgáló integritási parancsokat?
15. Miért kell azonosító nevet megadni az ASSERTION utasításnál? Mely parancsokban használható fel ez a név?
16. Mit jelent a késleltetett integritás ellenőrzési mód, és hogyan adható meg?
17. Miért lehet szükséges a DEFERRED kapcsoló az integritási objektumoknál?
18. Hol szerepelhet al-SELECT a DML utasításokban?
19. Mit jelent a NULL szimbólum?
20. Milyen operátorok és függvények vannak a NULL érték észlelésére?
21. Miért volt szükség a 3VL bevezetésére az SQL nyelvben?
22. Ismertesse a 3VL-hez kapcsolódó logikai műveleti táblákat.
23. Mit jelent a harmadik kizárt elve?
24. Lehet-e adni olyan lekérdezési igényt, melyet nem lehet SQL-ben még elviekben sem megvalósítani?
25. Mit jelent a hierarchikus SELECT és hogyan működik?
26. Ismertesse a hierarchikus SELECT-et megvalósító SQL utasítást.
27. Lehet a hierarchikus SELECT-tel a hivatkozási fában felfelé is haladni?
28. Hogyan lehet a hierarchikus SELECT-nél a feltárt mélységet szabályozni?
29. Mutassa be az indexek típusait és létrehozásuk módját.
30. Mikor jönnek létre implicit módon is indexek?
31. Mire szolgál a SEQUENCE objektum?
32. Miért előnyös az adatbázisban is tárolni egyes eljárásokat?
33. Milyen módszerekkel lehet numerikus kulcsmezőt egyedi értékkel feltölteni? Adja meg a szükséges SQL utasításokat is.
34. Mi történik, ha a szekvencia-változó eléri a megadott maximális értéket?
35. Mely SQL parancsokban lehet felhasználni a szekvencia objektumokat?
36. Hogyan szabályozható a bejárási irány a hierarchikus SELECT-nél?

37. Mire szolgál a trigger objektum?
38. Hogy működik a job objektum?
39. Mi a célja Codd szabályainak?
40. Mely Codd-szabályok vonatkoznak az adatbázis struktúrára?
41. Ismertesse a műveletekre vonatkozó Codd-szabályokat.
42. Milyen követelményeket támaszt Codd a VIEW kezelésre vonatkozólag?
43. Milyen függetlenségi szinteket igényelnek Codd szabályai?

Feladatok

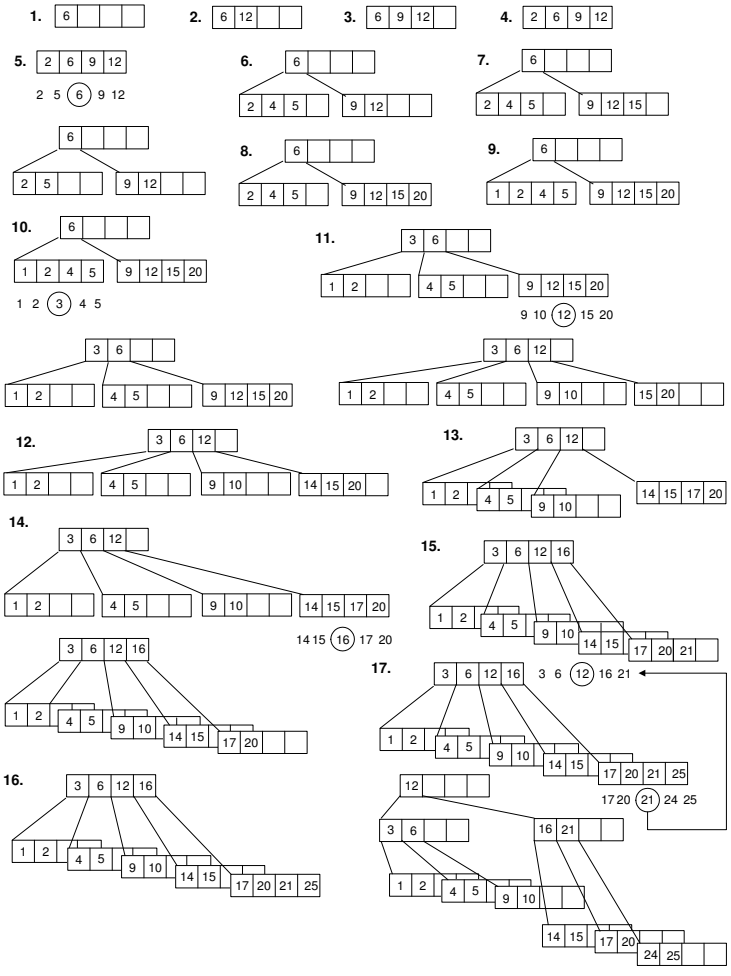
1. Adott az ÜGYNÖK[KÓD, BESZERVEZŐ, NÉV, VÁROS] tábla, amelyben a *kód* a kulcs és a *beszervező* mező idegen kulcs az ÜGYNÖK táblára. Egy ügynök minden olyan más ügynöktől kap részesedést, akiket ő vagy az általa beszervezett más ügynök szervezett be (maximum 3 szintig lemenve). Hogyan lehet meghatározni, hogy kiktől kap részesedést az XX kódú ügynök? Adja meg a műveletet hierarchikus SELECT-tel és normál SELECT utasítással.

*2. A DOLGOZÓ[KÓD, NÉV, BEOSZTÁS, FIZETÉS, FŐNÖK] táblából kérdezze le az 1111 kódú vezető 100000 Ft-nál kevesebbet kereső beosztottjait 2 szintig lemenve.

*3. Hozzon létre olyan származtatott táblát, amely kétnaponta frissül és az ingatlannal nem rendelkezők neveit és lakcímét tartalmazza (SQL utasítás). A felhasználandó táblák: SZEMÉLY[KÓD, NÉV, CÍM] és INGATLAN[AZON, CÍM, TULAJ].

Feladat megoldások

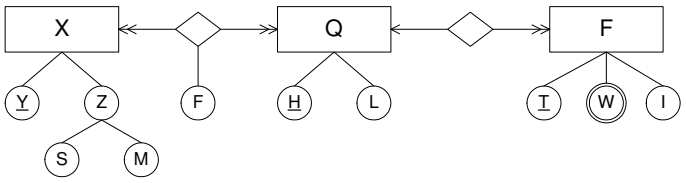
1.3. Építsen fel egy B-fát az alábbi elemekből, melyek beépülési sorrendje adott. A fa fokszáma 4, és a beszúrandó elemek listája: 6, 12, 9, 2, 5, 4, 15, 20, 1, 3, 10, 14, 17, 16, 21, 25, 24.

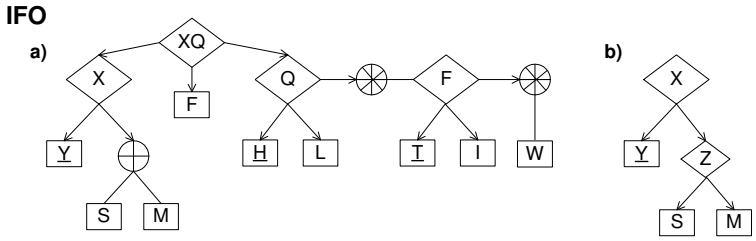


1.4. Építsen fel egy alap hash táblát az alábbi elemekből, melyek beépülési sorrendje adott. A hash függvény: x mod 7, egy bucket kapacitása 3 rekord, és az elemek listája: 45,2,34,1,67,21,26,54,12,43,28,32.

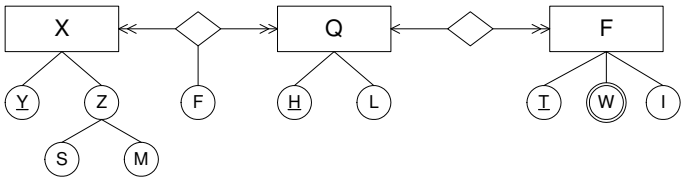
h(x) = x mod 7	Hash tábla		
0	21	28	
1	1	43	
2	2		
3	45		
4	67	32	
5	26	54	12
6	34		

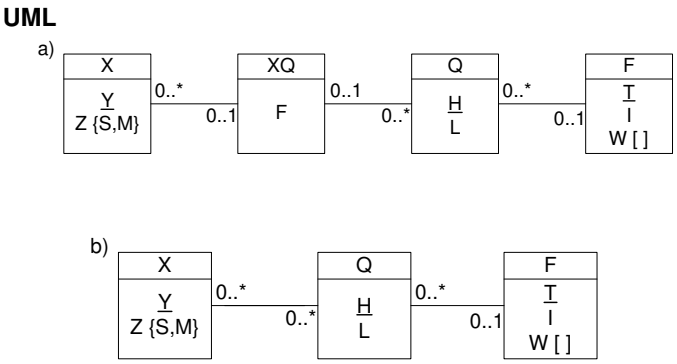
2.2. Konvertálja az alábbi ER modellt IFO modellre:





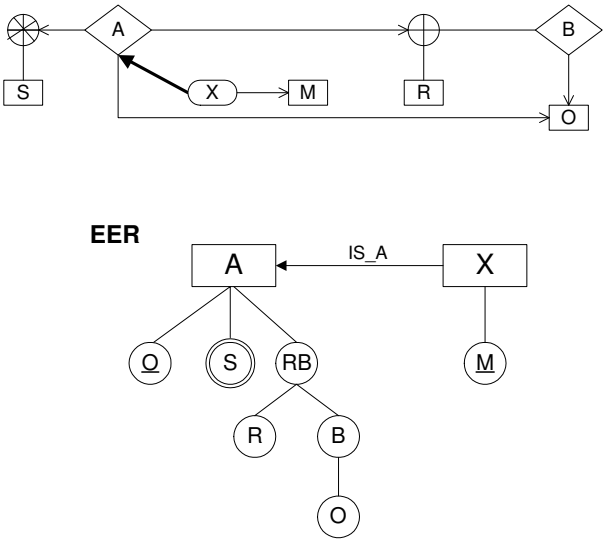
2.6. Konvertálja az alábbi ER modellt UML modellre:





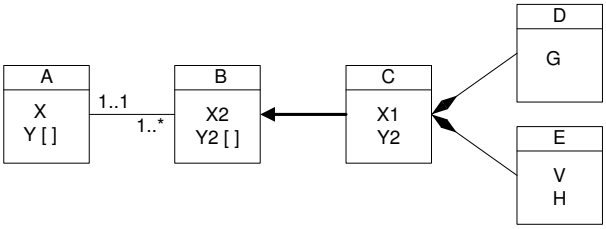
A b) esetben a kapcsolathoz rendelt F tulajdonságot nem tudjuk ábrázolni.

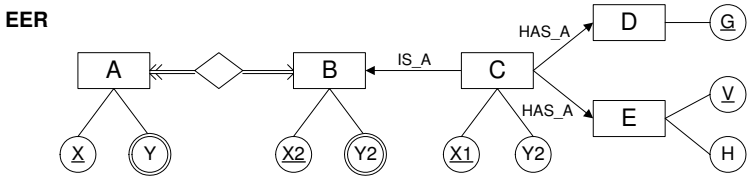
2.8. Konvertálja át a következő IFO sémát EER sémára:



A kulcs kijelölése tetszőleges.

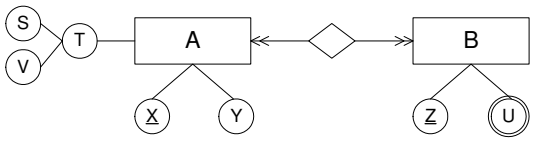
2.14. Konvertálja át a következő UML sémát EER sémára:

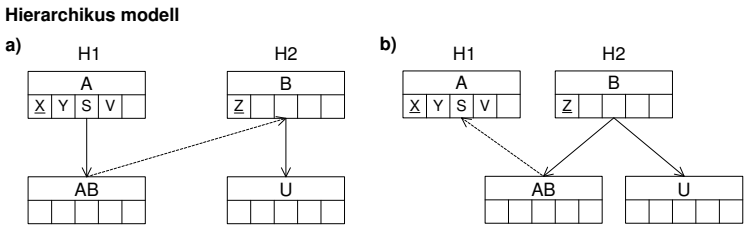




A kulcs kijelölése tetszőleges.

3.9. Konvertálja az alábbi ER sémát hierarchikus modellre, és adja meg a létrehozó HDDL utasításokat.





a)

SCHEMA NAME = FELADAT
HIERARCHIES = H1, H2
RECORD
NAME = A
TYPE = ROOT OF H1
DATA ITEMS =
X INTEGER
Y CHAR 15
S CHAR 20
V INTEGER
KEY = X
ORDER BY Y

RECORD
NAME = B
TYPE = ROOT OF H2
DATA ITEMS =
Z INTEGER
KEY = Z
ORDER BY Z

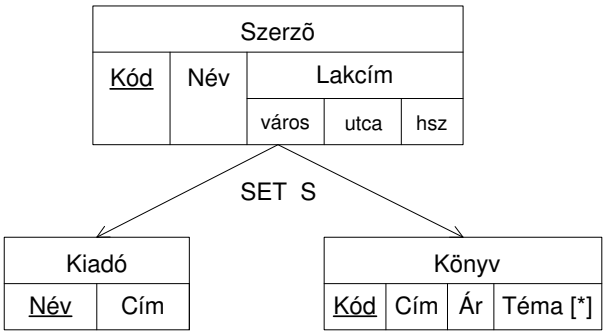
RECORD
NAME = AB
PARENT = A
CHILD NUMBER = 1
DATA ITEMS =
PB POINTER TO
VIRTUAL PARENT B

RECORD
NAME = U
PARENT = B
CHILD NUMBER = 1
DATA ITEMS =
...

b)
```
SCHEMA NAME = FELADAT
HIERARCHIES = H1, H2
RECORD
    NAME = A
    TYPE = ROOT OF H1
    DATA ITEMS =
       X   INTEGER
       Y   CHAR 15
       S   CHAR 20
       V   INTEGER
    KEY = X
    ORDER BY Y

RECORD
    NAME = B
    TYPE = ROOT OF H2
    DATA ITEMS =
       Z   INTEGER
    KEY = Z
    ORDER BY Z

RECORD
    NAME = AB
    PARENT = B
    CHILD NUMBER = 1
    DATA ITEMS =
       PA POINTER TO
       VIRTUAL PARENT A

RECORD
    NAME = U
    PARENT = B
    CHILD NUMBER = 2
    DATA ITEMS =
       ...
```

3.14. Adja meg a sémát megvalósító NDDL utasításokat az alábbi sémához:

Rekord: Szerző (név, kód, lakcím(város, utca, hsz))
Kiadó (név, cím)
Könyv(cím, kód, ár, téma(kulcsszavak*))
Set: S (tulajdonos: Szerző; tag: Kiadó, Könyv)

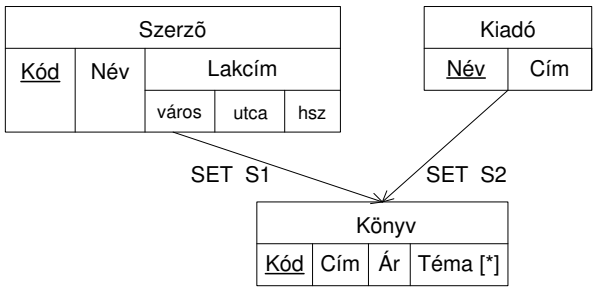



```
SCHEMA NAME IS FELADAT

RECORD NAME IS Szerző
    01 kód        N(4)
    01 név        C(20)
    01 lakcím
       02 város   C(15)
       02 utca    C(15)
       02 hsz     C(4)
LOCATION MODE CALC
USING kód
```

```
RECORD NAME IS Kiadó
    01 név        C(10)
    01 cím        C(50)
LOCATION MODE CALC
USING név

RECORD NAME IS Könyv
    01 kód        N(10)
    01 cím        C(30)
    01 ár         N(5)
    01 téma       C(15)
    OCCUR 0 TO 50 TIMES
LOCATION MODE CALC
USING kód

SET NAME IS S
OWNER IS Szerző
MEMBER IS Kiadó, Könyv
ORDER IS SYSTEM DEFAULT
```

3.17. Kérdezze le a miskolci kiadóknál megjelent könyvek szerzőinek a nevét az adott séma mellett:

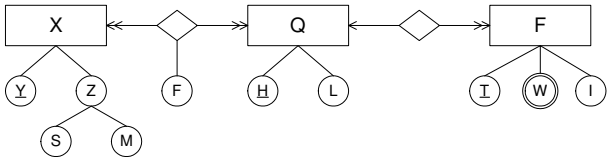
Rekord: Szerző (név, kód, lakcím(város, utca, hsz))
Kiadó (név, cím)
Könyv(cím, kód, ár, téma(kulcsszavak*))
Set: S1 (tulajdonos: Szerző; tag: Könyv)
S2 (tulajdonos: Kiadó; tag: Könyv)



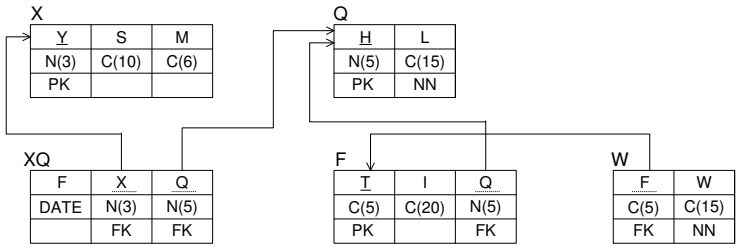

```
p[1] cim='Miskolc' (kiadó)
while (DB_STATUS == 0) {
   m[1](S2, könyv)
   while (DB_STATUS == 0) {
         o(S1, könyv)
         printf("%s \n", név)
         m[n](S2, könyv)
   }
   p[n] cim='Miskolc' (kiadó)
}

NEXEC FIND FIRST kiadó
   USING cim='Miskolc'
while (DB_STATUS == 0) {
```

```
   NEXEC FIND FIRST könyv
         IN CURRENT S2 SET
   while (DB_STATUS == 0) {
         NEXEC FIND OWNER OF
           könyv IN CURRENT
           S1 SET
         printf("%s \n", név)
         NEXEC FIND NEXT
           könyv IN CURRENT
           S2 SET
   }
   NEXEC FIND NEXT kiadó
         USING cim='Miskolc'
}
```

4.1. Hozza létre a relációs modellt az alábbi ER sémához:

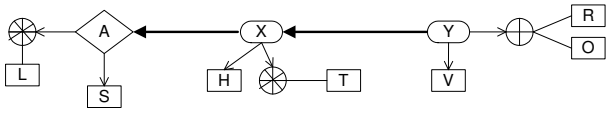


Relációs modell

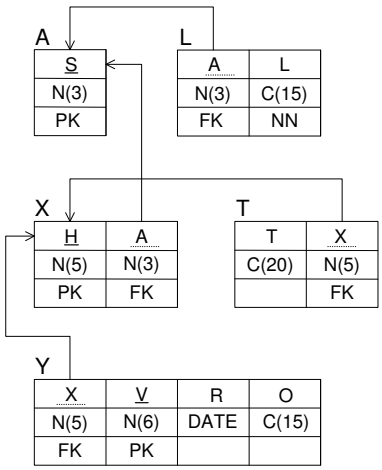


Az adattípusok megadása tetszőleges.

4.7. Konvertálja át az alábbi IFO sémát relációs adatmodellre:



Relációs modell



Az adattípusok megadása tetszőleges.

4.14. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

- Könyvek címe és szerző neve.
  $\pi_{konyv.cim,szerzo.nev}$ (KÖNYV $\bowtie_{konyv.szerzo=szerzo.kod}$ SZERZŐ)
- Mely szerzőknek nincs könyve.
  $\pi_{szerzo.nev}$ (SZERZŐ) \
  $\pi_{szerzo.nev}$ (KÖNYV $\bowtie_{konyv.szerzo=szerzo.kod}$ SZERZŐ)
- Az UNIVERSUM kiadónál megjelent könyvek átlagára.
  $\Gamma^{AVG(konyv.ar)}$ (KÖNYV $\bowtie_{konyv.kiado=kiado.kod}$
  ($\sigma_{kiado.nev=Universum}$ (KIADÓ)))
- Azon szerzők neve, akiknek 5-nél több könyve van.
  $\pi_{szerzo.nev}$ ($\sigma_{db>5}$ ($\Gamma_{nev}^{szerzo.nev,\ COUNT(konyv.cim)\ db}$
  (KÖNYV $\bowtie_{konyv.szerzo=szerzo.kod}$ SZERZŐ)))

4.15. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM, KOR].

- A kiadók neve és könyveik átlagára.
  $\Gamma_{nev}^{kiado.nev,\ AVG(konyv.ar)}$
  (KÖNYV $\bowtie_{konyv.kiado=kiado.kod}$ KIADÓ)
- Az átlagnál fiatalabb szerzők könyvei.
  $\pi_{konyv.cim}$ (KÖNYV $\bowtie_{konyv.szerzo=szerzo.kod}$
  ($\sigma_{szerzo.kor<\ \Gamma^{AVG(szerzo.kor)}\ (SZERZO)}$ (SZERZŐ)))
- Mely szerző adott ki minden kiadónál.
  $\pi_{szerzo.nev}$ (($\pi_{konyv.szerzo,konyv.kiado}$ (KÖNYV) $\div$
  $\pi_{kiado.kod}$ (KIADÓ)) $\bowtie_{szerzo=szerzo.kod}$ SZERZŐ)

4.16. Adja meg a relációs algebrai kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

- Mely szerzők nem adtak ki 2000 forintnál olcsóbb könyvet.
  $\pi_{szerzo.nev}$ (SZERZŐ) \
  $\pi_{szerzo.nev}$ (($\sigma_{konyv.ar<2000}$ (KÖNYV)) $\bowtie_{konyv.szerzo=szerzo.kod}$
  SZERZŐ)
- A szerzők neve és könyveik darabszáma (akinek nincs ott 0 álljon).
  $\Gamma_{nev}^{szerzo.nev,\ COUNT(konyv.cim)\ db}$
  (SZERZŐ $+\bowtie_{konyv.szerzo=szerzo.kod}$ KÖNYV)
- A legtöbb könyvet írt szerző neve.
  X = $\Gamma_{nev}^{szerzo.nev,\ COUNT(konyv.cim)\ db}$
  (SZERZŐ $\bowtie_{konyv.szerzo=szerzo.kod}$ KÖNYV)

Megoldás:
$\pi_{\ szerzo.nev}\ (\sigma_{\ db=\ \Gamma^{MAX(db)}\ (X)}\ (\mathrm{X}))$

4.20. Adottak az alábbi relációk: **VERSENYZŐ** [VKÓD, NÉV, KOR, CSAPAT] és **CSAPAT** [ID, NÉV, CÍM]. Adja meg a megfelelő relációs algebrai kifejezést:

– Versenyzők neve és csapatuk neve.
$\pi_{\ versenyzo.nev,csapat.nev}$ (VERSENYZŐ $\bowtie_{\ versenyzo.csapat=csapat.id}$ CSAPAT)

– A nem a SASOK csapatban játszó játékosok átlagéletkora.
$\Gamma^{\ AVG(versenyzo.kor)}$ ((CSAPAT $\setminus$ $\sigma_{\ csapat.nev=SASOK}$ (CSAPAT)) $\bowtie_{\ versenyzo.csapat=csapat.id}$ VERSENYZŐ)

– Azon játékosok, akik idősebbek csapatuk átlagéletkoránál.
$\pi_{\ versenyzo1.nev}\ (\sigma_{\ versenyzo1.kor>\ \Gamma^{AVG(versenyzo2.kor)}}$
$(\sigma_{versenyzo1.csapat=versenyzo2.csapat}\ (VERSENYZO2))$ (VERSENYZŐ1))

4.25. Adja meg a tuple relációs kalkulus kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

– Könyvek címe, kiadó neve és a szerző neve.
{k.cím, ki.név, sz.név | könyv(k) ∧ kiadó(ki) ∧ szerző(sz) ∧ k.szerző=sz.kód ∧ k.kiadó=ki.kód}

– A 2000 forintnál olcsóbb könyvek szerzőinek neve.
{sz.név | szerző(sz) ∧ ∃ k (könyv(k) ∧ k.szerző=sz.kód ∧ k.ár<2000)}

– A szerzők és könyveik darabszáma.
{sz.név, COUNT(k) | szerző(sz) ∧ ∃ k (könyv(k) ∧ k.szerző=sz.kód)}

4.26. Adja meg a domain relációs kalkulus kifejezéseket az alábbi műveletekhez, ha a példához tartozó sémák: **KÖNYV** [ISBN, CÍM, SZERZŐ, ÁR, KIADÓ], **KIADÓ** [KÓD, NÉV] és **SZERZŐ** [KÓD, NÉV, LAKCÍM].

– Könyvek címe, kiadó neve és a szerző neve.
{c, kn, szn | ∃ i, szk, a, kk, l (könyv(i,c,szk,a,kk) ∧ kiadó(kk,kn) ∧ szerző(szk,szn,l))}

– A 2000 forintnál olcsóbb könyvek szerzőinek neve.
{szn | ∃ szk, l, i, c, a, kk (szerző(szk,szn,l) ∧ könyv(i,c,szk,a,kk) ∧ a<2000)}

– A szerzők és könyveik darabszáma.
{szn, COUNT(c) | ∃ szk,l,i,a,kk (szerző(szk,szn,l) ∧ könyv(i,c,szk,a,kk))}

5.1. Adott **DOLGOZÓ** [KÓD, NÉV, BEOSZTÁS, FIZETÉS] és **PROJEKT** [NÉV, VÁROS, DOLGOZÓ] sémához adja meg az alábbi műveletek SQL megfelelőjét:

- PROJEKT tábla létrehozása.
  CREATE TABLE projekt (név CHAR(20) PRIMARY KEY, város CHAR(15) NOT NULL, dolgozó REFERENCES dolgozó);
- Kiss nevű dolgozók törlése.
  DELETE FROM dolgozó WHERE név LIKE 'Kiss%';
- Operátor beosztású dolgozók fizetésének növelése 10%-al.
  *UPDATE dolgozó SET fizetés=fizetés*1.1 WHERE beosztás='Operátor';*
- Operátor beosztású dolgozók neve ABC sorrendben.
  SELECT név FROM dolgozó WHERE beosztás='Operátor' ORDER BY név;
- 120000 forintnál többet keresők darabszáma.
  SELECT COUNT(*) FROM dolgozó WHERE fizetés > 120000;
- Miskolc városhoz tartozó projektek és azon résztvevő dolgozók neve.
  SELECT p.név, d.név FROM projekt p, dolgozó d WHERE p.dolgozó=d.kód AND p.város='Miskolc';
- Mely beosztásokban nagyobb az átlagfizetés 100000 forintnál.
  SELECT beosztás FROM dolgozó GROUP BY beosztás HAVING AVG(fizetés) > 100000;
- Mennyi a projektekben nem dolgozók összfizetése.
  a) *SELECT SUM(fizetés) FROM dolgozó WHERE kód NOT IN (SELECT dolgozó FROM projekt);*
  b) *(SELECT SUM(fizetés) FROM dolgozó) MINUS (SELECT SUM(fizetés) FROM dolgozó WHERE kód IN(SELECT dolgozó FROM projekt));*
  c) *(SELECT SUM(fizetés) FROM dolgozó) MINUS (SELECT SUM(fizetés) FROM dolgozó d, projekt p WHERE d.kód=p.dolgozó);*

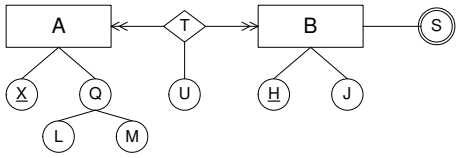
5.5. Adottak az alábbi SQL adattáblák: **CSAPAT** [CSAPATNÉV, CSAPATKÓD, PONTSZÁM] és **JÁTÉKOS** [NÉV, CSAPATKÓD, KOR]. Végezze el az alábbi SQL utasításokat:

- Szüntesse meg a JÁTÉKOS táblát.
  DROP TABLE játékos;
- Növelje a K-val kezdődő játékosok életkorát eggyel.
  UPDATE játékos SET kor=kor+1 WHERE név LIKE 'K%';
- Írassa ki mely csapatban hány játékos játszik (ahol nincs ott 0).
  SELECT csapatnév, COUNT(*) FROM csapat cs LEFT OUTER JOIN játékos j ON cs.csapatkód=j.csapatkód GROUP BY csapatnév;

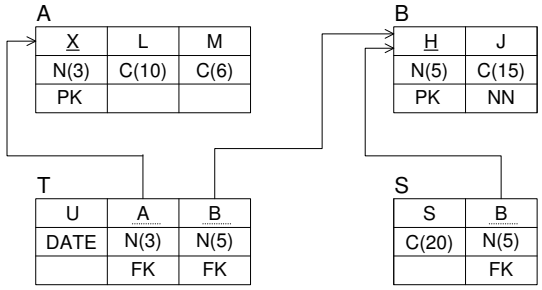
– Listázza ki azon csapatokat, ahol nincs 20 évnél idősebb játékos.
a) *SELECT DISTINCT csapatnév FROM csapat WHERE csapatkód NOT IN (SELECT csapatkód FROM játékos WHERE kor>20);*
b) *(SELECT csapatnév FROM csapat) MINUS (SELECT DISTINCT csapatnév FROM csapat WHERE csapatkód IN (SELECT csapatkód FROM játékos WHERE kor>20));*

– Azok a játékosok, akik idősebbek csapatuk átlagéletkoránál.
SELECT név FROM játékos j1 WHERE kor > (SELECT AVG(kor) FROM játékos j2 WHERE j1.csapatkód=j2.csapatkód);

5.7. Hozza létre a relációkat, táblákat SQL paranccsal az alábbi ER sémához:



Relációs modell



CREATE TABLE a (x NUMBER(3) PRIMARY KEY, l CHAR(10), m CHAR(6));
CREATE TABLE b (h NUMBER(5) PRIMARY KEY, j CHAR(15) NOT NULL);
CREATE TABLE t (u DATE, a REFERENCES a, b REFERENCES b);
CREATE TABLE s (s CHAR(20), b REFERENCES b);

5.9. Adottak az alábbi sémák: **OKTATÓ** [ID, NÉV, FOKOZAT] és **TÁRGY** [TID, CÍM, OKTATÓ, ÓRA]. Adja meg a megfelelő SQL utasításokat:

– Vigyen fel egy új tárgyat.
a) *INSERT INTO tárgy VALUES('IAL404','Adatbázisok','idd4q1',4);*
b) *INSERT INTO tárgy VALUES(cim='Adatbázisok', tid='IAL404', ora=4, oktato='idd4q1');*
c) *INSERT INTO tárgy (tid, ora) VALUES('IAL404', null, null, 4);*

– Növelje a 3-as kódú oktatóhoz tartozó tárgyak óraszámát eggyel.
UPDATE tárgy SET óra=óra+1 WHERE oktató=3;

– Engedélyezze a TÁRGY tábla olvasását Péternek.
GRANT SELECT ON tárgy TO Peter;

– A legtöbb tárgyat tanító oktató neve.
CREATE VIEW v AS SELECT oktató, COUNT(*) db FROM tárgy GROUP BY oktató;
a) *SELECT név FROM oktató, v WHERE oktató.id=v.oktató AND db IN (SELECT MAX(db) FROM v);*
b) *SELECT név FROM oktató, v WHERE oktató.id=v.oktató AND db= (SELECT MAX(db) FROM v);*

5.16. Adottak az alábbi sémák: **DOLGOZÓ** [KÓD, NÉV, FIZETÉS, BEOSZTÁS, ÜZEMKÓD] és **ÜZEM** [KÓD, NÉV]. A megadandó SQL parancsok:

– Dolgozók létszáma.
SELECT COUNT(*) FROM dolgozó;

– 100000 forintnál többet keresők neve.
SELECT név FROM dolgozó WHERE fizetés>100000;

– Dolgozók neve és üzemük neve.
SELECT d.név, u.név FROM dolgozó d, üzem u
WHERE d.üzemkód=u.kód;

– Mennyi az egyes beosztásokban az átlagfizetés.
SELECT beosztás, AVG(fizetés) FROM dolgozó GROUP BY beosztás;

– Az átlagnál többet keresők neve.
SELECT név FROM dolgozó WHERE fizetés>(SELECT AVG(fizetés) FROM dolgozó);

5.20. Végezze el az alábbi műveleteket SQL-ben, ha a felhasználói azonosítója JOE:

– A DOLGOZÓ tábla olvasási jogának átadása a PETER felhasználónak.
GRANT SELECT ON dolgozó TO Peter;

– PETER lekérdezi az engedélyezett táblát.
*SELECT * FROM joe.dolgozó;*

– A DOLGOZÓ tábla rekord bővítési jogának megvonása minden felhasználótól.
REVOKE INSERT ON dolgozó FROM PUBLIC;

– A DOLGOZÓ táblára a rekord törlési jog átadása JOHN felhasználónak úgy, hogy ő továbbadhatja ezt a jogot.
GRANT DELETE ON dolgozó TO John WITH GRANT OPTION;

5.21. Végezze el az alábbi műveleteket SQL-ben, ha a felhasználói azonosítója JOE:

- A DOLGOZÓ táblára minden műveleti jog kiadása JAMES felhasználónak.
  GRANT ALL ON dolgozó TO James;
- A DOLGOZÓ tábla tartalom módosítási jogának engedélyezése minden felhasználó számára.
  GRANT UPDATE ON dolgozó TO PUBLIC;
- A DOLGOZÓ tábla NÉV mezőjére vonatkozó olvasási jogosultság átadása JOHN felhasználónak.
  GRANT SELECT(név) ON dolgozó TO John;
- A DOLGOZÓ tábla rekord törlési jogának szigorú letiltása PETER felhasználó esetén.
  DENY DELETE ON dolgozó TO Peter;

5.23. Adottak az alábbi sémák: **DOLGOZÓ** [KÓD, NÉV, BEOSZTÁS, FIZETÉS, FŐNÖK, ÜZEM] és **ÜZEM** [NÉV, KÓD]. Adja meg a megfelelő SQL SELECT utasításokat:

- Azoknak a dolgozóknak a neve és főnökük neve, akiknek a fizetése 100000 forint.
  SELECT név, főnök FROM dolgozó WHERE fizetés=100000;
- Üzemenként az átlagfizetés.
  SELECT üzem.név, AVG(fizetés) FROM dolgozó, üzem WHERE dolgozó.üzem=üzem.kód GROUP BY üzem.név;
- Kik nem főnökök.
  SELECT név FROM dolgozó WHERE kód NOT IN (SELECT főnök FROM dolgozó);
- Mely beosztásban dolgoznak 3-nál kevesebben.
  SELECT beosztás FROM dolgozó GROUP BY beosztás HAVING COUNT(*)<3;
- Melyik üzemben dolgozik Nagy nevű ember.
  SELECT üzem.név FROM üzem WHERE üzem.kód IN (SELECT dolgozó.üzem FROM dolgozó WHERE dolgozó.név LIKE 'Nagy%');
- Melyik az az üzem, ahol a legmagasabb az átlagfizetés.
  CREATE VIEW v AS SELECT u.név, AVG(fizetés) atl FROM dolgozó d, üzem u WHERE d.üzem=u.kód GROUP BY u.név;
  SELECT név FROM v WHERE atl=(SELECT MAX(atl) FROM v);

5.24. Az alábbi sémákhoz adja meg a megfelelő SQL parancsokat: **ZENESZÁM** [KÓD, CÍM, TÍPUS, ELŐADÓ] és **ELŐADÓ** [KÓD, NÉV, LAKCÍM].

- A népdal típusú számok törlése.
  DELETE FROM zeneszám WHERE tipus='népdal';

– Az ELŐADÓ tábla módosítási jogának engedélyezése Péternek.
GRANT INSERT, UPDATE ON előadó TO Peter;

– A miskolci előadók számainak címe.
SELECT z.cim FROM zeneszám z, előadó e WHERE z.előadó=e.kód AND e.lakcim='Miskolc';

– Hány zeneszám van az egyes típusokból.
SELECT tipus, COUNT(*) FROM zeneszám GROUP BY tipus;

– Mely városbeli előadóknak nincs népdal típusú száma.
SELECT lakcim FROM előadó WHERE kód NOT IN (SELECT előadó FROM zeneszám WHERE tipus='népdal');

– Azok az előadók, akiknek az átlagnál több zeneszámuk van.
CREATE VIEW v AS SELECT név, COUNT(*) db FROM zeneszám z, előadó e WHERE z.előadó=e.kód GROUP BY név;
SELECT név FROM v WHERE db >
(SELECT AVG(db) FROM v);

6.3. Normalizálja az alábbi sémát 3NF-ig:
R(A,B,C,D,E,F) ahol A → C, C → E , (A,B) → F , B → D.

Armstrong 1. axiómája alapján:
(A,B) → A és (A,B) → B

Armstrong 3. axiómája alapján:
(A,B) → A és A → C ⇒ (A,B) → C
(A,B) → C és C → E ⇒ (A,B) → E
(A,B) → B és B → D ⇒ (A,B) → D

A mezők atomiságát feltesszük.
1NF: **R**(A,B,C,D,E,F)

2NF: **R1**(A,B,F) **R2**(A,C,E) **R3**(B,D)

3NF: **R1**(A,B,F) **R2**(A,C) **R3**(C,E) **R4**(B,D)

6.5. Normalizálja az alábbi sémát 3NF-ig:
R(X,Y,Z,Q,W) ahol Y → W, X → (Q,Z), Z → Y.

A szétvághatósági szabály alapján:
X → (Q,Z) ⇒ X → Q és X → Z

Armstrong 3. axiómája alapján:
X → Z és Z → Y ⇒ X → Y
X → Y és Y → W ⇒ X → W

A mezők atomiságát feltesszük.
1NF: **R**(X,Y,Z,Q,W)

2NF: = **1NF**

3NF: **R1**(<u>X</u>,Q,Z) **R2**(<u>Z</u>,Y) **R3**(<u>Y</u>,W)

6.7. Normalizálja az alábbi sémát BCNF-ig:
R(A,B,C,D,E) ahol C → E, A → D, E → B, (A,E) → A.

Armstrong 1. axiómája alapján:
(A,E) → A és (A,E) → E

Armstrong 3. axiómája alapján:
(A,E) → A és A → D ⇒ (A,E) → D
(A,E) → E és E → B ⇒ (A,E) → B

De C → E, ezért (A,C) a kulcs.

A mezők atomiságát feltesszük.
1NF: **R**(<u>A,C</u>,B,D,E)

2NF: **R1**(<u>A,C</u>) **R2**(<u>A</u>,D) **R3**(<u>C</u>,E,B)

3NF: **R1**(<u>A,C</u>) **R2**(<u>A</u>,D) **R3**(<u>C</u>,E) **R4**(<u>E</u>,B)

BCNF: = **3NF**

6.13. Normalizálja az alábbi sémát BCNF-ig:
R(X,Y,Z,Q,R,S) ahol (Y,Q) → Y , Q → Z, Y → S, (Y,Q) → R, S → X.

Armstrong 1. axiómája alapján:
(Y,Q) → Y és (Y,Q) → Q

Armstrong 2. axiómája alapján:
Q → Z ⇒ (Y,Q) → (Y,Z)

A szétvághatósági szabály alapján:
(Y,Q) → (Y,Z) ⇒ (Y,Q) → (Y) és (Y,Q) → (Z)

Armstrong 3. axiómája alapján:
(Y,Q) → Y és Y → S ⇒ (Y,Q) → S
(Y,Q) → S és S → X ⇒ (Y,Q) → X

A mezők atomiságát feltesszük.
1NF: **R**(<u>Y,Q</u>,X,Z,R,S)

2NF: **R1**(<u>Y,Q</u>,R) **R2**(<u>Y</u>,S,X) **R3**(<u>Q</u>,Z)

3NF: **R1**(<u>Y,Q</u>,R) **R2**(<u>Y</u>,S) **R3**(<u>S</u>,X) **R4**(<u>Q</u>,Z)

BCNF: = **3NF**

6.14. Normalizálja az alábbi sémát BCNF-ig:
R(A,B,C,D,E,F) ahol A → C, E → B, C → (F,C), (A,E) → D.

Armstrong 1. axiómája alapján:
(A,E) → A és (A,E) → E

A szétvághatósági szabály alapján:
C → (F,C) ⇒ C → (F) és C → (C)

Armstrong 3. axiómája alapján:
(A,E) → A és A → C ⇒ (A,E) → C
(A,E) → C és C → F ⇒ (A,E) → F
(A,E) → E és E → B ⇒ (A,E) → B

A mezők atomiságát feltesszük.
1NF: **R**(A,E,B,C,D,F)

2NF: **R1**(A,E,D) **R2**(A,C,F) **R3**(E,B)

3NF: **R1**(A,E,D) **R2**(A,C) **R3**(C,F) **R4**(E,B)

BCNF: = **3NF**

6.16. Normalizálja az alábbi sémát BCNF-ig:
R(A,B,C,D,E) ahol A → B, A → C, B → A, B → C, C → D, D → E.

Armstrong 3. axiómája alapján:
B → C és C → D ⇒ B → D
B → D és D → E ⇒ B → E

De A → B, tehát A vagy B lehet a kulcs.

A mezők atomiságát feltesszük.
1NF: **R**(B,A,C,D,E)

2NF: = **1NF**

3NF: **R1**(B,A) **R2**(A,C) **R3**(C,D) **R4**(D,E)

BCNF: **R1**(B,A,C) **R2**(C,D) **R3**(D,E)

7.3. Készítsen C gazdanyelv esetén beágyazott SQL felületű programot, mely bekér egy beosztás értéket és kiírja az ilyen beosztású dolgozók névsorát ABC sorrendben. A tábla szerkezete: DOLGOZÓ[KÓD, NÉV, BEOSZTÁS].

```
#include <stdio.h>
                              /* output és input változók deklarálása */
EXEC SQL BEGIN DECLARE SECTION;
    char beosztas[30];        /* beosztás mező */
    char nev[15];             /* név mező */
    short inev;               /* indikátor */
    VARCHAR nev[40];          /* felhasználó azonosító neve */
    VARCHAR jelszo[40];       /* felhasználó jelszava */
EXEC SQL END DECLARE SECTION;
                              /* hibakezelési kommunikációs terület */
EXEC SQL INCLUDE sqlca;
```

```
main()                          /* főprogram */
{
                                /* hibakezelés definiálása */
    EXEC SQL WHENEVER SQLERROR DO hiba();
                                /* bejelentkezés a rendszerbe */
    strcpy (nev.arr,"SCOTT");
    nev.len = strlen (nev.arr);
    strcpy (jelszo.arr,"TIGER");
    jelszo.len = strlen (jelszo.arr);
    EXEC SQL CONNECT :nev IDENTIFIED BY :jelszo;
                                /* kurzor deklarációja */
    EXEC SQL DECLARE kurz CURSOR FOR
        SELECT nev FROM dolgozo WHERE beosztas = :beosztas
        ORDER BY nev;
                                /* beosztás bekérése */
    printf ("beosztás = ");
    scanf ("%s", beosztas);
                                /* lekérdezés elindítása */
    EXEC SQL OPEN kurz;
                                /* lekérdező ciklus */
    EXEC SQL WHENEVER NOT FOUND GOTO veg;
    while (1) {
        EXEC SQL FETCH kurz INTO :nev:inev;
        if (inev == 0) {
            printf ("név= %s\n", nev);
        }
    }
                                /* kilépés a ciklusból */
    veg:
                                /* kurzor lezárása */
    EXEC SQL CLOSE kurz;

                                /* tranzakció lezárása */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}
hiba()                          /* hibakezelő rutin */
{
                                /* végtelen ciklus elkerülése */
    EXEC SQL WHENEVER SQLERROR CONTINUE;
                                /* hibaüzenet kiírása */
    printf ("hiba: %s\n", sqlca.sqlerrm.sqlerrmc);
                                /* tranzakció visszagörgetés */
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}
```

7.4. Készítsen C gazdanyelv esetén OCI felületű programot, mely felvisz egy új dolgozó rekordot a DOLGOZÓ[KÓD, NÉV, BEOSZTÁS] táblába. A kód egyediségét nem kell ellenőrizni.

```
...
struct crsdef cursor;
struct ldadef lda;
char uid[20];
...
strcpy (uid, "scott/tiger");
olon (&lda, uid, -1, (char *) -1, -1, 0);
oopen (&cursor, &lda, (char *) -1, -1, -1, (char *) -1, -1);
osql3 (&cursor,"INSERT INTO dolgozo
    VALUES(2345,'Nagy Peter','Operator')",-1);
...
oclose (struct crsdef *cursor);
ologof (struct ldadef *lda);
```

8.2. A DOLGOZÓ[KÓD, NÉV, BEOSZTÁS, FIZETÉS, FŐNÖK] táblából kérdezze le a 1111 kódú vezető 100000 Ft-nál kevesebbet kereső beosztottjait 2 szintig lemenve.

SELECT nev FROM dolgozo WHERE fizetes<100000 AND LEVEL <3 START WITH kod=1111 CONNECT BY PRIOR kod=fonok;

8.3. Hozzon létre olyan származtatott táblát, amely kétnaponta frissül és az ingatlannal nem rendelkezők neveit és lakcímét tartalmazza (SQL utasítás). A felhasználandó táblák: SZEMÉLY[KÓD, NÉV, CÍM] és INGATLAN[AZON, CÍM, TULAJ].

CREATE SNAPSHOT noing REFRESH START WITH SYSDATE NEXT 'SYSDATE+2' AS SELECT nev, cim FROM szemely WHERE kod NOT IN (SELECT tulaj FROM ingatlan);

Fogalom magyarázat

A, Á

Absztrakt objektum: összetett objektumok, melyek mögött nem egy elemi érték áll; az ember egy absztrakt objektum, mert több tulajdonság együttese jellemzi.

Adat: az információ hordozója; a tények, fogalmak számítási feldolgozásra alkalmas reprezentációja.

Adatabsztrakció: az adatok ábrázolásának részletektől mentes, lényegi leírása.

Adatbázis: integrált adatszerkezet, mely több különböző objektum előfordulási adatait adatmodell szerint szervezetten, perzisztens módon tárolja olyan segédinformációkkal, ún. metaadatokkal együtt, melyek a hatékonyság, integritásőrzés, adatvédelem biztosítását szolgálják.

Adatbázis adminisztrátor (DBA): az adatbázis rendszergazdája.

Adatbáziskezelő rendszer: programrendszer, melynek feladata az adatbázishoz történő hozzáférések biztosítása és az adatbázis belső karbantartási funkcióinak végrehajtása.

Adatbázis modellek: az adatbázisban letárolandó adatrendszer szerkezetének, az elvégezhető műveletek körének és a megkötések leírására szolgáló formalizmus. (lásd még *Adatmodell*)

Adatbázisrendszer: az adatbázis, az adatbáziskezelő és az alkalmazások együtteséből álló információs rendszer.

Adatdefiníciós nyelv: az adatbáziskezelő parancsnyelvének az a komponense, amellyel adatbázis objektumok (például táblák) hozhatók létre, szüntethetők meg; ez a komponens szolgál az objektum szerkezetének módosítására is.

Adatelőfordulás (instance): megadott adattípusra illeszkedő objektum, változó.

Adatfüggetlenség: az adatbázisrendszer kapcsolódó elemei között megvalósuló függetlenség az adatok kezelését illetően; az egyik szinten elvégzett módosítás nem hat ki a másik szintre; főbb megvalósulásai a logikai és fizikai adatfüggetlenség.

Adatkezelő nyelv: az adatbáziskezelő parancsnyelvének azon komponense, mellyel az adatbázis táblák tartalma kezelhető: módosítható, bővíthető vagy törölhető.

Adatkommunikációs komponens (DC): az adatbáziskezelő azon komponense, mely a kliensektől bejövő üzeneteket fogadja vagy oda küld üzeneteket.

Adatlekérdező utasítás: olyan utasítás, melynek célja az adatbázisban tárolt adatok kiolvasása az adatbázisból; a relációs algebra döntő része a lekérdezési utasítások lehetőségeit írja le.

Adatmodell: formalizmus az adatspecifikus elemek leírására és kezelésre az adatbázisban; magába foglalja a szerkezet megadását, az elvégezhető műveletek körét és az értelmezhető integritási feltételeket. (lásd még *Adatbázis modellek*)

Adatszerű tárolás: az információnak stukturált, elemekre bontott, kapcsolatokat és jelentést is hordozó alakban való letárolása.

Aggregáció: a relációs algebrában az a művelet, mely során az elemi adatokból összesített értékeket határozunk meg (avg, sum, min, max, count); az IFO modellben több elemből összeálló egységet jelent: egy lakcím mint aggregáció adható meg, mivel a város, utca, házszám elemekből áll elő.

Alap join: két reláció Descartes-szorzatát szokás alap join-nak is nevezni; ez a művelet önmagában ritkán használatos.

Állományszervezés: a háttértáron lévő állományok belső struktúrája.

ALTER: adatbázis objektum szerkezetének módosítása; ALTER TABLE a tábla szerkezetének módosítására szolgál. A mezők megszüntetése nem minden implementációban támogatott.

Anomália: az adatmodell helytelen felépítéséből fakadó adatkezelési problémák. Jelentheti felesleges adatelemek felvitelét vagy módosítását. Attól függően, hogy mely művelet során jelentkezik, lehet bővítési, beszúrási, módosítási anomália.

Apobetika: az információ mögött húzódó szándék.

Armstrong-axiómák: az FD függőségek származási szabályait leíró axiómák. Létező függőségekből újabb függőségek származtathatók.

Assertion: globális integritási feltétel, melyben több táblát érintő SELECT kifejezés is megadható operandusként. Önálló adatbázis objektumként viselkedik. Ügyelni kell a feltétel hatékony megfogalmazására.

Asszociáció: objektumok társítása egymással; kapcsolat, mely lehet ideiglenes is.

Atomi séma: olyan reláció séma, melyhez nem létezik független felbontás, minden dekompozíciója szétszakít valamely belső FD-t.

Attribútum: az objektumok tulajdonsága; az ember objektum esetében a név és az életkor mint attribútum jelenik meg.

Attribútum szelekció: a szelekció azon típusa, melynél a szelekciós feltételben több attribútum értéke kerül összehasonlításra.

B

Bázis reláció: olyan reláció az adatbázisban, melynek teljes előfordulása az adatbázis előfordulásban letárolásra kerül és értéke független a többi relációtól; a bázis vagy alaprelációk mellett az adatbázis tartalmazhat származtatott relációkat is.

Beágyazott SQL: a gazdanyelv utasításai közé beillesztjük az SQL nyelv utasításait. A beszúrt SQL parancsok alakjukban a szabvány SQL parancsokhoz illeszkednek.

Bejelentkezési adatterület: az OCI rendszerben a kliens programot azonosító adatstruktúra; a kapcsolatleíró információkat foglalja magába.

Beszúrási anomália: új rekord felvitelekor jelentkező többletköltség; az új információt leíró adatelemek mellett a már korábban letárolt információkat megadó adatelemek ismét felvitelre kerülnek.

BETWEEN: kétoperandusú relációoperátor; ellenőrzi, hogy a kifejezés értéke az operandusként megadott intervallumba esik-e.

B-fa: index fa, ahol egy csomópontban több elem is tárolható, és egy csomópontnak több gyereke is lehet; a B-fa kiegyensúlyozott és jó kihasználtságú.

Blokk: az állományszervezés azon egysége, melyet egységként lehet mozgatni a központi egység és a háttértár között; írási és olvasási egység.

Blokkok láncolása: a blokkok közötti sorrendiség mutató lánccal történő nyilvántartása.

Blokk címlista: olyan állományszervezési elem, amelyben egy listában van megadva az állományhoz tartozó blokkok halmaza és sorrendisége.

Boyce-Codd normálforma (BCNF): normalizálási lépcsőfok. A séma akkor van BCNF szinten, ha FD csak jelölt kulcsból indul ki.

Bucket: egy tárolási egység, amelyben az összetartozó rekordok, adatelemek kerülnek elhelyezésre.

C

CASE szerkezet: a SELECT utasítás projekciós részében szereplő tag, melynek segítségével egy eredménymező értékét más mennyiségektől függően lehet meghatározni. A CASE szokásos feltétele bizonyos kifejezések NULL értéke.

CHECK: általános értékellenőrzési feltétel kötése egy mezőhöz; operandusként a feltételt kell megadni, melyben a vizsgált mezőt nevével azonosítjuk.

CLI: Call Library Interface; a gazdanyelv utasításai között függvényhívások formájában adjuk meg az SQL parancsok végrehajtására szolgáló részt. A forráskód teljes egészében illeszkedik a gazdanyelv szintaktikai követelményéhez.

CODASYL szabályok: a CODASYL szervezet által megalkotott szabályok rendszere, melynek célja a hálós adatmodell struktúrájának egyértelmű leírása.

Codd-szabályok: Codd által megfogalmazott követelmények a relációs adatbáziskezelő rendszerekkel szemben. Céljuk a nem igazi RDBMS rendszerek felismerése és az RDBMS fejlesztési igények meghatározása.

COMMIT: tranzakció véglegesítése; hatására a tranzakció keretében végzett műveletek eredménye letárolódik az adatbázisba.

CONNECT: a beágyazott SQL azon utasítása, mellyel kiépíthető az adatkapcsolat a végrehajtó adatbáziskezelőhöz. A parancs legfontosabb paraméterei az adatbázis azonosító, a név és a jelszó.

CREATE: adatbázis objektumok létrehozásának utasítása; a CREATE TABLE szolgál új tábla létrehozására.

CURRENT OF: a kurzorkezeléshez kapcsolódó tag. A módosítás utasításánál a szelekciós részben megadott CURRENT tag jelöli ki, hogy a módosítást a kurzor által éppen kijelölt rekordra kell elvégezni.

CS

Csoportképzés: relációs algebrai művelet, mely során a reláció rekordjai egy megadott kifejezés értéke alapján diszjunkt csoportokba kerülnek szétosztásra.

Csoportképzési kifejezés: azon kifejezés, mely a csoportképzés során meghatározza a csoportbontás eredményét; az azonos helyettesítési értékű rekordok kerülnek egy csoportba.

Csoportmező: olyan mező struktúra, mely több elemi vagy összetett mezőt fog össze; a csoport különböző szerkezetű és megadott elnevezésű részt fog össze.

D

DBMS közeli adatmodellek: gépközeli, egzakt adatmodellek, melyek a DBMS rendszerekben megvalósíthatók.

DCL: Data Control Language, adatvezérlő nyelv; az SQL nyelv működési környezet beállító utasításokat tartalmazó komponense; védelmet és tranzakciót szabályozó utasításokat tartalmaz.

DDL: Data Definition Language; az SQL nyelv adatdefiníciós komponense, ide tartoznak az adatbázis objektumok létrehozásának, megszüntetésének és szerkezet módosításának utasításai.

DEFAULT: alapértelmezési mezőérték kijelölése, ha NULL értéket kapna a mező a rekord létrehozásakor, akkor az itt megadott adat kerül a mezőbe a NULL helyett.

Deferred: késleltetett integritás ellenőrzés. A megadott feltételek csak a tranzakciók végén kerülnek ellenőrzésre.

Dekompozíció: a reláció szerkezetének felbontása két vagy több relációra. A dekompozíció során egyszerűsödik az egyes relációk sémája. A felbontás során ügyelni kell arra, hogy létezzen kapcsolati elem az egyes eredményrelációk között.

DELETE: rekordelőfordulások törlése; egy szelekciós résszel lehet kijelölni a kitörlendő rekordok körét.

De-normalizálás: a normalizálási folyamat inverze. A normalizálás után szokás végrehajtani. Célja a szétszedett sémákat újra összehozni a lekérdezések hatékonyságának növelése céljából, hogy minél kevesebb join műveletre legyen szükség.

Deszkriptív: a problémát formálisan megadott elemekből, szabályok szerinti kifejezéssel leíró műveletkijelölés; nem magát a végrehajtandó műveletsort adja meg, hanem csak a megoldandó feladatot (az eredmény állapot jellemzését).

Dinamikus elem: az adatmodell olyan eleme, mely az adatbázisban végbemenő műveletekhez, változásokhoz kapcsolódik; az objektumok viselkedését, időbeli változásukat leíró rész.

Dinamikus SQL: olyan SQL parancsokat jelöl, melyek pontos alakja csak a program futása során határozódik meg. A forráskód megírásakor kell olyan kódrészletet megadni, mely alkalmas a futásidejű SQL parancs kiadására. Az SQL parancs jellegétől függően több módja van az előkészítésnek.

DISTINCT: a SELECT utasításban használható, hatására nem lesz ismétlődés az eredmény relációban.

Diszjunkt specializálás: minden általános típus csak maximum egy altípushoz rendelődhet.

DML: Data Manipulation Language; az SQL nyelv adatkezelő komponense, ide tartozik a rekord felvitel, tartalom módosítás és rekord törlés utasítása.

Domain: értelmezési tartomány, mely megadja az elemhez tartozó értékkészletet, és meghatározza a végrehajtható műveletek körét.

Domain kalkulus: DRC, Domain Relational Calculus; a relációs kalkulus azon típusa, amikor a változók attribútumokat reprezentálnak.

DQL: Data Query Language; az SQL nyelv adatlekérdező komponense, a SELECT utasítást tartalmazza.

DROP: adatbázis objektum megszüntetése, a DROP TABLE utasítás szolgál egy tábla megszüntetésére.

E,É

EER adatmodell: kiterjesztett ER modell, mely az ER modell specializáció és tartalmazási kapcsolat elemekkel való kibővítésével jött létre.

Egyed: a valóság egy önálló léttel bíró egysége, melyhez különböző információ elemeket lehet hozzárendelni.

Egyedelőfordulás: az egyedtípusnak megfelelő objektum.

Egyedtípus: az egyed jelentését, az azt leíró tulajdonságokat megadó séma.

Egyszerű kulcs: olyan kulcs, mely csak egyetlen egy mezőből áll.

Egyszerű tulajdonság: olyan tulajdonság, melynek értéke egy elemi (atomi) érték, azaz a modell szempontjából belső szerkezettel nem rendelkező érték. Például az életkor, mert egy számértékkel megadható.

Egytagú és többtagú set: a set-ben tárolt tagrekord típusok darabszámára utaló elnevezés.

Elemi egyedtípus: olyan típus, melynek struktúrája elemi érték tárolására alkalmas.

Elemi mező: olyan mező, melyben egy elemi érték kerülhet csak letárolásra. Ez az adatbázis struktúra legkisebb egységként elérhető eleme.

Elemi objektum: elemi egyedtípusra illő objektum.

Előfordító: a beágyazott SQL utasításainak feldolgozására szolgáló modul. Az előfordító a beszúrt SQL parancsokat CLI alakra konvertálja és kiegészíti a feldolgozáshoz szükséges egyéb adatelemekkel.

Előfordulási diagram: az adatbázis megadásakor különbséget teszünk a séma és az előfordulás megadása között. Az előfordulási diagramm megadja az egyes rekordelőfordulások létezését és kapcsolatát.

Elsődleges kulcs: a jelölt kulcsok közül az azonosításra kiválasztott mezőcsoport; egyedi és nem üres. A kapcsolódó relációk erre a mezőcsoportra fognak hivatkozni, azaz ennek értékét tartalmazzák a megfelelő idegen kulcsban.

Elsődleges kulcs integritási feltétel: olyan integritási feltétel, mely kimondja, hogy az elsődleges kulcs minden rekordban létezik és egyedi értéket hordoz.

ER adatmodell: egyed-kapcsolat adatmodell, mely a modellezett terület leírására szolgáló adatstruktúrát az egyedek, tulajdonságaik és kapcsolataik feltárásával adja meg.

Értékkorlátozás: a statikus integritási feltételek egyik fontos eleme a felvehető

adatértékek körének korlátozása; az egyediséget előíró integritási feltétel is hordoz értékkorlátozási elemeket.

EXEC SQL: a beágyazott SQL parancsok előtt szereplő tag, mely kijelöli az előfordító számára, hogy mely sorok szolgálnak az SQL parancsok kezelésére.

EXECUTE: dinamikus SQL parancs végrehajtására szolgál. Lehet előkészített vagy közvetlen a végrehajtás.

EXISTS: relációoperátor; igaz értékű, ha az operandus halmaz vagy al-SELECT eredményhalmaz nem üres.

F

Fagin tétele: a normalizálás egyik alaptétele, mely a veszteségmentes felbontás szükséges kritériumát adja meg. A tétel a többértékű FD-n alapszik.

FETCH INTO: a kurzor soronkövetkező rekordjának beolvasása a kijelölt gazdanyelvi változókba.

FOR UPDATE: a kurzor létrehozásakor használható tag, melynek hatására a kurzort módosíthatóként kezeli.

Fregment: az egyedtípus struktúráját leíró kapcsolati gráf egy algráfja, mely rendszerint egy aggregációt vagy csoportképzést takar.

Funkcionális adatmodell: olyan adatmodell, melyben az elemek közötti kapcsolatok nem struktúraszerűen, tartalmazás jelleggel íródnak le, hanem hozzárendeléssel, azaz függvényszerűen.

Funkcionális függőség (FD): alapvető függőségi viszony a mezőcsoportok között a relációs sémában. Az A → B FD teljesül, ha minden A-hoz maximum egy B érték tartozik.

G,GY

Gazda (host) nyelv: azon programozási nyelv, melynek kódjába beillesztjük az adatbázis kapcsolatot megvalósító DBMS utasításokat.

Gazdanyelvi változó: olyan programváltozó, melyet a gazdanyelvi programban annak formátuma szerint hozunk létre, de a vendégnyelvben (pl. SQL) is felhasználunk az adatforgalom lebonyolítására, hibák jelzésére.

Globális integritási feltétel: olyan integritási feltétel, melynek ellenőrzése több relációt is érint az adatbázisból; a hivatkozási integritási szabály egy példa a globális integritási feltételre.

Grafikus adatbázis-séma: az adatbázis szerkezetének megadásakor a táblák és kapcsolataik grafikus jelölés rendszerrel való ábrázolása.

GRANT: hozzáférési jogosultság engedélyezése; a jogosultság SQL műveletek szintjén definiálható.

GROUP BY: a SELECT utasítás része, a rekordok csoportosítására szolgál; az azonos csoportképzési értékkel rendelkező rekordok kerülnek egy csoportba.

Gyenge egyed: olyan egyedtípus, mely nem rendelkezik önálló azonosító tulajdonság csoporttal (jelölt kulccsal); az egyed azonosítása csak valamely kapcsolatán keresztül biztosítható.

H

Halmazorientáltság: az adatok tárolása halmazokban történik, és az adatkezelő műveletek is halmazokon értelmezettek.

Hálós adatmodell: olyan adatbázis adatmodell, melyben a rekordok közötti kapcsolatok set-ekbe történő csoportosítással adhatók meg; egy rekordtípus több set-ben is elhelyezkedhet a CODASYL szabályoknak megfelelően, hálós kapcsolati viszonyt eredményezve; a modellben egy mező összetett szerkezetű is lehet; a rekordok elérése a lerögzített kapcsolatok mentén haladva történik.

Hálós adatdefiníciós nyelv (NDDL): a hálós adatmodellhez definiált adatdefiníciós nyelv.

Háromértékű logika (3VL): olyan logikai kifejezések kezelését teszi lehetővé, melyekben a szokásos kétféle logikai érték, az igaz és hamis érték mellett egy harmadik logikai érték is megjelenik. Az U (nem ismert) logikai érték a kiértékelések legvégén hamis értékre konvertálódik.

Hashing rekordelérés: a hash technika egy függvény segítségével határozza meg a rekordot tároló bucket helyét; a függvény bemenő argumentuma a rekord azonosító tulajdonsága.

Heath tétele: a normalizálás egyik alaptétele, mely a dekompozíció veszteségmentességéhez kapcsolódik. A tétel kimondja, hogy megfelelő közös kapcsoló mezőcsoport létezése esetén biztosított a veszteségmentesség.

Hierarchikus adatdefiniáló nyelv (HDDL): a hierarchikus adatmodell adatdefiníciós nyelve; a mezők, rekordtípusok és a hierarchiák megadására, kezelésére.

Hierarchikus adatkezelő nyelv (HDML): a hierarchikus adatmodellhez értelmezett adatkezelő nyelv, a rekord előfordulások kezelésére.

Hierarchikus adatmodell: a legelső adatbázis adatmodell; a modellben a rekordok közötti kapcsolatok szülő-gyerek elemekből felépített fa struktúrájú kapcsolatrendszert, hierarchiát alkotnak; a mezők csak elemi értéket vehetnek fel; a rekordok elérése a lerögzített kapcsolatok mentén haladva történik.

Hierarchikus SELECT: olyan lekérdezés, amellyel a tábla többszöri rekurzív átfutásával többszörös mélységű navigációs bejárással adható meg az eredménytábla. A tábla többszöri átfutása helyett egy SELECT ... START WITH ... CONNECT BY ... utasítás elegendő a rekurzív bejáráshoz.

Homonima: az adatbázis tervezés azon hibája, amikor eltérő jelentésű fogalmakat, modell elemeket azonos elnevezéssel illetünk.

I

Idegen kulcs integritási feltétel: hivatkozási integritási feltétel, mely kimondja, hogy az idegen kulcs értéke vagy üres, vagy egy létező rekord előfordulás kulcsértékét tartalmazza.

Idegen kulcsmező: olyan mezőcsoport a relációsémában, melynek célja egy megadott másik reláció valamely rekord előfordulásának az egyértelmű kijelölése.

Ideiglenes eredményreláció: a relációs műveletek feldolgozásakor keletkező reláció, mely nem tárolódik perzisztensen az adatbázisban.

Ideiglenes tábla: a tábla fizikailag tárolódik, önálló, független tartalommal is rendelkezhet, de létezése csak egy megadott tevékenységi egység időtartamára korlátozott. A tábla a megadott végrehajtási egységből kilépve automatikusan megszűnik.

IFO adatmodell: olyan szemantikai adatmodell, amely objektum orientált alapokon nyugszik, és a kapcsolatok funkcionálisan vannak megadva.

Implementáció orientált adatmodell: olyan adatmodell, melyben az elsődleges cél nem a funkcionalitás minél szélesebb körű megvalósítása, hanem a hatékony megvalósíthatóság.

IN: halmazoperátor; a feltétel igaz értékű, ha a kifejezés értéke az operandusként megadott halmazban benne van.

Index: az állomány rekordjainak kulcsértékét és a rekord pozíciót tároló szerkezet, melyben a bejegyzések kulcsérték szerinti sorrendben helyezkednek el, gyors keresést lehetővé téve.

Indikátorváltozó: a hibakezelés egyik eszköze. Elsősorban a NULL érték ellenőrzésére szolgál. Ha a lekérdezés során a fogadó gazdanyelvi változóba NULL érték kerülne, akkor a kapcsolódó indikátorváltozó negatív értéket vesz fel.

Információ: jelsorozat a hozzá kapcsolódó jelentéssel együtt.

Információs rendszer: adatbázisra épülő számítógépes rendszer a vállalatok hatékony információ kezelésére.

Inkonnektivitás: a tervezés azon hibája, amikor a dekompozíció során az eredmény táblák közötti kapcsolat elvész, nincs kapcsoló elem.

INSERT: új rekord előfordulás felvitele a relációba; az INSERT utasítás alapvetően egy új rekord felvitelére szolgál, de van olyan alakja is, mely egy SELECT eredmény halmazát viszi fel.

Integritásőrzés: az adatbázisban definiált működési szabályok betartatása; a felvehető értékek körét vagy a műveletek körét korlátozó megkötések ellenőrzése.

Irreducibilis FD halmaz: minimális FD halmaz, melyet tovább nem szűkíthetünk anélkül, hogy az ekvivalencia meg ne szűnne az alap FD halmazzal. Az irreducibilis FD elemeiben az FD-k jobb oldala mindig elemi.

IS NULL: relációoperátor; a feltétel igaz értékű, ha az operandusként megadott kifejezés nem üres értékű.

J

Jelölt kulcs: olyan mezőcsoport a reláció sémában, mely egyedi, értéke sehol sem üres és nem vehető el belőle egyetlen egy mező sem, hogy ezen tulajdonságai meg ne sérülnének. A jelölt kulcsok valamelyike szerepel elsődleges kulcsként.

Job: munkaköteg; adatkezelő parancsokat tartalmazó utasítássor, melyet megadott időpontokban kell lefuttatnia az adatbáziskezelő rendszernek. Az ütemezett műveletsort rendszerint tárolt eljárásként kell megadni.

Join: két reláció összefűzése oly módon, hogy az eredmény reláció attribútum halmaza az alaprelációk attribútum halmazainak uniója.

Join függőség (JD): a mezőcsoportok közötti függőség azon fajtája, amikor az alapreláció a megadott mezőkhöz tartozó projekciókból a join művelettel előállítható.

K

Kapcsolat (1-1, 1-N, N-M, n-ed fokú, totális): egyedtípusok közötti asszociáció, melyek a kapcsolódó egyedek számában, a jellegben különbözhetnek egymástól. Az n-ed fokú kapcsolatban n egyed vesz részt. A totális kapcsolat kötelező jelleget takar.

Kapcsoló kulcs: az idegen kulcs egy másik elnevezése, utalva arra, hogy az idegen kulcs kapcsoló szerepet tölt be.

Késleltetett ellenőrzésű integritási feltétel: olyan integritási feltétel, melynek teljesülése nem közvetlenül a végrehajtáskor, hanem egy későbbi időpontban, rendszerint a tranzakció végén kerül ellenőrzésre.

Kibővítés, kiterjesztés: egy reláció sémájának kibővítése új attribútumokkal; az új attribútum a meglévő attribútumokból származtatható.

Kompozíció: szoros strukturális kapcsolat az osztályok között, az UML modell egy eleme.

Konstans szelekció: a szelekció azon típusa, melynél a szelekciós feltételben az attribútum értéke egy konstans értékkel kerül összehasonlításra.

Kulcs: a keresés, az azonosítás szempontjából meghatározó mező vagy mezőcsoport, mely a rekord azonosítására szolgál; azaz értéke nem ismétlődik és egyetlen egy rekordban sem üres az értéke. Fontosabb típusai: elsődleges kulcs, jelölt kulcs, idegen kulcs, szuper kulcs, index kulcs.

Kulcs tulajdonság: olyan tulajdonság vagy tulajdonság csoport, melynek értéke egyértelműen meghatározza az egyed előfordulását.

Kurzor szerkezet: a beágyazott és CLI felületek fontos adatkapcsolati eleme. A kurzor szolgál egy SQL lekérdezés eredményének rekordonkénti feldolgozására a gazdanyelvi programban. A kurzor megoldásnál az eredmény egy ideiglenes táblába letárolódik, melyből a gazdanyelvi program rekordonként el tudja érni az egyes eredményrekordokat. Egyes kurzortípusok tetszőleges mozgást, navigációt is megengednek az eredményhalmazban.

Kurzor struktúra: az OCI rendszerben egy SQL parancs végrehajtására szolgáló memória terület. Egy kurzor terület egy SQL parancsot tud feldolgozni, de egyidejűleg több kurzor struktúra is nyitva lehet.

Különbség: azon relációs algebrai művelet, mely két azonos szerkezetű reláció rekordjainak különbségét adja eredményül.

Külső (outer) join: a szelekciós join művelet egy fajtája, amikor a reláció azon rekordjai is bekerülnek az eredmény relációba, melyekhez nem létezik kapcsolódó rekord a másik oldalról; ezen rekordok üres értékű mezőkkel egészülnek ki.

L

Laza csatolás: a gazdanyelvbe ágyazott DBMS kezelő felület esetében arra utal, hogy szemantikailag élesen elválnak egymástól a gazdanyelvi és a DBMS kezelő utasítások.

LDA: Logon Data Area; kommunikációs terület az Oracle DBMS és a CLI program között.

Leszármaztatott tulajdonság: az ős típusból, osztályból örökölt tulajdonság.

LIKE: relációoperátor; egy szöveges kifejezés értékének egy operandusként megadott mintára való illeszkedését vizsgálja, melyben % és _ a joker karakterek.

Lokális integritási feltétel: olyan integritási feltétel, melynek kiértékelése csak egyetlen relációt érint. Az elsődleges kulcs, az egyediség, az általános értékellenőrzés feltétele egy-egy példa a lokális integritási feltételre.

Lost update: a tranzakciók párhuzamos végrehajtása során fellépő, az adatfelülírásból eredő adatvesztés.

M

Metaadat: a normál adatokat leíró adatok.

Meta egyedtípus: olyan típusok, melynek előfordulásai is típusok.

Metszet: azon relációs algebrai művelet, mely két azonos szerkezetű reláció rekordjainak metszetét adja eredményül.

Mező: az adatbázis struktúra azon egysége, melyből a rekordok felépülnek; a mező rendszerint a legkisebb DB struktúra egység.

Módosítási anomália: rekordok módosításakor jelentkező többletköltség; az új információt leíró adatelemek több helyen is jelen vannak az adatbázisban, így a módosítás során több helyen is módosítani kell.

Módosítható kurzor: a kurzor azon fajtája, amikor egy közvetlen és egyértelmű összerendelés képezhető a kurzor rekord és az alaptábla rekord között. Ebben az esetben a kurzor rekord tartalma módosítható, és a módosítás egyértelműen visszavezethető egy alaptábla módosítására.

Műveleti integritási feltétel: a végrehajtható műveletekre vonatkozó integritási, megkötési elemek.

N

Natural join (természetes join): a szelekciós join művelet egy típusa, amikor az illeszkedés az azonos elnevezésű attribútumok értékegyezőségén alapszik.

Nem triviális függőség: olyan FD, mely nem triviális. A triviális függőség minden mezőre vagy mezőcsoportra teljesül, így nem vesszük figyelembe a normalizálás során.

Név inkonzisztencia: az egyes adatmodell elemek elnevezésénél fellépő következetlenség; nem a jelentésre utaló elnevezés, azonos elnevezés eltérő tartalomra vagy eltérő elnevezés azonos tartalomra.

Normál egyed: olyan egyedtípus, melyhez társítható olyan tulajdonság(csoport), amely azonosító szerepet tölthet be (kulcs).

Normalizálás: azon folyamat, mely során a relációs sémában meglévő redundanciát okozó elemeket feltárjuk és megszüntetjük. A normalizálást több egymásra épülő lépésben hajtjuk végre. Minden lépéshez egy normálforma tartozik.

Normalizációs lépés: normalizálási lépcsőfok. Több egymásra épülő lépés létezik, melyekhez egy-egy normálforma tartozik. A lépés során biztosítjuk, hogy a megfelelő normálforma teljesüljön.

Normalizálási szabályok: a helyes, anomáliáktól mentes relációsémák megvalósítását szolgáló tervezési irányelvek.

Normálforma: a relációs sémára vonatkozó előírás. Teljesülése esetén bizonyos, anomáliákat és redundanciákat okozó elemek nem jelenhetnek meg a sémában. Több egymásra épülő normálforma van: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF.

NOT NULL: integritási feltétel, a hozzá kapcsolódó mező értéke nem lehet kitöltetlen (üres).

NULL érték: az adatbázisban az üres, nem kitöltött mező értéke; nem azonos a 0 értékkel.

O,Ö

Objektum-orientált adatmodell: az objektum orientáltság elveit megvalósító adatmodell, melyben megvalósul az öröklés, a metódusok tárolása is.

OCI: Oracle Call Interface; az Oracle rendszerhez kapcsolódó CLI interface, mely több különböző gazdanyelvhez is kapcsolódhat.

OPEN: kurzor megnyitása, a lekérdezés végrehajtása.

Operációk: műveletek valamely objektumokon.

ORDER BY: a SELECT parancs része, az eredményrekordok rendezésére szolgál; lehet növekvő (ASC) és csökkenő (DESC) rendezettséget előírni.

Osztás művelete: a join művelet inverze; eredménye azon legnagyobb rekordhalmaz, melynek alap join művelete az osztóval benne van az osztandóban.

OUTER JOIN ON: a SELECT parancs része, a külső join kijelölésére szolgál; az ON rész adja meg a kapcsolódási szelekciós feltételt.

Összetett kulcs: olyan kulcs, mely több mező együtteséből áll elő; mind az elsődleges kulcs, mind az idegen kulcs lehet összetett is.

Összetett mező: olyan mezőtípus, mely összetett struktúrájú adatértéket tartalmaz; ekkor DB szinten is elérhetők a mező részelemei.

Összetett szelekciós feltétel: logikai operátorokkal összekötött szelekciós feltétel.

Összetett tulajdonság: olyan tulajdonság, mely tagokból áll össze; a lakcím mint összetett tulajdonság adható meg, mivel a város, utca, házszám elemekből áll elő.

P

Permanens adatok: időben állandó, hosszú idejű adattárolás.

Pointer (mutató): olyan adatelem, melynek értéke egy másik objektumra való hivatkozás, a másik objektum címe.

Pointer lánc: olyan struktúra, melyben az elemek egy láncolatot alkotnak, és a soron következő elemet egy pointer jelöli ki.

Pointer háló: olyan struktúra, melyben az elemek egy hálót alkotnak, és a szomszédos elemeket pointerek jelölik ki.

Pragmatika: az információ gyakorlati, tevékenységben megnyilvánuló jelentése.

PREPARE: dinamikus SQL parancsok esetén az SQL parancsnak egy szövegláncból való előkészítésére szolgál.

PRIMARY KEY: az elsődleges kulcs kijelölésére szolgáló integritási feltétel.

ProC: az Oracle rendszer beágyazott SQL felülete a C gazdanyelvi környezethez. A rendszer előfordítót és modulkönyvtárakat foglal magába.

Procedurális nyelv: olyan nyelv, melyben rendelkezésre állnak vezérlési elemek is a folyamatok algoritmusának leírására. Az SQL jelenlegi változata nem procedurális.

Projekció: azon relációs algebrai művelet, mely során a relációt leszűkítjük megadott attribútumaira.

R

Random rekordelérés: a rekordok közvetlenül elérhetők, beolvashatók anélkül, hogy az előtte lévő rekordokat is át kellene olvasni.

RDBMS: relációs adatbáziskezelő rendszer.

Redundancia: az adatelemek ismétlődése a relációkban. A redundancia jelenléte nagyobb helyigényt, lassúbb módosítást és ellentmondás veszélyt rejt magában. A lekérdezés hatékonysága és a megbízhatóság szempontjából viszont előnyös lehet a redundancia.

REFERENCES: az idegenkulcs integritási feltétel kijelölése; a hivatkozott táblát kell megadni a kulcsszó után.

Rekord: adatbázis struktúra elem, mely a logikailag összetartozó, és egységként kezelhető elemi adatértékek együttesét jelöli.

Rekordkulcs: a rekord előfordulást azonosító mezőcsoport, a kulcs értéke nem lehet azonos két különböző rekordban.

Rekordorientált: az adatok tárolása listába rendezett rekordokban történik, és az adatkezelő műveletek is a rekordlista elemein értelmezettek.

Rekurzív set: olyan set a hálós adatmodellben, melyben a tag és a tulajdonos elem is ugyanazon rekordtípushoz tartozik.

Reláció: az azonos szerkezetű rekord előfordulások névvel ellátott halmaza; tárolási egység a relációs adatbázisban.

Reláció fokszáma: relációsémához tartozó mezők darabszáma.

Relációs adatbáziskezelés: a relációs adatmodellen nyugvó adatkezelés.

Relációs adatstruktúra: a relációs adatmodell struktúra részét megadó szerkezeti elemek rendszere; legfontosabb alkotó eleme a reláció.

Relációs algebra: a relációkon értelmezett azon operátorok, műveletek összessége, melyek szintén relációt eredményeznek.

Relációs integritási feltételek: a relációs modellben értelmezett integritási feltételek rendszere; a feltétel lehet lokális vagy globális.

Relációs kalkulus: a relációkon értelmezett deszkriptív jellegű lekérdezési felület, melynél az eredmény reláció tulajdonságait kell megadni az elvégzendő műveletsor helyett.

Relációs műveletek: a relációkon értelmezett kezelő műveletek rendszere; a relációs műveletek operandusai és eredményei is relációk.

Reláció számossága: reláció előfordulásban tartalmazott rekordok darabszáma.

Reláció típus: a reláció megvalósulásának és kezelésének módját meghatározó típus. Lehet: bázis vagy alap, származtatott, ideiglenes.

REVOKE: a hozzáférési jogosultság visszavonása; a továbbadott jogok is automatikusan visszavonásra kerülnek.

Rissanen tétele: a normalizálás egyik alaptétele. A tétel alapján a független felbontás kritériuma, hogy az FD halmaz ne sérüljön és biztosított legyen az összekapcsolhatóság az egyes eredményrelációk között.

ROLLBACK: a tranzakció visszavonásának utasítása; hatására a tranzakció keretében elvégzett összes utasítás visszagörgetődik.

S

Safe kifejezés: olyan kifejezés a relációs kalkulusban, mely véges számosságú eredmény halmazt eredményez.

SELECT: az adatlekérdezés SQL operátora, a relációs algebra elemeit tartalmazza; az eredmény feldolgozásáról a kezelő programnak kell gondoskodnia.

SELECT INTO: a SELECT utasítás azon alakja, melyben a lekérdezés eredményét közvetlenül letesszük egy gazdanyelvi változóba. A fogadó gazdanyelvi változót az INTO kulcsszót követően kell megadni.

Séma: felhasználói séma; közös tulajdonoshoz tartozó objektumok névvel ellátott együttese. Alapesetben minden felhasználóhoz rendelődik egy séma, melynek neve megegyezik a felhasználó nevével.

Séma (schema): szerkezeti séma; az adatbázis elemek, objektumok szerkezetének leírása.

Séma-diagramm: az adatbázis szerkezetét, sémáját leíró diagramm, mely tartalmazza a rekordtípusokat és kapcsolataikat.

Semi join: a join művelet azon típusa, amikor az eredmény reláció egyik alapreláció szerinti projekcióját kapjuk eredményül.

SE piramis: a szoftverfejlesztés főbb fázisait és azok jellegét egy piramis struktúrában megadó ábrázolás.

SEQUEL: Structured English Query Language; a kísérleti System/R rendszerhez megvalósított relációs algebrai alapokon nyugvó parancsnyelv.

Set: a kapcsolatok tárolásának elemi leíró eszköze a hálós adatmodellben; a set az egy rekordtípusból, mint szülőből kiinduló egy-több kapcsolatokat tartalmazza, ahol egy rekordtípus több set-ben is előfordulhat.

Snapshot: származtatott tábla, mely mögött egy pillanatkép jellegű reláció áll; a snapshot tartalma fizikailag is tárolódik, de kapcsolódik hozzá egy származtatási műveletsor is, mely alapján bizonyos időközönként automatikusan frissítésre kerül.

Soros rekordelérés: a rekordok egymásutáni olvasásán alapuló állományhozzáférés.

Specializációs kapcsolat: két osztály vagy egyedtípus közötti kapcsolat, mely azt jelöli, hogy az egyik elem a másik specializációja.

SQL: Structured Query Language; a relációs adatbáziskezelők szabvány parancsnyelve, mely a relációs algebrán alapszik. Több, egyre bővülő ANSI szabványa is létezik: SQL86, SQL89, SQL92, SQL3.

SQLCA: a ProC kommunikációs területe az adatbázissal. A DBMS-ből jövő üzenetek ezen változóban tárolódnak. Elsősorban hibaüzenetek átadására szolgál.

Statikus elem: az adatmodell olyan elemei, melyek a perzisztens, a tárolásra vonatkozó részekre adnak leírást.

Statikus integritási elemek: az objektumok struktúráját, a felvett értékek körét szabályozó megkötések, leírások.

SZ

Származtatott reláció: olyan reláció, melynek értéke nem független a többi relációtól; lehet snapshot és view jellegű is.

Származtatott tulajdonság: olyan tulajdonság, melynek értéke nem független a többi tulajdonság értékétől, hanem azokból származtatható.

Szekvencia (sequence): bizonyos adatbáziskezelő rendszerekben az adatbázisban létrehozhatók globális számlálók, melyek felhasználhatók mesterséges sorszám típusú értékek, egyedi azonosítók generálására.

Szelekciós művelet: azon relációs algebrai művelet, mely során az alapreláció rekordjaiból csak bizonyos feltételnek eleget tevők kerülnek át az eredmény relációba.

Szelekciós join: a join művelet azon típusa, amikor csak azon rekordpárosok kerülnek át az eredmény relációba, melyekre egy megadott szelekciós feltétel igaz értéket ad eredményül.

Szemantika: az adatelemek jelentését kifejező vetülete az adatmodellnek.

Szemantikai adatmodell (SDM): az emberi szemlélethez közel álló adatmodell, melyben nem az egzaktság és a teljesség az elsődleges, hanem a jelentés.

Szemi-strukturált tárolás: az információ tárolása olyan formában, mely az adatszerű strukturált és a struktúra nélküli szöveges formátum között van; az információknak csak egy része kötött formátumú.

Szinguláris set: olyan set, melynek nincs normál tulajdonosa; célja egy rekordtípus előfordulásainak listába rendezése.

Szinonima: az adatmodell elemek elnevezéséhez kapcsolódó ellentmondás. Ugyanazon jelentésű elemekhez eltérő elnevezés kapcsolódik.

Szintaktika: az adatok leírásának, megjelenési formájának a szabályrendszere.

Szintézis: a relációséma tervezés azon módszere, amikor összegyűjtjük az egyes mezőket és az FD előfordulásokat, és ez alapján az azonos magból kiinduló FD-k és célmezők határozzák meg a relációk sémáját.

Szövegszerű tárolás: az adatok struktúra nélküli, szabad formátumú megadása.

Szuperkulcs: olyan mezőcsoport mely magába foglal egy jelöltkulcsot.

Szülő-gyerek kapcsolat (PCR): egy-több jellegű kapcsolat, melyben egy szülő szerepkörű egyedhez több gyerek szerepkörű egyed kapcsolódhat.

T

Tagrekord: a set-ben a tulajdonos rekord alatt elhelyezkedő rekord, amelynek minden előfordulásához egy tulajdonos rekord tartozik.

Tárolt eljárás, függvény: adatbázisban tárolt rutin, függvény. Használata az adatbázis objektumokhoz hasonlóan védhető. A tárolt eljárások révén hatékonyabbá tehető a kliens felől jövő összetett műveletsorok végrehajtása.

Tartalmazási kapcsolat: két osztály vagy egyedtípus közötti kapcsolat, mely azt jelöli, hogy az egyik típus a másikat mindig magába foglalja.

Többdimenziós indexfa: olyan indexfa, melyben az elemeknek több kulcsa is van, és több kulcs értéket is figyelembe kell venni a rekord elhelyezésekor és keresésekor.

Többértékű függőség: Multivalued Dependency, MVD; olyan függőségi fajta, amikor egy mezőhöz egy másik mezőcsoport több értéke is társítható, és a relációban minden lehetséges érték egyenrangúan jelenik meg.

Többértékű tulajdonság: olyan tulajdonság, mely egy típus több előfordulását is tartalmazhatja; a hobbi mint többértékű tulajdonság adható meg, mivel egy embernek több hobbija is lehet.

Törlési anomália: rekordok törlésekor jelentkező információvesztés; a kitörlendő információt leíró adatelemek mellett más adatelemek is törlésre kerülnek feleslegesen.

Tranzitív függőség: az FD azon fajtája, amikor egy A mezőcsoport egy C mezőcsoportot egy B mezőcsoporton keresztül határoz meg: $A \rightarrow B \rightarrow C$.

Trigger: olyan aktív adatbáziselem, mely lehetővé teszi, hogy bizonyos adatbáziskezelő műveletek hatására más megadott műveletek automatikusan végrehajtásra kerüljenek. A trigger megadásakor definiálni kell a kiváltó eseményt és a választe-vékenységet.

Triviális függőség: olyan FD, mely minden mezőcsoportnál teljesül. Alapja, hogy az egész mindig meghatározza bármely részét.

Tulajdonos rekord: a set-ben a szülő rekord.

Tulajdonság: az egyedeket jellemző, leíró információ elemek.

Túlcsordulás: a tárolásra kijelölt bucket megtelik, és az oda irányított rekordoknak más helyet kell keresni.

Tuple: a relációt alkotó érték n-es; egy tuple előfordulás egy rekord előfordulásnak felel meg.

Tuple kalkulus: TRC, Tuple Relational Calculus; a relációs kalkulus azon típusa, amikor a változók rekordokat reprezentálnak; ekkor a pont operátorral lehet hivatkozni az attribútumokra.

U

UIT: User Interface Tools; a DBMS-hez integráltan tartozó segédprogramok angol rövidítése.

UML: Unified Modelling Language; általános célú, szabvány objektum orientált modellezési nyelv.

Unió: azon relációs algebrai művelet, mely két azonos szerkezetű reláció rekordjainak unióját adja eredményül.

UNIQUE: egyediséget előíró integritási feltétel; a táblában a megadott mezőnél minden mezőérték csak egyszer szerepelhet.

UPDATE: SQL parancs a rekordok tartalmának módosítására; egy szelekciós részben lehet kijelölni a módosítandó rekordokat.

Utasításértelmező: a DBMS azon része, mely a bejövő utasításokat szintaktikai és szemantikai oldalról is feldolgozza a végrehajtás céljából.

V

Veszteségmentes dekompozíció: olyan felbontás, mely során az induló reláció előállítható az eredményrelációkból adatvesztés nélkül a join művelet segítségével.

View: reláció típus, mely mögött egy virtuális reláció áll; a view (nézeti tábla) tartalma fizikailag nem tárolódik, de kapcsolódik hozzá egy származtatási műveletsor, mely alapján hivatkozáskor automatikusan meghatározásra kerül az aktuális tartalom.

Virtuális mező: olyan mező, mely mögött egy származtatott tulajdonság áll, ezért nem kerül fizikailag letárolásra; értéke hivatkozáskor határozódik meg.

Virtuális szülő-gyerek kapcsolat (VPCR): olyan egy-több jellegű kapcsolat a hierarchikus adatmodellben, mely nem a hierarchia szerinti szülő rekordra mutat; segítségével lehet egy rekordnak több szülőtípusa is.

VLDB: Very Large DataBases, nagyon nagy adatbázisok; egy szervezetet is jelöl, mely az adatbáziskezelés kutatási területeit széles körűen átfogja.

W

Well formed formula (wff): a helyes alakú kifejezés olyan kifejezés a relációs kalkulusban, mely megadott lépéseken keresztül állítható elő és egyértelműen kiértékelhető.

WHENEVER: automatikus hibakezelési lehetőség a ProC rendszerben. Az előfordító dolgozza fel. Az ezt követő SQL parancsok után automatikusan beszúrásra kerül a megfelelő hibakezelő programrész.

Tárgymutató

Ábrák jegyzéke

Irodalomjegyzék

Magyar nyelvű tankönyvek

[1] Stolnicki, Gy.: SQL kézikönyv, Computer Books, 1994, ISBN 9636180008

[2] Halassy, B.: Az adatbázistervezés alapjai és titkai, IDG, 1994, ISBN 9638287012

[3] Halassy, B.: Ember - információ - rendszer, IDG, 1996, ISBN 9638287039

[4] Quittner, P.: Adatbázis-kezelés a gyakorlatban, Akadémiai kiadó, 1993, ISBN 9630565870

[5] Arató, I. - Schwarzenberger, I.: Információs rendszerek szervezési módszertana, ComputerBooks, 1993, ISBN 9637642749

[6] Rolland, F.: Adatbázis rendszerek, Panem kiadó, 2002, ISBN 9635453485

[7] Tringer - Fodor: Adatbázis-kezelés, Kossuth , 1999, ISBN 9630940930

[8] Bálint, D.: Adatbázis-kezelés, Talentum, 1999, ISBN 9630348497

[9] Bártfai - Budavári: Adatbázis-kezelés, BBS-Info Kft., 2000, ISBN 9630034441

[10] Gazsó Z.: Adatbázis-kezelés FoxPro-ban - 2.5, 2.6, ComputerBooks, 1999

[11] Kende M. - Kotsis D. - Nagy: Adatbáziskezelés az Oracle rendszerben, Panem, 2002

[12] Szelezsán J.: Adatbázisok, LSI, 1999

[13] K. Fitus I.: Adatbázisok példatár, LSI, 1999

[14] Ullman - Widom: Adatbázisrendszerek, Panem, 1999

[15] Garcia - Jeffrey - Ullman - Widom: Adatbázisrendszerek megvalósítása, Panem, 2001

[16] Harmon: Delphi/Kylix alapú adatbázis-kezelés, Kiskapu Kft., 2002

[17] Gazsó Z.: Visual adatbázis-kezelők objektum-orientált programozása, ComputerBooks, 1999

[18] Váradi Zs.: Adatbázis kezelési ismeretek, Műszaki könyvkiadó, 2003

[19] Ensor - Stevenson: Oracle tervezés, Kossuth, 2002

Idegen nyelvű tankönyvek

[20] Date, C.J.: Database Systems, Addison-Wesley, 1995, ISBN 0201824582

[21] Parsaye, K. - Chignell, M. - Khoshafian, S. - Wong, H.: Intelligent Databases, Wiley, 1989, ISBN 0471503452

[22] Blieberger, J. - Schildt, G. - Schmid, U. - Stöckler, S.: Informatik, Springer, 1990, ISBN 3211823891

[23] Lusardi, F.: SQL Programmieren in Datenbanken, MCGraw-Hill, 1989, ISBN 38920281664

[24] Hughes, J.: Database Technology, Prentice Hall, 1988, ISBN 013197906

[25] Courtney, J. - Paradice, D.: Database Systems for Managament, Times Mirror, 1988, ISBN 0801611806

[26] Elmasri - Navathe: Fundamentals of Database Systems, The Benjamin/Cummings Publisher

[27] Özsu, T. - Valduriez, P.: Principles of Distributed Database Systems, Prentice Hall, 1991, ISBN 0137156812

[28] Date, C.J.: Relational Database Writings 1989-1991, Addison-Wesley, 1992

[29] Heuer, A.: Objektorientierte Datenbanken, Konzepte, Modelle, Systeme, Addison-Wesley, 1992, ISBN 3893193154

[30] Castano-Fugini-Martella-Samarati: Database Security, Addison-Wesley, 1995

[31] Kroenke: Database Processing: Fundamentals, Design & Implementation, MacMillin, 1994, ISBN 0023668814

[32] Silberschatz, K.: Database System Concepts, McGraw Hill, 1991, ISBN 0070447543

[33] Korth: Database System Concepts, McGraw Hill, 1997, ISBN 007044756

[34] Ramarkrishnan, R.: Database Management Systems, McGraw Hill, 1997, ISBN 0070507759

[35] Wiederhold, G.: Database Design, McGraw Hill, 1983, ISBN 0070701326

[36] Post, G.: Database Management Systems, McGraw Hill, 1999, ISBN 0071166777

[37] Houlette: SQL: A beginners guide, McGraw Hill, 2000, ISBN 0072130962

[38] Houlette: Troubelshooting SQL, McGraw Hill, 2001, ISBN 0072134895

[39] Seideman, C.: The Complete Reference mySQL, Osborne, 2003, ISBN 0072224770

[40] Groff: SQL The Complete Reference, Osborne, 2002, ISBN 0072225599

[41] Allen: Introduction to Relational Databases & SQL Programming, Osborne, 2003, ISBN 0072229241

[42] Post, G.: Database Management Systems: Designing and Building Business Application, McGraw Hill, 1998, ISBN 0072898933

[43] Atzeni, P.: Database Systems: Concepts, Languages and Architectures, McGraw Hill, 1999, ISBN 0077095006

[44] Abbey: Oracle8 A Beginner's Guide, McGraw Hill, 1997, ISBN 0078823935

[45] Harrington, L.: SQL clearly explained, Morgan Kaufmann, 1998, ISBN 012326426

[46] Simovoci, D.: Relational Database Systems, Academic Press, 1995, ISBN 0126443750

[47] Dietrich, S.: Understanding Relational Database Query Languages, Prentice Hall, 1991, ISBN 0130286524

[48] Ullman, J.: Database Systems: Complete Book, Prentice Hall, 2001, ISBN 0130319953

[49] Ullman, J.: First Course in Database Systems, Prentice Hall, 2001, ISBN 0130353000

[50] Patrick, J.: SQL Fundamentals, Prentice Hall, 2002, ISBN 0130669474

[51] Hernandez, M.: Complete SQL Training Course, Prentice Hall, 2000, ISBN 0130897272

[52] Holden, P.: ECDL Advanced Databases, Prentice Hall, 2003, ISBN 0131202405

[53] Hansen, G.: Database Management and Design, Prentice Hall, 1995, ISBN 0133088006

[54] Rolland, F.: Essence of Databases, Prentice Hall, 1997, ISBN 0137278276

[55] Johnson, J.: Database: Models, Languages, Design, Oxford University Press, 1997, ISBN 0195107837

[56] Connolly: Database Systems, Addison-Wesley, 1995, ISBN 0201422778

[57] Pascal, F.: Practical Concepts, Principles and Methods for Database Management, Addison-Wesley, 2000, ISBN 0201485559

[58] Lans, V.: SQL Guide for Oracle, Addison-Wesley, 1992, ISBN 0201565455

[59] Date, J.: Database Relational Model: A Retrospective Review and Analysis, Addsion-Wesley, 2001, ISBN 0201612941

[60] Hernandez: Database Design for Mere Mortals, Addison-Wesley, 1996, ISBN 0201694719

[61] Carlis - Maguire: Mastering Data Modeling, Addison-Wesley, 2000

[62] Connolly - Begg: Database Systems Practical Approach to Design, Implementation and Management, Addison-Wesley, 2001, ISBN 0201708574

[63] Date - Darwen: Guide to SQL Standard, Addison-Wesley, 1996, ISBN 0201964260

[64] Richard: Data Management: Databases and Organizations, John Wiley, 2003, ISBN 0471347116

[65] Stair: Fundamentals of Information Systems, Course Technology, 2001, ISBN 0619034165

[66] Melton - Simon: Understanding the new SQL - A complete guide

[67] Oracle server: SQL Language Reference Manual, Oracle Corporation

[68] Oracle server: Concepts Manual, Oracle Corporation

Webes források

Idegen nyelvű szakkönyvek
www.softwarestation.hu

Magyar nyelvű könyvek
www.kiskapu.hu

Hasznos link gyűjtemény
www-ccs.cs.umass.edu/db.html

Termékek
www.oracle.com
www.torolab.ibm.com/data/db2
www.sybase.com
www.microsoft.com/sql

Ingyenes termékek
www.postgresql.org
www.mysql.com

On-line folyóiratok
www.dbmsmag.com
www.dbpd.com

Hírcsoport
Comp.databases