9.11. Irodalomjegyzék

A [6] könyv az ütemezésről, valamint a zárolásról nyújt lényeges forrásanyagot. A [3] ugyanennek egy másik fontos forrása. A konkurenciavezérlés legújabb eredményei a [12]-ben és a [11]-ben találhatók.

Valószínűleg a legjelentősebb cikk a tranzakciófeldolgozásban a [4] a kétfázisú zárolásról. A figyelmeztető protokoll a szemcsézettség hierarchiákra az [5]-ből származik. A fák nem kétfázisú zárolása a [10]-ből ered. A kompatibilitási mátrixot a zárolási módok tanulmányozására a [7]-ben vezették be.

Az időbélyegzők mint konkurenciavezérlési módszerek a [2]-ben és az [1]-ben fordulnak elő. Az érvényesítésen alapuló ütemezés a [8]-ból származik. A többszörös változatok használatát a [9]-ben tanulmányozták.

1. P. A. Bernstein, Goodman, N., „Timestamp-based algorithms for concurrency controll in distributed database systems", *Proc. Intl. Conf. on Very Large Databases* (1980), pp. 285–300.
2. P. A. Bernstein, Goodman, N., Rothnie, J. B. Jr., Papadimitriou, C. H., „Analysis of serializability in SDD-1: a system of distributed databases (the fully redundant case)", *IEEE Trans. on Software Engineering* **SE-4**:3 (1978), pp. 154–168.
3. P. A. Bernstein, Hadzilacos, V., Goodman, N., *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, Reading MA, 1987.
4. Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, I. L., „The notions of consistency and predicate locks in a database system", *Comm. ACM* **19**:11 (1976), pp. 624–633.
5. Gray, J. N, Putzolo, F., Traiger, I. L, „Granularity of locks and degrees of consistency in a shared data base", in Nijssen, G. M. (ed.), *Modeling in Data Base Management Systems,* North Holland, Amsterdam, 1976.
6. Gray, J. N, Reuter, A., *Transaction Processing: Concepts and Techniques,* Morgan-Kaufmann, San Francisco, 1993.
7. Korth, H. F., „Locking primitives in a database system", *J. ACM* **30**:1, (1983), pp. 55–79.
8. Kung, H.-T., Robinson, J. T., „Optimistic concurrency control", *ACM Trans. on Database Systems* **6**:2 (1981), pp. 312–326.
9. Papadimitriou, C. H., Kanellakis, P. C., „On concurrency control by multiple versions", *ACM Trans. on Database Systems* **9**:1 (1984), pp. 89–99.
10. Silberschatz, A., Kedem, Z., „Consistency in hierarchical database systems", *J. ACM* **27**:1 (1980), pp. 72–80.
11. Thompsian, A., „Concurrency control: methods, performance, and analysis", *Computing Surveys* **30**:1 (1998), pp. 170–231.
12. Thuraisingham, B., Ko, H.-P., „Concurrency control in trusted database management systems: a survey", *SIGMOD Record* **22**:4 (1993), pp. 52–60.

10. fejezet

Bővebben a tranzakciókezelésről

Ebben a fejezetben a tranzakciókezelés olyan kérdéseiről lesz szó, amelyekkel a 8. és a 9. fejezetben nem foglalkoztunk. Megnézzük, hogyan egyeztethető össze az előző két fejezet nézőpontja: milyen kölcsönhatásban áll egymással a helyreállítás, a tranzakciók abortálthatóságának és a sorbarendezhetőség fenntartásának a szükségessége? Ezután megtárgyaljuk a tranzakciók közötti holtpontkezelés lehetőségeit. Holtpont tipikusan abban az esetben alakul ki, amikor több tranzakciónak kell várnia egy olyan erőforrásra (például egy zárra), amely az adott pillanatban egy másik tranzakció birtokában van.

A fejezet során bevezetést nyerünk az osztott adatbázisok világába is. Közelebbről megvizsgáljuk az esetleg többszörözött példányok segítségével megosztott adatok zárolási problémáját. Azt a kérdést is áttekintjük, hogy mi alapján lehet dönteni egy olyan tranzakció abortálásáról, illetve véglegesítéséről, amely egyszerre több helyszínen is végez műveletet.

Végül tárgyalásra kerülnek a „hosszú tranzakciókból" eredő problémák. Léteznek olyan alkalmazások, például CAD[1] vagy „munkafolyamat" rendszerek, amelyekben az emberi és a számítógépes eljárások akár több napon keresztül is kölcsönhatásban vannak egymással. Hasonlóan a rövid tranzakciós rendszerekhez (banki műveletek, repülőjegy-foglalás) itt is szükség van az adatbázis-állapot konzisztenciájának a megőrzésére. A 9. fejezetben bevezetett konkurenciavezérlő módszerek azonban nem működnek ésszerűen, amikor a zárak napokra vannak kiosztva, vagy az érvényesítési döntéseket több nappal ezelőtt végbement események kapcsán kell meghoznunk.

10.1. Tranzakciók, melyek nem véglegesített adatokat olvasnak

A 8. fejezetben néhány naplózási eljárással és ezeknek a rendszerhiba utáni helyreállításban betöltött szerepével ismerkedtünk meg. Az adatbázison végrehajtott számításokat olyan folyamatoknak tekintettük, amely során az értékek a nem felejtő lemez, a

[1] *Computer Aided Design,* magyarul Számítógéppel Támogatott Tervezés. *A fordító megjegyzése.*

felejtő központi memória és a tranzakció memóriaterülete között mozognak. A különféle naplózási módszerek segítségével egy esetleges rendszerhiba esetén biztosan helyreállíthatók voltak a véglegesített tranzakciók műveletei és ezek hatása az adatbázislemezen tárolt példányán. A naplózási rendszerek azonban nem tesznek kísérletet a sorbarendezhetőség támogatására; az adatbázis állapotát még akkor is visszaállítják az eredeti formájára, ha az nem sorba rendezhető műveletek eredményeképpen jött létre. Ami azt illeti, a forgalomban lévő adatbázisrendszerek sem mindig ragaszkodnak a sorbarendezhetőséghez, és néhányuk csak a felhasználó kifejezett kérésére tesz erre irányuló törekvéseket.

A 9. fejezetben viszont egyedül a sorbarendezhetőségről volt szó. A fejezet alapelvei szerint tervezett ütemező olyan dolgokat művelhet, amelyeket a naplóvezérlő nem tűrhet el. Például a sorbarendezhetőség definíciójában semmi sem gátolja meg a tranzakciót abban, hogy a véglegesítés előtt a megfelelő zár birtokában A-nak új értéket adjon az adatbázisban, és ezzel megsértse a naplózási politikát. Még rosszabb esetben a tranzakció az adatbázisba írás után abortál, ami könnyen eredményezhet inkonzisztens állapotot még rendszerhiba nélkül is, annak ellenére, hogy az ütemező elméletileg támogatja a sorbarendezhetőséget.

10.1.1. A piszkos adat probléma

Már volt arról szó, hogy az adatot „piszkosnak" nevezzük, ha egy olyan tranzakció írta, amely még nem fejeződött be. Piszkos adat feltűnhet a pufferben, a lemezen vagy mindkét helyen; ezek közül bármelyik problémát okozhat.

10.1. példa: Vegyük újra a 9.13. ábra sorba rendezhető ütemezését, de tegyük fel, hogy T_1 a B olvasása után valamilyen oknál fogva abortál. Ekkor a 10.1. ábrán látható események követik egymást. T_1 abortáltatása után az ütemező feloldja a B-re T_1-nek kiosztott zárat. Ez a lépés elkerülhetetlen, mert különben egyik tranzakció sem zárolhatná B-t.

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$;	250	
	$l_2(B)$ **Elutasítva**		
$r_1(B)$;			
Abort; $u_1(B)$;			
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		50

10.1. ábra. *T_1 piszkos adatot ír, majd abortál*

T_2 azonban most olyan adatot olvasott, amely egy inkonzisztens adatbázisállapothoz tartozik. Azaz T_2 a T_1 által megváltoztatott A értéket és a T_1 működése előtti B-t olvasta. Ebben az esetben nem számít, hogy a T_1 által írt $A = 125$ érték kikerült-e a lemezre vagy sem, T_2 úgyis a pufferből kapja az értékeket. Az inkonzisztens állapot olvasása következtében T_2 inkonzisztens állapotban hagyja az adatbázist (a lemezen), ahol $A \neq B$.

A 10.1. ábra ütemezésében az okozza a problémát, hogy a T_1 által írt A piszkos, akár a pufferben van, akár a lemezen. Az a tény, hogy T_2 olvasta és a saját számításaihoz felhasználta A-t, megkérdőjelezi a tranzakció helyes működését. A 10.1.2. részben látni fogjuk, hogy ha ilyen eset előfordulhat, akkor T_2-t és T_1-et is abortáltatni kell, és vissza kell görgetni (rollback). Ha még azt is megengedjük, hogy a T_1 által írt A érték a lemezen is megjelenjen, az elég sokba kerül, hiszen a változtatás meg nem történtté tevéséhez a naplót kell felhasználnunk. Ezért olyan szabályokat kell kidolgoznunk, amelyek segítségével megelőzhető, hogy a piszkos adat kikerüljön a lemezre. □

10.2. példa: Most nézzük a 10.2. ábrán látható eseménysorozatot, amely a 9.8. részben megismert időbélyegző alapú ütemező felügyelete alatt játszódik le. Tegyük fel azonban, hogy ez az ütemező nem használja a 9.8.1. részben bevezetett véglegesítés bitet. Abban a részben láthattuk, hogy ennek a bitnek a segítségével megelőzhető, hogy egy tranzakció olyan adatot olvasson, amelyet egy még folyamatban lévő másik tranzakció írt. Így, amikor a második lépésben T_1 B-t olvassa, nincs véglegesítés bit ellenőrzés, ami késleltethetné T_1-et. T_1 továbbléphet, akár a lemezre is írhat, és véglegessé válhat. B olvasása utáni műveleteit nem tüntettük fel az ábrán.

Végül T_2 fizikailag nem megvalósítható módon próbálja C-t olvasni, ezért abortál. T_2-nek a B-n végrehajtott változtatásait érvénytelenítjük, B értékét és írási idejét visszaállítjuk arra, amit T_2 felülírt. Mindezek ellenére T_1 hozzáférhetett B érvénytelen értékéhez, és ezt bármire felhasználhatja, például új értékeket számolhat A-nak, B-nek és/vagy C-nek, és ezeket kiírhatja a lemezre. Vagyis mivel T_1 B piszkos értékét olvasta, inkonzisztens állapotba hozta az adatbázist. Vegyük észre, hogyha használtuk volna a véglegesítés bitet, akkor a 2. lépésben B olvasását késleltethettük volna egészen addig, amíg T_2 nem abortál, és az általa írt B értéket vissza nem állítjuk az előző (feltehetően véglegesített) értékére. □

T_1	T_2	T_3	A	B	C
200	150	175	RT = 0	RT = 0	RT = 0
			WT = 0	WT = 0	WT = 0
	$w_2(B)$;			WT = 150	
$r_1(B)$;					
	$r_2(A)$;		RT=150		
		$r_3(C)$;			RT = 175
	$w_2(C)$;				
	Abort;			WT = 0	
		$w_3(A)$;	WT = 175		

10.2. ábra. *T_1 a T_2-ből olvas piszkos adatot, ezért T_2-vel együtt őt is abortáltatni kell*

Az elkülönítés szintjei SQL-ben

Az SQL2-szabvány nem teszi fel, hogy a tranzakciók sorba rendezhető módon futnak, ennek megfelelően a forgalomban lévő rendszerek a felhasználóra bízzák a kívánt konkurenciavezérlési szint beállítását. SQL2-ben a legmagasabb „elkülönítési szint" a *sorba rendezhető* (serializable), amely pontosan azt jelenti, amit a szó takar: ezen a szinten a tranzakciónak úgy kell lefutnia, mintha egy pillanat alatt történt volna, és mintha minden más tranzakció teljes egészében vagy előtte, vagy utána ment volna végbe.

Az „olvasásbiztos" elkülönítési szint nem követeli meg a sorbarendezhetőséget, viszont megtiltja a piszkos adat olvasását. Lehetséges azonban, hogy ha ezen a szinten egy T tranzakció kétszer olvassa A-t, akkor két különböző értéket kap, amelyeket két különböző véglegesített tranzakció írt. Ehhez kapcsolódik a „megismételhető olvasás" szintje, amely egy kicsit erősebb feltételt szab: nemcsak hogy ha T olvassa A-t, akkor A nem lehet piszkos, de ha A egy reláció, akkor A minden további olvasásánál csak bővebb részhalmazt kaphatunk; vagyis amíg T aktív, A-nak semmilyen része sem tűnhet el.

A negyedik, „nem olvasásbiztos" szinten semmilyen megszorítás sincs arra vonatkozólag, hogy a tranzakció milyen adatokhoz férhet hozzá. Csak ez a szint engedi meg a piszkos adat olvasását.

10.1.2. Továbbgyűrűző visszagörgetés

A fenti példákból látszik, hogy ha a tranzakciók hozzáférhetnek piszkos adatokhoz, akkor néha végre kell hajtanunk egy *továbbgyűrűző visszagörgetést* (cascading rollback). Ez azt jelenti, hogy amikor T abortál, meg kell határoznunk, hogy milyen tranzakciók olvasták a T által írt adatokat, ezeket abortáltatnunk kell, és rekurzívan az összes olyan tranzakciót is, amelyek az ezek által írt adatokat olvasták. Tehát meg kell találnunk minden olyan U tranzakciót, amely a T által írt piszkos adato(ka)t olvasta, abortáltatnunk kell U-t, majd meg kell találnunk minden olyan V tranzakciót, amely az U által írt piszkos adatot olvasta, abortáltatnunk kell V-t és így tovább. Hogy az abortáltatott tranzakció hatását érvénytelenítsük az adatbázison, használhatjuk a naplót, ha megtalálhatók benne a változók előző értékei (semmisségi vagy semmisségi/helyrehozó naplózás). Ha a piszkos adat hatása még nem érte el a lemezt, akkor az adatok helyreállításához az adatbázis lemezen található változata is megfelelő alapot nyújt. Ezekről a módszerekről részletesebben a következő részben lesz szó.

Mint azt már megjegyeztük, az időbélyegző alapú ütemező a véglegesítés bit használatával megakadályozza az olyan tranzakció továbblépését, amely piszkos adatot olvasott, azaz ebben az esetben nem lesz szükség visszagörgetésre. Az érvényesítés alapú ütemező is elkerüli a továbbgyűrűző visszagörgetést, hiszen az adatbázisba írás (sőt már a pufferbe írás is) csak azután történik meg, miután az ütemező megállapította, hogy a tranzakció végleges lesz.

10.1.3. A visszagörgetés kezelése

Most nézzük meg, hogyan kezelhető a továbbgyűrűző visszagörgetés problémája a zárolás alapú ütemezők esetén. Egyszerűen biztosítható, hogy ne legyen szükség visszagörgetésre:

- *Szigorú zárolás:* A tranzakció egészen addig nem engedheti el az írás zárjait (illetve az olyan típusú zárjait, például növelési zár, amelyek birtokában megváltoztathatja az adat értékét), amíg nem lett végleges vagy nem abortált, és az ennek megfelelő naplóbejegyzés nem került ki a lemezre.

Vegyük észre, hogy a feltétel, amely szerint a tranzakció befejezésére vonatkozó feljegyzés lemezre írása megelőzi a zárakról való lemondást, biztosítja a következőt: ha rendszerhiba következik be, miután a T tranzakció eleresztette a zárjait, és egy másik tranzakció olvasta a T által írt adatokat, akkor ezek az adatok nem lehetnek piszkosak amiatt, hogy a helyreállító eljárás abortáltatja T-t. A tranzakciók olyan ütemezését, amely megfelel a szigorú zárolás szabályának, *helyreállíthatónak* (recoverable) nevezzük.

Világos, hogy egy helyreállítható ütemezésben a tranzakciók nem olvashatnak piszkos adatot, hiszen egy még folyamatban lévő tranzakció által pufferbe írt adat egészen addig zár alatt marad, amíg a tranzakció be nem fejeződik. A pufferben lévő adatok helyreállítása azonban még mindig problémát jelent abban az esetben, ha a tranzakció abortál, hiszen a végrehajtott változtatások hatását érvényteleníteni kell. Hogy ez mennyire bonyolult, az attól is függ, hogy az adatbáziselemek blokk méretűek vagy annál kisebbek. Most mindkét esetet megvizsgáljuk.

Blokkok visszagörgetése

Ha a zárolható adatbáziselemek blokkok, akkor létezik a visszagörgetésnek egy egyszerű módja, amelyhez nem kell a naplót felhasználnunk. Tegyük fel, hogy T tranzakció rendelkezik az A blokk kizárólagos zárjával. A új értékét beírta a pufferbe, majd abortálnia kellett. Mivel A zárolva volt, mióta T felülírta, más tranzakció nem férhetett hozzá az értékéhez. Könnyű visszaállítani A régi értékét, ha a következő szabály szerint járunk el:

- A még folyamatban lévő tranzakciók által írt blokkok a központi memóriában vannak csatolva, ami azt jelenti, hogy ezeket a puffereket nem írhatjuk ki a lemezre.

Ilyenkor T abortálása után a „visszagörgetés" annyiból áll, hogy utasítjuk a pufferkezelőt, hogy ne vegye figyelembe A értékét. Vagyis az A által elfoglalt puffer nem íródik ki sehova, hanem az elérhető „szabad" pufferek közé kerül. Biztosak lehetünk benne, hogy A értéke a lemezen a legutóbbi végleges tranzakció eredménye, amely pontosan az, amit szeretnénk.

Ha a 9.8.5. és 9.8.6. részben megismert többverziós időbélyegzős rendszert használjuk, akkor is könnyű dolgunk van a visszagörgetéssel. Ebben az esetben is fel kell tennünk, hogy a még folyamatban lévő tranzakciók által írt blokkok a memóriában csatolva vannak. Ekkor az A értékeit tartalmazó listából egyszerűen töröljük azt, amely T hatására jött létre. Vegyük észre, hogy mivel T értékváltoztató tranzakció volt, ezért az általa írt A érték egészen addig el volt zárva a többi tranzakció elől, amíg T nem abortált (feltéve, hogy a 9.8.6. rész időbélyegző/zár sémáját használtuk).

Kis adatbáziselemek visszagörgetése

Amikor a zárolható adatbáziselemek nem egész blokkok, csak blokkrészletek (például sorok vagy objektumok), akkor az abortált tranzakciók által megváltoztatott pufferek helyreállítása egyszerű módszerrel nem oldható meg. Az a probléma ugyanis, hogy egy pufferben több adatelem is megtalálható, és ezeket két vagy több tranzakció is írhatja, vagyis ha ezek közül az egyik abortál, a többi által véghezvitt változtatásokat még meg kell őriznünk. Egy abortált tranzakció által írt kis A adatbáziselem régi értékének visszaállítására több lehetőségünk is van.

1. A eredeti értékét a pufferben a lemezen található adatbázis alapján állítjuk vissza.
2. Semmisségi vagy semmisségi/helyrehozó naplózás esetén A előző értékét a napló alapján állítjuk vissza.
3. Készíthetünk minden tranzakcióhoz a központi memóriában egy külön naplót arra az időre, amíg a tranzakció aktív. Ebbe a tranzakciók által végrehajtott változtatásokat jegyezzük fel, tehát A régi értéke is megtalálható a megfelelő „naplóban".

Egyik módszer sem ideális. Az elsőben biztosan a lemezen tárolt információhoz kell fordulnunk. A második esetben, ha a napló megfelelő része még mindig a pufferben van, akkor lehet, hogy nem kell a lemezhez fordulnunk. Rosszabb esetben azonban alaposan át kell vizsgálnunk a lemezen tárolt naplót, hogy az A-ra vonatkozó bejegyzést megtaláljuk. Az utolsó módszerrel nem kell a lemezről olvasnunk, viszont a központi memória „naplók" sok helyet foglalhatnak a memóriában.

10.1.4. Csoportos véglegesítés

Bizonyos körülmények között akkor is elkerülhető a piszkos adat olvasása, ha nem írjuk ki azonnal a lemezre a naplóban található commit rekordokat. Ha a naplóbejegyzéseket abban a sorrendben írjuk ki a lemezre, ahogy a naplóban szerepelnek, akkor amint a napló pufferben lévő részébe kerül egy commit rekord, feloldhatjuk a megfelelő zárakat.

10.3. példa: Tegyük fel, hogy T_1 tranzakció az X írása után véglegessé válik, kiírja a commit rekordját a naplóba, de ez a pufferben marad. Bár T_1 nem végleges abban az értelemben, hogy a commit rekordja túlélne egy rendszerhibát, mégis feloldjuk a zárjait. Ezután T_2 olvassa X-et és „véglegessé válik", de a commit rekord, amely T_1 commit rekordját követi a naplóban, szintén a pufferben marad. Mivel a naplóbejegyzések az eredeti sorrendben kerülnek ki a lemezre, a helyreállítás-kezelő csak akkor foghatja fel T_2-t végleges tranzakciónak (a lemezen talált commit rekord következtében), ha T_1-et is véglegesként kezeli. Így a helyreállítás-kezelő szempontjából három eset különböztethető meg:

Mikor végleges igazán egy tranzakció?

A csoportos véglegesítés kifinomultságáról eszünkbe juthat, hogy egy lezárult tranzakció több különböző állapotban lehet a műveletei befejezése és a valódi „végleges" állapota között, ahol ez utóbbi azt jelenti, hogy a tranzakció hatása semmilyen körülmények között, még a rendszer összeomlása esetén sem vész el. Ahogy a 8. fejezetben megjegyeztük, lehetséges, hogy egy tranzakció véget ér, és még a `COMMIT` feljegyzés is bekerül a központi memória egy pufferébe, de ha ez nem éri el a lemezt, rendszerhiba esetén mégis elvész a tranzakció hatása. Sőt a 8.5. részben azt láttuk, hogy ha a `COMMIT` rekord már el is érte a lemezt, de mentés még nem készült róla, az adathordozó hibája esetén a tranzakció ugyanúgy semmissé válhat, és elveszheti a hatását.

Ha a hiba lehetőségét kizárjuk, ezek az állapotok mind ekvivalensek abban az értelemben, hogy minden tranzakció a lezárulását követően biztosan eléri a tartósság szintjét, azaz még az adathordozó hibája esetén sem vész el a hatása. Ha azonban a hibákat és ezek helyreállítását is számításba kell vennünk, fontos felismernünk a fenti állapotok közti különbséget, amelyekre egyébként mindre „véglegesként" hivatkozhatnánk.

1. Sem T_1, sem T_2 commit rekordja nem kerül ki a lemezre. Ekkor a helyreállítás-kezelő mindkettőt abortáltatja, és az a tény, hogy T_2 a le nem zárult T_1 által írt X-et olvasta, lényegtelen.
2. T_1 végleges, T_2 nem. Két okból kifolyólag sincs probléma: T_2 tranzakció X-et egy véglegesített tranzakció után olvasta, vagyis X nem volt piszkos, másrészt T_2-t egyébként is abortáltatjuk az adatbázisra való hatás nélkül.
3. Mindkettő végleges. Ekkor T_2 nem piszkos adatot olvasott.

Ezzel szemben tegyük fel, hogy a T_2 commit rekordját tartalmazó puffer kikerült a lemezre (például, mert a pufferkezelő ezt a puffert valami másra akarta használni), T_1 commit rekordja viszont nem. Ha ezen a ponton a rendszer összeomlik, a helyreállítás-kezelő számára úgy fog tűnni, mintha T_2 véglegesített lett volna, T_1 pedig nem. T_2 hatása végleges marad az adatbázison, annak ellenére, hogy ez az X piszkos adat olvasásán alapult. □

Konkurencia és helyreállíthatóság

Talán feltűnt, hogy a 8. fejezetben megismert három naplózási módszer nem mindig felel meg a helyreállítható ütemezés követelményeinek. Az olyan rendszereket, amelyek mind a konkurencia, mind a helyreállíthatóság elvárásainak megfelelnek, gyakran nevezzük ACID- (atomosság, konzisztencia, elkülönítés, tartósság) rendszereknek (lásd az 1.2.4. rész idevágó részeit). Itt a megfelelő alkalom tehát, hogy megjegyezzük: a naplózás és a zárolás (vagy akármilyen más konkurenciavezérlő módszer) egymás ellen ható mechanizmusok.

Például lehet egy olyan rendszerünk, amely naplózza a tranzakciók műveleteit, hogy biztosítsa az atomosságukat, még akkor is, ha semmi szükségünk a sorbarendezhetőségre vagy a piszkos adatok olvasásának az elkerülésére. Ami ezt illeti, a forgalomban lévő rendszerek gyakran a felhasználóra bízzák, hogy igényli-e a tranzakciók sorbarendezhetőségét, illetve a piszkos olvasás megakadályozását; ahogy ezt a 10.1.1. részben az SQL2 „elkülönítési szintjeit" tárgyaló dobozban is említettük. Megfordítva, a megszokott ABKR-ekben lehetőség van a naplózás kikapcsolására, de ezzel együtt az összes vagy néhány tranzakció futása sorba rendezhető. Ennek az a következménye, hogy egy esetleges rendszerhiba esetén a sorbarendezhetőség és az adatbázis konzisztenciája nincs biztosítva, de a felhasználónak esetleg más módszerei is lehetnek a konzisztencia visszaállítására.

A 10.3. példa alapján levonhatjuk azt a következtetést, hogy a zárakat még azelőtt feloldhatjuk, mielőtt a tranzakció commit rekordja a lemezen megjelenne. Ez az eljárás, amit gyakran *csoportos véglegesítésnek* (group commit) nevezünk, a következőkben foglalható össze:

- Ne oldjuk fel a zárakat, amíg a tranzakció be nem fejeződik, és amíg a commit naplóbejegyzés legalább a pufferben fel nem tűnik.
- A naplóblokkokat abban a sorrendben írjuk ki a lemezre, amelyben keletkeztek.

A csoportos véglegesítés elve éppúgy, mint a 10.1.3. részben tárgyalt „helyreállítható ütemezés" elve, biztosítja, hogy piszkos adat olvasására soha ne kerüljön sor.

10.1.5. Logikai naplózás

A 10.1.3. részben láthattuk, hogy a „piszkos olvasásokat" könnyebb megelőzni, illetve ha az olvasás már megtörtént, akkor könnyebb kijavítani abban az esetben, ha a zárolható egységek blokkok, illetve lapok. Azonban még ekkor is legalább két probléma adódik:

1. Minden naplózási módszer számon tartja az adatbáziselemek régi és/vagy új értékét. Ha egy blokkon belül csak kis változtatás történik, például egy sor egy mezőjét átírjuk, vagy beszúrunk/törlünk egy sort, akkor a naplóba nagy mennyiségű redundáns adat kerül be.
2. Az a követelmény, hogy az ütemezés helyreállítható legyen, azaz a zárak feloldása csak a tranzakció véglegesítése után következzen be, a konkurenciát komolyan gátolhatja. Például emlékezzünk vissza a 9.7.1. részre, ahol éppen a korai zárfeloldás előnyeit tárgyaltuk B-fa-index használata esetén. Ha megköveteljük, hogy a zárakat egészen a tranzakció véglegesítéséig fenn kell tartani, akkor ez az előny nem használható ki, és gyakorlatilag egyszerre csak egy értékváltoztató tranzakciónak engedjük meg, hogy a B-fához hozzáférjen.

Mindkét probléma indokolja a *logikai naplózás* használatát, ahol csak a blokkok változtatásai vannak feltüntetve. A változtatás természetétől függően a bonyolultságnak különböző fokai lehetnek.

1. Az adatbáziselemeknek csak néhány bájtján változtatunk, például egy rögzített hosszúságú mezőt módosítunk. Ebben a helyzetben egy kézenfekvő módszer, ha csak a megváltoztatott bájtokat és azok pozícióját jegyezzük fel. A 10.4. példában bemutatjuk ezt az esetet, és a módosítást leíró naplórekord megfelelő formáját.
2. Az adatbáziselemen végrehajtott változtatás egyszerűen leírható és könnyen visszaállítható, de az adatbáziselem legtöbb/összes bájtját érinti. A 10.5. példában egy gyakori szituáció kerül elő, ahol egy változó hosszú mezőn változtatunk, és a mező rekordjait és még más rekordokat is el kell csúsztatnunk a blokkon belül. A blokk új és régi értéke nagyon különböző lehet, ha nincs a tudatunkban, és nem jelöljük a változás egyszerű okát.
3. Az adatbáziselemen sok bájtot érintő változtatást hajtunk végre, és a további módosítások megakadályozhatják a meg nem történtté tevését. Ez valódi „logikai" naplózás, mert a semmisségi/helyrehozó folyamatot még csak nem is az adatbáziselemeken, hanem az általuk képviselt magasabb szintű „logikai" struktúrán keresztül látjuk. A 10.6. példában B-fák (a lemezblokkoknak mint adatbáziselemeknek megfelelő logikai struktúra) segítségével szemléltetjük a logikai naplózásnak ezt a komplex formáját.

10.4. példa: Tegyük fel, hogy az adatbáziselemek olyan blokkok, amelyek mindegyike tartalmaz néhány sort valamilyen relációból. Egy attribútum módosítását egy ehhez hasonló naplóbejegyzéssel fejezhetjük ki: „t sor a attribútumát v_1-ről v_2-re változtattuk". Ha a blokk egy üres részére beszúrunk egy új sort, azt a következőképpen fejezhetjük ki: „a p eltolási érték pozíciótól kezdve beszúrtunk egy t sort az $(a_1, a_2, \ldots, a_k)$ értékkel". Ha a módosított attribútum vagy a beszúrt sor méretben nem összemérhető a blokk nagyságával, akkor ezek a feljegyzések sokkal kisebb helyet foglalnak el, mint maga a blokk. Ezenfelül a semmissé tevő és helyrehozó műveletek végrehajtását is támogatják.

Vegyük észre, hogy a naplóba feljegyzett mindkét művelet idempotens: ha egy blokkon többször is végrehajtjuk őket, ugyanazt kapjuk, mintha csak egyszer alkal-

maztuk volna a műveletet. Hasonlóan az inverz műveleteik ($t[a]$ visszaállítása v_2-ről v_1-re, illetve t sor törlése) is idempotensek. Vagyis az ilyen típusú bejegyzések a helyreállítás során pontosan ugyanúgy használhatók, mint a 8. fejezetben használt módosítást leíró rekordok. □

10.5. példa: Ismét tegyük fel, hogy az adatbáziselemek sorokat tartalmazó blokkok, de most a sorok valamilyen változó hosszúságú mezőt is tartalmaznak. Ha a 10.4. példában említett módosításhoz hasonlóra kerül sor, akkor lehet, hogy a blokk nagy részét el kell csúsztatnunk ahhoz, hogy a hosszabb mezőnek helyet biztosítsunk, vagy hogy helyet takarítsunk meg, ha a mező megrövidül. Rendkívüli esetben túlcsordulásblokkot kell létrehoznunk az eredeti blokk egy részének tárolására (lásd 3.5. részt), illetve a túlcsordulásblokkot el kell távolítanunk, ha egy rövidebb mező miatt lehetővé válik két blokk egyesítése.

Ha a blokkot a túlcsordulásblokkjaival együtt egy adatbáziselem részeként tekintjük, akkor kézenfekvő a módosított mező régi és/vagy új értékeinek a használata a változtatás meg nem történtté tevéséhez illetve újra elvégzéséhez. A blokk-plusz-túlcsordulásblokk(ok)ra azonban úgy kell gondolnunk, mintha az adott sorokat „logikai" szinten tartalmaznák. Talán nem is leszünk képesek a blokkok bájtjait az eredeti állapotukba visszaállítani egy semmissé tevés vagy helyrehozás után, hiszen olyan módosítások miatt, amelyek más mezők hosszúságát változtatták, a blokkok újraszervezésére is sor kerülhetett. Ha viszont az adatbáziselemekre úgy gondolunk, mint blokkok gyűjteményére, amelyek együtt jelentenek bizonyos sorokat, akkor a helyrehozás, illetve semmissé tevés csakugyan visszaállítja az elem logikai „állapotát". □

Nem mindig lehet azonban a blokkokat a túlcsordulásblokkok mechanizmusán keresztül kiterjeszthető egységként kezelni, mint ahogy ezt a 10.5. példában láttuk. Ezért lehet, hogy a műveletek semmissé tevésére, illetve helyrehozására csak a blokkoknál magasabb szinten van lehetőség. A következő példa a B-fa-indexek fontos esetét tárgyalja, ahol a blokkok kezelése nem teszi lehetővé a túlcsordulásblokkok alkalmazását, és a semmissé tevés, illetve helyrehozás műveletekre úgy kell gondolnunk, mintha a blokkok helyett a B-fa logikai szintjén történnének meg.

10.6. példa: Nézzük meg, hogyan lehetne a B-fán történt változtatásokat a logikai naplózás módszerével nyilvántartani. Az egész csúcspont (blokk) régi és/vagy új értéke helyett csak egy rövid bejegyzést írunk a naplóba, amely a következő módosítások valamelyikét írja le:

1. Kulcs-mutató pár beszúrása vagy törlése a leszármazott (gyerek) csúcspontba vagy csúcspontból.
2. A mutatóhoz rendelt kulcs megváltoztatása.
3. Csúcspontok szétválasztása vagy egyesítése.

A fenti módosítások mindegyike leírható egy rövid naplóbejegyzéssel. Még a szétválasztás műveletéhez is csak azt kell megmondanunk, hogy melyik csúcs osztódik és

melyek az új csúcsok. Hasonlóan az egyesítés műveleténél is csak az érintett csúcspontokat kell megadnunk, hiszen az egyesítés módját a használt B-fa-kezelő algoritmus határozza meg.

A logikai módosító rekordok használatával lehetőségünk nyílik arra, hogy a zárakat korábban feloldjuk, mint ahogy erre egy helyreállítható ütemezés szerint sor kerülhetne. Ennek az az oka, hogy a piszkos B-fa-blokkok olvasása nem jelent problémát a tranzakció részéről, feltéve, hogy az olvasás egyetlen célja a tranzakció számára szükséges adatok helymeghatározása.

Tegyük fel például, hogy T tranzakció az N levelet olvassa, de az U tranzakció, amely N-t utoljára írta, abortál, ezért az N-en történt változtatásokat (például az U által beszúrt sor miatt az N-be beszúrtunk egy új kulcs-mutató párt) érvényteleníteni kell. Ha T is beszúrt egy kulcs-mutató párt N-be, akkor N-et már nem lehet az U működése előtti állapotába visszavinni. U-nak az N-re gyakorolt hatását azonban érvényteleníteni tudjuk: ebben a példában az U által beszúrt kulcs-mutató párt törölnénk. Az így kapott N persze nem lesz ugyanaz, mint ami U előtt volt, hiszen tartalmazni fogja a T által beszúrt kulcs-mutató párt. Az adatbázis azonban konzisztens maradt, mivel a B-fa továbbra is csak a végleges tranzakciók hatását tükrözi. A B-fát tehát visszaállítottuk logikai szinten, de fizikai szinten nem. □

10.1.6. Feladatok

* **10.1.1. feladat:** Hogyan lehetne az

$r_1(A)$; $r_1(B)$; $w_1(A)$; $w_1(B)$;

műveletsorozatba a 9.3. részben megismert egyszerű zárolásokat beilleszteni úgy, hogy a T_1 tranzakció alkalmazza:

a) A kétfázisú zárolás és a szigorú zárolás módszerét?
b) A kétfázisú zárolás módszerét, de ne alkalmazzon szigorú zárolást?

Adjuk meg az összes lehetséges megoldást!

10.1.2. feladat: Tegyük fel, hogy a következő ütemezések mindegyike a T tranzakció abortálásával zárul. Mely tranzakciókat kell visszagörgetnünk?

* a) $r_1(A)$; $r_2(B)$; $w_1(B)$; $w_2(C)$; $r_3(B)$; $r_3(C)$; $w_3(D)$;
b) $r_1(A)$; $w_1(B)$; $r_2(B)$; $w_2(C)$; $r_3(C)$; $w_3(D)$;
c) $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$; $r_3(B)$; $w_2(C)$; $r_3(C)$;
d) $r_2(A)$; $r_3(A)$; $r_1(A)$; $w_1(B)$; $r_3(B)$; $w_2(C)$; $r_3(C)$;

10.1.3. feladat: Vegyük a 10.1.2. feladat ütemezéseit, de most tegyük fel, hogy mindhárom tranzakció véglegessé válik, és az utolsó műveletük után rögtön be is írják a

naplóba a commit rekordjukat. A rendszer összeomlása miatt azonban a napló vége már nem kerül ki a lemezre, így az utolsó néhány rekord elvész. Válaszoljunk a következő kérdésekre attól függően, hogy a napló mekkora része élte túl a rendszerhibát!

i) Mely tranzakciók nem tekinthetők véglegesnek?
ii) Létrejön-e piszkos olvasás a helyreállítás közben? Ha igen, mely tranzakciókat kell visszagörgetni?
iii) Milyen további piszkos olvasások jöhettek volna létre, ha a bejegyzések nem a napló végéről, hanem valahonnan a közepéről vesztek volna el?

10.2. Nézet-sorbarendezhetőség

A 9.1.4. részben azt mondtuk, hogy az ütemezők tervezésekor az a valódi célunk, hogy csak sorba rendezhető ütemezések jöhessenek létre. Azt is láttuk, hogy egy ütemezés sorbarendezhetősége a tranzakciók adatokon végzett műveleteitől is függhet. A 9.2. részben pedig azt is megtudtuk, hogy az ütemezők rendszerint a „konfliktus-sorbarendezhetőséget" juttatják érvényre, amely a tranzakciók adatokon végrehajtott módosításaira való tekintet nélkül garantálja a sorbarendezhetőséget.

Azonban a konfliktus-sorbarendezhetőségnél gyengébb feltételek is biztosíthatják a sorbarendezhetőséget. Ebben a részben egy ilyen feltétellel, a „nézet-sorbarendezhetőséggel" foglalkozunk. A nézet-sorbarendezhetőség veszi az összes olyan esetet T és U tranzakció kapcsán, amikor T írja az U által olvasott adatbáziselemet. A kétféle sorbarendezhetőség közötti lényeges különbség akkor jön elő, amikor T tranzakció írja A-t, de azt az értéket nem olvassa más tranzakció (mert később egy másik tranzakció felülírja A-t). Ilyenkor a $w_T(A)$ művelet az ütemezés egyéb olyan pozícióira is kerülhet, ahol éppúgy sohasem olvassák, viszont a konfliktus-sorbarendezhetőség definíciója mellett nem lennének megengedhetők. Ebben a részben pontosan definiáljuk a nézet-sorbarendezhetőség fogalmát, majd megadunk egy eljárást, amely segítségével ellenőrizhető, hogy egy adott ütemezés sorba rendezhető-e ilyen értelemben.

10.2.1. Nézetekvivalencia

Tegyük fel, hogy ugyanazokhoz a tranzakciókhoz van két ütemezés: S_1 és S_2. Az ütemezések legelejére, illetve legvégére vegyünk fel további két hipotetikus tranzakciót, T_0-t, illetve T_f-t. Úgy képzeljük, hogy a többi tranzakció által olvasott adatbáziselemek kezdeti értékét T_0 írta, az általuk írt értékeket pedig T_f olvassa az ütemezés végén. Ekkor minden $r_i(A)$ olvasáshoz megadható az a $w_j(A)$ írásművelet, amely megelőzi $r_i(A)$-t, és a legközelebb esik hozzá.[2] Ilyenkor a T_j tranzakciót az $r_i(A)$ olvasás *forrásának* nevezzük. Vegyük észre, hogy T_j lehet a hipotetikus T_0 kezdeti tranzakció, illetve T_i lehet a T_f tranzakció is.

[2] Bár az eddigiekben nem akadályoztuk meg a tranzakciókat abban, hogy ugyanazt az elemet kétszer is írják, erre általában nincs szükségük. A továbbiakban hasznos feltennünk, hogy a tranzakciók az egyes elemeket csak egyszer írják.

Ha az olvasásműveletek forrása mindkét ütemezésben ugyanaz, akkor azt mondjuk, hogy az S_1 és az S_2 ütemezések nézetekvivalensek. Kétségtelen, hogy a nézetekvivalens ütemezések valóban ekvivalensek: akármilyen adatbázis-állapoton hajtjuk őket végre, azonos lesz a hatásuk. Ha az S ütemezés nézetekvivalens egy soros ütemezéssel, akkor S-t *nézet-sorbarendezhető* ütemezésnek nevezzük.

10.7. példa: Vegyük a következő S ütemezést:

T_1:			$r_1(A)$			$w_1(B)$				
T_2:	$r_2(B)$	$w_2(A)$						$w_2(B)$		
T_3:					$r_3(A)$					$w_3(B)$

A tranzakciók műveleteit most függőleges irányban tagoltuk, hogy jobban lehessen követni, melyik tranzakció mit csinál; az ütemezés a megszokott módon balról jobbra olvasandó.

S-ben T_1 és T_2 is írja B-t, de ezek az értékek elvesznek (felülíródnak), csak a T_3 által írt B érték éri meg az ütemezés végét, ahol a hipotetikus T_f tranzakció „kiolvassa". S nem konfliktus-sorbarendezhető. Hogy lássuk, miért, először vegyük észre, hogy T_2 azelőtt írja A-t, mielőtt azt T_1 olvasná, azaz egy feltételes konfliktus ekvivalens soros ütemezésben T_2-nek meg kell előznie T_1-et. Az a körülmény viszont, hogy a $w_1(B)$ művelet megelőzi $w_2(B)$-t, azt vonja maga után, hogy a feltételes konfliktus ekvivalens soros ütemezésben T_1-nek kell megelőznie T_2-t. Pedig sem $w_1(B)$-nek, sem $w_2(B)$-nek nincs hosszú távú hatása az adatbázisra. A nézet-sorbarendezhetőség az ilyen típusú, lényegtelen írásműveleteket hagyja figyelmen kívül, amikor megállapítja, hogy melyek azok a tényleges megszorítások, amelyekhez igazodnia kell egy ekvivalens soros ütemezésnek.

Formálisabban, tekintsük S-ben az egyes olvasásműveletek forrásait:

1. $r_2(B)$ forrása T_0, mivel ezelőtt nem írjuk B-t S-ben.
2. $r_1(A)$ forrása T_2, mivel az olvasás előtt T_2 írta legutóbb A-t.
3. Hasonlóan, $r_3(A)$ forrása T_2.
4. A hipotetikus $r_{T_f}(A)$ művelet forrása T_2.
5. A hipotetikus $r_{T_f}(B)$ művelet forrása T_3, amely legutoljára írta B-t.

Természetes, hogy T_0, illetve T_f a valódi tranzakciók előtt, illetve után tűnik fel, akármilyen ütemezést is veszünk. Ha a valódi tranzakciók (T_2, T_1, T_3) sorrendjét vesszük, akkor minden olvasás forrása ugyanaz, mint S-ben. Azaz T_2 olvassa B-t és biztosan T_0 az utolsó „író" tranzakció. T_1 olvassa A-t, de T_2 már írta A-t, vagyis $r_1(A)$ forrása T_2, ahogy S-ben. T_3 is olvassa A-t, de mivel az ezt megelőző T_2 írta A-t, $r_3(A)$ forrása T_2, mint S-ben. Végül a hipotetikus T_f olvassa A-t és B-t, de A, illetve B utolsó írói a (T_2, T_1, T_3) ütemezésben rendre T_2, illetve T_3, szintén, mint S-ben. Megállapíthatjuk, hogy S egy nézet-sorbarendezhető ütemezés, és hogy (T_2, T_1, T_3) egy vele nézetekvivalens ütemezés. □

10.2.2. Poligráfok és nézet-sorbarendezhetőségi teszt

A 9.2.2. részben a konfliktus-sorbarendezhetőség tesztelésére a megelőzési gráfot használtuk. Ez a gráf általánosítható úgy, hogy a nézet-sorbarendezhetőség definíciójában megkövetelt összes megelőzési megszorítást tükrözze. Az ütemezéshez tartozó *poligráfot* definíció szerint a következő alkotóelemekből kell összeállítani:

1. Minden tranzakcióhoz vegyünk egy-egy csomópontot, és még további kettőt a hipotetikus T_0-nak és T_f-nek.
2. Ha $r_i(X)$ művelet forrása T_j, vegyük fel a T_j-ből T_i-be mutató irányított élt.
3. Tegyük fel, hogy az $r_i(X)$ olvasás forrása T_j, de T_k is írja X-t. Nem engedhetjük meg, hogy T_k tranzakció T_j és T_i közé essen, tehát vagy T_j előtt vagy T_i után kell állnia. Ezt a feltételt a T_k-ból T_j-be és a T_i-ből T_k-ba vezető élpár felvételével (szaggatott vonallal rajzolva) adhatjuk meg. Az élpár egyik fele „valódi", de nem foglalkozunk vele, hogy melyik. Amikor majd körmentessé próbáljuk a gráfot tenni, az élpárnak azt a felét tartjuk meg, amelyik ebben jobban segít. Vannak azonban fontos speciális esetek, amikor az élpár csak egyetlen élből áll:

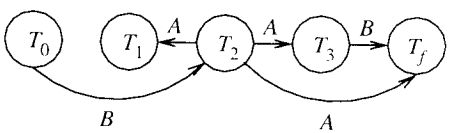
   a) Ha $T_j = T_0$, akkor T_k-nak nincs lehetősége arra, hogy T_j előtt szerepeljen. Ilyenkor az élpár helyett a $T_i \rightarrow T_k$ élt használjuk.
   b) Ha $T_i = T_f$, akkor T_k nem követheti T_i-t. Ilyenkor az élpár helyett a $T_k \rightarrow T_j$ élt használjuk.

10.8. példa: Vegyük a 10.7. példa ütemezését. A 10.3. ábrán látható, hogyan kezdtük el az S-hez tartozó poligráfot felépíteni. A csúcsokon kívül csak a 2. szabály alapján berajzolható élek vannak meg; ezekre azt az adatelemet is ráírtuk, ami miatt bekerültek a gráfba. A-t tehát T_2 továbbadja T_1-nek, T_3-nak és T_f-nek, míg B T_0-tól T_2-höz, illetve T_3-tól T_f-hez kerül.

Most meg kell gondolnunk, hogy milyen tranzakciók léphetnek közbe azáltal, hogy ugyanazt az adatelemet írják, mint ami összeköttetésekben szerepel. Ezeket a lehetséges beavatkozásokat zárjuk ki a 3. szabály élpárjaival, amelyek ebben a példában mind egyetlen élt jelentenek majd valamelyik speciális eset miatt.

Vegyük a $T_2 \rightarrow T_1$ irányított élt. Az A adatelemet csak T_0 és T_2 írja, és ezek közül egyik sem kerülhet T_2 és T_1 közé, mivel T_0 nem változtathatja meg a pozícióját, T_2 pedig már ott szerepel az él egyik végén. Vagyis nem kell új élt berajzolnunk. Hasonló érvelés alapján a $T_2 \rightarrow T_3$ és a $T_2 \rightarrow T_f$ esetén sem kell újabb éleket rajzolnunk ahhoz, hogy az A-t író tranzakciókat kívül tartsuk ezektől az élektől.

Most nézzük a B-hez tartozó éleket. Vegyük észre, hogy T_0, T_1, T_2 és T_3 is írja B-t.

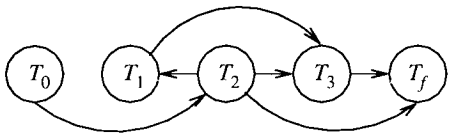


10.3. ábra. *Poligráf kezdemény a 10.8. példához*

Vegyük először a $T_0 \rightarrow T_2$-t. Ahogy az előbb láttuk, T_0-val és T_2-vel most nem kell foglalkoznunk, hiszen az él végpontjai nem okozhatnak problémát; a B adatelemet író tranzakciók közül tehát marad T_1 és T_3. Mivel T_1 nem szerepelhet T_0 és T_2 között, elméletileg be kellene rajzolnunk a ($T_1 \rightarrow T_0$, $T_2 \rightarrow T_1$) élpárt. T_0-t azonban semmi sem előzheti meg, vagyis a $T_1 \rightarrow T_0$ lehetőség valójában nem lehetőség. Ebben a speciális esetben elég csak a $T_2 \rightarrow T_1$ irányított él felvétele. De ez az él már megvan A miatt, vagyis gyakorlatilag semmit sem változtatunk a poligráfon annak érdekében, hogy T_1 ne kerülhessen T_0 és T_2 közé.

T_3 sem kerülhet T_0 és T_2 közé. Az előző esethez hasonlóan itt is az élpár helyett csak a $T_2 \rightarrow T_3$ élt kellene felvennünk a gráfba, de ez ugyancsak szerepel már A miatt. Vagyis most sem változtatunk semmin.

Most nézzük a $T_3 \rightarrow T_f$ irányított élt. Mivel T_3-on kívül B-t még T_0, T_1 és T_2 írja, ezeket mind kívül kell tartanunk ezen az élen. T_0 nem eshet T_3 és T_f közé, de T_1 vagy T_2 igen. Mivel semelyikük sem tehető T_f mögé, biztosítanunk kell, hogy T_1 és T_2 mind T_3 előtt jelenjenek meg. A $T_2 \rightarrow T_3$ él már szerepel a gráfban, de a $T_1 \rightarrow T_3$-at hozzá kell adnunk. Ez az egyetlen változtatás, amit végre kell hajtanunk a poligráfon. Az S-hez tartozó végleges poligráf a 10.4. ábrán látható. □



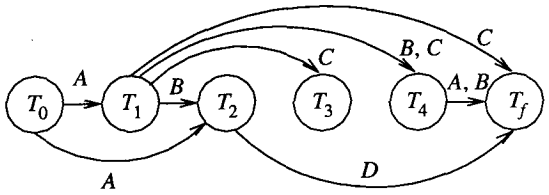
10.4. ábra. *Kész poligráf a 10.8. példához*

10.9. példa: A 10.8. példában minden élpárról kiderült, hogy valamelyik speciális eset miatt tulajdonképpen csak egyetlen élből áll. A 10.5. ábrán egy olyan ütemezést láthatunk, amely poligráfjában egy valódi élpár is szerepel.

T_1	T_2	T_3	T_4
	$r_2(A)$;		
$r_1(A)$; $w_1(C)$;			
		$r_3(C)$;	
$w_1(B)$;			
			$r_4(B)$;
		$w_3(A)$;	
			$r_4(C)$;
	$w_2(D)$; $r_2(B)$;		
			$w_4(A)$; $w_4(B)$;

10.5. ábra. *Az ütemezéshez tartozó poligráf valódi élpárt tartalmaz*

A 10.6. ábrán látható poligráf csak a forrás-olvasó kapcsolatokat mutatja. Mint a 10.3. ábrán, itt is ráírtuk az élekre, hogy melyik adatelem miatt kellett berajzolnunk. Ezek után végig kell mennünk az összes lehetséges eseten, amikor élpárt kéne felven-



10.6. ábra. *Poligráfkezdemény a 10.9. példához*

nünk. Ahogy a 10.8. példában láttuk, többféleképpen is egyszerűsíthető ez az eljárás. A $T_j \rightarrow T_i$ él vizsgálata esetén T_k szerepében (amelyek nem lehetnek középen) csak a következő tranzakciókat kell megvizsgálnunk:

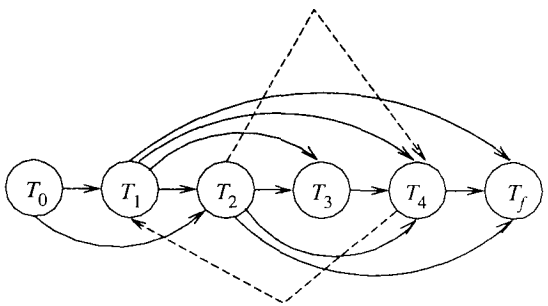
- A $T_j \rightarrow T_i$ élhez tartozó adatelemek írói.
- De nem T_0 vagy T_f, amelyek sohasem vehetik fel T_k szerepét, és
- Nem T_i vagy T_j, a kérdéses él végpontjai.

Ezek fényében nézzük végig az A adatbáziselemhez tartozó éleket. Az A-t író tranzakciók: T_0, T_3, T_4. Ezek közül T_0-val egyáltalán nem kell foglalkoznunk. T_3 nem kerülhet T_4 és T_f közé, ezért felvesszük a $T_3 \rightarrow T_4$ élt a gráfba; az élpár másik fele: a $T_f \rightarrow T_3$ nem jöhet számításba. Hasonlóan T_3 nem kerülhet T_0 és T_1, illetve a T_0 és T_2 közé sem, így a gráf a $T_0 \rightarrow T_3$, illetve $T_2 \rightarrow T_3$ élekkel bővül tovább.

Most nézzük meg, milyen megszorítások vonatkoznak T_4-re A írása miatt. T_4 a $T_4 \rightarrow T_f$ él egyik végpontja, azaz ezzel az éllel nem kell foglalkoznunk. T_4 nem kerülhet T_0 és T_1, illetve T_0 és T_2 közé, tehát fel kell vennünk a gráfba a $T_1 \rightarrow T_4$, illetve a $T_2 \rightarrow T_4$ éleket.

Következő lépésként vegyük a B-hez tartozó éleket. A B-t író tranzakciók: T_0, T_1, T_4. T_0-t most is figyelmen kívül hagyhatjuk. B miatt a következő éleket kell sorba vennünk: $T_1 \rightarrow T_2$, $T_1 \rightarrow T_4$, $T_4 \rightarrow T_f$. T_1-nek nincsen jelentősége az első két éllel kapcsolatban, viszont a harmadik miatt fel kell vennünk a $T_1 \rightarrow T_4$ élt.

T_4-nek csak a $T_1 \rightarrow T_2$ élre lehet hatása. Mivel ennek egyik végpontja sem T_0 vagy T_f, egy valódi élpárral bővül a gráf: $(T_4 \rightarrow T_1, T_2 \rightarrow T_4)$-gyel. Az új élekkel kibővített poligráf a 10.7. ábrán látható.



10.7. ábra. *Kész poligráf a 10.9. példához*

A C adatelemet T_0 és T_1 írja. Az előbbiekhez hasonlóan T_0 most sem jelenthet problémát. T_1 viszont minden C-hez tartozó élnek része, tehát vele sem kell foglalkoznunk. A helyzet gyakorlatilag D-vel kapcsolatban is ugyanez, vagyis megállapíthatjuk, hogy nincs szükség további élek felvételére. A végleges poligráf tehát megegyezik azzal, amelyet a 10.7. ábrán láthatunk. □

10.2.3. A nézet-sorbarendezhetőség tesztelése

Mivel minden élpárból csak az egyik élt kell kiválasztanunk, az S ütemezéshez pontosan akkor adható meg vele ekvivalens soros ütemezés, ha az S-hez tartozó poligráf az élpárok egy-egy tagjának elhagyásával körmentessé tehető. Először tegyük fel, hogy van egy ilyen körmentes gráfunk. Ekkor a gráf egy topologikus sorrendje a tranzakciók egy olyan rendezését adja, amelyben a forrás és az olvasó tranzakció közé nem ékelődhet be író tranzakció, továbbá minden író tranzakció megelőzi a hozzá tartozó olvasó tranzakciókat. Így a soros ütemezésben az olvasó-forrás kapcsolatok pontosan ugyanazok, mint S-ben; a két ütemezés nézetekvivalens, tehát S nézet-sorbarendezhető.

Megfordítva, ha S nézet-sorbarendezhető, akkor létezik hozzá egy S' nézetekvivalens soros ütemezés. Az S-hez tartozó poligráfban minden $(T_k \rightarrow T_j, T_i \rightarrow T_k)$ élpár esetén az S' ütemezésben T_k vagy megelőzi T_j-t vagy T_i után következik. Ellenkező esetben T_k írásművelete megszakítaná a T_j és T_i közötti kapcsolatot, ami azt jelentené, hogy S és S' nem nézetekvivalens. Hasonlóan, a poligráf minden irányított éle is megjelenik az S'-beli tranzakciók sorrendjében. Ebből az következik, hogy a poligráf minden élpárjából ki tudjuk választani az egyik élt úgy, hogy az így kapott gráf irányított éleinek megfelel az S' soros ütemezése. Tehát a gráf körmentes.

10.10. példa: Vegyük a 10.4. ábrán látható poligráfot. Ez már ebben az állapotában is körmentes gráf. Ehhez egyetlen topologikus sorrend adható, így a 10.8. példában bemutatott ütemezés nézetekvivalens soros ütemezése (T_2, T_1, T_3).

Most nézzük a 10.7. ábra poligráfját. Két esetet kell megvizsgálnunk aszerint, hogy az élpárból melyik él marad meg. Ha a $T_4 \rightarrow T_1$ élt választjuk, akkor létrejön egy irányított kör a gráfban. A $T_2 \rightarrow T_4$ választása esetén azonban a gráf körmentes lesz. A gráfhoz megadható egyetlen topologikus sorrend (T_1, T_2, T_3, T_4), amely egyben nézetekvivalens soros ütemezésként is szolgál, és igazolja, hogy az eredeti ütemezés nézetsorbarendezhető. □

10.2.4. Feladatok

10.2.1. feladat: Adjuk meg a következő ütemezésekhez tartozó poligráfokat és az összes nézetekvivalens soros ütemezést:

* a) $r_1(A)$; $r_2(A)$; $r_3(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$;
b) $r_1(A)$; $r_2(A)$; $r_3(A)$; $r_4(A)$; $w_1(B)$; $w_2(B)$; $w_3(B)$; $w_4(B)$;

c) $r_1(A)$; $r_3(D)$; $w_1(B)$; $r_2(B)$; $w_3(B)$; $r_4(B)$; $w_2(C)$; $r_5(C)$; $w_4(E)$; $r_5(E)$; $w_5(B)$;
d) $w_1(A)$; $r_2(A)$; $w_3(A)$; $r_4(A)$; $w_5(A)$; $r_6(A)$;

! **10.2.2. feladat:** Határozzuk meg, hogy az alábbi soros ütemezésekkel az előző feladat ütemezései közül hány i) konfliktusekvivalens, illetve ii) nézetekvivalens:

* a) $r_1(A)$; $w_1(B)$; $r_2(A)$; $w_2(B)$; $r_3(A)$; $w_3(B)$; azaz mindhárom tranzakció először olvassa A-t, majd írja B-t.
b) $r_1(A)$; $w_1(B)$; $w_1(C)$; $r_2(A)$; $w_2(B)$; $w_2(C)$; azaz mindhárom tranzakció először olvassa A-t, majd írja B-t és C-t.

10.3. Holtpontkezelés

Már többször megfigyelhettük, hogy az egyszerre végrehajtott tranzakciók versenyezhetnek egymással az erőforrásokért, és ezáltal *holtpontra* (deadlock) juthatnak, ami azt jelenti, hogy minden tranzakció egy másik tranzakció által birtokolt erőforrásra vár, és egyik sem tud továbblépni.

- A 9.3.4. részben láttuk, hogy a kétfázisú zárolást használó tranzakciók szokásos műveletei még mindig vezethetnek holtponthoz, ha mindegyik tranzakció valami olyasmit zárolt, amelyre egy másiknak is szüksége van.
- A 9.4.3. részben pedig azt láttuk, hogy az a lehetőség, hogy az osztott zár kizárólagos zárrá minősíthető fel, szintén okozhat holtpontot, ha minden tranzakció rendelkezik ugyanarra az elemre egy osztott zárral, és ezt egyszerre akarják felminősíteni.

A holtpontkezelés problémája két fő irányból közelíthető meg. Vagy valahogy rájövünk, hogy néhány tranzakció holtpontra jutott, és ebből a helyzetből keresünk kiutat, vagy már eleve úgy kezeljük a tranzakciókat, hogy soha ne juthassanak holtpontra.

10.3.1. Holtpontérzékelés időkorláttal

Ha a holtpont már bekövetkezett, akkor általában nem lehet a helyzeten úgy javítani, hogy minden tranzakció továbbléphessen. Azaz legalább egy tranzakciót vissza kell görgetni – abortáltatni kell, majd újraindítani.

A holtpontok érzékelésére és feloldására a legegyszerűbb megoldást az *időtúllépés* (timeout) módszere adja. Időkorlátokat vezetünk be, amely arra vonatkozik, hogy az egyes tranzakciók mennyi ideig lehetnek aktívak, és ha ezt a határt túllépik, akkor visszagörgetjük őket. Például egy egyszerű rendszerben, ahol a tipikus tranzakciók ezredmásodpercek alatt lefutnak, az egyperces időkorlátnak tényleg csak a holtpontra jutott tranzakcióra lenne hatása. De ha van néhány összetettebb tranzakció is, akkor az időtúllépés bekövetkezéséhez hosszabb időt választhatunk.

Vegyük észre, hogyha a holtpontra jutott tranzakció túllépi az időkorlátját, akkor a többi erőforrással együtt az eddig birtokolt zárjairól is lemond. Így tehát van esély arra, hogy a holtponton álló többi tranzakció még azelőtt be tudja fejezni a tevékenységét, mielőtt kifutna az időből. De mivel a holtpontra jutott tranzakciók valószínűleg körülbelül ugyanabban az időpontban indultak (különben az egyik befejeződött volna, még mielőtt a másik elkezdődött volna), az is lehetséges, hogy a rendszer hamis időtúllépéseket érzékel, azaz úgy görgeti vissza a tranzakciókat, hogy azok már túljutottak a közös holtponton.

10.3.2. A várakozási gráf

Azok a helyzetek, amikor a holtpont azért alakul ki, mert az egyik tranzakció a másik birtokában lévő zárakra vár, jól kezelhetők a *várakozási gráffal* (waits-for graph). Ebben a gráfban azt tartjuk nyilván, hogy melyik tranzakció melyikre vár. A várakozási gráf segítségével érzékelhetővé válnak a már kialakult holtpontok, de meg is előzhető a kialakulásuk. Mi az utóbbival próbálkozunk, ami azzal jár, hogy a várakozási gráfot egész idő alatt nyilván kell tartanunk, és az olyan műveleteket, amelyek következtében a gráfban kör alakulna ki, nem szabad megengednünk.

Idézzük fel a 9.5.2. rész alapján, hogy a zártáblában minden X adatbáziselemhez létezik egy lista, amelyben azon tranzakciók mellett, amelyek arra várnak, hogy zárolhassák X-t, azok is fel vannak sorolva, amelyek rendelkeznek X zárjával. A várakozási gráf csúcsai a listában található tranzakcióknak felelnek meg. A gráfban irányított él fut T-ből U-ba, ha létezik olyan A adatbáziselem, hogy:

1. U zárolja A-t.
2. T arra vár, hogy zárolhassa A-t, és
3. T csak akkor kapja meg a számára megfelelő módban A zárját, ha először U lemond róla.[3]

Ha nincsen (irányított) kör a gráfban, akkor végül minden tranzakció be tudja fejezni a működését. Lesz legalább egy olyan tranzakció, amelyik nem vár semelyik másikra, így ez biztosan befejeződhet. Ekkor viszont megint lesz legalább egy tranzakció, amelyik nem várakozik, ezért továbbléphet és így tovább.

Ha azonban a gráf nem körmentes, akkor a körben részt vevő tranzakciók nem léphetnek tovább, azaz holtpontra jutottak. A holtpont-megelőzési stratégia tehát abból áll, hogy minden olyan tranzakciót visszagörgetünk, amelynek valami olyan igénye van, ami kört idézne elő a várakozási gráfban.

[3] Egyszerű esetekben, mint amikor osztott és kizárólagos zárakat használunk, minden egyes várakozó tranzakciónak meg kell várnia, amíg *minden* zárral rendelkező tranzakció feloldja a saját zárját. Vannak azonban olyan zármód rendszerek is, ahol egy tranzakció már akkor is megkaphatja a zárat, ha még csak néhány zárbirtokos mondott le róla. Lásd 10.3.6. feladat.

10.11. példa: Tegyük fel, hogy a következő négy tranzakcióval rendelkezünk:

T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $l_2(C)$; $r_2(C)$; $l_2(A)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;
T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;
T_4: $l_4(D)$; $r_4(D)$; $l_4(A)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

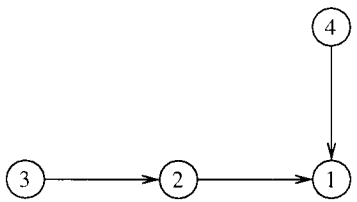
Mindegyik az egyik elemet olvassa, a másikat írja. Egy egyszerű zárrendszert használunk egyféle zárral, bár ugyanezt a jelenséget figyelhetnénk meg, ha egy osztott/kizárólagos rendszerben a szokásos módon osztanánk a zárakat: osztott zárat az olvasáshoz, kizárólagos zárat az íráshoz.

A 10.8. ábrán egy lehetséges ütemezés kezdeti szakasza látható. Az első négy lépésben mindegyik tranzakció zárolja azt az elemet, amelyet olvasni szeretne. Az 5) lépésben T_2 megpróbálja zárolni A-t, de nem tudja, mert a zár már T_1 birtokában van. T_2 tehát várakozik T_1-re, ezért a várakozási gráfba berajzolunk egy élt a T_2-nek megfelelő csúcsból a T_1-nek megfelelő csúcs felé.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(C)$; $r_2(C)$;		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(D)$; $r_4(D)$;
5)		$l_2(A)$; **Elutasítva**		
6)			$l_3(C)$; **Elutasítva**	
7)				$l_4(A)$; **Elutasítva**
8)	$l_1(B)$; **Elutasítva**			

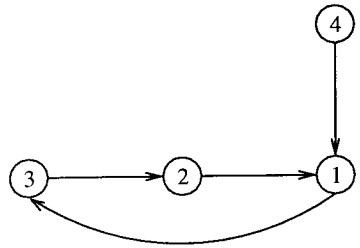
10.8. ábra. *Egy ütemezés első néhány lépése, amelyben holtpont alakul ki*

Hasonlóan, a 6) lépésben T_3 nem tudja zárolni C-t T_2 miatt, a 7) lépésben pedig T_4 vall kudarcot A zárolásával T_1 miatt. Az ebben az állapotban egyelőre körmentes várakozási gráf a 10.9. ábrán látható.



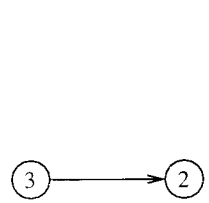
10.9. ábra. *A várakozási gráf az ütemezés 7) lépése után*

A 8) lépésben T_1-nek várnia kell B zárolásával T_3 miatt. Ha megengednénk T_1-nek, hogy várjon erre a zárra, akkor T_1, T_2 és T_3 mentén kör jönne létre a várakozási gráfban, ahogy ezt a 10.10. ábra is mutatja. Mivel a körben mindegyik tranzakció arra vár, hogy a másik befejeződjön, egyik sem tud továbblépni, vagyis ennek a három tranzakciónak a részvételével holtpont alakul ki. Véletlen egybeesés, hogy T_4 sem fejeződhet be annak ellenére, hogy nincs benne a körben. Az ő előrejutása azonban T_1 továbblépésén múlik.



10.10. ábra. *A várakozási gráf az ütemezés 8) lépése után kört tartalmaz*

Mivel a kört okozó tranzakciókat visszagörgetjük, így teszünk T_1-gyel is. A várakozási gráf a 10.11. ábrának megfelelően alakul. T_1 feloldja A zárolását, amelyet vagy T_2 vagy T_4 vesz át. Tegyük fel, hogy a zár T_2 birtokába kerül. T_2 befejeződik, ezáltal feloldódik a zár A-n és C-n. Most T_3, amely C-t akarja zárolni, és T_4 is, amely A-t, lezárulhat. Valamivel később T_1-et újraindítjuk, de nem kaphatja meg sem A, sem B zárját, amíg T_2, T_3 és T_4 be nem fejeződött. □



10.11. ábra. *A várakozási gráf T_1 visszagörgetése után*

10.3.3. Holtpontmegelőzés az elemek sorbarendezésével

Most ismerkedjünk meg néhány más módszerrel is, amellyel megelőzhető a holtpont kialakulása. Az elsőben sorrendbe kell raknunk az adatbáziselemeket valamilyen tetszőleges, de előre rögzített rendezés szerint. Ha például az adatbáziselemek blokkok, akkor rendezhetjük őket lexikografikusan a fizikai címük szerint. Emlékezzünk vissza, a 3.3.1. részben volt arról szó, hogy egy blokk fizikai címe általában egy olyan bájtsorozat, amely a blokknak a tárrendszeren belüli helyét írja le.

Ha minden tranzakciótól megkívánjuk, hogy az adatbáziselemeket a megadott sorrendnek megfelelően próbálja meg zárolni (ami egyébként a legtöbb alkalmazásban nem reális feltétel), akkor a (foglalt) zárakra való várakozások miatt nem alakulhat ki holtpont. Ugyanis tegyük fel, hogy T_2 várakozik A_1 zárolásával T_1 miatt, T_3 várakozik A_2 zárolásával T_2 miatt, és így tovább, T_n várakozik A_{n-1} zárolásával T_{n-1} miatt, T_1 pedig A_n zárolásával várakozik T_n miatt. Mivel T_2 zárolta A_2-t, de várnia kell A_1-re,

$A_2 < A_1$ relációnak kell teljesülnie az adott rendezésen. Hasonlóan, $A_i < A_{i-1}$, ahol $i = 3, 4, \ldots, n$. De mivel T_1 zárolta A_1-et és várakozik A_n-re, $A_1 < A_n$ is fennáll. Ekkor a következő teljesül: $A_1 < A_n < A_{n-1} < \ldots < A_2 < A_1$, ami lehetetlen, hiszen ebből $A_1 < A_1$ következne.

10.12. példa: Tegyük fel, hogy az adatbáziselemek alfabetikusan vannak rendezve. Ekkor, ha a 10.11. példa négy tranzakcióját vesszük, akkor T_2-t és T_4-t át kell írnunk, hogy a megfelelő sorrendben zárolják az elemeket. Így a négy tranzakció a következő:

T_1: $l_1(A)$; $r_1(A)$; $l_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $l_2(A)$; $l_2(C)$; $r_2(C)$; $w_2(A)$; $u_2(C)$; $u_2(A)$;
T_3: $l_3(B)$; $r_3(B)$; $l_3(C)$; $w_3(C)$; $u_3(B)$; $u_3(C)$;
T_4: $l_4(A)$; $l_4(D)$; $r_4(D)$; $w_4(A)$; $u_4(D)$; $u_4(A)$;

A 10.12. ábrán látható, hogy mi történik, ha ugyanazt az ütemezést használjuk, mint ami a 10.8. ábrán adott. Az első lépésben T_1 zárolja A-t. A következő lépésben T_2 próbálja végrehajtani az első műveletét, de A zárolásával T_1 miatt várnia kell. Ezek után T_3 zárolja B-t, majd T_4 is megpróbálja zárolni A-t, de várakoznia kell.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(A)$; **Elutasítva**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Elutasítva**
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)		$l_2(A)$; $l_2(C)$;		
10)		$r_2(C)$; $w_2(A)$;		
11)		$u_2(A)$; $u_2(C)$;		
12)				$l_4(A)$; $l_4(D)$;
13)				$r_4(D)$; $w_4(A)$;
14)				$u_4(A)$; $u_4(D)$;

10.12. ábra. *Az elemeket a tranzakciók alfabetikus sorrendben zárolják, ezzel megelőzhető a holtpont kialakulása*

Mivel T_2 elakadt, a 10.8. ábra ütemezése szerint T_3 következik. Sikeresen zárolja C-t, majd a 6) lépésben lezárul. Mivel ezzel egy időben elengedi B-t és C-t, T_1 is befejeződhet, ami meg is történik a 8) lépésben. Ezen a ponton A felszabadul a zárolás alól, és feltehető, hogy FIFO alapon T_2 birtokába kerül a zár. Most T_2 mindkét szükséges adatelemét zárolja, majd a 11) lépésben lezárul. Végül T_4 is megkapja az óhajtott zárakat és befejeződik. □

10.3.4. Holtpontérzékelés időbélyegzővel

A 10.3.2. részben láttuk, hogy a várakozási gráf segítségével észlelni tudjuk a holtpontok kialakulását. Ez a gráf azonban nagy lehet, és minden alkalommal kört keresni benne, valahányszor egy tranzakciónak várnia kell egy zárra, nagyon időigényessé válhat. A várakozási gráf mellett egy másik lehetőség az időbélyegzők bevezetése. Minden egyes tranzakcióhoz hozzárendelünk egy-egy időbélyegzőt, amely:

- Csak a holtpont érzékelésére használatos. Ez nem ugyanaz, mint az időbélyegző, amelyet a 9.8. részben a konkurenciavezérléshez használtunk, még akkor sem, ha éppen az időbélyegző alapú konkurenciavezérlés van érvényben.
- Például, ha egy tranzakciót vissza kell görgetni, akkor egy új, későbbi konkurencia időbélyegzővel kezdi újra a működését, de a holtpontészleléshez használt időbélyegzője sohasem változik.

Az időbélyegzőt akkor használjuk, amikor egy T tranzakciónak az U tranzakció birtokában lévő zárra kell várakoznia. Attól függően, hogy T vagy U az *idősebb* (korábbi időbélyegzővel rendelkező), két különböző dolog történhet. Két különféle eljárásmód használható a tranzakciók kezelésére és a holtpontok érzékelésére.

1. *Megvár-meghal séma:*

   a) Ha T idősebb U-nál (azaz T időbélyegzője korábbi, mint U időbélyegzője), akkor megengedjük, hogy T az U által birtokolt zár(ak)ra várakozzon.
   b) Ha U idősebb T-nél, akkor T „meghal": visszagörgetjük.

2. *Megsebez-megvár séma:*

   a) Ha T idősebb U-nál, akkor „megsebzi" U-t. A „seb" általában végzetes: U-t vissza kell görgetni, és le kell mondania a szükséges zárakról T javára. Egyetlen eset képez kivételt: ha, mire a „sebzésnek" hatása lenne, U befejeződik, és elengedi a zárjait. Ilyenkor U életben marad, és nem kell visszagörgetni.
   b) Ha U idősebb T-nél, akkor T az U birtokában lévő zár(ak)ra várakozik.

10.13. példa: Vegyük a 10.12. példa tranzakcióit, és nézzük meg, hogyan működik a megvár-meghal séma. Feltesszük, hogy T_1, T_2, T_3, T_4 az időbélyegzők sorrendje, azaz T_1 a legidősebb tranzakció. Azt is feltesszük, hogy amikor egy tranzakciót visszagörgetünk, az nem indul újra olyan hamar, hogy még azelőtt aktívvá válna, mielőtt a többi tranzakció lezárulna.

Az események egy lehetséges sorrendje a megvár-meghal sémát használva a 10.13. ábrán látható. A zárját először T_1 kapja meg. Amikor T_2 akarja zárolni A-t, meghal, mert T_1 idősebb nála. A 3) lépésben T_3 zárolja B-t, de a 4) lépésben T_4 akarja zárolni A-t, és meghal, mert a zár birtokosa, T_1, idősebb T_4-nél. A következő lépésben T_3 megkapja C zárját, majd befejeződik. Amikor T_1 folytatja a működését, elérhető lesz számára B zárja, majd a 8) lépésben szintén befejeződik.

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(A)$; **Meghal**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Meghal**
5)			$l_3(C)$; $w_3(C)$;	
6)			$u_3(B)$; $u_3(C)$;	
7)	$l_1(B)$; $w_1(B)$;			
8)	$u_1(A)$; $u_1(B)$;			
9)				$l_4(A)$; $l_4(D)$;
10)		$l_2(A)$; **Várakozik**		
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)		$l_2(A)$; $l_2(C)$;		
14)		$r_2(C)$; $w_2(A)$;		
15)		$u_2(A)$; $u_2(C)$;		

10.13. ábra. *A tranzakciók műveletei a megvár-meghal sémában érzékelik a holtpontot*

Most újrakezdődik a két visszagörgetett tranzakció: T_2 és T_4. A holtpontkezeléshez használt időbélyegzőjük nem változik; T_2 továbbra is idősebb, mint T_4. Feltesszük azonban, hogy T_4 kezdődik előbb, a 9) lépésben, és amikor a 10) lépésben az idősebb T_2 zárolni akarja A-t, akkor erre várnia kell, de nem abortál. T_4 lezárul a 12) lépésben, majd az utolsó három lépésben T_2 is befejezi a működését. □

10.14. példa: Most a 10.14. ábra segítségével kövessük nyomon, hogy ugyanezek a tranzakciók hogyan viselkednek, ha a megsebez-megvár sémát használjuk. Mint a 10.13. ábrán látható, T_1 itt is A zárolásával indul. Amikor a 2) lépésben T_2 akarja zá-

	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$			
2)		$l_2(A)$; **Várakozik**		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(A)$; **Várakozik**
5)	$l_1(B)$; $w_1(B)$;		**Megsebezve**	
6)	$u_1(A)$; $u_1(B)$;			
7)		$l_2(A)$; $l_2(C)$;		
8)		$r_2(C)$; $w_2(A)$;		
9)		$u_2(A)$; $u_2(C)$;		
10)				$l_4(A)$; $l_4(D)$;
11)				$r_4(D)$; $w_4(A)$;
12)				$u_4(A)$; $u_4(D)$;
13)			$l_3(B)$; $r_3(B)$;	
14)			$l_3(C)$; $w_3(C)$;	
15)			$u_3(B)$; $u_3(C)$;	

10.14. ábra. *A tranzakciók műveletei a megsebez-megvár sémában érzékelik a holtpontot*

Miért működik jól az időbélyegző alapú holtpontérzékelés?

Azt állítjuk, hogy sem a megvár-meghal, sem a megsebez-megvár sémával nem alakulhat ki kör a várakozási gráfban, így holtpont sem jöhet létre. Tegyük fel az ellenkezőjét, vagyis hogy létezik egy $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$ kör. A legidősebb tranzakció legyen mondjuk T_2.

A megvár-meghal sémában mindig csak újabb tranzakciókra lehet várni. Így nem lehetséges, hogy T_1 várjon T_2-re, mivel T_2 biztosan idősebb T_1-nél. A megsebez-megvár sémában csak idősebb tranzakciókra lehet várni. Tehát semmiképp sem lehetséges, hogy T_2 az újabb T_3-ra várakozzon. Ezekből következik, hogy a kör nem létezhet, ezért holtpont sincsen.

rolni A-t, várakoznia kell, hiszen T_1 idősebb T_2-nél. Miután a 3) lépésben T_3 zárolja B-t, T_4-nek is várnia kell A zárjára.

Ez után tegyük fel, hogy az 5) lépésben T_1 folytatja a működését, zár alá akarja helyezni B-t. B zárja már T_3 birtokában van, de mivel T_1 az idősebb, „megsebzi" T_3-t. T_3 még nem zárult le, a seb halálos: feloldjuk T_3 zárjait, őt pedig visszagörgetjük, így T_1 befejeződhet.

Amikor T_1 feloldja A zárját, tegyük fel, hogy ez T_2-höz kerül, amely ekkor továbbléphet. T_2 után a zárat T_4 kapja meg, amely ezután befejeződik. Végül T_3 újraindul és közbeavatkozás nélkül lezárul. □

10.3.5. A holtpontkezelő módszerek összehasonlítása

A megvár-meghal és a megsebez-megvár sémában is az idősebb tranzakciók maguknál újabb tranzakciókat végeznek ki. Mivel a tranzakciók mindig az eredeti időbélyegzőjükkel indulnak újra, egyszer mindegyikre teljesülni fog, hogy éppen ő a legidősebb a rendszerben, és mint ilyen, biztosan lezárul. Erre a biztosítékra, hogy végül mindegyik tranzakció befejeződik, úgy hivatkozunk, hogy *nincs kiéheztetés*. Vegyük észre, hogy a fejezetben leírt többi séma nem feltétlenül előzi meg a kiéheztetést; ha nem teszünk további intézkedéseket, a tranzakciókkal újra és újra megtörténhet, hogy az indításuk után holtpontra jutnak, és ezért visszagörgetjük őket. Ezzel kapcsolatban nézzük meg a 10.3.7. feladatot.

A megvár-meghal és a megsebez-megvár sémák működése között azonban van egy finom különbség. A megsebez-megvár sémában valahányszor egy idősebb tranzakciónak egy újabb tranzakció birtokában lévő zárra van szüksége, az újabb tranzakció meghal. Ha feltesszük, hogy a tranzakciók az indítás után nem sokkal már birtokba veszik a szükséges zárakat, akkor ritkán fog az előfordulni, hogy egy újabb tranzakció megszerzi a zárakat az idősebb elől. Ebben a sémában tehát a tranzakciók visszagörgetésére várhatóan csak ritkán kerül sor.

Másrészt, amikor a megvár-meghal séma görget vissza egy tranzakciót, akkor az

még mindig a „zárgyűjtő" stádiumában van, amely feltehetően a tranzakció legkorábbi szakaszát jelenti. Így, habár a megvár-meghal séma talán több tranzakciót görget vissza, mint a megsebez-megvár séma, ezek a tranzakciók csak rövid ideig működtek az abortálás előtt. Ezzel szemben a megsebez-megvár sémában visszagörgetett tranzakció valószínűleg már birtokba vette az összes szükséges zárat, és tekintélyes mennyiségű processzoridőt vett igénybe az eddigi működése. A feldolgozandó tranzakcióktól függően tehát hol az egyik, hol a másik módszer eredményez több fölösleges munkát.

Most vizsgáljuk meg a két séma előnyeit és hátrányait a várakozási gráf kézenfekvő konstrukciójával és használatával szemben is. A következő szempontok a lényegesek:

- A megsebez-megvár és a megvár-meghal sémát is könnyebb megvalósítani, mint egy olyan rendszert, amely folyamatosan nyilvántartja vagy időszakosan létrehozza a várakozási gráfot. A várakozási gráf felépítésével járó hátrány még rendkívülibb, amikor osztott adatbázisokkal van dolgunk, és a gráfot a különböző munkaállomásokról[4] összegyűjtött zártáblák alapján kell elkészíteni. Erről bővebben a 10.6. részben lesz szó.
- A várakozási gráf használatával minimálisra csökkenthető a holtpont miatt abortáltatott tranzakciók száma. A tranzakciót mindig csak akkor abortáltatjuk, ha tényleg holtpontra jutott. Másrészt azonban, a megsebez-megvár és a megvár-meghal sémával is előfordulhat néha, hogy olyankor görget vissza egy tranzakciót, amikor nem jött volna létre holtpont, és a tranzakció életben hagyásával sem alakulhatott volna ki.

10.3.6. Feladatok

10.3.1. feladat: Tegyük fel, hogy az alábbi műveletsorozatokban minden egyes olvasás-, illetve írásműveletet közvetlenül megelőzi az osztott, illetve kizárólagos zár igénylése. Tegyük fel továbbá, hogy a zárak feloldása rögtön a tranzakció utolsó művelete után történik meg. Adjuk meg azokat a műveleteket, amelyeknek a végrehajtását az ütemező megtagadja, és mondjuk meg, hogy létrejön-e holtpont vagy sem! Adjuk meg továbbá, hogy hogyan alakul a műveletek végrehajtása során a várakozási gráf! Ha létrejön egy holtpont, abortáltassuk az egyik tranzakciót, és mutassuk meg, hogyan folytatódik a műveletsorozat!

* a) $r_1(A)$; $r_2(B)$; $w_1(C)$; $r_3(D)$; $r_4(E)$; $w_3(B)$; $w_2(C)$; $w_4(A)$; $w_1(D)$;
b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;
c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;
d) $r_1(A)$; $r_2(B)$; $w_1(C)$; $w_2(D)$; $r_3(C)$; $w_1(B)$; $w_4(D)$; $w_2(A)$;

10.3.2. feladat: Hogyan játszódnak le a 10.3.1. feladat műveletsorozatai a megsebez-megvár holtpont-megelőzési rendszerben? Tegyük fel, hogy a holtpont-időbélyegzők sorrendje megegyezik a tranzakciók indexével, azaz T_1, T_2, T_3, T_4. Tegyük fel azt is, hogy a tranzakciók esetleges újraindítása a visszagörgetés sorrendjében történik.

[4] Angolul *site*. A hálózatban részt vevő számítógépeket, „helyszíneket" nevezzük így. *A fordító megjegyzése.*

10.3.3. feladat: Hogyan játszódnak le a 10.3.1. feladat műveletsorozatai a megvár-meghal holtpont-megelőzési rendszerben? Használjuk ugyanazokat a feltevéseket, mint a 10.3.2. feladatban!

! **10.3.4. feladat:** Igaz-e, hogy minden $n > 1$ egészhez létezik olyan várakozási gráf, amelyben a legrövidebb kör hossza n? Mi a helyzet $n = 1$, vagyis hurokél esetén?

!! **10.3.5. feladat:** A holtpontok elkerülésére a következő módszer is megadható: minden tranzakció a futása elején bejelenti, hogy mely zárakra lesz szüksége, ezeket vagy mind megkapja, vagy egyiket sem kapja meg, és várakoznia kell. Elkerülhető-e ezzel a módszerrel a zárolások miatt kialakuló holtpont? Ha igen, állításunkat indokoljuk; ha nem, adjunk ellenpéldát!

! **10.3.6. feladat:** Vegyük a 9.6. részben megismert szándékjelölő zárolási rendszert. Írjuk le, hogyan lehetne ehhez a zárrendszerhez várakozási gráfot készíteni! Vegyük például azt a lehetőséget, amikor az A adatbáziselemet különböző tranzakciók S, illetve IX módban zárolják. Milyen éleket rajzolunk a gráfba, ha A zárolásával valamelyik tranzakciónak várnia kell?

*! **10.3.7. feladat:** A 10.3.5. részben rámutattunk arra, hogy a megsebez-megvár, illetve megvár-meghal sémától eltérő holtpontérzékelő módszerek nem feltétlenül előzik meg a tranzakciók kiéheztetését, amikor is a tranzakciót ismételten visszagörgetjük, így az sohasem éri el a végpontját. Adjunk példát arra, hogy hogyan vezethet a tranzakciók kiéheztetéséhez annak a módszernek a használata, amikor minden olyan tranzakciót visszagörgetünk, amely kört hozna létre a várakozási gráfban! Megelőzhető-e a kiéheztetés, ha a tranzakciók csak egy rögzített sorrendben zárolhatják az elemeket? Mit mondhatunk az időtúllépésről, mint holtpontfeloldó mechanizmusról?

10.4. Osztott adatbázisok

Ebben a fejezetben az osztott adatbázisrendszerek alapelemeit tekintjük át. Egy osztott rendszerben sok, viszonylag autonóm processzorral rendelkezünk, amelyek mind részt vehetnek az adatbázis-műveletekben. Az osztott adatbázisok alkalmazásában számos lehetőség rejlik:

1. Mivel egyszerre több gép is dolgozhat ugyanazon a problémán, jobbak a lehetőségek a párhuzamosításra és arra, hogy a lekérdezésekre gyorsan kapjuk meg a választ.
2. Mivel az adatok többszörözés útján számos helyszínen is jelen lehetnek, a rendszernek valószínűleg nem kell leállnia a feldolgozással csak azért, mert az egyik munkaállomás vagy alkotóelem az adott pillanatban nem érhető el.

Másrészt azonban az osztott feldolgozás minden tekintetben növeli az adatbázisrendszer összetettségét, ezért újra kell gondolnunk az ABKR legalapvetőbb elemeinek a tervezését is. Sok osztott környezetben a kommunikációs költség túlnőhet a feldolgozás költségén, így kritikus témává válik az elküldött üzenetek száma. Ebben a fejezetben az osztott adatbázisok fő kérdései kerülnek bevezetésre, az utána következő részekben pedig az osztott adatbázisok két fontos problémájának, az osztott véglegesítésnek és az osztott zárolásnak megoldására összpontosítunk.

10.4.1. Osztott adatok

Az adatok szétosztásának egyik fontos oka lehet, hogy az adatbázist felhasználó szervezet maga is több helyszínre van szétosztva, és minden munkaállomáson megvannak az elsősorban odatartozó adatok. Lássunk néhány példát:

1. Egy banknak lehet sok fiókja. Minden fiók (vagy egy adott város fiókcsoportja) rendelkezik a nála (vagy a városban) vezetett számlák adatbázisával. Az ügyfelek akármelyik fiókban elintézhetik a pénzügyeiket, de általában a „saját" fiókjukba járnak, ahol a számla adatait is tárolják. A banknak lehetnek a központi fiókban tárolt adatai is, például az alkalmazottakról vagy a banki politikáról, amely például az aktuális kamatlábat is jelentheti. Természetesen az egyes fiókokban tárolt adatrekordokról készült biztonsági másolatot (backup) is tárolják valahol, de valószínűleg nem az adott és nem is a központi fiókban.
2. Egy üzlethálózat sok önálló áruházból állhat. Minden üzlet (vagy egy város üzletcsoportja) rendelkezik az adott üzletben történt vásárlásokra vonatkozó adatbázissal és egy raktáradatbázissal. Lehet, hogy van egy központi részleg, ahol az alkalmazottakról, az üzletláncszintű leltárról vagy a hitelkártyával fizető vásárlókról tárolnak adatokat, a szállítókról pedig olyan információkat tartanak nyilván, mint például a nem teljesített megrendelések, vagy hogy mennyivel tartozik még az üzletlánc. Ezenfelül létezhet még egy „adattárház" is, amelyben megtalálható az összes üzletben nyilvántartott vásárlási adatok másolata. Ezt az elemzők arra használják, hogy ad hoc lekérdezések segítségével analizálják, illetve előre jelezzék a várható keresletet (lásd 11.3. részt).
3. On-line könyvekkel és egyéb dokumentumokkal rendelkező egyetemek közös erővel létrehozhatnak egy digitális könyvtárat. Akármelyik munkaállomáson indítunk egy keresést, az az elérhető dokumentumok *uniójának* a katalógusát fogja megvizsgálni, és ha valamelyik munkaállomáson elérhető a keresett dokumentum, a felhasználó megkapja az elektronikus másolatát.

Néhány esetben az a reláció, amire logikailag úgy gondolunk, mintha egyetlen reláció lenne, valójában sok különböző helyen megtalálható részre van szétosztva. Úgy képzelhetjük például, hogy az üzlethálózat egyetlen vásárlási relációval rendelkezik:

```
Eladások(árucikk, dátum, ár, vevő)
```


A kommunikációs költségben szerepet játszó tényezők

Mivel manapság egyre olcsóbban egyre nagyobb sávszélesség érhető el, eltűnődhetünk azon, vajon számításba kell-e vennünk a kommunikációs költséget az osztott adatbázisrendszer tervezése közben. A legnagyobb elektronikusan kezelt objektumok között most az adatok bizonyos fajtái is megtalálhatók, tehát még egy nagyon olcsó kommunikációs megoldás mellett sem hanyagolható el egy terabájt méretű adatdarab továbbításának a költsége. A legtöbb esetben azonban a kommunikációs költség nem csupán a bitek átküldéséből áll – figyelembe kell vennünk a különböző protokollrétegeket is, amelyek előkészítik, majd a vevő oldalon helyreállítják a továbbításra szánt adatokat, és kezelik az egész kommunikációt. Ezen protokollok mindegyike tekintélyes mennyiségű számítást igényel. A számítások elvégzése is egyre kevesebbe kerül, a kommunikáció miatt végrehajtott számítások mennyisége azonban valószínűleg még mindig jelentős marad ahhoz képest, amennyire a legfontosabb adatbázis-műveletek elvégzéséhez a hagyományos, egyprocesszoros rendszerekben szükség van.

Ez a reláció azonban fizikailag nem létezik, csak mint a hálózat üzleteiben tárolt, azonos sémával rendelkező számos reláció uniója. Ezeket a helyi relációkat *töredékeknek* (fragments) hívjuk, a logikai reláció fizikai részekre bontását pedig az `Eladások` reláció *vízszintes irányú (horizontális) dekompozíciójának* nevezzük. A felbontásra azért hivatkozunk „vízszintesként", mert a „nagy" `Eladások` reláció részekre bontását úgy is elképzelhetjük, mintha vízszintes vonalakkal választanánk szét az egyes üzletekhez tartozó sorokat egymástól.

Más helyzetekben úgy tűnik, mintha az osztott adatbázis a relációt „függőlegesen" bontotta volna fel, mégpedig úgy, hogy ami logikailag egy reláció lenne, azt két vagy több, más-más munkaállomáson megtalálható relációra osztja szét, amelyek attribútumai az eredeti attribútumhalmaz részhalmazát képezik. Például, ha meg akarjuk keresni azokat a vásárlásokat a bostoni áruházban, ahol a vevő több mint 90 napos hátralékban van a hitelkártyaszámla kiegyenlítésével, akkor hasznos lenne egy olyan reláció (vagy nézettábla), amely az `Eladások` táblázatból az `árucikk`, `dátum` és `vevő` információkkal együtt a vevő utolsó hitelkártyaszámla befizetésének az időpontját is tartalmazná. Az itt leírt esetben azonban ez a reláció függőlegesen fel van bontva, ezért össze kellene kapcsolnunk a központban tárolt hitelkártyás vásárló relációt a bostoni áruházban nyilvántartott `Eladások` töredékkel.

10.4.2. Osztott tranzakciók

Az adatok megosztásának az az egyik következménye, hogy a tranzakciók különböző helyeken végrehajtandó folyamatokat is magukban foglalnak. Így az eddigi tranzakciómodellünket meg kell változtatni. A tranzakciót nem tekinthetjük többé egy darab kódnak, amit az egyetlen ütemezővel és az egyetlen naplókezelővel kommunikáló

egyetlen processzor egyetlen számítógépen hajt végre. A tranzakció most egymással kapcsolatot tartó, különböző munkaállomásokon megtalálható *tranzakció-alkotórészekből* áll, amelyek mindegyike az adott munkaállomáson működő lokális ütemezővel és naplókezelővel kommunikál. Két fontos kérdést kell újra átgondolnunk:

1. Hogyan kezeljük a véglegesítés/abortálás döntést osztott tranzakciók esetén? Mi történik, ha a tranzakció egyik alkotórésze abortáltatni akarja az egész tranzakciót, de a többi alkotórésznek nincs problémája, és inkább véglegessé szeretne válni? A 10.5. részben megtárgyaljuk a „kétfázisú véglegesítés" technikáját, melynek segítségével helyes döntés hozható, és amely gyakran akkor is figyelembe veszi a még elérhető munkaállomásokat, ha néhány másik már átmenetileg működésképtelenné vált.
2. Hogyan biztosítható az olyan tranzakciók sorbarendezhetősége, amelyek különböző helyeken végrehajtott alkotórészeket foglalnak magukban? A 10.6. részben különös figyelmet szentelünk a zárolás problémájának, és megnézzük, hogyan használhatók a lokális zártáblák az adatbáziselemek globális zárolásához, és így hogyan támogatják egy megosztott környezetben a tranzakciók sorbarendezhetőségét.

10.4.3. Adattöbbszörözés

Az osztott rendszereknek egy fontos előnye az, hogy az adatokat *többszörözni* tudjuk, azaz az adatelemekről másolatokat tarthatunk több helyen is. Ez egyrészt hasznos abban az esetben, ha az egyik munkaállomás meghibásodik, mert ilyenkor egy másik helyen is megtalálhatók azok az adatok, amelyekhez most a meghibásodás miatt nem lehet hozzáférni. Másrészről pedig növelhetjük a válaszadás sebességét azzal, hogy a szükséges adatokból másolatokat tárolunk azon a munkaállomáson, ahol a lekérdezést elindítják.

Például:

1. A bank minden fiókjában tárolhat egy-egy másolatot az aktuális kamatlábakról, így az erre vonatkozó lekérdezéseket nem kell elküldeni a központi fiókba.
2. Az áruház minden egyes üzletében tárolhat egy-egy másolatot a szállítókról nyilvántartott információkról, így ha lokálisan szükség van valamilyen adatra (például az üzletvezető tudni szeretné egy szállító telefonszámát, mert ellenőrizni akarja a szállítmányt), akkor ehhez nem kell a központi részlegbe üzeneteket küldeni.
3. A digitális könyvtár ideiglenesen eltárolhatja (cache-elheti) egy népszerű dokumentum másolatát abban az iskolában, ahol a diákoknak ez kötelező olvasmány.

A többszörözött adatokkal kapcsolatban azonban számos problémával is szembe kell néznünk.

a) Hogyan oldható meg, hogy a másodpéldányok mind azonosak legyenek? A többszörözött adat módosítása lényegében egy olyan osztott tranzakcióvá válik, amely az összes másolatot módosítja.

b) Hogyan döntsük el, hogy hol, mennyi másodpéldányt tároljunk? Minél több másolattal rendelkezünk, annál nagyobb erőfeszítést vesz igénybe az adat módosítása, viszont annál könnyebb lesz a lekérdezések végrehajtása. Egy ritkán módosított relációból például a maximális hatásfok érdekében mindenhol tárolhatunk másolatot, viszont egy gyakran módosított relációból csak egy vagy kettő másodpéldányunk lehetne.
c) Mi történik hálózati kommunikációs hiba esetén, amikor ugyanannak az adatnak különböző másodpéldányai egymástól független változásokon mehetnek keresztül, és amikor a hálózat újra helyreáll, az adat másolatait valahogy össze kell egyeztetni?

10.4.4. Osztott lekérdezésoptimalizálás

A megosztott adatok jelenléte hatást gyakorol a fizikai lekérdezési terv létrehozásakor választható lehetőségekre és a terv összetettségére is (lásd 7.7. részt). Többek között a következő kérdéseket kell eldöntenünk, amikor fizikai tervet választunk:

1. Ha a szükséges R relációnak több másolata is létezik, melyikből vegyük R értékét?
2. Ha egy kétrelációs műveletet, például összekapcsolást alkalmazunk R-re és S-re, akkor számos lehetőség közül kell kiválasztanunk egyet. Néhány ezekből:

   a) Lemásolhatjuk S-t az R-t tároló munkaállomásra, és a számítást ott végezzük el.
   b) Lemásolhatjuk R-t az S-t tároló munkaállomásra, és a számítást ott végezzük el.
   c) Lemásolhatjuk R-t és S-t is egy harmadik helyre, és a számítást ott végezzük el.

   Hogy melyik a legjobb választás, az többek között olyan tényezőktől is függ, hogy melyik helyen van elérhető számítási kapacitás, és hogy a művelet eredményét kombináljuk-e egy harmadik munkaállomáson található adattal. Például ha $(R \bowtie S) \bowtie T$ eredményét kell kiszámítanunk, azt a lehetőséget is választhatjuk, hogy R-t és S-t is továbbítjuk a T-t tároló munkaállomásra, és mindkét összekapcsolást ott végezzük el.

Ha az R reláció az $R_1, R_2, \ldots, R_n$ töredékek képében van több helyre szétosztva, akkor a logikai lekérdezési terv választásakor R helyett mindenhol az

$$R_1 \cup R_2 \cup \ldots \cup R_n$$

uniót kell írnunk. A lekérdezéstől függően aztán jelentősen egyszerűsíthetjük a kifejezést. Például, ha mindegyik R_i a 10.4.1. részben tárgyalt `Eladások` reláció egy-egy töredéke, és minden töredék egy-egy üzlethez van hozzárendelve, akkor a bostoni áruház eladásaival kapcsolatos lekérdezésekben a bostoni töredéken kívül minden más töredéket elhagyhatunk az unióból.

10.4.5. Feladatok

*!! **10.4.1. feladat:** Ebben a feladatban lehetőségünk lesz néhány olyan probléma felvetésére, amelyek az adattöbbszörözési stratégia kiválasztásakor jönnek elő. Tegyük fel, hogy az R relációhoz n munkaállomás férhet hozzá. Az i-edik munkaállomás másodpercenként q_i lekérdezést és u_i módosítást hajt végre R-en, $i = 1, 2, \ldots, n$. A lekérdezés végrehajtásának a költsége az R relációval rendelkező munkaállomáson c, de ha az adott munkaállomás nem tárol másodpéldányt R-ről, ezért a lekérdezést továbbítani kell egy másikra, a költség $10c$. A módosítás végrehajtása egy „helyi" R-en d, de minden távoli példányon $10d$. Ezen paraméterek függvényében, nagy n esetén, hogyan döntenénk el, hogy mely munkaállomások rendelkezzenek R másodpéldányával és melyek ne?

10.5. Osztott véglegesítés

Ebben a fejezetben arról lesz szó, hogy hogyan biztosítható a különböző helyeken végrehajtott alkotórészekből álló osztott tranzakció atomossága. A következő fejezet az osztott tranzakciók egy másik fontos tulajdonságát, a sorbarendezhetőséget tárgyalja. A következő példán keresztül bemutatjuk, hogy milyen problémák merülhetnek fel.

10.15. példa: Vegyük a 10.4. részben említett üzlethálózatot. Tegyük fel, hogy a hálózat igazgatója minden egyes áruházról tudni akarja, hogy mennyi fogkeféje van raktáron, majd ezek alapján kiegyenlíti a készleteket, azaz néhány áruháznak olyan utasítást fog adni, hogy valamelyik másik áruházba helyezze át a fogkefekészletének egy bizonyos részét. Az egész műveletet egyetlen globális T tranzakció végzi el, amelynek több alkotórésze is van: az i-edik üzletben T_i, az igazgató irodájában pedig T_0. T a következő tevékenységeket hajtja végre:

1. Létrehozza T_0 alkotórészt az igazgató munkaállomásán.
2. T_0 minden áruházba üzenetet küld, utasítva őket, hogy hozzák létre a T_i alkotórészeket.
3. Minden T_i végrehajt egy lekérdezést az i-edik üzletben, amelyből megtudja a raktáron lévő fogkefék számát, majd továbbítja ezt az értéket T_0-nak.
4. T_0 a visszajelzett értékek alapján, valamilyen, itt nem tárgyalt algoritmus segítségével meghatározza, hogy a fogkefeállományt hogyan kell átszervezni. Ezután T_0 olyan üzeneteket küld a megfelelő üzleteknek (ebben az esetben a 7-esnek és a 10-esnek), mint „a 10-es számú áruház szállítson át 500 fogkefét a 7-es számú áruházba".
5. Az üzletek a kapott utasítások alapján módosítják a leltárukat, és végrehajtják a szükséges szállításokat. □

10.5.1. Az osztott atomosság támogatása

A 10.15. példában számos dolog elromolhat, és ezek közül sok eredményezheti azt, hogy sérül T atomossága, azaz néhány műveletét végrehajtja a rendszer, de a többit nem. A naplózás és a helyreállítás mechanizmusa, amelyről feltesszük, hogy minden munkaállomáson jelen van, biztosítja az egyes T_i-k atomosságát, de nem biztosítja, hogy maga T is atomosan fusson le.

10.16. példa: Tegyük fel, hogy a fogkeféket újraosztó algoritmus hibás és ennek következtében a 10-es áruháznak több fogkefét kellene átszállítania, mint amennyi a raktárán van. Ezért T_{10} abortálni fog, és a 10-es üzletből nem szállítanak át egyetlen fogkefét sem, és az üzlet leltárát sem módosítják. T_7 viszont nem talál semmi problémát, és miután a várt fogkefeszállítmánynak megfelelően módosította a 7-es üzlet leltárát, véglegessé válik. Most tehát T nemcsak hogy nem volt atomos (mivel T_{10} soha nem fut le), de ráadásul még inkonzisztens állapotban is hagyta az adatbázist: a leltárban szereplő fogkefék száma nem egyenlő a raktárban ténylegesen megtalálható fogkefék számával. □

A problémák egy másik forrását jelenti annak a lehetősége, hogy egy munkaállomás meghibásodik, vagy leszakad a hálózatról az osztott tranzakció futása közben.

10.17. példa: Tegyük fel, hogy T_{10} még válaszol T_0 első kérdésére, vagyis megküldi a raktáron lévő fogkefék számát, de a 10-es üzlet számítógépe ezután működésképtelenné válik, így T_{10} soha nem kapja meg T_0 utasításait. Véglegessé válhat-e valaha is az osztott T tranzakció? Mit kellene T_{10}-nek tennie, miután a számítógép megjavul? □

10.5.2. Kétfázisú véglegesítés

Annak érdekében, hogy a 10.5.1. részben jelzett problémák elkerülhetők legyenek, az osztott ABKR-ek egy összetett protokollt használnak annak az eldöntésére, hogy egy osztott tranzakció véglegessé váljon-e vagy ne. Ez a *kétfázisú véglegesítés protokoll*[5], és ebben a fejezetben az eljárás alapötletével ismerkedünk meg. Hogy globális döntést hozunk a tranzakció véglegesítéséről, az azt jelenti, hogy vagy minden alkotórésze véglegessé válik, vagy egyetlenegy sem. Ahogy szokásos, most is feltesszük, hogy az egyes munkaállomásokon a lokális alkotórészek vagy véglegessé válnak, vagy nincs hatásuk egyáltalán az adott adatbázisra, vagyis hogy a tranzakció alkotórészei atomosak. Így annak a szabálynak a követésével, hogy az osztott tranzakció vagy minden alkotórésze véglegessé válik, vagy egyik sem, maga az osztott tranzakció is atomossá tehető.

Most következzék néhány kiugróan lényeges dolog a kétfázisú véglegesítés protokollal kapcsolatban:

[5] Ne keverjük össze a kétfázisú véglegesítést a kétfázisú zárolással. Ez két, egymástól független elgondolás, egymástól eltérő problémák megoldására.

- Föltesszük, hogy minden munkaállomás naplózza a saját eseményeit, és hogy nincs globális napló.
- Azt is feltesszük, hogy az egyik munkaállomás speciális szerepet játszik annak eldöntésében, hogy az osztott tranzakció véglegessé válhat-e vagy sem. Ezt a munkaállomást *koordinátornak* nevezzük. Koordinátor lehet például az a munkaállomás, ahonnan a tranzakció ered; ez a 10.5.1. rész példáiban a T_0 munkaállomása.
- A kétfázisú véglegesítés protokoll során a koordinátor és a többi munkaállomás bizonyos üzeneteket váltanak egymással. Minden üzenetet küldő munkaállomás naplózza az általa küldött üzeneteket, ezzel segítve az esetleg szükséges helyreállítás műveletét.

Ezeket észben tartva most már jellemezhető a két fázis a munkaállomások közti üzenetcserék leírásával.

Első fázis

A kétfázisú véglegesítés első fázisában a T osztott tranzakció koordinátora eldönti, hogy mikor kísérli meg T véglegesítését. A kísérlet feltehetően akkor történik meg, amikor a koordinátornál futó alkotórész már készen áll a véglegesítésre, de elvben a lépéseket akkor is végre kell hajtani, ha ez az alkotórész abortálni szándékozik (persze nyilvánvaló egyszerűsítések mellett, ahogy ezt majd látni fogjuk). A koordinátor a T tranzakció alkotórészeihez tartozó összes munkaállomást megszavaztatja arról, hogy a véglegesítés vagy az abortálás mellett van-e.

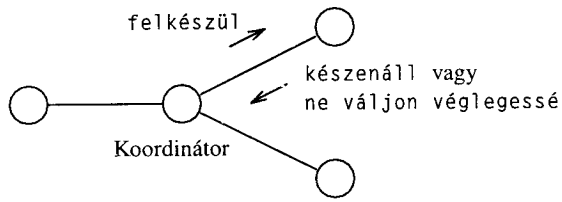
1. A koordinátor a saját naplójába felveszi a <T `Felkészül`> bejegyzést.
2. A koordinátor minden munkaállomásra (elméletileg a sajátjára is) elküldi a T `felkészül` üzenetet.
3. Minden munkaállomás, amely megkapta a T `felkészül` üzenetet, eldönti, hogy a nála található T alkotórészt véglegesíteni vagy abortáltatni akarja. A döntés késleltethető, ha az adott alkotórész még folyamatban van, de a munkaállomásnak végül vissza kell jeleznie.
4. Ha a munkaállomás véglegesíteni akarja az alkotórészt, akkor ennek az *előzetesen véglegesített* állapotba kell lépnie. Az alkotórészt ebben az állapotban a munkaállomás már csak akkor abortáltathatja, ha erre a koordinátortól utasítást kap. Hogy T alkotórésze az előzetesen véglegesített állapotba jusson, a következőket kell megtenni:

   a) Az összes olyan lépést végre kell hajtani, amely annak a biztosításához szükséges, hogy T lokális alkotórészének ne kelljen majd abortálnia, a munkaállomáson bekövetkező rendszerhibát követő helyreállítás során sem. Így nem csak a lokális T műveleteit kell elvégezni, hanem a megfelelő naplózási műveleteket is, hogy egy esetleges helyreállítás során T hatását ne semmisítsük meg, legfeljebb újra futtassuk a tranzakciót. A tényleges műveletek a naplózási módszertől függenek, de a lokális T működéséhez tartozó naplóbejegyzéseket mindeneképpen ki kell írni a lemezre.
   b) Fel kell venni a <T `Készenáll`> bejegyzést a lokális naplóba, és a naplót ki kell írni a lemezre.
   c) El kell küldeni a koordinátornak a T `készenáll` üzenetet.

T alkotórészét azonban még nem véglegesíti ebben a lépésben a munkaállomás; ezzel várnia kell a második fázisig.

1. Ha a munkaállomás inkább abortáltatni akarja az alkotórészt, akkor naplózza a <T `Nem Válik Véglegessé`> bejegyzést és elküldi a T `ne váljon véglegessé` üzenetet a koordinátornak. Biztonságos már ebben a lépésben abortáltatni az alkotórészt, hiszen T is biztosan abortál, még akkor is, ha csak egyetlenegy alkotórész szavazott a véglegesítés ellen.

Az első fázisban váltott üzeneteket a 10.15. ábrán foglaltuk össze.



10.15. ábra. *A kétfázisú véglegesítés első fázisában váltott üzenetek*

Második fázis

A második fázis akkor kezdődik, amikor a koordinátor mindegyik munkaállomástól megkapta a `készenáll` vagy a `ne váljon véglegessé` üzenetet. Lehetséges azonban, hogy valamelyik munkaállomás nem válaszol, talán mert meghibásodott, vagy leszakadt a hálózatról. Ebben az esetben a koordinátor egy megfelelő időkorlát túllépése után úgy fogja venni, mintha ez a munkaállomás a `ne váljon véglegessé` választ küldte volna.

1. Ha a koordinátor a T `készenáll` üzenetet kapta T minden alkotórészétől, akkor a T véglegesítése mellett fog dönteni. A koordinátor

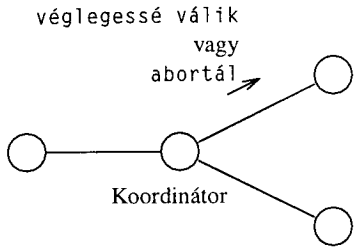
   a) Felveszi a saját naplójába a <`Commit` T> bejegyzést, és
   b) Elküldi a T `véglegessé válik` üzenetet minden T-hez tartozó munkaállomásra.

2. Ha a koordinátor egy vagy több munkaállomástól a `ne váljon véglegessé` üzenetet kapta, akkor

a) Felveszi a saját naplójába az <Abort *T*> bejegyzést, és
b) Elküldi a *T* abortál üzenetet minden *T*-hez tartozó munkaállomásra.

3. Ha egy munkaállomás *T* véglegessé válik üzenetet kap, akkor véglegesíti *T* alkotórészét, és felveszi a naplóba a <Commit *T*> bejegyzést.
4. Ha egy munkaállomás *T* abortál üzenetet kap, akkor abortáltatja *T* alkotórészét, és felveszi a naplóba az <Abort *T*> bejegyzést.

A második fázis üzeneteit a 10.16. ábrán foglaltuk össze.



10.16. ábra. *A kétfázisú véglegesítés második fázisában váltott üzenetek*

10.5.3. Az osztott tranzakciók helyreállítása

A kétfázisú véglegesítés folyamata alatt bármikor meghibásodhat egy munkaállomás. Biztosítanunk kell, hogy ami a helyreállításkor történik, az megfelel annak a döntésnek, amelyet a *T* osztott tranzakcióról hoztunk. Attól függően, hogy az adott helyen mi a *T*-vel kapcsolatos utolsó naplóbejegyzés, a következő eseteket kell végiggondolnunk:

1. Ha a *T*-re vonatkozó utolsó naplóbejegyzés <Commit *T*>, akkor a koordinátor biztosan véglegesítette *T*-t. A naplózási módszertől függően szükség lehet arra, hogy a helyreállítás során *T* alkotórészét újra végrehajtsák.
2. Ha az utolsó feljegyzés <Abort *T*>, akkor az előző esethez hasonlóan itt is ismerjük a globális döntést: *T* abortált. Ha a naplózási módszer igényli, a *T* alkotórész hatását meg kell semmisíteni.
3. Ha az utolsó bejegyzés <*T* Nem válik véglegessé>, akkor a globális döntés biztosan *T* abortálása volt. Ha szükséges, *T* hatását a lokális adatbázison megsemmisítjük.
4. Nehéz a dolgunk, ha *T*-vel kapcsolatban az utolsó feljegyzés <*T* Készenáll>. Ilyenkor a magát helyreállítani próbáló munkaállomás nem tudja, hogy mi lehetett a globális döntés, ezért ezt egy másik munkaállomástól kell megtudnia. Kommunikálhat a koordinátorral, ha az éppen nem működésképtelen, a koordinátor meghibásodása esetén azonban megkérdezhet egy másik munkaállomást is, amely majd a naplója alapján megadja a szükséges információt. Legrosszabb esetben, ha semelyik másik munkaállomással sem tud kapcsolatba lépni, *T* lokális alkotórészét függőben kell tartania addig, amíg meg nem tudja állapítani, hogy mi volt a *T*-re vonatkozó véglegesítés/abortálás döntés.[6]
5. Az is előfordulhat, hogy a lokális napló egyáltalán nem tartalmaz a kétfázisú véglegesítés protokollal kapcsolatban *T*-re vonatkozó feljegyzést. Ebben az esetben az önhelyreállítást végző munkaállomás egyoldalúan a *T* alkotórész abortáltatása mellett dönthet, ami minden naplózási módszernek megfelel. Lehetséges, hogy a koordinátor egyébként is abortáltatni akarta *T*-t a meghibásodott munkaállomás időkorlát-túllépése miatt. Ha ez csak rövid ideig volt működésképtelen, akkor a többi helyen *T* még mindig aktív lehet, de a lokális *T* alkotórész abortáltatása soha nem okoz problémát, ha a munkaállomás a később kezdődő első fázisban a *T* ne váljon véglegessé választ adja.

A fenti elemzésben feltettük, hogy a meghibásodott munkaállomás nem a koordinátor volt. Ha a koordinátor válik működésképtelenné a kétfázisú véglegesítés folyamata alatt, akkor új problémák merülnek fel.

Először is, a túlélő résztvevőknek vagy meg kell várniuk, amíg a koordinátort helyreállítják, vagy új koordinátort kell választani. Mivel nem lehet előre tudni, hogy a koordinátor mikor lesz újra üzemképes, erős az indíttatás az új vezető választására, legalábbis abban az esetben, amikor a koordinátor még egy rövid várakozási idő eltelte után sem állt helyre.

A *vezetőválasztás* az osztott rendszerek egy méltán összetett problémája – részletesebb vizsgálata túlmutat e könyv célkitűzésein. Egy egyszerű módszer azonban a legtöbb esetben működni fog. Például feltehető, hogy minden résztvevő munkaállomás rendelkezik egy egyedi azonosítóval; az IP-cím sok helyzetben megfelel erre a célra. Minden résztvevő az összes többi résztvevőnek küld egy üzenetet a saját azonosítójával, jelezvén, hogy őt meg lehet választani vezetőnek. Megfelelő idő eltelte után minden résztvevő a legalacsonyabb azonosítószámmal rendelkező munkaállomást ismeri el új koordinátorként azok közül, amelyekről hallott, és erről minden más munkaállomásnak értesítést küld. Ha minden résztvevő azonos üzenetet kap, akkor egyértelmű és mindenki számára ismeretes az új koordinátor személye. Ha az üzenetek nem egyeznek meg egymással, vagy egy résztvevő nem válaszolt, erről szintén mindenki tud majd, és a választást újrakezdhetik.

Most az új vezető szavazás formájában információt gyűjt a munkaállomásoktól a *T* osztott tranzakcióról. Minden munkaállomás elküldi a naplójában található utolsó bejegyzést *T*-vel kapcsolatban, ha van ilyen. A lehetséges esetek a következők:

1. Valamelyik munkaállomás naplójában szerepel a <Commit *T*> bejegyzés. Az eredeti koordinátor bizonyára a *T* véglegessé válik üzenetet akarta mindenhova elküldeni, tehát *T* véglegesítése egy biztonságos megoldás.
2. Hasonlóan, ha valamelyik munkaállomás naplójában az <Abort *T*> bejegyzés sze-

[6] Ez az állapot, *T* blokkolódása az adott csomópontban, akkor is bekövetkezik, ha minden elérhető csomópontban a *T*-vel kapcsolatos utolsó feljegyzés <*T* Készenáll>. *A szerkesztő megjegyzése.*

repel, akkor az eredeti koordinátor biztos, hogy abortáltatni akarta T-t, tehát biztonságos lépés, ha az új koordinátor is ezt teszi.

3. Most tegyük fel, hogy egyetlen munkaállomás naplójában sem szerepel a <Commit T> vagy az <Abort T> bejegyzés, és legalább egy munkaállomás naplójában a <T Készenáll> feljegyzés sem található meg. Mivel az elküldött üzeneteket még a továbbítás előtt naplózzák, világos, hogy a régi koordinátor ettől a munkaállomástól nem kapott T készenáll üzenetet, ezért nem dönthetett T véglegesítése mellett. Az új koordinátor tehát nyugodt lélekkel abortáltathatja T-t.
4. Nehéz a dolgunk abban az esetben, ha egyik helyen sem található <Commit T> vagy <Abort T>, viszont minden túlélő munkaállomás naplójában szerepel a <T Készenáll> bejegyzés. Ilyenkor nem lehetünk biztosak abban, hogy a régi koordinátornak nem volt valamilyen oka T abortáltatására; a saját munkaállomásán történt események, vagy egy másik, jelenleg hibás munkaállomás T ne váljon véglegessé üzenete például megfelelő alapot nyújthatott ehhez a döntéshez. De az is lehet, hogy véglegesíteni akarta T-t, és a saját alkotórésze már véglegessé is vált. Vagyis az új koordinátor nem tudja eldönteni, hogy mit csináljon T-vel, ezért meg kell várnia, amíg az eredeti koordinátor helyreáll. A valódi rendszerekben az adatbázis-adminisztrátornak megvan a lehetősége arra, hogy közbelépjen és manuálisan kényszerítse a várakozó tranzakciókat a továbblépésre. Lehetséges, hogy ezzel sérül a tranzakció atomossága, de a blokkolt tranzakciót végrehajtó személy ennek tudatában megfelelően ellensúlyozza majd ezt a hatást.

10.5.4. Feladatok

! **10.5.1. feladat:** Vegyünk egy olyan T tranzakciót, amelyet egy otthoni számítógépről indítanak, és a segítségével a B bank egy számlájáról 10 000 dollárt akarnak átutalni egy C bankbeli számlára.

* a) Melyek lesznek az osztott T tranzakció alkotórészei? Mi a feladata a B-ben, illetve C-ben található alkotórésznek?

b) Milyen hibát okozhat az, ha a B bankban vezetett számlán nincs 10 000 dollár?

c) Milyen problémát okoz, ha az egyik vagy mindkét bank számítógépe meghibásodik, vagy ha a hálózati kapcsolat megszűnik?

d) Ha a c)-ben felsoroltak közül valamelyik bekövetkezik, hogyan folytatódna helyesen a tranzakció, amikor a számítógépek, illetve a hálózat rendbe jön?

10.5.2. feladat: Ebben a feladatban valahogy jelölnünk kell a kétfázisú véglegesítés folyamata során váltott üzeneteket. Jelentse (i, j, M) a következőt: i munkaállomás az M üzenetet küldi a j munkaállomásnak, ahol M az alábbi értékeket veheti fel: P (jelentése felkészül), R (készenáll), D (ne váljon véglegessé), C (commit), A (abort). Nézzünk meg egy egyszerű esetet, ahol a 0-s sorszámú munkaállomás a koordinátor, amelynek a tranzakcióban máskülönben nincs része, az 1-es és a 2-es munkaállomásokon pedig fut egy-egy alkotórész. Ekkor a tranzakció sikeres véglegesítése során például a következő üzenetsorozat jöhet létre:

$(0, 1, P)$, $(0, 2, P)$, $(2, 0, R)$, $(1, 0, R)$, $(0, 2, C)$, $(0, 1, C)$

* a) Adjunk példát egy olyan üzenetsorozatra, amely akkor jön létre, ha az 1-es munkaállomás a véglegesítés, a 2-es pedig az abortálás mellett van!

*! b) Hány, a fentihez hasonló lehetséges üzenetsorozat jöhet létre, ha a tranzakció sikeresen véglegessé válik?

! c) Hány különböző üzenetsorozat jöhet létre, ha az 1-es munkaállomás a véglegesítés, a 2-es viszont az abortálás mellett van, és feltehető, hogy nem fordul elő semmiféle hibajelenség?

! d) Hány különböző üzenetsorozat jöhet létre, ha az 1-es munkaállomás a véglegesítés mellett szavaz, viszont a 2-es valamilyen meghibásodásnál fogva nem válaszol az üzenetekre?

!! **10.5.3. feladat:** Az előző feladat jelöléseit használva tegyük fel, hogy a koordinátor mellett n másik munkaállomásunk van, és ezek adják a tranzakció alkotórészeit. Adjuk meg n függvényében, hogy hány különböző üzenetsorozat jöhet létre abban az esetben, ha a tranzakció sikeresen véglegessé válik!

10.6. Osztott zárolás

Ebben a fejezetben azzal fogunk foglalkozni, hogy hogyan lehet a zárolás alapú ütemezőket kiterjeszteni egy olyan környezetben, ahol a tranzakciók meg vannak osztva, és különböző helyeken futó alkotórészekből állnak. Feltesszük, hogy a zártáblákat az egyes munkaállomások külön-külön kezelik, és hogy az alkotórész csak azokat az adatbáziselemeket zárolhatja, amelyek azon a helyen megtalálhatók.

Amikor többszörözött adatokkal van dolgunk, úgy kell rendeznünk, hogy egy X adatbáziselem minden másodpéldánya egyformán tükrözze az X-en végrehajtott változtatásokat. Ezért különbséget kell tennünk az X *logikai* adatbáziselem és az X egy vagy több másolatának a zárolása között. Ebben a részben megismerkedünk egy költségmodellel, amelyet az osztott zárolási algoritmusokhoz fejlesztettek ki, és amely akkor is alkalmazható, ha nem többszörözött adatokat használunk. Ennek tárgyalása előtt azonban bemutatjuk a központi zárolás technikáját, amely egy kézenfekvő (és néha éppen megfelelő) megoldás az osztott zárolás problémájára.

10.6.1. Központosított zárolási rendszerek

Talán a legegyszerűbb módszer az, ha a munkaállomások közül kijelölünk egyet, hogy az legyen a *zárállomás* (lock site), azaz hogy az tartsa nyilván a logikai elemek zártábláit, függetlenül attól, hogy rendelkezik-e az elemek másodpéldányával vagy sem. Ha egy tranzakció zárolni akar egy X logikai elemet, akkor ezt az igényét a zárállomásnak nyújtja be, az pedig az adott helyzettől függően vagy engedélyezi, vagy

megtagadja a zárolást. Mivel X globális zárolása megegyezik X lokális zárolásával a zárállomáson, biztosak lehetünk benne, hogy a globális zárak helyesen működnek, feltéve, hogy a lokális zárak adminisztrációja a hagyományos módon történik. A költség általában három üzenet zárolásonként (igénylés, engedélyezés, feloldás), ha a tranzakció nem a zárállomáson fut.

Hogy csak egyetlen zárállomást használunk, ez néhány esetben megfelelő lehet, de ha sok munkaállomáson egyszerre sok tranzakció fut, a zárállomásnál hamar kialakulhat a torlódás. Ráadásul, ha az állomás összeomlik, ez teljesen lebénítja a rendszert, hiszen ilyenkor egyetlen tranzakció sem juthat hozzá semelyik zárhoz, függetlenül attól, hogy éppen hol fut. Ezen problémák miatt számos más módszer is született az osztott zárolás megoldására; ezekre a költségbecslés tárgyalása után kerül sor.

10.6.2. Költségmodell az osztott zárolási algoritmusokhoz

Tegyük fel, hogy minden adatelem pontosan egy helyen fordul elő (vagyis hogy nincs adattöbbszörözés), és hogy az egyes munkaállomásokon működő zárkezelő tartja nyilván az ott elérhető adatelemek zárjait és az azokra benyújtott igényeket. Lehetnek osztott tranzakciók a rendszerben, és mindegyikük egy vagy több különböző helyen futó alkotórészből áll.

A zárkezeléshez ugyan többféle költség is kapcsolódik, de sokuk rögzített, független a hálózaton keresztül történt zárigénylés módjától. Az egyetlen költségtényező, amelyet befolyásolni tudunk, a zárak kiosztásakor és feloldásakor váltott üzenetek száma. Ezért a különböző zárolási sémáknál mindig ezt fogjuk figyelni azon feltevés mellett, hogy az igényelt zárakat mindig ki is osztják. Persze lehet, hogy a zárkezelő nem engedélyezi a zárolást, és ezzel további üzenetek küldésére kerül sor a zárolás megtagadásával, illetve a későbbi engedélyezéssel kapcsolatban. Mivel azonban nem látjuk előre, hogy ez milyen arányban fog előfordulni, és ezt az értéket egyébként sem tudjuk befolyásolni, az összehasonlításban ezeket a pluszüzeneteket nem fogjuk figyelembe venni.

10.18. példa: Ahogy ezt már a 10.6.1. részben említettük, a központi zárolás módszerével a zárkérelmekhez jellemzően három üzenet szükséges: egy az igénylés bejelentéséhez, egy a központi zárállomástól a zár kiadásához, és a harmadik a zár feloldásához. Kivételt a következő esetek jelentenek:

1. Nincs szükség az üzenetekre, ha maga a központi zárállomás igényli a zárolást.
2. További üzenetek küldésére van szükség, ha az első kérelmet nem lehet teljesíteni.

Feltesszük azonban, hogy mindkét eset viszonylag ritkán fordul elő, azaz a legtöbb zárigénylés a központi zárállomástól különböző helyről fut be, és hogy a kérelmek többsége kielégíthető. A zárankénti három üzenet tehát egy jó becslés a központi zár módszer költségére. □

Most vizsgáljunk meg a központi zárolásnál egy rugalmasabb helyzetet, ahol minden X adatbáziselemhez a saját munkaállomásán tartjuk nyilván a hozzá tartozó zárat.

Úgy tűnhet, hogy mivel az X-et zárolni kívánó tranzakció az X munkaállomásán is rendelkezik egy alkotórésszel, a munkaállomások közötti üzenetcserére nincs is szükség: az alkotórész egyszerűen az adott munkaállomás zárkezelőjével tárgyal X-t illetően. Ha azonban az osztott tranzakciónak több elemet is zárolnia kell, mondjuk X-t, Y-t és Z-t is, akkor nem fejezheti be addig a számításait, amíg meg nem szerzi mindhárom elem zárját. Ha X, Y és Z különböző munkaállomásokon található, akkor az ott futó alkotórészeknek legalább szinkronizáló üzeneteket kell cserélniük, hogy a tranzakció nehogy „előbbre járjon saját magánál".

Az összes lehetséges variáció áttekintése helyett csak egy egyszerű modellt veszünk arra, hogy a tranzakciók hogyan gyűjtik be a szükséges zárakat. Feltesszük, hogy minden tranzakciónak az egyik alkotórésze, a *zárkoordinátor* felelős az egyes alkotórészek által igényelt zárak összegyűjtéséért. A zárkoordinátor a saját munkaállomásán üzenetcsere nélkül zárolja az elemeket, de egy másik helyen található X zárolásához három üzenet szükséges:

1. Az igény benyújtása X munkaállomására.
2. A válasz üzenet, amellyel a zárolást engedélyezik (korábban feltettük, hogy az igényeket azonnal kielégítik; ha mégsem, akkor ebben a pontban az üzenet a zárolás elutasítására vonatkozik, amelyet majd később követ az engedélyező üzenet).
3. A zárról való lemondás továbbítása X munkaállomására.

Mivel az osztott zárolási protokollokat csak összehasonlítani akarjuk és nem kívánjuk megadni az üzenetek átlagos számát, ez a leegyszerűsítés megfelel a céljainknak.

Ha azt a munkaállomást választjuk zárkoordinátornak, ahonnan a legtöbb zárat kell beszereznie a tranzakciónak, akkor minimalizáljuk a szükséges üzenetek számát. Ez a többi munkaállomáson található adatbáziselemek számának háromszorosával egyenlő.

10.6.3. Többszörözött elemek zárolása

Óvatosan kell az X adatelem zárolását értelmeznünk, amikor X-ből különböző helyeken másodpéldányok is megtalálhatók.

10.19. példa: Tegyük fel, hogy az X adatbáziselem két példányban létezik, ezek X_1 és X_2. Tegyük fel továbbá, hogy a T tranzakció X_1 munkaállomásán osztott zár alá helyezte X_1-et, U viszont kizárólagos zárat birtokol X_2 munkaállomásán X_2-n. Ilyenkor U csak X_2-t módosíthatja, X_1-et nem, és ez azt eredményezi, hogy X két példánya egymástól eltérő lesz. Sőt mivel T és U más elemeket is zár alá helyezhet, és az, hogy milyen sorrendben olvassák, illetve írják X-et, független a másodpéldányokon birtokolt záraktól, T-nek és U-nak arra is lehetősége van, hogy nem sorba rendezhető módon viselkedjenek. □

A 10.19. példában bemutatott probléma lényege, hogy többszörözött adatok esetén különbséget kell tennünk az X logikai elem osztott, illetve kizárólagos zárolása, és az

X egy másodpéldányának az adott munkaállomáson történő helyi zárolása között. Vagyis annak érdekében, hogy biztosítani lehessen a sorbarendezhetőséget, a tranzakcióknak globálisan kell zárolniuk a logikai elemeket. Viszont a logikai elemek fizikailag nem léteznek – csak a másodpéldányaik –, és globális zártábla sincs. Így a tranzakció csak egyetlen módon juthat hozzá X globális zárjához: ha megszerzi X egy vagy több másodpéldányán az adott munkaállomáson a lokális zárakat. Most olyan módszerekkel fogunk foglalkozni, amelyek segítségével a lokális zárak globális zárakká alakíthatók, és rendelkeznek a szükséges tulajdonságokkal:

- Semelyik két tranzakció sem birtokolhat globális kizárólagos zárat egy X logikai elemen ugyanabban az időben.
- Ha egy tranzakció birtokában van az X logikai elem globális kizárólagos zárja, akkor semelyik másik tranzakció nem birtokolhat globális osztott zárat X-en.
- Bármennyi tranzakció rendelkezhet X globális osztott zárjával, amíg nincs olyan tranzakció, amely birtokolná a globális kizárólagos zárat.

10.6.4. Az elsődleges példány zárolása

A központi zárolás módszere továbbfejleszthető úgy, hogy megosztjuk a zárállomás feladatát, de továbbra is ragaszkodunk ahhoz az elképzeléshez, hogy minden logikai elemhez tartozik egy olyan egyedi munkaállomás, amely az elem globális zárjáért felelős. Ezt az osztott zárolási módszert az *elsődleges példány* (primary copy) módszerének nevezzük. Ezzel a változtatással sikerül a központosított módszer egyszerűségét megőrizni, ugyanakkor elkerülhető annak a lehetősége, hogy a központi zárállomás túlterheltté váljon.

Az elsődleges példány zárolási módszerben minden X logikai elem másodpéldányai közül kijelölünk egyet „elsődleges példánynak". Ha egy tranzakció zárolni szeretné az X logikai elemet, akkor arra a munkaállomásra kell elküldenie az igényét, ahol X elsődleges példánya található. Ez a munkaállomás tartja nyilván X-et a helyi zártáblázatban, és az adott helyzettől függően vagy engedélyezi, vagy megtagadja a zár kiadását. Az előző módszerhez hasonlóan a globális (logikai) zárak helyes működése itt is az elsődleges példányhoz tartozó zárak megfelelő helyi nyilvántartásán múlik.

Ahogy ezt a központi zárállomás esetében láttuk, ennél a módszernél is a legtöbb zárigénylés három üzenettel jár, kivéve, ha a tranzakció az elsődleges példány munkaállomásán fut. Ha azonban az elsődleges példányokat okosan választjuk, várhatóan ez sokkal gyakrabban fog előfordulni, mint az ellenkezője.

10.20. példa: Az üzletláncos példánkban a vásárlásokra vonatkozó adatok elsődleges példányait úgy kellene kiválasztanunk, hogy azok annak az üzletnek az adatbázisában legyenek, ahol maga a vásárlás történt. Az adat többi példánya, amely például a központi részlegben vagy az elemzők által használt adattárházban van, nem elsődleges. A jellemző tranzakciók valószínűleg az áruházakban futnak le, és csak az adott áruházra vonatkozó vásárlási adatokat módosítják. Amikor egy ilyen típusú tranzakció zárolja

Osztott holtpontok

Miközben a tranzakció többszörözött adatot próbál globális zár alá helyezni, számos alkalma van arra, hogy holtpontra jusson. Annak is számos módja van, hogy globális várakozási gráfot készítsünk, és így észlelni tudjuk a holtpontok kialakulását. Osztott környezetben azonban gyakran az a legegyszerűbb és leghatékonyabb megoldás, ha az időkorlátos módszert alkalmazzuk. Bármely tranzakcióról, amely a megfelelő időmennyiség eltelte után még mindig nem zárult le, feltesszük, hogy holtpontra jutott, és ezért visszagörgetjük.

az elemeket, akkor nincs szükség üzenetek küldésére. A zárolással kapcsolatos üzenetekre csak akkor lenne szükség, ha a tranzakció egy másik áruház adatait vizsgálná vagy módosítaná. □

10.6.5. A lokális zártól a globálisig

Egy másik megoldás, ha több, összegyűjtött lokális zárból képzünk globális zárat. Ezekben a sémákban az X adatbáziselem egyik példánya sem „elsődleges" – az elem másolatai szimmetrikusak, és bármelyikükre igényelhető lokális osztott vagy kizárólagos zár. Egy jól működő globális záró séma nyitja, hogy a tranzakcióknak be kell gyűjteniük bizonyos számú lokális zárat X példányain ahhoz, hogy az X-en lévő globális zárat birtokolhassák.

Tegyük fel, hogy az A adatbáziselem n példányban létezik. Választunk két értéket:

1. s – A példányai közül s számút kell osztott módban zárolni ahhoz, hogy egy tranzakció globális osztott módban zárolhassa A-t.
2. x – A példányai közül x számút kell kizárólagos módban zárolni ahhoz, hogy egy tranzakció globális kizárólagos módban zárolhassa A-t.

Ha $2x > n$ és $s + x > n$, akkor megvannak a kellő tulajdonságok: csak egy globális kizárólagos zár létezhet A-n, és nem létezhet egyszerre globális osztott és globális kizárólagos zár rajta. Ezeket a következőképpen magyarázhatjuk: ha két tranzakció birtokában is lenne egy-egy globális kizárólagos zár A-n, akkor mivel $2x > n$, A-nak legalább egy másodpéldányán mindkét tranzakció lokális kizárólagos zárat birtokolna (mert több lokális kizárólagos zár van kiosztva, mint ahány másodpéldánya van A-nak). Ekkor azonban a lokális zárolási módszer nem működne helyesen. Hasonlóan, ha egy tranzakció globális osztott módban, egy másik pedig globális kizárólagos módban zárolja A-t, akkor, mivel $s + x > n$, valamelyik másodpéldányon az egyikük lokális osztott, a másikuk pedig lokális kizárólagos zárat birtokolna egyszerre.

Általában, a globális osztott, illetve kizárólagos zár megszerzéséhez szükséges üzenetek száma rendre $3s$, illetve $3x$. Ez a szám igen nagynak tűnik, összehasonlítva a

központi módszerekkel, ahol záranként átlagosan három vagy annál kevesebb üzenet szükséges. Vannak azonban ezt ellensúlyzó tulajdonságai a rendszernek speciális (s, x) választása esetén, ahogy ezt a következő két példában látjuk:

- *Egy-olvasás-zár; Minden-írás-zár.* Itt $s = 1$, $x = n$. A globális kizárólagos zár megszerzése nagyon drága, de globális osztott zár esetén legfeljebb három üzenet is elég. Ráadásul ennek a sémának van egy előnye az elsődleges példány módszerével szemben: míg az utóbbival elkerülhető az üzenetek küldése, ha az elsődleges példányt olvassuk, addig az egy-olvasás-zár sémával ugyanez megtehető, valahányszor a tranzakció egy olyan helyen fut, ahol megtalálható az olvasni kívánt adatbáziselem *akármelyik példánya*. Így ez a séma jobban megfelelhet, amikor a tranzakciók többsége csak olvas, de az X-et olvasó tranzakciók más-más helyeken futnak. Példának hozható fel az osztott digitális könyvtár, amely egy dokumentum másodpéldányait azokon a helyeken tárolja el, ahol azokat gyakran olvassák.
- *Többségi zárolás.* Itt $s = x = \lceil (n+1)/2 \rceil$. Úgy tűnik, ebben a rendszerben a tranzakció helyétől függetlenül mindenképpen sok üzenetre lesz szükség. Van azonban számos más tényező, amely figyelembe vételével már elfogadhatóvá válik a séma. Ezek közé tartozik, hogy sok hálózati rendszer támogatja a *csomagszórást*[7], amely segítségével lehetővé válik, hogy a tranzakció egyetlen általános igény elküldésével próbálja begyűjteni az X elem lokális zárjait, hiszen ez az üzenet minden munkaállomáshoz eljut. Hasonlóan, a zárak feloldásához is elég lehet egyetlen üzenet. A módszer javára írandó még az is, hogy s és x értékének fenti megválasztása az üzenetek nagy számának ellenére olyan előnnyel jár, amely más értékek esetén nem érhető el: lehetőség van a rendszer részleges működésére, még akkor is, ha a hálózat több részre szakadt. Amíg a hálózatnak van olyan része, amely az X másodpéldányait tároló munkaállomások többségét tartalmazza, addig lehetősége van a tranzakciónak arra, hogy zárolni próbálja X-et. Ha más, a hálózatról leszakadt munkaállomások aktívak is maradnak, még osztott zárat sem kaphatnak X-en, így nincs veszélye annak, hogy a hálózat különböző (egymástól elszakadt) részein futó tranzakciók nem sorba rendezhető módon fognak viselkedni.

10.6.6. Feladatok

! **10.6.1. feladat:** Megmutattuk, hogyan lehet lokális osztott, illetve kizárólagos zárból a globális változatot létrehozni. Hogyan hoznánk létre a megfelelő típusú lokális zárból a következőket:

* a) Globális osztott, kizárólagos és növelési zárakat.
b) Globális osztott, kizárólagos és módosítási zárakat.
!! c) Globális osztott, kizárólagos és mindenféle típusú szándékjelölő zárakat.

[7] Angolul *broadcast*. Olyan üzenettovábbítás, amely a hálózat minden elemének szól. *A fordító megjegyzése.*

10.6.2. feladat: Tegyük fel, hogy van öt munkaállomás, amelyek mindegyike rendelkezik az X adatbáziselem egy-egy másodpéldányával. Ezek közül P a legfontosabb X-re nézve, és az elsődleges példány osztott zárolási rendszerben ezt használjuk X elsődleges munkaállomásaként. Az X elérésére vonatkozó statisztikák a következők:

i) A hozzáférések 50%-a P-ből ered és ezek során X-et csak olvassák.
ii) A többi négy munkaállomás mindegyike egyenként 10%-ban igényel hozzáférést, és ezek csak olvasó-hozzáférések.
iii) A hozzáférések maradék 10%-a kizárólagos, és az öt munkaállomás közül akármelyik igényelheti egyenlő valószínűséggel (azaz mindegyik 2%-ban igényli).

Az alábbi zárolási módszerek mindegyikéhez adjuk meg az egyes zárak megszerzéséhez szükséges üzenetek átlagos számát! Feltehető, hogy minden kérelmet jóváhagynak, azaz hogy elutasító üzenetekre nem lesz szükség.

* a) Egy-olvasás-zár; minden-írás-zár.
b) Többségi zárolás.
c) Elsődleges példány zárolás, az elsődleges példány P-nél található.

10.7. Hosszú tranzakciók

Az adatbázis-alkalmazásoknak van egy olyan csoportja, ahol az adatok nyilvántartásához megfelel az adatbázisrendszer, de a sok, rövid tranzakciós modell, amelyre az adatbázis konkurenciavezérlő mechanizmusai alapoznak, alkalmatlan. Ebben a fejezetben ilyen alkalmazásokkal és a velük kapcsolatban felmerülő problémákkal foglalkozunk, majd bemutatjuk a „kiegyenlítő tranzakciókon" alapuló megoldást. Ezek a tranzakciók érvénytelenítik az olyan véglegesített tranzakciók hatását, amelyeknek nem kellett volna véglegessé válniuk.

10.7.1. A hosszú tranzakciók problémái

Durván fogalmazva a *hosszú tranzakció* egy olyan tranzakció, amely túl sokáig tart ahhoz, hogy megengedhető legyen számára az, hogy olyan elemeket tartson zár alatt, amelyekre más tranzakcióknak is szükségük van. A „túl hosszú" a környezettől függően jelenthet másodperceket, perceket vagy órákat; mi feltesszük, hogy egy „hosszú" tranzakció legalább percekig, talán órákig is eltart. A hosszú tranzakciókkal kapcsolatos alkalmazások három nagy osztálya a következő:

1. *Hagyományos ABKR-alkalmazások.* A közönséges adatbázis-alkalmazások főleg rövid tranzakciókat futtatnak, sok esetben azonban szükség van alkalmanként hosszú tranzakciókra is. Például egy tranzakció végigvizsgálhatja egy bank összes

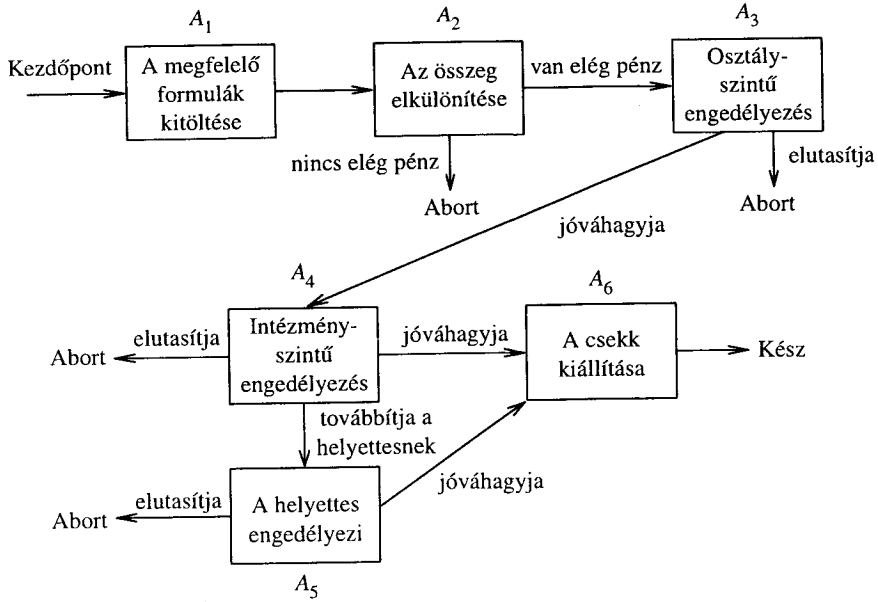
számláját, hogy megállapítsa, hogy a végösszeg helyes-e, egy másik alkalmazásban pedig alkalmanként szükség lehet egy-egy index újraszervezésére, hogy a teljesítmény továbbra is maximális legyen.

2. *Tervezőrendszerek.* Függetlenül attól, hogy a tervezni kívánt eszköz mechanikus (például gépkocsi) vagy elektronikus (például mikroprocesszor vagy programrendszer), a tervezőrendszerek egy dologban mindig megegyeznek: a terv alkotóelemeire van szétbontva (például fájlokra egy felhasználói program esetén), és a különböző részeken különböző tervezők dolgoznak egyszerre. Nem akarjuk viszont, hogy két tervező egyszerre ugyanazon a fájlon dolgozzon, hiszen ezzel csak felülírnák egymás munkáját, ezért egy *ki- és bejelentkező* rendszer segítségével lehetővé tesszük, hogy a tervező „kijelentsen" egy fájlt, majd a változtatások végeztével, talán órákkal vagy napokkal később, bejelentse. Egy tervező azonban valamilyen oknál fogva talán akkor is bele akar nézni egy fájlba, amikor azon éppen egy másik tervező dolgozik. Ha a fájl kijelentése egyenértékű lenne egy kizárólagos zárolással, akkor néhány indokolt és ésszerű műveletet talán csak napokkal később lehetne végrehajtani.
3. *Munkafolyamat-rendszerek.* Ezek a rendszerek olyan folyamatokat foglalnak magukban, amelyek közül néhányat egy szoftver hajt végre egyedül, néhánynak emberi beavatkozás is része, és néhány talán csak ember által végrehajtott műveletekből áll. Rövidesen láthatunk egy példát, amely egy számla kifizetésével járó hivatalos papírmunkát mutat be. Az ilyen alkalmazások végrehajtása napokig is eltarthat, és ez alatt az idő alatt lehet, hogy néhány adatbáziselemen változtatni kell. Ha a rendszer engedélyezné a tranzakcióhoz szükséges adatok kizárólagos zárolását, akkor a többi tranzakció napokig nem férhetne hozzájuk.

10.21. példa: Vegyük azt az esetet, amikor egy alkalmazott megpróbálja elszámoltatni az utazási költségeit. Azt szeretné, hogy a cége az A123-as számláról visszafizesse a kiadásait – a költségtérítés folyamata a 10.17. ábrán látható. Az eljárás az A_1 művelettel kezdődik, amikor az utazó titkárnője egy on-line kérdőív kitöltésével leírást ad az utazásról, megadja a terhelendő számla számát és a visszatérítendő összeget. Feltesszük, hogy ebben a példában a számlaszám A123 és az összeg 1000 dollár.

Az utazó számláit fizikailag átküldik az osztályos titkárságra, a kitöltött formula pedig a hálózaton keresztül eljut az A_2 automatikus eljáráshoz. A_2 ellenőrzi, hogy a megadott számlán (A123) van-e elég pénz a költségek (1000 dollár) fedezésére, és ha van, akkor a számlán elkülönít egy ennek megfelelő összeget; vagyis próbaképpen levon a számlán található pénzmennyiségből 1000 dollárt, de erről még nem állít ki csekket. Ha nincs elég pénz a számlán, akkor a tranzakció abortál, és feltehetően majd újraindul, vagy amikor már rendelkezésre áll a kívánt pénzösszeg, vagy ha egy másik számláról kérik a kifizetést.[8]

[8] Természetesen az utazó (aki úgysem a stanfordi egyetemen dolgozik) *soha* sem írná a költségeit – helytelenül – egy másik kormányszámla terhére, hanem megfelelő pénzügyi forrásokat keresne. Ezt azért kell hangsúlyoznunk, mert a kormány számvevői, akiknek fogalmuk sincs arról, hogy egy egyetemnek hogyan kéne működnie, még mindig itt nyüzsögnek Stanford körül.



10.17. ábra. *Munkafolyamat-ábra az utazási költségtérítésekhez*

Az A_3 műveletet, amelyre lehet, hogy csak napokkal később kerül sor, az osztály ügyintézője hajtja végre: megvizsgálja a benyújtott számlákat és az on-line formulát. Ha mindent rendben talál, a kérvényt jóváhagyja, és a számlákkal együtt továbbküldi egy magasabb adminisztrációs szintre, az intézményes ügyintézőhöz. Ellenkező esetben a tranzakció abortál, az utazónak pedig feltehetően módosított formában újra be kell adnia a jelentkezését.

Az A_4 művelet során, amelyre lehet, hogy csak napokkal később kerül sor, az intézményes ügyintéző vagy jóváhagyja, vagy elutasítja a kérelmet, vagy továbbadja a helyettesének, aki az A_5 műveletben dönt. Ha a kérvényt elutasítják, akkor a tranzakció abortál, és a jelentkezést újra be kell adni. Jóváhagyás esetén az A_6 művelet során kiállítják a csekket, és véglegesítik az 1000 dollár levonását a számláról.

Tegyük fel, hogy ez a munkafolyamat csak a hagyományos zárak használatával valósítható meg. Ekkor például, mivel a sikeresen lezárult tranzakció megváltoztathatja az A123-as számla egyenlegét, az A_2 műveletnél kizárólagos zár alá kell helyezni a számlát, és azt addig nem szabad feloldani, amíg a tranzakció nem abortál, vagy az A_6 művelet le nem zárul. A zárat talán napokig fenn kell tartani, amíg a kérvény elbírálásával megbízott emberek végre foglalkozni tudnak az üggyel. Ilyenkor viszont az A123-as számláról semmilyen más kifizetés sem történhet, még próbaképpen sem. Másrészről viszont, ha a számlához való hozzáférést egyáltalán nem szabályozzuk, akkor különböző tranzakciók egyszerre terhelhetik a számlát, illetve foglalhatnak le a számlán különféle összegeket, ez pedig a hitelkeret túllépéséhez vezethet. A két szélsőséges módszer helyett tehát egy kompromisszumos megoldást kell találni. □

10.7.2. Regék

A *rege* (saga) olyan műveletek összessége, amelyek együtt egy hosszú „tranzakciót" alkotnak (ilyen műveleteket láttunk a 10.21. példában). A rege tehát a következő részekből áll:

1. Valamilyen műveletek összessége.
2. Egy gráf, amelyben a csúcspontok vagy műveletek, vagy a speciális *Abort*, illetve *Kész* csúcsok, az élek pedig a csúcsok között futnak. A kétféle speciális csúcsból, amelyeket *végpontoknak* (terminal node) nevezünk, nem indulnak ki élek.
3. Azon csúcspont megjelölése, ahonnan az egész folyamat indul. Ez a csúcs a *kezdőpont* (start node).

A kezdőpontból akármelyik végpontba vezető út a műveletek egy lehetséges sorozatát adja. Az *Abort* csúcspontba vezető utak olyan műveletsorozatot jelentenek, amely után az egész tranzakciót vissza kell görgetni, a műveleteknek pedig változatlanul kell hagyniuk az adatbázist. A *Kész* csúcsba vezető utak sikeres műveletsorozatokat jelentenek, és az ezen műveletek által az adatbázisrendszeren véghezvitt változtatásoknak meg kell maradniuk az adatbázisban.

10.22. példa: A 10.17. ábrán látható gráfban az *Abort* csúcsba vezető utak a következők: A_1A_2, $A_1A_2A_3$, $A_1A_2A_3A_4$ és $A_1A_2A_3A_4A_5$. A *Kész* csúcsba vezetnek $A_1A_2A_3A_4A_6$ és $A_1A_2A_3A_4A_5A_6$. Vegyük észre, hogy most a gráf nem tartalmaz kört, vagyis a végpontokba vezető utak száma véges. Általános esetben azonban létrejöhetnek körök a gráfban, és ilyenkor az utak száma végtelen is lehet.[9] □

A regék konkurenciavezérlésére kétféle lehetőség kínálkozik:

1. Minden egyes műveletet tekinthetünk egy-egy önálló (rövid) tranzakciónak, amely a végrehajtáskor egy olyan hagyományos konkurenciavezérlő mechanizmust használ, mint a zárolás. Az A_2 művelet például megvalósítható úgy is, hogy arra a (rövid) időre, amíg az A123-as számla egyenlegét csökkenti az utazási elismervényen jelölt összeggel, zárolja a számlát, majd a zárat feloldja. A zárolás segítségével megelőzhető, hogy egyszerre két tranzakció próbáljon a számla egyenlegének új értéket adni, amivel az első változtatás hatása elveszne, és „csodával határos módon pénz jelenne meg a számlán".
2. Az egész tranzakciót, amelyet a végpontokba vezető utak bármelyike jelenthet, a „kiegyenlítő tranzakciók" mechanizmusán keresztül kezeljük, amelyek a rege csúcspontjaiban található tranzakciók inverzei. Feladatuk az, hogy a véglegessé vált műveletek hatását olyan módon görgessék vissza, amely nem függ attól, hogy mi történt az adatbázissal a véglegesített művelet és a kiegyenlítő tranzakció végrehajtása közötti időben. A kiegyenlítő tranzakciókról a következő részben lesz szó.

[9] A bürokrácia útvesztői... *A fordító megjegyzése.*

Mikor „ugyanaz" két adatbázis-állapot?

A kiegyenlítő tranzakciók tárgyalása közben óvatosnak kell lennünk azzal kapcsolatban, hogy mit jelent az adatbázist „ugyanabba" az állapotba visszaállítani, mint amelyben előtte volt. Már kaptunk egy kis ízelítőt ebből a problémából, amikor a 10.6. példában a B-fák logikai naplózásáról volt szó. Láthattuk, hogy amikor „visszaállítottunk" egy műveletet, a B-fa állapota nem feltétlenül volt azonos a művelet előtti állapottal, viszont ekvivalens volt vele a B-fa elérési műveleteit illetően. Általánosabban fogalmazva, megtörténhet, hogy egy műveletet és a hozzá tartozó kiegyenlítő tranzakciót végrehajtva, az adatbázis nem áll vissza bitről bitre ugyanabba az állapotba, amelyben előtte volt, de a különbségeket nem érzékelheti egyetlen, az adatbázis által támogatott felhasználói program sem.

10.7.3. Kiegyenlítő tranzakciók

Egy regében minden A művelethez tartozik egy *kiegyenlítő tranzakció* (compensating transaction), amit A^{-1}-gyel jelölünk. Ha végrehajtjuk A-t, majd később A^{-1}-et, akkor eredményül ugyanazt az adatbázis-állapotot kapjuk, mintha sem A-t, sem A^{-1}-et nem hajtottuk volna végre. Formálisabban fogalmazva:

- Ha D egy tetszőleges adatbázis-állapot, $B_1B_2...B_n$ pedig műveletek és kiegyenlítő tranzakciók sorozata (akár a kérdéses regéből, akár egy másikból vagy akármilyen más tranzakcióból, amely szabályosan fut az adatbázison), akkor ugyanazt az adatbázis-állapotot kapjuk D-ből a $B_1B_2...B_n$ sorozat futtatása után, mint $AB_1B_2...B_nA^{-1}$ futtatása után.

Ha a rege végrehajtása az *Abort* csúcspontba vezet, akkor a visszagörgetés úgy történik, hogy a végrehajtott műveletek fordított sorrendjében lefuttatjuk az egyes műveletekhez tartozó kiegyenlítő tranzakciókat. A kiegyenlítő tranzakciók fent említett tulajdonsága miatt a rege hatása érvénytelenné válik, és az adatbázis-állapota ugyanaz lesz, mintha a regét soha nem is hajtottuk volna végre. Hogy miért biztos az, hogy a visszagörgetett rege hatása érvénytelenné válik, arra részletesebb magyarázatot a 10.7.4. részben találunk.

10.23. példa: Vegyük a 10.17. ábra műveleteit, és nézzük meg, hogy mik lehetnek a megfelelő kiegyenlítő tranzakciók. Elsőként A_1 létrehoz egy on-line dokumentumot. Ha ezt az adatbázisban tároljuk, akkor A_1^{-1}-nek ezt el kell onnan távolítania. Vegyük észre, hogy ez a kiegyenlítés engedelmeskedik a kiegyenlítő tranzakciók alaptulajdonságának: ha létrehozzuk a dokumentumot, és összeállítunk egy α műveletsorozatot (amely akár a dokumentum törlését is tartalmazhatja, ha akarjuk), akkor $A_1\alpha A_1^{-1}$ hatása megegyezik α hatásával.

A_2 implementálásával óvatosan kell bánnunk. A pénzt úgy „foglaljuk le", hogy a megfelelő mennyiséget levonjuk a számlaegyenlegből. Ez a pénz egészen addig törölve marad, amíg az A_2^{-1} kiegyenlítő tranzakció vissza nem helyezi a számlára. Azt állítjuk, hogy A_2^{-1} kiegyenlítő tranzakció helyesen fog működni, ha a számlavezetés általános szabályait követjük. Hogy jobban ráérezzünk a probléma lényegére, hasznos, ha megnézünk egy hasonló tranzakciót, ahol a logikusnak tűnő kiegyenlítés nem működik. A 10.24. példában látunk majd egy ilyen esetet.

Az A_3, A_4 és A_5 műveletek mindegyikében egy jóváhagyással egészül ki az on-line nyomtatvány. A kiegyenlítő tranzakciók tehát egyszerűen eltávolíthatják ezeket a jóváhagyásokat.[10]

Végül A_6-hoz, a csekket kiállító művelethez nem tudunk egyértelműen kiegyenlítő tranzakciót megadni. A gyakorlatban erre nincs is szükség, hiszen ha A_6-ot egyszer már végrehajtottuk, akkor a regét már nem görgethetjük vissza. A_6-nak viszont a szó szoros értelmében úgy sincs hatása az adatbázisra, mivel a csekket fedező összeget a számláról már levonta A_2. Ha az „adatbázist" esetleg tágabb határok között kéne értelmeznünk, ahol a csekk beváltásához hasonló eseményeknek hatásuk van az adatbázisra, akkor A_6^{-1}-et úgy kéne megterveznünk, hogy először próbálja meg a csekket visszavonni, ha ez nem megy, akkor levélben követelje vissza a pénzt attól, aki a csekket beváltotta, végül, ha ez a kísérlet is kudarcba fulladt, nyilatkozzon a behajthatatlan követelés miatti veszteségről, és állítsa vissza az egyenleg eredeti értékét. □

Most fogjunk hozzá a 10.23. példában említett eset felvázolásához, ahol is a számlán végrehajtott változtatást nem lehet kiegyenlíteni a változtatás inverzével. A problémát az okozza, hogy a számlák egyenlege általában nem lehet negatív.

10.24. példa: Tegyük fel, hogy B egy olyan tranzakció, amely 1000 dollárt rak egy olyan számlára, amelyen eredetileg 2000 dollár van, B^{-1} pedig a kiegyenlítő tranzakció, amely ugyanezt az összeget leszedi a számláról. Ésszerű továbbá azt is feltennünk, hogy az a tranzakció, amely egy számláról a rendelkezésre álló összegnél nagyobbat akar eltávolítani, nem mindig futhat le sikeresen. Legyen C egy olyan tranzakció, amely 2500 dollárt emel le ugyanarról a számláról. Ekkor BCB^{-1} nem ekvivalens C-vel. Ennek az az oka, hogy C magában nem fut le sikeresen, így a számlán ott marad 2000 dollár, viszont ha először B-t, majd C-t is végrehajtjuk, a számlán 500 dollár marad, ami után B^{-1} nem zárulhat le sikeresen.

Azt a következtetést kell levonnunk, hogy egy számlák közti tetszőleges átutalásokból álló regét és azt a szokást, hogy a számlák nem mehetnek negatívba, nem lehet egyszerűen kiegyenlítő tranzakciókkal támogatni. Valamilyen változtatást kell bevezetnünk a rendszerbe, például megengedhetjük a negatív egyenleget is. □

[10] A 10.17. ábrán bemutatott regében ezeket a műveleteket csak akkor kell kiegyenlíteni, amikor a nyomtatványt különben is megsemmisítenék. A kiegyenlítő tranzakciók definíciója azonban megköveteli, hogy a tranzakciók egymástól elkülönítve is működjenek, arra való tekintet nélkül, hogy valamelyik másik tranzakció esetleg lényegtelenné teszi az általuk eszközölt változtatásokat.

10.7.4. Miért működnek jól a kiegyenlítő tranzakciók?

Azt mondjuk, hogy két műveletsorozat *ekvivalens* (≡), ha bármely D adatbázis-állapotot azonos állapotba visznek. A kiegyenlítő tranzakciókról tett alapvető feltevésünk ekkor a következő alakban írható fel:

- Ha A egy művelet, α pedig legális műveletek és kiegyenlítő tranzakciók sorozata, akkor $A\alpha A^{-1} \equiv \alpha$.

Most azt kell megmutatnunk, hogy ha egy $A_1A_2\ldots A_n$ rege végrehajtása után a fordított sorrendben vett kiegyenlítő tranzakciók is lefutnak, akkor akármilyen műveletek is estek a rege műveletei illetve a tranzakciók közé, ennek olyan a hatása, mintha sem a rege műveletei, sem a kiegyenlítő tranzakcióik nem futottak volna le egyáltalán. Ezt az állítást n-re vonatkozó indukcióval bizonyítjuk.

Alap: Ha $n = 1$, akkor az A_1 és az ezt kiegyenlítő A_1^{-1} tranzakció közti műveletsorozat: $A_1\alpha A_1^{-1}$. A kiegyenlítő tranzakciókról való alapfeltevés szerint $A_1\alpha A_1^{-1} \equiv \alpha$; azaz a regének nincs hatása az adatbázis-állapotra.

Indukció: Tegyük fel, hogy az állítás teljesül minden legfeljebb $n-1$ műveletből álló útra. Vegyük most egy n hosszú út műveleteit, amelyeket a fordított sorrendben vett kiegyenlítő tranzakciók követnek, és a műveletek és a tranzakciók közé másféle műveletek is eshetnek. A sorozatot ilyen alakban írhatjuk fel:

$$A_1\alpha_1A_2\alpha_2\ldots\alpha_{n-1}A_n\beta A_n^{-1}\gamma_{n-1}\ldots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (10.1.)$$

ahol minden görög betű nulla vagy több művelet sorozatát jelöli. A kiegyenlítő tranzakciók definíciója miatt $A_n\beta A_n^{-1} \equiv \beta$. Vagyis (10.1.) ekvivalens a következővel:

$$A_1\alpha_1A_2\alpha_2\ldots A_{n-1}\alpha_{n-1}\beta\gamma_{n-1}A_{n-1}^{-1}\gamma_{n-1}\ldots\gamma_2A_2^{-1}\gamma_1A_1^{-1} \qquad (10.2.)$$

Az indukciós feltevés miatt pedig ez ekvivalens

$$\alpha_1\alpha_2\ldots\alpha_{n-1}\beta\gamma_{n-1}\ldots\gamma_2\gamma_1$$

kifejezéssel, hiszen (10.2.)-ben csak $n-1$ művelet szerepel. Tehát ha a rege után a kiegyenlítését is lefuttatjuk, ez olyan állapotban hagyja az adatbázist, mintha a regét nem is hajtottuk volna végre.

10.7.5. Feladatok

*! **10.7.1. feladat:** Szoftverek „eltávolítását" (uninstalling), mint folyamatot felfoghatjuk az adott szoftver telepítésének (installing) a kiegyenlítő tranzakciójaként is. A telepítés és eltávolítás egy egyszerű modelljében feltehető, hogy egy művelet egy vagy több fájl *feltöltéséből* (loading) áll a forrásról (például CD-ROM) a számítógép merevle-

mezére. Egy *f* fájl feltöltésekor átmásoljuk *f*-et a CD-ROM-ról, felülírva az *f*-fel azonos nevű és útnevű fájlt, ha volt ilyen. Hogy meg tudjuk különböztetni az ilyen értelemben azonos fájlokat, feltehetjük, hogy minden fájl rendelkezik egy időbélyegzővel.

a) Mi a kiegyenlítő tranzakciója *f* fájl feltöltésének? Gondoljuk meg mindkét esetet: amikor létezett ugyanazon a helyen egy ugyanolyan nevű *f'* fájl, illetve amikor nem.
b) Magyarázzuk meg, hogy az a) feladat megoldásaként adott tranzakció miért lesz valóban kiegyenlítő tranzakció! *Segítség:* alaposan gondoljuk át azt az esetet, amikor *f*-fel felülírtuk *f'*-t, majd egy későbbi művelet egy másik, azonos útnevű fájllal felülírta *f*-et.

! **10.7.2. feladat:** Adjunk meg a repülőjegy-foglalás folyamatához egy regét! Vegyük számításba azt a lehetőséget, hogy az ügyfél csak érdeklődik egy jegy iránt, de nem foglalja le. A jegy lefoglalása után az ügyfél lemondhatja a foglalást, de az is lehetséges, hogy nem fizeti ki a jegyet időre, vagy kifizeti, de végül mégsem repül. Minden művelethez adjuk meg a megfelelő kiegyenlítő tranzakciót is!

10.8. Összefoglalás

- *Piszkos adat:* A központi memória puffereiben vagy a lemezen található adatot, amelyet egy, még folyamatban lévő tranzakció írt, „piszkos" adatnak nevezünk.
- *Továbbgyűrűző visszagörgetés:* Olyan naplózás és konkurenciavezérlés együttese esetén, amely megengedi, hogy egy tranzakció piszkos adatot olvasson, szükség lehet az olyan tranzakciók visszagörgetésére, amelyek egy később abortált tranzakció által írt (piszkos) adatokat olvastak.
- *Szigorú zárolás:* A szigorú zárolás elve azt követeli meg a tranzakcióktól, hogy a zárakat (az osztott zárak kivételével) ne csak a tranzakció lezárulásáig tartsák fenn, hanem még azt követően egészen addig, amíg a commit vagy abort naplóbejegyzés ki nem kerül a lemezre. A szigorú zárolás biztosítja, hogy egyetlen tranzakció sem olvas piszkos adatot, még visszamenőlegesen sem egy rendszerhiba és annak helyreállítása során sem.
- *Csoportos véglegesítés:* Gyengíthetünk a szigorú zárolásnak a naplóbejegyzésre vonatkozó feltételén, ha biztosítjuk, hogy a bejegyzések abban a sorrendben kerülnek ki a lemezre, amelyben eredetileg a pufferbe írtuk őket. Ekkor továbbra is garantált, hogy a tranzakciók nem olvasnak piszkos adatot, még egy esetleges hiba és helyreállítása során sem.
- *Az adatbázis állapotának helyreállítása abort után:* Ha egy tranzakció abortál; miután a pufferbe írt, az adatelemek régi értékét a napló vagy az adatbázis lemezen található példánya alapján állíthatjuk vissza. Ha az új értékek már elérték a lemezt, a napló még akkor is használható a régi értékek visszaállítására.
- *Logikai naplózás:* Olyan nagy adatbáziselemek esetén, mint a lemezblokkok, sok helyet takaríthatunk meg, ha a régi és az új értékeket a naplóba növekményesen jegyezzük be, azaz csak a változtatásokat tüntetjük fel. Néhány esetben a változtatások logikai feljegyzése – azaz a blokkok tartalmának absztrakt leírása – lehetővé teszi, hogy egy tranzakció abortálása után az adatbázis állapotát logikailag visszaállítsuk, még akkor is, ha szigorúan ugyanabba az állapotba való visszaállítás lehetetlen.
- *Nézet-sorbarendezhetőség:* Olyan ütemezésekben, ahol a tranzakciók talán olyan értékeket is írnak, amelyeket a későbbiekben egyetlen más tranzakció sem olvas, csak egy következő felülír, a konfliktus-sorbarendezhetőség túl erős feltételnek bizonyul. Egy gyengébb feltétel, amit nézet-sorbarendezhetőségnek nevezünk, csak azt követeli meg, hogy az ekvivalens soros ütemezésben minden tranzakció ugyanabból a forrásból olvassa az adatelemek értékeit, mint az eredeti ütemezésben.
- *Poligráf:* A nézet-sorbarendezhetőség ellenőrzése magában foglalja egy poligráf felépítését, amelyben az élek az értékek az írótól az olvasó irányába haladását jelölik, az élpárok pedig azt a követelményt tükrözik, amely szerint bizonyos írásműveletek nem eshetnek az adott írás- és olvasásművelet közé. Az ütemezés pontosan akkor nézet-sorbarendezhető, ha minden élpárnak elhagyhatjuk az egyik felét úgy, hogy eredményül egy körmentes gráfot kapjunk.
- *Holtpont:* Ez bármikor kialakulhat, amikor a tranzakcióknak olyan erőforrásokra, például zárakra, kell várakozniuk, amelyek az adott pillanatban egy másik tranzakció birtokában vannak. Megfelelő előkészületek nélkül fennáll a veszélye egy várakozási kör kialakulásának, amikor is az ebben részt vevő tranzakciók egyike sem képes a továbblépésre.
- *Várakozási gráf:* Rajzoljunk egy csúcsot minden várakozó tranzakcióhoz, és kössük össze azokkal a tranzakciókkal, amelyekre várakozik. A holtpont kialakulása pontosan azt jelenti, hogy létrejött egy vagy több kör a várakozási gráfban. A holtpontok kialakulása elkerülhető, ha a várakozási gráf nyilvántartásával minden olyan tranzakciót abortáltatunk, amely várakozásával kör jönne létre a gráfban.
- *Holtpontmegelőzés az erőforrások sorbarendezésével:* Ha a tranzakcióktól megkívánjuk, hogy az erőforrásokat valamilyen lexikografikus sorrendben használják fel, akkor ezzel megelőzhető a holtpontok kialakulása.
- *Időbélyegző alapú holtpontmegelőzés:* Más sémák időbélyegzőket vezetnek be, és ezek alapján döntik el, hogy az erőforrást igénylő tranzakciót abortáltassák-e vagy várakoztassák. A megvár-meghal sémában az erőforrást birtokló tranzakciónál idősebb tranzakció várakozik, az újabbakat pedig visszagörgetjük, de az időbélyegzőjük nem változik. A megsebez-megvár sémában az újabb tranzakció várakozik, az idősebb viszont kényszeríti az erőforrást birtokló tranzakciót, hogy mondjon le az erőforrásairól, és később kezdje el újra a működését.
- *Osztott adatok:* Egy osztott adatbázisban az adatok részekre bonthatók vízszintesen (a reláció sorai különböző munkaállomásokra vannak szétszórva), vagy függőlegesen (a reláció sémája több sémára van bontva [dekompozíció], és az ezekhez tartozó relációk különböző munkaállomásokon találhatók). Lehetőség van az adatok többszörözésére is, vagyis arra, hogy a relációnak egymással feltehetőleg azonos másodpéldányai legyenek különböző helyeken.
- *Osztott tranzakciók:* Az osztott adatbázisban egy logikai tranzakció több alkotórészből állhat, amelyek mindegyike különböző munkaállomáson fut. A konzisztén-

11. fejezet

Információk egyesítése

Bár a modern adatbázisrendszerek sokféle irányba fejlődnek, az *információegyesítés* általános iránya az új alkalmazások népes családját tudhatja magáénak. Az ilyen alkalmazások két vagy több adatbázisban *(információforrásban)* tárolt adatokból hoznak létre egy nagy (virtuális) adatbázist, amelyben a forrásokban fellelhető összes információ megtalálható. Így a több helyen „szétszórtan" tárolt adatokat egységesen lehet lekérdezni. Az adatforrások lehetnek hagyományos vagy más típusú adatbázisok, például világhálós weboldalak gyűjteményei.

Ebben a fejezetben az információegyesítés fontos szempontjait vezetjük be. Először vázlatosan áttekintjük az információegyesítés legfontosabb módszereit: a szövetséget, a tárház létrehozását és a közvetítést. Ezután néhány alkalmazásban megtalálható integrált adatszervezési módszeren keresztül megismerkedünk egy különleges adatbáziselemmel, az úgynevezett „adatkockával". Az információegyesítéshez kapcsolódó speciális alkalmazásokat is áttekintjük. Így kerül sorra az „OLAP" (on-line analitikus feldolgozás)[1], és az „adatbányászat" témaköre.

11.1. Az információegyesítés módjai

Sokféle módszer létezik, amelynek segítségével adatbázisok vagy más (osztott) információforrások együttműködhetnek. Ebben a részben a három leggyakrabban alkalmazott eljárást mutatjuk be:

1. *Adatbázis-szövetség.* Az adatforrások egymástól függetlenek, de mindegyik kérhet a másiktól információt.
2. *Tárház létrehozása.* A különböző adatforrásokban megtalálható adatok másolatait egyetlen adatbázisban, az *adattárházban* tároljuk. Lehetséges, hogy az adatok még a tárházbavétel előtt valamilyen feldolgozó eljáráson mennek keresztül, például szűrhetjük vagy összesíthetjük (aggregáljuk) őket, vagy összekapcsolhatunk néhány relációt. Az adattárházat valamilyen időközönként, talán éjszakánként, frissítjük. Mivel az adatokat az adatforrásokból másoljuk, szükség lehet bizonyos átalakításokra, hogy az összes adat megfeleljen az adattárház sémájának.
3. *Közvetítés.* A közvetítő egy szoftverkomponens, amely egy olyan *virtuális adatbázist* támogat, amelyet a felhasználó úgy kérdezhet le, mintha az *valódi* lenne (azaz mintha fizikailag létezne, mint egy adattárház). Maga a közvetítő nem tárol semmiféle adatot – más módszert alkalmaz. A felhasználó lekérdezését lefordítja az adatforrások számára, egy vagy több lekérdezés képében, majd az adatforrások válaszait egységbe fogva adja meg a választ a felhasználónak.

[1] Az OLAP mozaikszó az angol *On-Line Analytic Processing* kifejezésnek felel meg. *A fordító megjegyzése.*

Mindegyik módszerrel részletesebben is megismerkedünk a fejezet során. Valamennyi megközelítésben kulcskérdés, hogy hogyan alakítjuk át az információforrásból nyert adatot. Az efféle információalakítók, a *borítékolók* (wrappers) vagy *adatkinyerők* (extractors) felépítését tárgyaljuk a 11.2. részben.[2] A következőkben bemutatunk néhány olyan problémát, amelyeket a borítékolóknak kell megoldaniuk.

11.1.1. Az egyesítés problémái

Akármilyen információegyesítő felépítést is választunk, kényes problémákba ütközünk, amikor a különböző adatforrások nyers adatainak próbálunk jelentést tulajdonítani. Az olyan adatforrások összességét, amelyek ugyanolyan, mégis sok apró részletben eltérő adatokkal dolgoznak, *heterogén* adatforrásnak nevezzük. Egy hosszabb példa segítségével bemutatjuk, miről van szó.

11.1. példa: A Süni Gépkocsigyártó Rt.-nek 1000 forgalmazója van, mindegyik adatbázist tart fenn a maga készletének a nyilvántartására. A cég létre akar hozni egy olyan egyesített adatbázist, amely az összes forgalmazójánál (azaz az 1000 adatforrásban) tárolt információt tartalmazza.[3]

Ha az egyik kereskedőnek éppen nincs raktáron valamilyen modellje, az egyesített adatbázis segítségével könnyen meg tudja állapítani, hogy melyik másiknál találhatja meg a hiányzó modellt. A cég elemzői pedig az előrelátható keresletre tehetnek megfigyeléseket, így a termelést a várhatóan népszerű modellekre lehet összpontosítani. Az 1000 forgalmazó azonban nem ugyanazt az adatbázissémát használja. Például az egyikük tárolhatja az autók adatait egyetlen relációban:

[2] Általában borítékolóról beszélünk közvetítő esetén, és adatkinyerőről tárház esetén, de nem mindig következetes a szóhasználat. *A fordító megjegyzése.*

[3] A legtöbb valódi gépkocsigyártó cég ma hasonló eszközökkel rendelkezik, bár ezeknek a fejlesztése eltérhet a példában leírtaktól. Lehet például, hogy a központi adatbázis jön létre először, majd később a forgalmazók, akik ebből letölthetik a saját adatbázisukba a megfelelő részeket. Az itt említett eset mégis arra szolgál például, amivel manapság sokféle szférában próbálkoznak.

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...)
```

ahol minden lehetséges extra felszereléshez (automata sebességváltó, CD-lejátszó...) egy logikai érték van rendelve. Egy másik forgalmazó viszont nyilvántarthatja az extra felszereléseket egy külön relációban, például így:

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```

Vegyük észre, hogy nem csak a két séma eltérő, hanem a nyilvánvalóan „azonos" nevek is megváltoztak: `Kocsik`ból `Autók` lett, `sorszám`ból `sorsz`.

Rosszabb esetben az azonos jelentésű adatok a különböző adatbázisokban másképpen vannak ábrázolva.

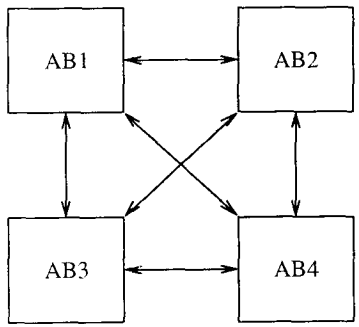
1. *Adattípus-különbségek.* A sorszámok lehetnek változó hosszúságú karakterláncok az egyik helyen vagy rögzített hosszúak a másikon. Maga a rögzített hossz is lehet más és más az egyes adatbázisokban, sőt néhány adatforrás használhat egész számokat a karakterlánc helyett.
2. *Értékkülönbségek.* Ugyanazt a fogalmat ábrázolhatják különböző konstansokkal a különböző adatforrásokban. A fekete színt például ábrázolhatják egy egész számmal, mint kóddal az egyik helyen, a `FEKETE` karakterlánccal egy másikon, és a `FE` kóddal egy harmadikon. Még rosszabb a helyzet, ha a `FE` a „fehéret" jelenti egy negyedik helyen.
3. *Szemantikus különbségek.* Sok fontos kifejezést másképpen értelmezhetnek az egyes adatforrások. Az egyik forgalmazó nyilvántarthat tehergépkocsikat is a `Kocsik` relációban, míg a másik csak személygépkocsikat tárol ugyanebben a relációban. Az egyik forgalmazó talán különbséget tesz furgon és kisteherautó között, a másik pedig nem.
4. *Hiányzó értékek.* Lehet, hogy az egyik adatforrás egyáltalán nem tárol egy olyan típusú információt, amit az összes többi (vagy a legtöbb) forrás nyilvántart. Például az egyik forgalmazó a színeket egyáltalán nem tartja nyilván. Hogy ezeket a helyzeteket kezelni tudjuk, néha használhatunk `NULL` értéket vagy valamilyen alapértelmezett értéket. A fejlődés iránya azonban a „felstrukturált" adatmodell használata felé mutat, amellyel olyan egyesített adatokat ábrázolhatunk, amelyek nem egészen egyeztethetők össze. Ezzel a témával kapcsolatban található néhány hasznos olvasmány a fejezet irodalomjegyzékében.

Az adatforrások közti minden efféle összeférhetetlenséget meg kell szüntetni valamilyen átalakítás, „fordítás" segítségével, még mielőtt az egyesített adatbázis létrejönne. □

11.1.2. Adatbázis-szövetség

Több adatbázis összefogásának talán az a legegyszerűbb módja, ha 1-1 kapcsolatot építünk azon adatbázispárok között, amelyek kommunikálni akarnak egymással. Ezeken a kapcsolatokon keresztül D_1 adatbázisrendszer lekérdezheti D_2-t, D_2 által érthető formában. A probléma ezzel a felépítéssel abból adódik, hogy ha az n adatbázis mindegyike szeretne kommunikálni a másik $(n-1)$-gyel, akkor $n(n-1)$ „rendszerközi" lekérdezéseket támogató kódot is kell írnunk. A 11.1. ábrán látható szövetségben a négy adatbázis mindegyikének három fordító komponensre van szüksége, hogy a másik hármat el tudja érni.

Mindezek ellenére bizonyos körülmények között lehet, hogy az adatbázis-szövetséget a legkönnyebb létrehozni, különösen, ha az adatbázisok egymás közti kommunikációja természeténél fogva korlátozott. Egy példán keresztül bemutatjuk, hogyan működhet a fordító komponens.



11.1. ábra. *A négy adatbázisból álló szövetségnek összesen 12 fordító komponensre van szüksége*

11.2. példa: Tegyük fel, hogy a Süni típusú gépkocsi forgalmazói meg akarják osztani egymás közt a leltárukat, de minden forgalmazónak csak arra van szüksége, hogy egy megfelelő lekérdezés segítségével megállapíthassa, hogy a nála megrendelt kocsik közül melyek találhatók meg valamelyik tőle nem messze működő forgalmazónál. Kicsit formálisabban, tekintsük Forgalmazó 1-et, aki a következő relációval rendelkezik:

```
IgényeltKocsik(modell, szín, autoSeb)
```

A reláció minden sora egy vásárló által igényelt kocsi leírása: milyen modell, milyen színű, és akarnak-e bele automata sebességváltót vagy nem. Forgalmazó 2 a leltárához a 11.1. példában bevezetett kétrelációs sémát használja:

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```

Forgalmazó 1 ír egy programot, amely egy távoli lekérdezés segítségével olyan kocsikat keres Forgalmazó 2 adatbázisában, amelyek leírása megfelel valamelyik olyan

```
for(minden IgényeltKocsik-hoz tartozó (:m, :c, :a) sorra) {
   if(:a= TRUE) {/*automata sebességváltót akarnak*/
      SELECT sorsz
      FROM Autók, Extrák
      WHERE Autók.sorsz = Extrák.sorsz AND
            Extrák.extra = 'autoSeb' AND
            Autók.modell = :m AND
            Autók.szín = :c;
   }
   else { / *nem akarnak automata sebességváltót */
      SELECT sorsz
      FROM Autók
      WHERE Autók.modell = :m AND
         Autók.szín = :c AND
         NOT EXISTS (
            SELECT *
            FROM Extrák
            WHERE sorsz = Autók.sorsz AND
                  extra = 'autoSeb'
         );
   }
}
```

11.2. ábra. *Forgalmazó 1 lekérdezi Forgalmazó 2 adatbázisát*

autónak, amely Forgalmazó 1 `IgényeltKocsik` relációjában szerepel. A program vázlata a 11.2. ábrán látható.

A beágyazott SQL rész jelenti Forgalmazó 2 adatbázisának távoli lekérdezését. Ennek az eredményét kapja vissza Forgalmazó 1. A szabvány SQL-ben használatos konvenciónak megfelelően kettőspontot tettünk az olyan változók elé, amelyek az adatbázisból visszakapott konstans értékeket jelölik.

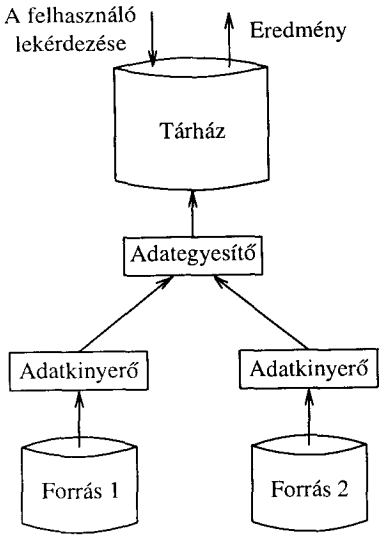
Ez a lekérdezés Forgalmazó 2 sémáját használja. Ha Forgalmazó 1 ugyanezt a lekérdezést Forgalmazó 3 adatbázisán is végre akarná hajtani, aki a 11.1. példában bemutatott

```
Kocsik(sorszám, modell, szín, autoSeb,...)
```

egyrelációs sémát használja, akkor a lekérdezés egész másképpen nézne ki. De mindegyik lekérdezés helyesen működik azon az adatbázison, amelyikre írták. □

11.1.3. Adattárházak

Az *adattárházban* mint információegyesítő architektúrában, a különböző információforrásokból nyert adatokat egy *globális* sémába illesztjük. Ezután az adatot a tárházban tároljuk, amely a felhasználó számára úgy néz ki, mint egy közönséges adatbázis.



11.3. ábra. *Az adattárház az egyesített információt egy külön adatbázisban tárolja*

A 11.3. ábrán látható egy adattárház felépítése, bár az ott szereplő két forrásnál sokkal többet is felvehettünk volna a rendszerbe.

Ha az adat egyszer már bekerült a tárházba, akkor a felhasználó ugyanolyan lekérdezéseket hajthat rajta végre, mint bármilyen más adatbázison. Azt viszont általában megtiltják, hogy a felhasználó módosítsa az itt tárolt adatokat, mivel ezek a változtatások nem jelennének meg a tárház adatforrásaiban, és így az adattárház inkonzisztenssé válna a forrásaival szemben.

A tárházban tárolt adat létrehozásának legalább három megközelítési módja van:

1. A tárházat a források aktuális adatai alapján újra és újra létrehozzuk. Ez az eljárás a leggyakoribb, általában az éjszakánként vagy akár hosszabb időközönként végrehajtott rekonstrukcióval (az éjszakai időpont azért nagyon kedvező, mert ilyenkor ki lehet kapcsolni a rendszert, így a tárház építése idején nincsenek lekérdezések). A módszer fő hátránya, hogy a rendszert le kell állítani, és hogy néha a tárház létrehozása hosszabb ideig tart, mint egy normális „éjszaka". Néhány alkalmazásban további hátrányt jelent, hogy a tárházban tárolt adat sokat veszthet az aktualitásából.
2. A tárházat időszakosan frissítjük (például minden éjjel) a forrásokon a tárház utolsó frissítése óta végrehajtott változtatások alapján. Ez az eljárás kisebb adattömeggel dolgozik, mint az előző módszer, és ez nagyon fontos szempont, amikor egy nagy tárházat (sok gigabájt, illetve terabájt nagyságúak vannak manapság használatban) rövid idő alatt kell módosítani. Hátrányt jelent azonban, hogy a tárházon végzendő változtatásokat számító eljárás, a *növekményes frissítés* (incremental update), nagyon összetett azokhoz az algoritmusokhoz képest, amelyek a semmiből építik újjá az adattárházat.

```
INSERT INTO AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
   SELECT sorszám, modell, szín, 'igen', 'forgalmazó2'
   FROM Autók, Extrák
   WHERE Autók.sorsz = Extrák.sorsz AND
         Extrák.extra = 'autoSeb';

INSERT INTO AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
   SELECT sorszám, modell, szín, 'nem', 'forgalmazó2'
   FROM Autók
   WHERE NOT EXISTS (
      SELECT *
      FROM Extrák
      WHERE sorsz = Autók.sorsz AND
            extra = 'autoSeb'
   );
```

11.4. ábra. *Az adatkinyerő Forgalmazó 2 adatait áthelyezi a tárházba*

3. A tárházat azonnal változtatjuk, mihelyst valami módosítás történik valamelyik adatforráson. Ez a megközelítés túl sok kommunikációt és feldolgozást igényel, hogy a gyakorlatban is alkalmazható legyen. Lassan változó forrásokból épített kis tárházak esetén azonban alkalmas lehet a használata. Mindezek ellenére ez a téma jó kutatási terület, és egy ilyen típusú tárház sikeres megvalósításának számos fontos alkalmazása lehetne, például az automatizált értékpapír-kereskedés területén.

11.3. példa: Az egyszerűség kedvéért tegyük fel, hogy a Süni rendszerben csak két forgalmazó szerepel, amelyek rendre a következő sémákat használják:

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...),
```

illetve

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```

Létre akarunk hozni egy adattárházat a következő séma szerint:

```
AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
```

A globális séma tehát hasonló ahhoz, amelyet az első forgalmazó használ, de a tárházban az extra felszerelések közül csak az automata sebességváltót tartjuk nyilván, és felveszünk egy új attribútumot is, amiből megtudhatjuk, hogy melyik forgalmazónál található meg az adott gépkocsi.

A szoftver, amely a második forgalmazó adatbázisából nyert adatokból a globális sémának megfelelően feltölti az adattárházat, SQL-lekérdezések segítségével is megírható. Az első forgalmazóhoz intézett lekérdezés egyszerű:

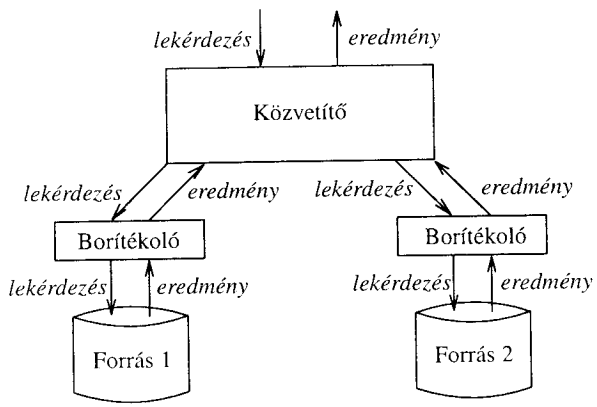

```
INSERT INTO AutóTház(sorszám, modell, szín, autoSeb, forgalmazó)
   SELECT sorszám, modell, szín, autoSeb, 'forgalmazó1'
   FROM Kocsik;
```

A második forgalmazóhoz írt adatkinyerő bonyolultabb, mert el kell döntenünk, hogy az adott autóban van-e automata sebességváltó vagy nincs. A nyilvánvaló jelentéssel bíró `'igen'`, illetve `'nem'` karakterláncokat használjuk az `autoSeb` attribútum értékeként. Az adatkinyerő SQL-kódja a 11.4. ábrán látható.

Ebben az egyszerű példában nincs adategyesítő. Mivel a tárház a forrásokból kiszűrt relációk uniója, az adatokat közvetlenül, előzetes feldolgozás nélkül töltöttük a tárházba. Sok tárház azonban a forrásokból nyert relációkon különböző műveleteket hajt végre. Például két relációt összekapcsolnak, és az eredményt teszik a tárházba, vagy több reláció unióján valamilyen összesítést hajtanak végre. Még általánosabban, minden forrásból több relációt is kiszűrhetnek, és különböző relációkat sokféle módon kombinálhatnak. □

11.1.4. Adatközvetítő

Az adatközvetítő egy vagy több virtuális nézettáblát támogat, amelyek az adattárházban testet öltött relációkhoz nagyon hasonló módon egyesítik az egyes forrásokat. Mivel azonban maga a közvetítő nem tárol adatokat, az adattárháztól elég eltérő módon működik. A 11.5. ábrán látható közvetítő két forrást koordinál. Ahogy azt az adattárház felépítésénél is említettük, kettőnél több forrás használata a tipikus. Első lépésként a felhasználó megad egy lekérdezést a közvetítőnek. Mivel maga a közvetítő nem tárol semmilyen adatot, az információforrásoktól kell megszereznie a szükséges adatokat, és ezek alapján kell választ adnia a felhasználónak.



11.5. ábra. *A közvetítő és a borítékolók átalakítják a lekérdezéseket a források számára, és összegyűjtik az eredményt*

A 11.5. ábrán látható, hogy a közvetítő mindegyik borítékolójának küld egy lekérdezést, és ezek a következő lépésben a saját adatforrásukhoz intéznek egy-egy lekérdezést. Valójában a közvetítő több lekérdezést is küldhet a borítékolónak, és lehet, hogy néhány borítékolónak egyáltalán nem küld lekérdezést. A visszakapott adatokat a közvetítő egyesíti. Nem tüntettünk fel egy kifejezett adategyesítő alkotóelemet, mint ahogy ezt a 11.3. ábrán az adattárház esetén tettük, mert az adatközvetítő esetén a forrásokból visszakapott eredmények egyesítése a közvetítő egyik feladata.

11.4. példa: Tekintsük a 11.3. példában tárgyalt helyzetet, de most adattárház helyett használjunk adatközvetítőt. A közvetítő tehát ugyanazt a két gépkocsi-adatforrást egyesíti, mint az előző példában, és egy nézettáblát hoz létre az alábbi sémával:

```
AutóKözv(sorszám, modell, szín, autoSeb, forgalmazó)
```

Tegyük fel, hogy a felhasználó a piros autók iránt érdeklődik, és a következő lekérdezést teszi fel:

```
SELECT sorszám, modell
FROM AutóKözv
WHERE szín = 'piros';
```

A közvetítő a felhasználó kérdésére reagálva továbbíthatja a lekérdezést mindkét borítékolójának. Hogy hogyan lehet ilyen és ehhez hasonló lekérdezéseket kezelő borítékolókat tervezni és megvalósítani, az a 11.2. rész témája. Bonyolultabb esetekben szükség lehet a lekérdezés alkotórészeinek az átalakítására és szétosztására is, de ebben az esetben az átalakítás műveletét egyedül a borítékolóra is lehet hagyni.

A Forgalmazó 1-hez tartozó borítékoló lefordítja a lekérdezést a forgalmazó sémájának megfelelően, amely emlékeztetőül a következő:

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...)
```

Egy megfelelő fordítás:

```
SELECT sorszám, modell
FROM Kocsik
WHERE szín = 'piros';
```

A `sorszám-modell` párokból álló válaszhalmazt visszaküldi a borítékoló a közvetítőnek.

Eközben a Forgalmazó 2-höz tartozó borítékoló ugyanazt a lekérdezést a második forgalmazó sémájának megfelelően alakítja át. A séma:

```
Autók(sorsz, modell, szín)
Extrák(sorsz, extra)
```


és az ennek megfelelő lekérdezés:

```
SELECT sorsz, modell
FROM Autók
WHERE szín = 'piros';
```

A borítékoló válaszként visszaküldi a `sorsz-modell` párokból álló halmazt a közvetítőnek, esetleg előtte végrehajtva azt a nem túl megerőltető feladatot, hogy az eredménytáblázat sémájában a `sorsz` attribútumot `sorszám`-ra cserélje.

A közvetítő ezek után veheti a két reláció unióját, és ezzel megadhatja a választ a felhasználó kérdésére. Mivel a kocsisorszámtól elvárjuk, hogy „globális kulcsként" működjön, vagyis hogy ne legyen két autó, még különböző adatbázisokban sem, amely ugyanazzal a sorszámmal rendelkezik, vehetjük a két reláció zsák-unióját, feltételezve, hogy úgysem lesznek az eredményrelációban azonos sorok. □

A közvetítőnek számos olyan választási lehetősége van egy lekérdezés megválaszolása közben, amelyre a 11.4. példában nem került sor. Például megteheti, hogy először csak egy adatforrást kérdez le, majd ennek az eredménye alapján dönt a következő lekérdezés(ek) felől. Ez a módszer megfelelő lenne például, ha a felhasználót az érdekelte volna, hogy van-e Süni „Tüskés" sportkupé kékben. A közvetítő először csak Forgalmazó 1-et kérdezné le, és ha ennek az eredménye az üres halmaz lenne, csak akkor fordulna Forgalmazó 2-höz a következő lekérdezéssel.

11.1.5. Feladatok

11.1.1. feladat: Ellenőrizzük néhány on-line könyvkereskedés weboldalán, hogy milyen információk találhatók erről a könyvről. Hogyan hoznánk létre ezek alapján egy olyan globális sémát, amely megfelel egy adattárház vagy egy közvetítő számára?

! **11.1.2. feladat:** Az *A* számítógépekkel foglalkozó cég a következő sémának megfelelően tartja nyilván az árusított modelleket:

```
Szgépek(szám, proc, seb, memória, hd)
Monitorok(szám, képernyő, maxX, maxY)
```

A (123, PIII, 500, 128, 18,7) sor az `Szgépek` relációban például azt jelenti, hogy a 123-as számú modellhez egy 500 MH-es Pentium–III processzor, 128 Mb memória és 18,7 Gb merevlemez tartozik. A (456, 19, 1600, 1200) sor a `Monitorok` relációban azt jelenti, hogy a 456-os számú modell képernyője 19 inches és a maximális felbontása 1600 × 1200.

A *B* számítógépekkel foglalkozó cég csak teljes konfigurációkkal foglalkozik, számítógépet és monitort csak együtt ad el. Ez a következő sémát használja:

```
Konfig(azon, processzor, mem, lemez, képMéret)
```

A `processzor` attribútum értéke egész szám és a processzor sebességét adja meg. A processzor típusa (például Pentium III) és a monitor maximális felbontása nincs nyilvántartva. Az `azon`, `mem` és `lemez` attribútumok az *A* cég `szám`, `memória` és `hd` attribútumaihoz hasonló jelentéssel bírnak azzal a különbséggel, hogy a merevlemez mérete itt gigabájt helyett megabájtban van megadva.

a) Milyen SQL-utasítást kellene kiadnia az *A* cégnek, ha a *B* cég által árusított cikkekről szeretne információt tárolni a saját adatbázisában?

* b) Milyen SQL-utasítás lenne *B* számára a legmegfelelőbb, ha az *A* számítógépeiből és monitoraiból összeállítható konfigurációkról a lehető legtöbb információt akarja felvenni a saját `Konfig` relációjába?

*! **11.1.3. feladat:** Milyen globális séma segítségével tudnánk a lehető legtöbb információt nyilvántartani a 11.1.2. feladat *A* és *B* cégei által kínált termékekről?

11.1.4. feladat: Adjuk meg azt a lekérdezéssorozatot, amely az *A* és *B* cégek adataiból gyűjtött információt egy olyan adattárházba teszi, amely a 11.1.3. feladat megoldásaként adott globális sémát használja! Szükség esetén használható a szerzők által megadott séma is.

11.1.5. feladat: Tegyük fel, hogy egy közvetítő a 11.1.3. feladat sémáját (akár a saját megoldást, akár a szerzők megoldását) használja. Hogyan lehetne megadni az 500 MHz-es számítógépekkel együtt elérhető maximális merevlemez méretét?

! **11.1.6. feladat:** Adjunk meg két másik sémát, amelyek segítségével a számítógépes cégek a 11.1.2. feladatban leírtakhoz hasonló adatokat tarthatnak nyilván!

! **11.1.7. feladat:** A 11.3. példában szó volt a Forgalmazó 1 `Kocsik` relációjáról, amely tartalmazott egy kényelmesen használható `autoSeb` attribútumot, amelynek csak „igen"/„nem" értékei lehettek. Mivel a globális sémában ehhez az attribútumhoz ugyanezeket az értékeket használtuk, az `AutóTház` relációt nagyon könnyű volt létrehozni. Most tegyük fel, hogy a `Kocsik.autoSeb` attribútum egész értékeket vehet fel; 0 jelenti azt, hogy nincs automata sebességváltó beszerelve, $i > 0$ pedig azt, hogy i sebességes automata sebességváltó tartozik a kocsihoz. Adjuk meg azt az SQL-lekérdezést, amely segítségével a `Kocsik` reláció átültethető az `AutóTház` relációba!

11.1.8. feladat: Hogyan fordítaná a 11.4. példa közvetítője a következő lekérdezéseket:

* a) Adjuk meg az automata sebességváltós gépkocsik sorszámát!

b) Adjuk meg azon gépkocsik sorszámát, amelyek nem rendelkeznek automata sebességváltóval!

! c) Adjuk meg a Forgalmazó 1-nél elérhető kék gépkocsik sorszámát!

11.2. Borítékolók a közvetítő alapú rendszerekben

A 11.3. ábrához hasonló adattárház adatkinyerői a következő alkotóelemekből állnak:

1. Egy vagy több beépített lekérdezés, amelyek végrehajtásával az adatforrásból adatok állíthatók elő az adattárház számára.
2. Megfelelő kommunikációs mechanizmus, amely segítségével az adatkinyerő képes az alábbi feladatokra:

   a) ad hoc lekérdezéseket továbbítani a forrásnak,
   b) válaszokat fogadni a forrástól és
   c) információt átadni az adattárháznak.

A forrás számára beépített lekérdezések lehetnek SQL-lekérdezések, ha az adatforrás egy olyan SQL-adatbázis, mint amilyenekkel a 11.1. rész példáiban találkoztunk. Ha a forrás azonban nem hagyományos adatbázis, az adatkinyerőbe beépített lekérdezések akármilyen más, a forrás által értelmezhető műveletek is lehetnek. Például a borítékoló kitölthet egy weboldalon található űrlapot, lekérdezhet egy on-line bibliográfiát a rendszer saját, speciális nyelvén, vagy számtalan más eszközt is igénybe vehet, hogy a forrással meg tudja értetni a lekérdezést.

Az adatközvetítőnek azonban sokkal összetettebb borítékolókra van szüksége, mint a legtöbb adattárháznak. A közvetítő lekérdezések egész változatát intézheti a borítékolóhoz, és a borítékolónak képesnek kell lennie ezeket elfogadni és akármelyiket lefordítani a megfelelő adatforrás nyelvére. Ezek után természetesen a lekérdezés eredményét továbbítania kell a közvetítő felé, ahogy ezt a borítékoló a tárház alapú rendszerben is megteszi. Ebben a részben a közvetítők számára használható rugalmas borítékolók konstrukciójával foglalkozunk.

11.2.1. Sablonok lekérdezési formákhoz

Az adatforrás és közvetítő kapcsolatát alkotó borítékoló tervezésének létezik egy szisztematikus módja: a közvetítő által feltett lehetséges lekérdezéseket egy-egy *sablonnak* megfelelő osztályba soroljuk. A sablonok olyan paraméteres lekérdezések, ahol a paraméterek csak konstans értékeket vehetnek fel. A borítékoló a közvetítő által kínált konstansokkal végzi el a lekérdezést. A következő példán keresztül bemutatjuk a módszer lényegét. $T \Rightarrow S$ azt jelöli, hogy a borítékoló a T sablont a forrás által feldolgozható S lekérdezéssé, úgynevezett forráslekérdezéssé alakítja át.

11.5. példa: Olyan borítékolót akarunk létrehozni, amely a közvetítő és Forgalmazó 1 között teremti meg a kapcsolatot. Forgalmazó 1, illetve a közvetítő rendre a következő sémát használja:

```
Kocsik(sorszám, modell, szín, autoSeb, cdJátszó,...)
AutóKözv(sorszám, modell, szín, autoSeb, forgalmazó)
```

Gondoljuk meg, hogyan tudna a borítékoló adott színű kocsikat megadni a közvetítőnek. Ha a szín számára bevezetjük a `$c` paramétert, akkor az adott színtől függetlenül mindig használható lesz a 11.6. ábrán látható sablon. Hasonlóan, a borítékoló rendelkezhet egy másik sablonnal is, a modellnek megfelelő `$m` paraméterrel, sőt egy harmadikkal az automata sebességváltó paraméterezésével és így tovább. Ha a lekérdezés e három attribútum közül bármelyiket rögzítheti, akkor összesen nyolc sablonra van szükségünk.[4] Általában, ha n attribútum rögzítésére van lehetőség, a szükséges sablonok száma 2^n.[5] Másfajta lekérdezésekhez, mint például „bizonyos típusú autók száma" vagy „van-e raktáron egy bizonyos típusú autó?", más sablonok kellenek. A sablonok száma túlságosan is nagyra nőhet, de a borítékoló finomításával egyszerűsíthető a helyzet. Erről a 11.2.3. részben lesz szó. □

```
SELECT *
FROM AutóKözv
WHERE szín = '$c';
   =>
SELECT sorszám, modell, szín, autoSeb, 'forgalmazó1'
FROM Kocsik
WHERE szín = '$c';
```

11.6. ábra. *Borítékoló sablon adott színű kocsik lekérdezésére*

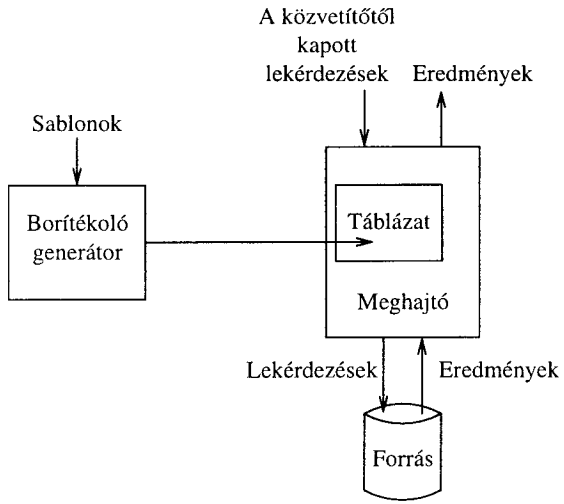
11.2.2. Borítékoló generátor

A borítékolót meghatározó sablonok alapján olyan programot kell írni, amely létrehozza magát a borítékolót. Ez a program a *borítékoló generátor*. Szellemében nagyon hasonló az elemző generátorokhoz (például YACC[6]), amelyek magas szintű specifikációk alapján hoznak létre fordítóprogram-komponenseket. A 11.7. ábrán bemutatott eljárás azzal kezdődik, hogy a specifikációt, vagyis a sablonkészletet, átadjuk a borítékoló generátornak.

[4] A három alapsablonon kívül szükségünk lesz még azokra is, amelyekben egyszerre adható meg a szín és modell, szín és automata sebességváltó és így tovább. A nyolcba beleszámoljuk az „üres" sablont is, amelyben a három paraméter közül egy sem szerepel, vagyis amely az egész egyesített adatbázist adja eredményül. *A fordító megjegyzése.*

[5] Ha az adatforrás ehhez a példához hasonlóan egy olyan adatbázis, amely lekérdezhető SQL-ben, jogosan elvárhatnánk, hogy egy sablon akárhány konstansértékű attribútumot képes legyen kezelni, egyszerűen úgy, hogy a `WHERE` záradékot paraméterré teszi. Ez a módszer működik az SQL-források és az olyan lekérdezések esetén, amelyekben az attribútumok értéke egy konstanshoz van kötve, de nem szükségszerű, hogy bármilyen adatforrás, például egy weboldal esetén is megfelel, ahol csak bizonyos formák használhatók interfészként. Általában nem tehetjük fel, hogy ahogy mi fordítunk le egy lekérdezést, az bármiben is hasonlít ahhoz, ahogy egy hasonló lekérdezés van lefordítva.

[6] *Yet Another Compiler-Compiler*, magyarul Ismét Egy Újabb Fordító Fordító. Értelmező- és fordítóprogramok fejlesztéséhez használt eszköz. *A fordító megjegyzése.*



11.7. ábra. *A borítékoló generátor létrehozza a táblázatokat a meghajtó számára; a meghajtó és a táblázatok alkotják a borítékolót*

A borítékoló generátor ez alapján táblázatot készít, amely a sablon és az annak megfelelő forráslekérdezési formapárokat tartalmazza. Minden borítékolóban használunk egy *meghajtót* (driver), amely általában lehet ugyanaz mindegyik generált borítékolóhoz. A meghajtó feladata a következő:

1. A közvetítőtől érkezett lekérdezések fogadása. Mivel a kommunikációs mechanizmus közvetítőfüggő lehet, a kommunikációs rutint mint „plug-in"-t adjuk a meghajtónak, így ugyanaz a meghajtó más plug-in-nel másféle kommunikációt használó rendszerekben is használható.
2. A lekérdezésnek megfelelő sablon keresése a táblázatban. Ha a keresés sikeres, akkor a lekérdezésből vett paraméterértékekkel kell végrehajtani a forráslekérdezést. Ha a keresés sikertelen, a borítékoló negatív választ küld a közvetítőnek.
3. A forráslekérdezés továbbítása az adatforrás felé, a „plug-in" kommunikációs rutin használatával. A lekérdezésre adott választ a borítékoló fogadja.
4. A válasz továbbítása a közvetítő felé, amelyen szükség esetén a borítékoló valamilyen feldolgozó eljárást hajt végre. Hogy ezzel a borítékolók hogyan támogathatják a lekérdezések egy nagyobb osztályát, azt a következő fejezetekben tárgyaljuk.

11.2.3. Szűrők

Tegyük fel, hogy az egyik forgalmazó adatbázisához tartozó borítékoló rendelkezik a 11.6. ábrán látható sablonnal, azonban az adatközvetítőtől modell *és* szín szerint kérnek tájékoztatást. A borítékoló talán fel van szerelve egy kicsit összetettebb sablonnal

is, amely segítségével kezelhetők a modellt és színt is rögzítő lekérdezések (11.8. ábra). De ahogy a 11.5. példa végén említettük, a gyakorlatban nem mindig valósítható meg, hogy az összes lehetséges lekérdezéshez megírjuk a megfelelő borítékoló sablont.

```
SELECT *
FROM AutóKözv
WHERE modell = '$m' AND szín = '$c';
   =>
SELECT sorszám, modell, szín, autoSeb, 'forgalmazól'
FROM Kocsik
WHERE modell = '$m' AND szín = '$c';
```

11.8. ábra. *Borítékoló sablon, amely az adott modellhez tartozó, adott színű kocsikat adja meg*

Létezik azonban egy másik megoldás is, amellyel növelhető a támogatott lekérdezések száma. Ennek az a lényege, hogy a borítékoló a forráslekérdezések eredményén egy *szűrést* (filter) hajt végre. Ha a borítékoló rendelkezik egy olyan sablonnal, amely (a paraméterek gondos behelyettesítése után) az eredeti lekérdezés eredményének egy bővebb részhalmazát adja, akkor lehetőség van arra, hogy a sablonnak megfelelő sorokból „kiszűrje" azokat, amelyek a kívánt lekérdezés eredményét adják, és csak ezeket küldje tovább a közvetítőnek. Annak eldöntése, hogy a közvetítő által igényelt lekérdezés végeredménye részhalmaza-e a borítékoló egyik sablonjának megfelelő lekérdezés végeredményének, általában nehéz probléma, bár az eddig látott példákhoz hasonló egyszerű esetekre az elmélet jól ki van dolgozva. A kérdés további tanulmányozásához jó kiindulópontok találhatók az irodalomjegyzékben.

11.6. példa: Tegyük fel, hogy egyetlen sablonnal rendelkezünk, azzal, amellyel az adott színű kocsikat kérdezhetjük le (11.6. ábra). A közvetítőnek azonban a kék „Tüskés" autókra van szüksége, azaz a következő lekérdezés végeredményét várja:

```
SELECT *
FROM AutóKözv
WHERE szín = 'kék' and modell = 'Tüskés';
```

A válaszadás egy lehetséges módja:

1. A 11.6. ábra sablonját `$c = 'kék'` helyettesítéssel használva megkeressük az összes kék autót.
2. Az eredményt az alábbi ideiglenes relációban tároljuk:

   ```
   TmpAutók(sorszám, modell, szín, autoSeb, forgalmazó)
   ```

3. Ebből a következő lekérdezés segítségével kiválasztjuk a Tüskéseket, az eredményt pedig visszaküldjük.

Hol végezzük a szűrést?

A példákban eddig mindig feltettük, hogy a szűrés műveletét a borítékoló végzi el. Az is lehetséges azonban, hogy a borítékoló továbbadja a közvetítőnek a még feldolgozatlan információt, és a szűrésre ott kerül sor. Ha viszont a sablon által visszaadott adatok többsége nem felel meg a közvetítő lekérdezésének, akkor a legjobb megoldás még a borítékoló szintjén elvégezni a szűrést, így kerülve el a szükségtelen sorok továbbításából eredő többletköltséget.

```
SELECT *
FROM TempAutók
WHERE modell = 'Tüskés';
```

A végeredmény a kívánt személygépkocsik halmaza. A gyakorlatban a `TempAutók` relációnak egyszerre csak egy sora jönne létre, amit rögtön meg is szűrnénk, csővezetékes jellegű feldolgozással. Inkább ezt a módszert használnánk, minthogy az egész relációt tároljuk és szűrjük a borítékolóban. □

11.2.4. A borítékoló más műveletei

A szűrésen kívül másféle átalakításokat is elvégezhet a borítékoló, amennyiben biztosak vagyunk benne, hogy az ehhez szükséges adatokat hiány nélkül visszakapja a sablon forráslekérdezés részétől. A közvetítőnek való továbbítás előtt elvégezhet például egy vetítést, de az is elképzelhető, hogy összekapcsolások vagy valamilyen összesítések után küldi csak tovább az eredményt.

11.7. példa: Tegyük fel, hogy a közvetítő az egyes forgalmazóknál fellelhető kék Tüskésekről akar információt szerezni, de csak a sorszámra, forgalmazóra és a sebességváltó típusára (automata-e) van szüksége, mivel a `modell` és `szín` mezők értéke a kérdésből nyilvánvaló. A borítékoló követheti ugyanazokat a lépéseket, mint a 11.6. példában, de az utolsó pontban, amikor az eredményeket kell visszaküldenie a közvetítőnek, el kell végeznie egy újabb „szűrést" is a `SELECT` záradékbeli vetítés képében a Tüskés modellek `WHERE` záradékbeli szűrése mellett:

```
SELECT sorszám, autoSeb, forgalmazó
FROM TempAutók
WHERE modell = 'Tüskés';
```

A 11.6. példához hasonlóan a `TempAutók` reláció itt sem jönne létre fizikailag a borítékolóban, a vetítést valószínűleg csővezetékes jellegű feldolgozás segítségével soronként végeznénk el. □

11.8. példa: Egy összetettebb példa kedvéért tegyük fel, hogy a közvetítőnek olyan forgalmazó-modell párokat kell találnia, ahol a forgalmazónak ebből a modellből van két piros példánya, az egyik automata sebességváltóval, a másik e nélkül. Még azt is tegyük fel, hogy Forgalmazó 1-hez csak a 11.6. ábráról már jól ismert sablon használható egyedül. A közvetítő tehát a 11.9. ábrán megadott lekérdezésre várja az eredményt a borítékolótól. Vegyük észre, hogy a forgalmazót nem kell megadnunk sem `A1` sem `A2` esetén, mert ez a borítékoló úgyis csak Forgalmazó 1 adataihoz fér hozzá. A közvetítő ugyanezt a lekérdezést küldi el az összes többi forgalmazó borítékolójához is.

```
SELECT A1.modell, A1.forgalmazó
FROM AutóKözv A1, AutóKözv A2
WHERE A1.modell = A2.modell AND
   A1.szín = 'piros' AND
   A2.szín = 'piros' AND
   A1.autoSeb = 'nem' AND
   A2.autoSeb = 'igen';
```

11.9. ábra. *Lekérdezés a közvetítőtől a borítékolóhoz*

Egy okosan tervezett borítékoló felismerné, hogy a közvetítő lekérdezését úgy is meg tudja válaszolni, ha először elkészíti a Forgalmazó 1-nél található összes piros autó adatait tartalmazó relációt:

```
PirosAutók(sorszám, modell, szín, autoSeb, forgalmazó)
```

Ehhez a 11.6. ábra sablonját használja, amelynek a segítségével az olyan lekérdezéseket lehet kezelni, amelyek csak egy színt adnak meg. Valójában a borítékoló úgy viselkedik, mintha a következő lekérdezésre válaszolna:

```
SELECT *
FROM AutóKözv
WHERE szín = 'piros';
```

A 11.6. ábra sablonját a `$c = 'piros'` behelyettesítéssel használva létre tudja hozni a `PirosAutók` relációt a Forgalmazó 1 adatbázisa alapján. Ahhoz, hogy a 11.9. ábrán látható lekérdezés végeredményét adja meg, a következő lépésben össze kell kapcsolnia a `PirosAutók` relációt saját magával, és el kell végeznie a szükséges kiválasztást. A 11.10. ábrán látható a borítékoló[7] által végrehajtott lekérdezés. □

```
SELECT DISTINCT A1.modell, A1.forgalmazó
FROM PirosAutók A1, PirosAutók A2
WHERE A1.modell = A2.modell AND
   A1.autoSeb = 'nem' AND
   A2.autoSeb = 'igen';
```

11.10. ábra. *A borítékoló (vagy a közvetítő) által végrehajtott lekérdezés a 11.9. ábra lekérdezésére ad választ*

[7] Néhány információegyesítő architektúrában ezt a feladatot inkább a közvetítő látná el.

11.2.5. Feladatok

* **11.2.1. feladat:** A 11.6. ábrán egy olyan borítékoló sablont láttunk, amely a közvetítő adott színű gépkocsikra vonatkozó lekérdezéseit a `Kocsik` relációval rendelkező forgalmazó számára értelmezhető lekérdezésekké alakította át. Most tegyük fel, hogy a közvetítő sémájában használt színkódolás más, mint amit a forgalmazó használ, de a következő konverziós táblázattal rendelkezünk: `GblL(globSzín, LokSzín)`. Ennek megfelelően írjunk új sablont a borítékoló számára!

11.2.2. feladat: A 11.1.2. feladatban két számítógépes cégről, *A*-ról és *B*-ről beszéltünk, amelyek különböző sémákat használtak az adatbázisaikban. Tegyük fel, hogy létezik egy közvetítő a következő sémával:

```
PCKözv(gyártó, seb, mem, lemez, képernyő)
```

A reláció egy sora megadja a gyártót (*A* vagy *B*) és az annál a cégnél vásárolható összeállítás jellemzőit: a processzor sebességét, a központi memória, a merevlemez és a képernyő méretét. Írjunk a borítékolóban használható sablonokat a következő típusú lekérdezésekhez (lekérdezésenként összesen két sablont kell készíteni, egyet-egyet mindkét gyártó számára):

* a) Adjuk meg azokat a sorokat, amelyekben a sebesség nagysága egy előre meghatározott értékkel egyenlő!
b) Adjuk meg azokat a sorokat, amelyekben a képernyő mérete egy előre meghatározott értékkel egyenlő!
c) Adjuk meg azokat a sorokat, amelyekben a memória és a merevlemez mérete egy előre meghatározott értékkel egyenlő!

11.2.3. feladat: Tegyük fel, hogy mindkét információforrás (számítógépgyártó) borítékolója rendelkezik az előző feladatban leírt sablonokkal. Hogyan tudná ezeket a közvetítő felhasználni a következő lekérdezések megválaszolásában:

* a) Adjuk meg az összes olyan konfiguráció gyártóját, memóriájának és képernyőjének a méretét, amelynek a processzora 400 MHz-es és merevlemeze 12 Gb!
! b) Adjuk meg az 500 MHz-es konfigurációkban megtalálható maximális merevlemez-méretet!
c) Adjuk meg az összes olyan konfigurációt, amely 128 Mb memóriával rendelkezik, és a képernyő mérete (inchben) meghaladja a merevlemez méretét (gigabájtban)!

11.3. On-line analitikus feldolgozás

Ebben a fejezetben az egyesített információs rendszerek, különösen a tárház alapú rendszerek köré nőtt alkalmazások egy fontos osztályával ismerkedünk meg. Különböző cégek és szervezetek hoznak létre hatalmas adatbázisok felhasználásával olyan

Adattárházak és OLAP

Különböző okai vannak annak, hogy a tárházak miért játszanak olyan fontos szerepet az OLAP-alkalmazásokban. Egyrészt lehetséges, hogy az adataink kezdetben sok különböző adatbázisban vannak szétszórva, vagyis ahhoz, hogy OLAP-lekérdezéseket tudjunk végrehajtani, egy adattárházat kell létrehoznunk, amelyben megfelelően tudjuk szervezni és központosítani az egyesített adatokat. Másrészt viszont legtöbbször fontosabb annak a ténye, hogy az OLAP-lekérdezések végrehajtása – mivel ezek összetettek és az adatbázis nagy részét érintik – túl sok időt vesz igénybe az olyan tranzakciókezelő rendszerekben, amelyektől nagy teljesítményt várunk el. Az OLAP-lekérdezéseket gyakran tekinthetjük a 10.7. rész értelmében „hosszú tranzakcióknak".

Az egész adatbázist záró hosszú tranzakciók miatt a szokásos OLTP-műveletek elvégzése lehetetlenné válna (például az eladásokból származó átlagos bevételt számító OLAP-lekérdezés futtatása közben nem vehetnénk fel az újabb vásárlások adatait az adatbázisba). Ezt a problémát rendszerint úgy oldjuk meg, hogy a még feldolgozatlan, nyers adatok másolatait összegyűjtjük egy adattárházba, itt futtatjuk az OLAP-lekérdezéseket, az adatváltoztató és az OLTP-lekérdezéseket pedig az adatforrásokon hajtjuk végre. A tárházat legtöbbször csak éjszakánként módosítjuk, napközben az elemzők a „befagyasztott" adatokon dolgoznak. Vagyis az adattárház adatai már 24 óra alatt elavulnak, ami az OLAP-lekérdezések eredményének az időszerűségét elég jól korlátozza, de ez az „időeltolódás" sok döntéstámogató alkalmazásban még a tűréshatáron belül marad.

tárházakat, amelyek alapján az arra kijelölt elemzők a szervezet számára fontos mintákat, irányvonalakat próbálnak megállapítani. Ez a tevékenység, az *on-line analitikus feldolgozás (OLAP)*, általában nagyon összetett, egy vagy több összesítő függvényt is felhasználó lekérdezéseket foglal magában. Ezeket a lekérdezéseket gyakran hívjuk *OLAP-* vagy *döntéstámogató lekérdezéseknek*. A 11.3.1. részben láthatunk ezekre néhány példát. Tipikus kérdés az olyan termékek keresése, amelyek iránt mindent egybevéve növekszik vagy éppen csökken a kereslet.

Az OLAP-alkalmazásokban használt lekérdezések jellemzően nagyon nagy mennyiségű adatot vizsgálnak át, még ha az eredmény nagyon kicsi is lesz. Ezzel szemben a mindennapos adatbázis-műveletek (például banki befizetés vagy repülőjegy-foglalás) az adatbázisnak ennél csak lényegesen kisebb hányadát érintik. Az ilyen típusú műveleteket gyakran nevezzük *on-line tranzakció-feldolgozásnak (OLTP)*.[8]

Az utóbbi időben olyan új lekérdezésfeldolgozó technikákat dolgoztak ki, amelyek különösen jól használhatók az OLAP-lekérdezések hatékony végrehajtásához. Az OLAP-lekérdezések bizonyos osztályainak különböző természete miatt pedig speciá-

[8] Az OLTP mozaikszó az angol *On-Line Transaction Processing* kifejezésnek felel meg. *A fordító megjegyzése.*

lis adatbázis-kezelő rendszereket, az *adatkocka*rendszereket fejlesztették ki és dobták piacra, hogy az OLAP-alkalmazásokat megfelelően támogassák. Ezek a rendszerek a 11.4. részben kerülnek tárgyalásra.

11.3.1. OLAP-alkalmazások

Az OLAP-alkalmazások általában fogyasztásra vonatkozó adatokból összeállított tárházat használnak. Nagyobb üzlethálózatok terabájtnyi információt halmoznak fel arról, hogy melyik üzletükben melyik cikkből mennyit adtak el. A cégre váró problémák vagy nagy lehetőségek előrejelzésében nagy szerepe lehet az olyan lekérdezéseknek, amelyek a fogyasztási adatok csoportosítása, összesítése alapján a valamilyen szempontból jelentős csoportokat azonosítani tudják.

11.9. példa: Tegyük fel, hogy a Süni cég létrehoz egy adattárházat, amely alapján elemezni tudja majd a gépkocsik iránti keresletet. A tárház sémája lehet a következő:[9]

```
Eladások(sorszám, dátum, forgalmazó, ár)
Autók(sorszám, modell, szín)
Forgalmazók(név, város, állam, tel)
```

Egy jellemző döntéstámogató lekérdezés az 1999. április 1-jén vagy azóta történt értékesítések alapján megállapíthatná, hogy az eladott járművek átlagára hogyan változik államonként. Egy ilyen lekérdezést mutatunk be a 11.11. ábrán.

```
SELECT állam, AVG(ár)
FROM Eladások, Forgalmazók
WHERE Eladások.forgalmazó = Forgalmazók.név AND
      dátum >= '1999-01-04'
GROUP BY állam;
```

11.11. ábra. *Államonkénti átlagos fogyasztói ár*

Figyeljük, meg, ahogy a lekérdezés végigveszi az adatbázis nagy részét, miközben az `Eladások` relációban tárolt vásárlási adatokat osztályozza a forgalmazó címe szerint. Ezzel szemben a „milyen áron adták el a 123-as sorszámú autót?" típusú gyakori OLTP-lekérdezések az adatbázis egyetlen sorát érintik csupán. □

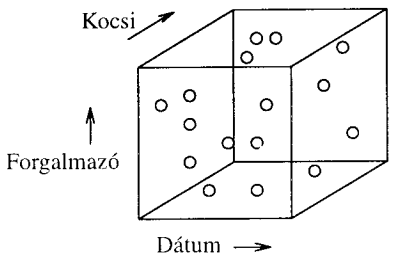
Hogy lássunk egy másik OLAP-példát is, vegyünk egy hitelkártya-kibocsátó céget, amely a kártyaigénylőiről akarja eldönteni, hogy hitelképesek lesznek-e. Készítenek egy adattárházat, amely a cég összes ügyfeléről és az eddigi befizetéseiről tartalmaz információkat. Az OLAP-lekérdezések olyan tényezőket (például életkor, jövedelem,

[9] Az Egyesült Államokban ésszerű a forgalmazó címében az államot is nyilvántartani, ezért szerepel a `Forgalmazók` relációban az `állam` attribútum. *A fordító megjegyzése.*

ingatlantulajdon, irányítószám) fognak keresni, amelyek segíthetnek annak az előrejelzésében, hogy egy adott ügyfél be fogja-e fizetni időben a számláit vagy nem. Ehhez hasonlóan kórházak is rendelkezhetnek betegek adatait (felvétel időpontja, elvégzett laborvizsgálatok, ezek eredménye, diagnózis, gyógykezelés leírása és így tovább) tartalmazó adattárházzal, hogy elemezni tudják a kezeléssel járó kockázatot, és ennek tükrében válasszák ki a legjobbnak ítélt módszert.

11.3.2. OLAP-adatok többdimenziós nézete

A tipikus OLAP-alkalmazásokban mindig létezik egy központi reláció vagy adatgyűjtemény: a *ténytáblázat* (fact table). A ténytáblázat olyan érdekes eseményeket vagy objektumokat tartalmaz, mint a 11.9. példában megismert vásárlási adatok. Gyakran segít, ha a ténytáblázatban tárolt objektumokra úgy gondolunk, mintha egy többdimenziós térben vagy „kockában" lennének elrendezve. A 11.12. ábrán háromdimenziós adatobjektumok láthatók, a kocka belső pontjaiként ábrázolva.[10] A dimenziók elnevezései: kocsi, forgalmazó, dátum, a korábbi személygépkocsi értékesítős példának megfelelően. A kocka minden egyes belső pontjára gondolhatunk úgy, mint egy-egy gépkocsi eladására, a dimenziókra pedig úgy, mintha ennek az eladásnak a körülményeit, jellemzőit írnák le.



11.12. ábra. *Az adatok többdimenziós kockába szervezése*

A 11.4. részben bevezetünk egy, az OLAP-adatok kezelésére alkalmas, speciális architektúrát, az „adatkockát". Ez a felépítés egy kicsit más szempontból közelíti meg a többdimenziós adatokat, ugyanis az adatkockában a pontok jelenthetnek összesített adatot is. Például ahelyett, hogy a „kocsi" dimenzió mentén minden egyes pont más és más kocsit jelölne, elképzelhető, hogy a dimenzió modellenként van összesítve, és a 11.12. ábra pontjai nem egyes autók, hanem egyes modellek összeladását jelentik a forgalmazó és a dátum dimenziók által meghatározott paraméterek mellett. A többdimenziós adat e kétfajta értelmezése közti különbség tükröződik a kockaszerkezetű OLAP-adatot támogató speciális rendszerek által vett két fő irányvonalban is:

[10] Háromnál több dimenzióval rendelkező objektumok esetén természetesen legalább négydimenziós kockát kellene felhasználnunk, de azt jóval nehezebb elképzelni és lerajzolni is. *A fordító megjegyzése.*

1. *ROLAP*[11] vagy *Relációs OLAP*. Ebben a megközelítésben az adatot relációkban tároljuk, de egy speciális szerkezetben, a „csillag sémában". A relációk egyike a ténytáblázat, amely a *nyers*, vagyis a „még nem összesített" adatokat tartalmazza. A rendszer lekérdezőnyelve és más képességei ezt az adatszervezési módot kihasználva alakíthatók. A csillag sémával a 11.3.3. részben foglalkozunk.
2. *MOLAP*[12] vagy *Többdimenziós OLAP*. Ebben az esetben a fent említett speciális szerkezetet, az „adatkockát" használjuk az adat tárolására. Ahogy már említettük, ez az adat gyakran már részben összesített. Az efféle adat OLAP-lekérdezéseit támogatandó, a rendszer nem relációs műveleteket is megvalósíthat.

11.3.3. A csillag séma

A *csillag séma* a ténytáblázat sémájából áll, amely több más relációhoz, a „dimenziótáblázatokhoz" kapcsolódik. A dimenziótáblázatokról kicsit később lesz részletesebben szó. A „csillag" központjában található a ténytáblázat, a csúcsaiban pedig a dimenziótáblázatok. A ténytáblázatnak általában van néhány dimenzió attribútuma – ezek az egyes *dimenziókat* jelölik –, és egy vagy több *függő* attribútuma, amelyek az adat, mint a többdimenziós tér egy pontjának, mint egésznek érdekes tulajdonságait jelölik. Például a vásárlásra vonatkozó adatok dimenziói között szerepelhet a vásárlás időpontja, helyszíne (melyik üzletben történt), a vásárolt cikk típusa, a fizetési mód (készpénz vagy hitelkártya) és így tovább. Függő attribútum(ok) lehet(nek) például a fogyasztói ár, a kereskedelmi ár vagy az adó mennyisége.

11.10. példa: Az előző példában bevezetett `Eladások` reláció

```
Eladások(sorszám, dátum, forgalmazó, ár)
```

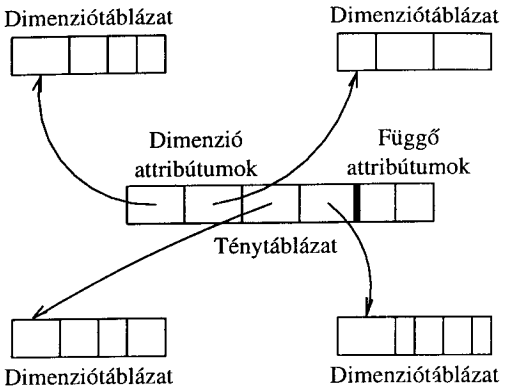
a ténytáblázat. A dimenziók a következők:

1. `sorszám`, az eladott gépkocsit jelöli, azaz az összes gépkocsi terében az ezzel a sorszámmal rendelkező autónak megfelelő pontot.
2. `dátum`, a vásárlás időpontját jelöli, azaz a vásárlás, mint esemény pozícióját az idő dimenzióban.
3. `forgalmazó`, az esemény pozícióját jelöli az összes forgalmazó terében.

Az egyetlen függő attribútum az `ár`, amire az adatbázis OLAP-lekérdezéseinek jellemzően valamilyen összesített formában lesz majd szüksége. Persze az összár vagy az átlagár megállapítása mellett a leszámlálós lekérdezéseknek is lehet értelmük. Például „adjuk meg, hogy 1999 májusában forgalmazóként összesen hány autót adtak el". □

[11] A ROLAP mozaikszó az angol *Relational OLAP* kifejezésnek felel meg. *A fordító megjegyzése.*

[12] A MOLAP mozaikszó az angol *Multidimensional OLAP* kifejezésnek felel meg. *A fordító megjegyzése.*



11.13. ábra. *A ténytáblázat dimenzió attribútumai a dimenziótáblázatok kulcsmezőire hivatkoznak*

A ténytáblázat kiegészítéseképpen használhatók a *dimenziótáblázatok,* amelyek az egyes dimenziók lehetséges értékeit írják le. A ténytáblázat mindegyik dimenzió attribútuma rendszerint idegen kulcsként működik a megfelelő dimenziótáblázatban, ahogy ezt a 11.13. ábra szemlélteti. A dimenziótáblázat attribútumai azt is leírják, hogy melyek azok a lehetséges csoportosítások, amelyeknek értelmük lenne egy SQL-lekérdezés `GROUP BY` záradékában. A következő példán keresztül bemutatjuk az új fogalmakat.

11.11. példa: A 11.9. példa gépkocsi-adatbázisában a dimenziótáblázatok közül kettő nyilvánvaló:

```
Autók(sorszám, modell, szín)
Forgalmazók(név, város, állam, tel)
```

A ténytáblázat

```
Eladások(sorszám, dátum, forgalmazó, ár)
```

`sorszám` attribútuma idegen kulcs, és az `Autók` dimenziótáblázatban[13] a sorszám attribútumra hivatkozik. Az `Autók.modell` és `Autók.szín` attribútumok pedig az adott autó tulajdonságait írják le. Szerepelhetne még sokkal több attribútum is ebben a relációban, igaz/hamis értékekkel jelölve, hogy a kocsiban van-e automata sebességváltó vagy akármilyen más extra felszerelés. Ha az `Eladások` ténytáblázatot összekapcsoljuk az `Autók` dimenziótáblázattal, akkor a `modell` és `szín` attribútumok szerint különböző érdekes módon lehet csoportosítani a vásárlásokat. Például lebonthatjuk az adatokat szín szerint, vagy a Tüskés modell értékesítéseket hónap és forgalmazó szerint.

[13] Most véletlenül a `sorszám` az `Eladások` relációban is kulcsként működik, de nem kell, hogy legyen olyan attribútum, amely a ténytáblázatnak is kulcsa és ideiglenes kulcs is egyben valamelyik dimenziótáblázatban.

Hasonlóan, az `Eladások` táblázat `forgalmazó` attribútuma idegen kulcs a `Forgalmazó` dimenziótáblázatban a `név` attribútumra hivatkozva. Ha az `Eladások` és `Forgalmazó` táblázatokat összekapcsoljuk, további lehetőségek adódnak az adatok csoportosítására. Például lebonthatjuk a vásárlásokat állam, város vagy forgalmazó szerint.

Eltűnődhetünk azon, hogy hol találjuk az időnek (az `Eladások` táblázat `dátum` attribútumának) megfelelő dimenziótáblázatot. Mivel az idő egy fizikai adottság, nincs értelme az adatbázisban időre vonatkozó tényeket tárolni, hiszen a „melyik évre esik 2000. július 5.?" típusú lekérdezések eredményét úgysem tudjuk megváltoztatni. De minthogy az elemzőknek gyakran van szükségük különböző időegységek, például hét, hónap, negyedév, év szerinti csoportosításra, segít, ha az idő fogalmát beépítjük az adatbázisba, éppen úgy, mintha a következő dimenziótáblázattal rendelkeznénk:

```
Napok(nap, hét, hónap, év)
```

A „reláció" egy jellegzetes, 2000. július 5-nek megfelelő sora lenne a következő:

```
(5, 27, 7, 2000)
```

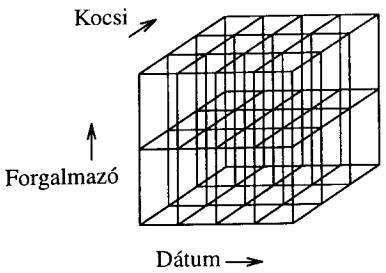
Ezt úgy értelmezzük, hogy ez a nap a 2000. év hetedik hónapjának az ötödik napjára esik, és történetesen a 2000. év 27-edik teljes hetéhez tartozik. Ez bizonyos fokig redundáns, mert a hetet ki lehet számolni a másik három attribútum alapján, viszont a hetek nem illeszthetők pontosan a hónapokhoz, azaz hetek szerinti csoportosításból nem nyerhetünk hónapok szerinti csoportosítást, és megfordítva sem. Tehát van értelme annak, ha úgy képzeljük el, hogy a hetek és a hónapok is jelölve vannak ebben a „dimenziótáblázatban". □

11.3.4. Szeletelés és kockázás

Az adatkockára úgy is gondolhatunk, mintha minden dimenzió mentén valamekkora finomsággal fel lenne osztva. Például az idő dimenziót feloszthatjuk (SQL-es szóhasználattal „csoportosíthatjuk"[14]) napok, hetek, hónapok, évek szerint, de azt is megtehetjük, hogy nem osztjuk fel egyáltalán. Az autó dimenziót feloszthatjuk modell szerint, szín szerint, modell és szín szerint, a forgalmazó dimenziót pedig a forgalmazó neve, a város vagy az állam szerint. Természetesen e két dimenzió esetén is megtehetjük, hogy egyáltalán nem csoportosítunk.

Ha minden dimenzióhoz választunk egy felosztást, ez „felkockázza" az adatkockát, ahogy ez a 11.14. ábrán látható. A kockázás eredményeképpen az adatkockában kisebb kockák jönnek létre. Az ezeket alkotó pontcsoportok jellemzőit az a lekérdezés

[14] Az angol *group by* kifejezésből. *A fordító megjegyzése.*



11.14. ábra. *A dimenziók felosztása darabolja (kockázza) a kockát*

összesíti, amely a GROUP BY záradékban a felosztásnak megfelelő csoportosítást hajtja végre. A WHERE záradékon keresztül a lekérdezésnek arra is lehetősége van, hogy csak bizonyos felosztásokra koncentráljon egy (vagy több) dimenzió mentén, azaz a kockának csak bizonyos „szeleteivel" foglalkozzon. Ennek az lesz az eredménye, hogy a lekérdezés a kockának csak bizonyos alterein végez összesítést.

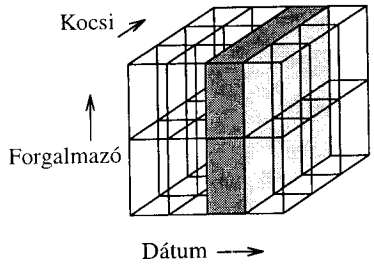
11.12. példa: A 11.15. ábra egy olyan lekérdezésre utal, amely a kockából az egyik dimenzió (dátum) mentén kivág egy szeletet, majd ezt a másik két dimenzió mentén (kocsi és forgalmazó) tovább kockázza. A dátum négy részre van felosztva, talán arra a négy évre, amelyek alatt ezek az adatok összegyűltek. Az ábra satírozása azt akarja kifejezni, hogy ebből a négy évből csak egyetlenegyre vagyunk kíváncsiak.

A kocsik három csoportra vannak osztva, például szedánokra (négyüléses, zárt autókra), kupékra (sportkocsikra) és a kabrioletekre (leereszthető tetejű kocsikra). A forgalmazók kettőre, például a keleti, illetve a nyugati országrészen működőkre. A lekérdezés eredménye egy olyan táblázat, amely a számunkra érdekes évben megadja a gépkocsi-értékesítésekből származó összbevételt mind a hat kategóriában. □

Az úgynevezett „szeletelős-kockázós" lekérdezés általános formája tehát a következő:

```
SELECT csoportosító attribútumok és összesítések
FROM ténytáblázat nulla vagy több dimenziótáblával
     összekapcsolva
WHERE bizonyos attribútumok konstans értékekkel vannak
     összehasonlítva
GROUP BY csoportosító attribútumok;
```

11.13. példa: Folytassuk a személygépkocsis példánkat, de a 11.11. példában megtárgyalt Napok fogalmi szintű dimenziótáblázattal együtt. Ha a Tüskés modellt nem vásárolják annyian, mint gondoltuk, megpróbálhatjuk kiszűrni azokat a színeket, amelyek nem mennek olyan jól. Ez a lekérdezés csak az Autók dimenziótáblázatot használja, és SQL-ben így nézhet ki:



11.15. ábra. *Egy szelet kiválasztása a darabolt (kockázott) kockából*

```
SELECT szín, SUM(ár)
FROM Eladások NATURAL JOIN Autók
WHERE modell = 'Tüskés'
GROUP BY szín;
```

Először szín szerint kockázza, majd modellenként szeleteli az adatkockát. Eközben a Tüskés modellre összpontosít, és a többi adatot figyelmen kívül hagyja.

Tegyük fel, hogy a lekérdezés eredménye nem mondott sokat, minden színből körülbelül ugyanakkora jövedelmünk származott. Mivel a lekérdezés az időt nem bontotta részekre, a színek szerinti végösszeget az egész időtartamra számította ki. Feltehetjük, hogy egy vagy több szín gyenge szereplése csak az utóbbi időre jellemző. Ekkor megpróbálkozhatunk a következő, immár bővített lekérdezéssel, amely hónapok szerint is particionál:

```
SELECT szín, hónap, SUM(ár)
FROM (Eladások NATURAL JOIN Autók) JOIN Napok ON dátum = nap
WHERE modell = 'Tüskés'
GROUP BY szín, hónap;
```

Fontos észben tartanunk, hogy a Napok reláció nem egy szokásos módon tárolt reláció, bár kezelhetjük úgy, mintha a következő sémával rendelkezne.

```
Napok(nap, hét, hónap, év)
```

Az adatkockarendszerek[15] többek között azért is tekinthetők speciális adatbázis-kezelő rendszereknek, mert képesek az efféle „relációk" használatára is.

Az előző lekérdezés alapján esetleg azt állapíthatjuk meg, hogy a piros Tüskések iránt az utóbbi időben nem volt túl nagy kereslet. Következő lépésként megpróbálhatnánk kideríteni, hogy ez általánosan az összes forgalmazóra igaz, vagy csak egy részüknél figyelhető meg ez a probléma. Az újabb lekérdezésben csak a piros Tüskésekre összpontosítunk, és a forgalmazó dimenzió szerint is csoportosítunk. Ily módon a lekérdezés:

[15] Adatkockarendszereknek az adatkocka adatmodellt támogató rendszereket hívjuk. Ezekről bővebben a 11.4. részben lesz szó. *A fordító megjegyzése.*

```
SELECT forgalmazó, hónap, SUM(ár)
FROM (Eladások NATURAL JOIN Autók) JOIN Napok ON dátum = nap
WHERE modell = 'Tüskés' AND szín = 'piros'
GROUP BY hónap, forgalmazó;
```

Ezen a ponton azt vesszük észre, hogy a piros Tüskésekből havonta olyan keveset adtak el, hogy ez alapján nem figyelhető meg könnyen semmilyen jellemző irányvonal. Úgy döntünk tehát, hogy hiba volt a hónapok szerinti felosztás. Jobb ötlet lenne csak évekre bontani az adatokat és csak az utolsó két évet vizsgálni (ebben a képzeletbeli példában 1999-et és 2000-et). A végleges lekérdezés a 11.16. ábrán látható. □

```
SELECT forgalmazó, év, SUM(ár)
FROM (Eladások NATURAL JOIN Autók) JOIN Napok ON dátum = nap
WHERE modell = 'Tüskés' AND
      szín = 'piros' AND
      (év = 1999 OR év = 2000)
GROUP BY év, forgalmazó;
```

11.16. ábra. *A végleges szeletelő-kockázó lekérdezés a piros Tüskés vásárlásokról*

11.3.5. Feladatok

* **11.3.1. feladat:** Egy on-line számítógép-értékesítő cég adatbázisban akarja nyilvántartani a vásárlók megrendeléseit. A vevők a PC-jükbe különböző processzorok, merevlemezek és CD-, illetve DVD-olvasók közül választhatnak, és megadhatják, hogy mekkora központi memóriára van szükségük. Az adatbázis ténytáblázata lehet a következő:

```
Megrendelések(vásárló, dátum, proc, memória, lemez, cd, menny, ár)
```

A `vásárló` attribútum a vevő azonosítójaként működik, és egyúttal idegen kulcs a vásárlókat nyilvántartó dimenziótáblázathoz. Ugyanez igaz a `proc`, `lemez` és `cd` attribútumokra is. A lemezazonosító szerepelhet például abban a dimenziótáblázatban, amely megadja a merevlemez gyártóját és más jellemzőit. A `memória` attribútum egy egész szám, a megrendelt memória méretét adja meg megabájtban. A `menny` attribútum a vásárló által megrendelt konfigurációk számát jelenti, az `ár` attribútum pedig ebből egy darab összköltségét.

a) Melyek a dimenzió attribútumok és melyek a függő attribútumok?
b) Néhány dimenzió attribútumhoz valószínűleg dimenziótáblázat szükséges. Adjunk meg ezekhez megfelelő sémákat!

! **11.3.2. feladat:** Tegyük fel, hogy az előző feladat adatbázisát vizsgáljuk annak érdekében, hogy a jellemző trendek alapján előre jelezhessük a cégnek, hogy milyen alkotórészekből kell majd többet rendelnie. Adjuk meg mélyre ásó és felgörgető lekérdezések egy olyan sorozatát, amely ahhoz a következtetéshez vezethet, hogy a vásárlók egyre inkább előnyben részesítik a DVD-meghajtót a CD-meghajtóval szemben!

Mélyre ásás és felgörgetés

A 11.13. példában az adatkockát szeletelő és kockázó lekérdezések sorozatában két gyakori mintával találkozunk.

1. A *mélyre ásás* az a folyamat, amely során a megfelelő dimenziókat egyre finomabb részekre osztjuk, és/vagy a dimenzió bizonyos értékeire koncentrálunk. A 11.13. példában az utolsó lépés kivételével mindegyik a mélyre ásás folyamatába illik.
2. A *felgörgetés* az egyre durvább particionálás folyamata. Az utolsó két lépést hozhatjuk fel erre példaként, amelyben hónapok helyett évek szerint csoportosítottunk, hogy az adatok esetlegességét kiküszöböljük.

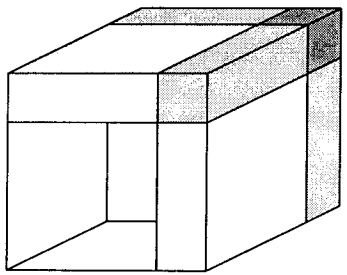
11.4. Adatkocka

A döntéstámogató lekérdezések eddig mindig ad hoc lekérdezések formájában jelentek meg. Ezzel szemben egy másik lehetőség az összes lehetséges összesítés módszeres kiszámítása, előre. Meglepő, de az ehhez szükséges tártöbblet mennyisége gyakran a még ésszerű határon belül marad, és amíg az adattárházban tárolt adat nem változik, az összesített adat frissítése sem jelent többletköltséget. Ebben a fejezetben az adatbázis-kezelő rendszerek egy családjával, az *adatkocka*rendszerekkel, más néven MOLAP- (többdimenziós OLAP) rendszerekkel foglalkozunk. Ezek közvetlenül a 11.12. ábrán bemutatott (adat)kocka adatmodellt támogatják, és a legfontosabb OLAP-műveletek elvégzését is biztosítják.

Az adatkockarendszerekben teljesen hétköznapi dolognak számít, ha még az adatkocka-tároló rendszerbe való felvétel előtt a ténytáblázat nyers adata néhány szempont szerint összesítve van. Az autós adatbázisunkban például a csillag sémában szereplő sorszám dimenziót kicserélhetjük a modell dimenzióra, így minden bejegyzés az adatkockában egy modell, egy forgalmazó, egy dátum, és ezzel együtt az adott modellből az adott napon az adott forgalmazónál történt vásárlások összegének leírásává válik. Az adatkocka pontjainak összességét továbbra is „ténytáblázatnak" nevezzük, még ha a pontok értelmezése egy kicsit el is tér a csillag séma ténytáblázatától.

11.4.1. A kockaművelet

Adott F ténytáblázat esetén definiálhatunk egy kibővített KOCKA(F) táblázatot, amely egy további, *-gal jelölt értéket ad minden dimenzióhoz. A *-nak a jelentése „bármi", és összesítést jelent annak a dimenziónak a mentén, ahol feltűnik. A 11.17. ábra szemlélteti azt az eljárást, amely során a kockához minden dimenzió mentén a * értéket, és az ennek megfelelő összesített értéket képviselő új határvonalakat adjuk hozzá. Az ábrán három dimenziót látunk, a leghalványabb részek jelölik az egy dimenzió mentén vett összesítéseket, a kicsit sötétebbek a két dimenzió mentén, a legsötétebb kis kocka a sarokban pedig a mind a három dimenzió mentén vett összesítéseket. Gondoljuk meg, hogy ha az egyes dimenziók lehetséges értékeinek száma elég nagy, de nem annyira, hogy a legtöbb pont a kocka belsejében szabad legyen (azaz, hogy ne tartozzon hozzá sor a ténytáblázatban), akkor a „határ" csak kis többletet jelent a kocka terjedelméhez (azaz a ténytáblázat sorainak számához) képest. Ebben az esetben a KOCKA(F) táblázat mérete nem sokkal nagyobb az F ténytáblázat méreténél.



11.17. ábra. *A kockaművelet a dimenziók minden lehetséges kombinációján kiszámított összesítéseknek megfelelő határvonalakkal bővíti az adatkockát*

A KOCKA(F) táblázat egy olyan sora, amelyben egy vagy több dimenzió attribútum mentén * található, a függő attribútumoknak megfelelő mezőkben egy összeget (vagy egy más összesítő függvénnyel számított mennyiséget) tartalmaz. Ez úgy számítható ki, hogy vesszük az összes olyan sort, amelyben a dimenzió attribútum értéke nem *, hanem egy valódi érték, és ezen sorok függő attribútumai mentén végezzük el az összesítést. Gyakorlatilag beépítjük az adatokba az összes lehetséges dimenzióhalmazon végzett összesítés eredményét. Vegyük azonban észre, hogy a KOCKA művelet nem támogatja a „köztes szinten" számított összesítéseket. Az adatokat például vagy hagyjuk napokra (illetve az idő dimenzió legfinomabb felbontásának megfelelő időközre) lebontva, vagy a teljes időtartamra nézve összesítünk. A KOCKA művelettel önmagában nem összesíthetünk hetek, hónapok vagy évek szerint.

11.14. példa: A 11.9. példa Süni adatbázisát vizsgáljuk meg a KOCKA művelet lehetőségeinek fényében. Emlékeztetőül, az ott használt ténytáblázat a következő:

```
Eladások(sorszám, dátum, forgalmazó, ár)
```


A `sorszám` attribútumnak megfelelő dimenzió azonban nem alkalmas arra, hogy a kockában használjuk, mert a sorszám egyértelműen azonosítja a gépkocsikat, és így a `sorszám` kulcsként működik az `Eladások` relációban. Vagyis ha a kocsik árát összegezzük a teljes időtartamra vagy az összes forgalmazóra nézve úgy, hogy közben a sorszám rögzítve marad, akkor ennek nem lesz hatása – az adott sorszámú (egyetlen) autó árának az összegét kapjuk. Használhatóbb az adatkocka, ha a sorszámot két másik attribútummal, a modellel és a színnel cseréljük fel. Ezek azok az attribútumok, amelyekhez a sorszám az `Autók` dimenziótáblázaton keresztül az `Eladások` relációt kapcsolja. Vegyük észre, hogy ha a `sorszám` attribútumot kicseréljük a `modell` és `szín` attribútumokra, akkor a kockának már egyetlen dimenziója sem működik kulcsként. Így a kocka egy-egy bejegyzése az olyan kocsik összértékét adja, amelyek adott színűek, adott modellek és az adott napon az adott forgalmazónál vásárolták meg őket.

Ha az `Eladások` ténytáblázatot az adatkockarendszerben akarjuk megvalósítani, egy további változtatás is hasznunkra válhat. Mivel a KOCKA operátor a függő attribútumokat általában összegezni szokta, ezért ha kíváncsiak vagyunk az átlagárra is, szükségünk lehet az eladott kocsik összértékére minden kategóriában (adott szín és modell az adott napon az adott forgalmazónál) és az abba a kategóriába eső vásárlások darabszámára is. Vagyis az `Eladások` reláció, amire a KOCKA műveletet alkalmazzuk, a következő:

```
Eladások(modell, szín, dátum, forgalmazó, összért, db)
```

Az `összért` attribútum az adott modellhez, színhez, dátumhoz és forgalmazóhoz tartozó eladott kocsik összértéke, a `db` pedig az ebbe a kategóriába eső autók darabszáma. Vegyük észre, hogy ebben az adatkockában az egyes autók külön-külön nem azonosíthatók, vagyis ilyen értelemben nincsenek jelen a kockában. Természetes viszont, hogy a saját kategóriájukon belül az `összért` és `db` attribútumok értékére kifejtik a hatásukat.

Vegyük szemügyre a KOCKA(`Eladások`) relációt. Ennek egy lehetséges sora, amely az `Eladások` relációban is megtalálható, a következő:

```
('Tüskés', 'piros', '1999-05-21', 'Undok Ubul', 45000, 2)
```

Ezt úgy értelmezzük, hogy 1999. május 21-én Undok Ubul eladott két piros Tüskést, összesen 45 000 dollár értékben. A következő sor:

```
('Tüskés', *, '1999-05-21', 'Undok Ubul', 152000, 7)
```

azt jelenti, hogy 1999. május 21-én Undok Ubul összesen hét Tüskést adott el, amelyek összértéke 152 000 dollár. Vegyük észre, hogy ez a sor előfordulhat a KOCKA(`Eladások`) relációban, de az `Eladások` biztosan nem tartalmazza.

A KOCKA(`Eladások`) reláció olyan sorokat is tartalmaz, amelyek egynél több attribútum mentén is tartalmaznak összesítést. Például a következő sor:

```
('Tüskés', *, '1999-05-21', *, 2348000, 100)
```

azt jelenti, hogy 1999. május 21-én a forgalmazók összesen 100 Tüskést adtak el, és ezek összértéke 2 348 000 dollár volt.

```
('Tüskés', *, *, *, 1339800000, 58000)
```

Ez a sor pedig azt jelenti, hogy a nyilvántartott egész időtartam alatt a forgalmazók összesen 58 000 Tüskést adtak el (mindenféle színből), összesen 1 339 800 000 dollár értékben. Végül, ebből a sorból:

```
(*, *, *, *, 3521727000, 198000)
```

azt tudhatjuk, meg, hogy a nyilvántartott időtartam alatt a Süni típusú személygépkocsikból összesen 198 000 példányt adtak el (színre és forgalmazóra való tekintet nélkül), és ezek értéke összesen 3 521 727 000 dollár. □

Nézzük meg, hogyan lehet válaszolni az olyan lekérdezésekre, amelyekben az `Eladások` reláció bizonyos attribútumaira feltételeket adunk meg, más attribútumok szerint csoportosítunk, eredményül pedig az összeget, darabszámot vagy az átlagos fogyasztói árat várjuk. A KOCKA(`Eladások`) relációban az olyan *t* sorokat keressük, amelyek rendelkeznek a következő tulajdonságokkal:

1. Ha a lekérdezés az *a* attribútum értékeként *v*-t adja meg, akkor a *t* sor *a* mezőjében *v* szerepel.
2. Ha a lekérdezés *a* attribútum szerint csoportosít, akkor a *t* sor *a* mezőjében tetszőleges nem-* érték szerepel.
3. Ha a lekérdezés nem ad meg *a*-nak értéket és nem is csoportosít *a* szerint, akkor a *t* sor *a* mezőjében * szerepel.

Minden *t* sor tartalmazza az összeget és a darabszámot a csoportosítások egyikére nézve. Ha az átlagárra van szükségünk, akkor minden *t* sorban az összeg és darabszám hányadosát kell venni.

11.15. példa: Ezt a lekérdezést

```
SELECT szín, AVG(ár)
FROM Eladások
WHERE modell = 'Tüskés'
GROUP BY szín;
```

úgy válaszoljuk meg, hogy kikeressük a KOCKA(`Eladások`) reláció minden olyan sorát, amely a következő formában írható:

('Tüskés', *c*, *, *, *v*, *n*)

A „kocka" különféle fogalmai

A 11.3. részben kezdtünk el „kockákkal" foglalkozni, de ott még ez a fogalom sokkal filozofikusabb jellegű volt. A relációs adatok bizonyos fajtáira hasznos úgy gondolnunk, mintha egy ténytáblázatból és dimenziótáblázatokból állnának. Ebben az esetben a „kocka" a ténytáblázat egy megjelenési formája.

A 11.4. részben bevezettünk egy formális kocka műveletet, amely összesített értékeket számol ki egy vagy több dimenzió mentén. Ez a művelet a 11.3. rész ténytáblázataira is alkalmazható, hogyha csak a mindent-vagy-semmit összegzéseknek van értelmük. A 11.14. példában azonban azt is láttuk, hogyan lehet egy dimenziót, mondjuk az „autók" dimenziót több, a dimenziótáblázatból nyert dimenzióval helyettesíteni, amelyek az autókat más nézőpontból közelítik meg (ebben a példában a kocsikat a sorszámuk helyett a színükkel és a modellel írtuk le). E változtatás után sokkal jobban ki tudtuk használni a KOCKA művelet lehetőségeit. Addig csak kétféle módon csoportosíthattunk: vagy minden autót összevontunk, vagy nem vontuk össze őket egyáltalán. A régi dimenzió lecserélése után azonban már bármely új dimenzió mentén tudtunk összesíteni (modell, szín vagy mindkettő szerint).

A 11.4.2. részben aztán olyan esetekkel ismerkedtünk meg, ahol a dimenzió több, független dimenzióra osztása (mint a kocsik színre és modellre osztása) nem volt elegendő. A dimenziók rendelkezhetnek ennél összetettebb struktúrával, ahol az egyedeket (például a forgalmazókat) különböző finomságú szinteken vagy még egy ennél is bonyolultabb rendszer szerint lehet csoportosítani, ahogy azt az időegységek példáján láttuk. Az olyan dimenziók esetén, amelyek nem oszthatók további dimenziókra, a KOCKA művelet lehetőségeit kevésbé lehet jól kihasználni, de a bizonyos felosztások szerinti előösszesítés a KOCKA művelet egy olyan általánosítása, amely hatékony lehet és számos forgalomban lévő rendszer is alkalmazza.

ahol *c* bármilyen jellemző szín lehet. Ebben a sorban *v* a megfelelő színű eladott Tüskések összértékét, *n* pedig az ilyen színű eladott Tüskések számát adja meg. A lekérdezés (*c*, *v*/*n*) formában írható sorokat vár eredményül. Bár a átlagár nem egy közvetlen attribútum az `Eladások` vagy a KOCKA(`Eladások`) relációban, az értéke kiszámítható az összérték és a darabszám hányadosaként. A lekérdezésre adott válasz tehát a ('Tüskés', *c*, *, *, *v*, *n*) sorokból nyert (*c*, *v*/*n*) párok halmaza. □

11.4.2. Kockaimplementáció megvalósított nézettáblákkal

A 11.17. ábra kapcsán azt állítottuk, hogy a kocka kibővítése az összesített értékekkel nem jár jelentős tártöbblettel, sőt időt takarítunk meg vele a leggyakrabban előforduló döntéstámogató lekérdezések tekintetében. Ez az elemzésünk azonban azon a feltevé-

sen alapult, hogy a lekérdezés vagy mindent átfogóan összesít egy dimenzióra nézve, vagy nem összesít egyáltalán. Néhány dimenzió esetén viszont a csoportosítás többféle finomsági fokon is elvégezhető.

Már említettük az idő dimenziót, ahol számos lehetőségünk van a csoportosításra. Az alapvető „mindent vagy semmit", azaz az egész időtartamra, illetve a napokra való lebontás mellett összesíthetünk például hetek, hónapok, negyedévek vagy évek szerint is. Egy másik példa kedvéért gondoljunk a személygépkocsis adatbázisra. Lehetőségünk volt a forgalmazók teljes vagy „egyáltalán nem" összevonására. Dönthetünk azonban a város, az állam vagy más kisebb-nagyobb területegység szerinti összegzés mellett is. Vagyis az idő szerinti csoportosítás esetén legalább hat, forgalmazók szerint pedig legalább négy választási lehetőségünk van.

Ha a választható csoportosítási szintek száma minden dimenzió mentén növekszik, egyre drágább az összes lehetséges csoportosítási kombináció összesített eredményeit tárolni. Nem csak az jelent gondot, hogy túl sok adatot kell tárolni, hanem az is, hogy nem olyan könnyen szervezhetők, mint a 11.17. ábrán a „mindent vagy semmit" esetén. Ezért a forgalomban lévő adatkockarendszerek segítenek a felhasználónak az adatkocka valamilyen *megvalósított nézettábláit* kiválasztani. A megvalósított nézettábla egy lekérdezés végeredménye, amelyet inkább az adatbázisban tárolunk, mint hogy újra és újra előállítsuk az egészet vagy bizonyos részeit egy lekérdezés megválaszolása közben. A megvalósított nézettáblák rendszerint az egész adatkocka különböző szempontok szerint számított összesítéseit tartalmazzák.

Minél durvább a csoportosításnak megfelelő felosztás, annál kevesebb helyet foglal a megvalósított nézettábla. Másrészről azonban, ha a nézettáblát fel akarjuk használni egy bizonyos lekérdezés megválaszolásához, akkor nem szabad egyetlen egy dimenziót sem durvábban osztani, mint ahogy azt a lekérdezés teszi. Vagyis, hogy maximálisan kihasználhassuk a megvalósított nézettáblákban rejlő lehetőségeket, nagy nézettáblákat veszünk, amelyek elég finoman osztják fel a dimenziókat csoportokra. Hogy milyen nézettáblákat valósítsunk meg, azt még az elemzőktől várt lekérdezések várható típusa is erősen befolyásolja. A következő példán keresztül bemutatjuk, hogy milyen problémákra kell odafigyelni.

11.16. példa: Térjünk vissza a 11.14. példa adatkockájához:

```
Eladások(modell, szín, dátum, forgalmazó, összért, db)
```

Egy lehetséges megvalósított nézettábla a dátumokat hónaponként, a forgalmazókat városonként csoportosítja. Ezt a nézettáblát, amelyet `EladásokN1`-nek nevezünk, a 11.18. ábrán látható lekérdezés hozza létre. Ez nem szigorú SQL-lekérdezés, mivel úgy képzeljük, hogy a dátumokat és a hozzá tartozó csoportosítási egységeket, mint például a hónap, az adatkockarendszer anélkül is értelmezni tudja, hogy az `Eladások` és a 11.11. példa képzetes `Napok` relációját összekapcsolná.

Egy másik lehetséges megvalósított nézettábla a színeket teljesen összevonja, a dátumokat hetenként, a forgalmazókat pedig államonként csoportosítja. Ezt a nézettáblát, az `EladásokN2`-t a 11.19. ábra lekérdezése adja meg. Mindkét nézettábla


```
INSERT INTO EladásokN1
   SELECT modell, szín, hónap, város, SUM(összért) AS összért,
      SUM(db) AS db
   FROM Eladások JOIN Forgalmazók ON forgalmazó = név
   GROUP BY modell, szín, hónap, város;
```

11.18. ábra. *Az* `EladásokN1` *megvalósított nézettábla*

```
INSERT INTO EladásokN2
   SELECT modell, hét, állam,
      SUM(összért) AS összért, SUM(db) AS db
   FROM Eladások JOIN Forgalmazók ON forgalmazó = név
   GROUP BY modell, hét, állam;
```

11.19. ábra. *Az* `EladásokN2` *egy másik megvalósított nézettábla*

használható az olyan lekérdezések megválaszolására, amelyek egyiknél sem bontják finomabb részekre a dimenziókat. Ez a lekérdezés tehát:

```
L1:  SELECT modell, SUM(összért)
FROM Eladások
GROUP BY modell;
```

megválaszolható így:

```
SELECT modell, SUM(összért)
FROM EladásokN1
GROUP BY modell;
```

de akár így is:

```
SELECT modell, SUM(összért)
FROM EladásokN2
GROUP BY modell;
```

Az L_2 lekérdezést

```
L2:  SELECT modell, év, állam, SUM(összért)
     FROM Eladások JOIN Forgalmazók ON forgalmazó = név
     GROUP BY modell, év, állam;
```

azonban csak az `EladásokN1` nézettábla felhasználásával válaszolhatjuk meg, a következő formában:

```
SELECT modell, év, állam, SUM(összért)
FROM EladásokN1
```

```
GROUP BY modell, év, állam;
```

Mellékesen jegyezzük meg, hogy ez a lekérdezés, hasonlóan azokhoz, amelyek valamilyen időegység szerint csoportosítanak, nem szigorú SQL-lekérdezés. Ez azt jelenti, hogy az `EladásokN1` nézettáblának az `állam` nem attribútuma, csak a `város`. Fel kell tennünk, hogy az adatkockarendszer tudja, hogyan lehet városokat államonként összevonni, valószínűleg a forgalmazókat leíró dimenziótáblázat segítségével.

Az L_2 lekérdezéshez nem használható az `EladásokN2` nézettábla. Ugyan a városokat össze tudtuk vonni az államok szerint úgy, hogy az `EladásokN1`-t használni tudjuk, ugyanezt a hetekkel nem tudjuk megtenni, hogy az `EladásokN2`-t is használhassuk, mert az évek nem egyenletesen vannak hetekre osztva. Az 1999. december 29-ével kezdődő hét adatainak egy része például még az 1999-es évhez tartozik, a másik része viszont már 2000-hez. Hogy pontosan hol lehetne a kettőt elválasztani egymástól, azt a hetenként összesített adatokból nem tudjuk megállapítani.

Végül egy olyan lekérdezés, mint:

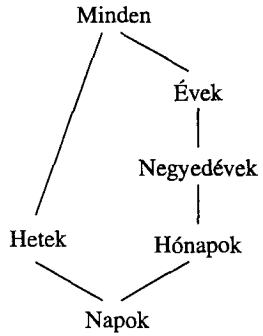
```
L3:  SELECT modell, szín, dátum, SUM(összért)
     FROM Eladások
     GROUP BY modell, szín, dátum;
```

nem válaszolható meg sem az `EladásokN1`, sem az `EladásokN2` segítségével. Nem használhatjuk az `EladásokN1`-t, mert a napok havonkénti összevonása túl durva ahhoz, hogy ez alapján a vásárlásokat napok szerint listázzuk. Nem használható az `EladásokN2` sem, mert ez a nézettábla nem csoportosít színek szerint. Az L_3 lekérdezést közvetlenül az egész adatkocka felhasználásával kellene megválaszolnunk. □

11.4.3. Nézetháló

Hogy a 11.16. példában tett megfigyeléseinket formalizálni tudjuk, hasznunkra válhat egy olyan háló, amely a kocka egyes dimenziói mentén az összes lehetséges csoportosítást tartalmazza. A háló pontjai az adott dimenzióhoz tartozó dimenziótáblázat egy vagy több attribútuma szerinti csoportosítási módot jelölik. Azt mondjuk, hogy a P_1 partíció a P_2 partíció alá esik, ha a P_1 partíció minden egyes csoportját tartalmazza P_2 valamely csoportja. Jelölés: $P_1 \leq P_2$.

11.17. példa: Az idő dimenzió felosztásához választhatnánk a 11.20. ábrán látható hálót. Ha létezik valamilyen P_2 csúcsból P_1-be vezető út, akkor ez azt jelenti, hogy $P_1 \leq P_2$. Az ábrán nem szerepel az összes lehetséges időegység, de mindenesetre jó példáját láthatjuk annak, hogy melyek azok az egységek, amelyeket egy rendszer támogathat. Vegyük észre, hogy a napok a hetek, illetve a hónapok alá esnek, de a hetek nem esnek a hónapok alá. Ennek az az oka, hogy egy adott napon bekövetkezett események biztosan egy megfelelő héten, illetve hónapon belül történtek, de az már nem igaz, hogy egy adott hét folyamán bekövetkezett események összessége szükségszerűen



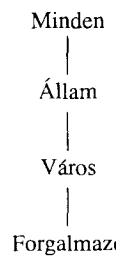
11.20. ábra. *Partíciók hálója az idő intervallumok felosztásához*

egy megfelelő hónapon belül történt volna. Hasonlóan, a hetek eseménycsoportjait nem lehet negyedévek vagy évek eseménycsoportjai alá sorolni. A háló legmagasabb pontjában a „minden" partíció áll. Ez azt jelenti, hogy az események egyetlenegy csoportba vannak beosztva, azaz a különböző időpillanatok között nem teszünk különbséget.

A 11.21. ábrán egy másik háló látható, ezúttal a gépkocsis példánk forgalmazó dimenziójához. Ez egy egyszerűbb háló, amely alapján megtudhatjuk, hogy az autóvásárlások forgalmazó szerinti csoportosítása finomabb, mintha a forgalmazóhoz tartozó város szerint csoportosítottunk volna, a város szerinti felosztás viszont finomabb az állam szerinti felosztásnál. A háló legtetején az a partíció áll, amely az összes forgalmazót egy csoportba osztja be. □

Most, hogy már minden dimenzióhoz meg tudunk adni egy hálót, definiálhatunk egy hálót az adatkocka azon megvalósított nézettábláihoz is, amelyek a dimenziók valamilyen felosztásának megfelelő csoportosítással jönnek létre. Ha N_1 és N_2 a dimenziók valamely partíciója (csoportosítása) alapján megvalósított nézettáblák, akkor $N_1 \leq N_2$ azt jelenti, hogy minden egyes dimenzió N_1-hez használt P_1 partíciója legalább olyan finom, mint ugyanennek a dimenziónak a N_2-höz használt P_2 partíciója, azaz $P_1 \leq P_2$.

A nézettáblák hálójába sok OLAP-lekérdezés is elhelyezhető. Tulajdonképpen az OLAP-lekérdezések gyakran ugyanolyan formájúak, mint az előbb leírt nézettáblák, azaz minden dimenzióhoz megadnak egy-egy felosztást (lehet, hogy a „minden" vagy a „semmi" partíciót). Más OLAP-lekérdezések egy ugyanilyen típusú csoportosítás

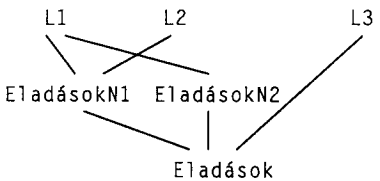


11.21. ábra. *Háló a személygépkocsi-forgalmazók felosztásához*

után még „szeletelik" is a kockát (ahogy a 11.15. ábrán láttuk), hogy az adatoknak csak egy részhalmazára fordítsák a figyelmüket. Az általános szabály a következő:

- Egy L lekérdezés pontosan akkor válaszolható meg a N nézettábla segítségével, ha $N \leq L$.

11.18. példa: A 11.22. ábrán a 11.16. példa nézettábláihoz és lekérdezéseihez adtunk meg egy hálót. Vegyük észre, hogy gyakorlatilag maga az `Eladások` adatkocka is egy nézettábla, ahol minden dimenzió mentén a lehető legfinomabb a felosztás. Ahogy azt az eredeti példában is megfigyeltük, L_1 lekérdezés az `EladásokN1` és `EladásokN2` nézettáblák bármelyikével megválaszolható. Természetesen az egész `Eladások` adatkockát is segítségül hívhattuk volna, de ha a többi nézettáblázat közül legalább az egyik megvalósított (azaz fizikailag jelen van az adatbázisban), akkor nincs okunk az adatkockát választani. L_2 megválaszolható akár az `EladásokN1`, akár az `Eladások` felhasználásával, míg L_3 lekérdezéshez csak az `Eladások` használható. Ezeket a kapcsolatokat a 11.22. ábrán a lekérdezéstől az őket támogató nézettáblákhoz vezető utak fejezik ki. □



11.22. ábra. *Nézettáblák és lekérdezések hálója a 11.16. példához*

Ha a nézettáblák hálójába a lekérdezéseket is beillesztjük, ez segít az adatkocka-adatbázisok tervezésében. Néhány újonnan fejlesztett adatkockarendszer-tervező eszköz az alkalmazás szempontjából „tipikus" lekérdezések halmazából indul ki. Ezután a megvalósítandó nézettáblák halmazát úgy választja ki, hogy minden lekérdezést legalább egy nézettábla támogasson (azaz a hálóban a lekérdezés legalább egy nézettábla fölé essen), de még jobb, ha a lekérdezés valamelyik nézettáblával azonos vagy majdnem azonos (vagyis a legtöbb dimenzió mentén a lekérdezés és a nézettábla ugyanazt a csoportosítást használja).

11.4.4. Feladatok

11.4.1. feladat: Adjuk meg a KOCKA(F) és az F ténytáblázat méretarányát, ha F a következő tulajdonságokkal rendelkezik:

* a) F-nek tíz dimenzió attribútuma van, mindegyik tíz különböző értékkel.
b) F-nek tíz dimenzió attribútuma van, mindegyik két különböző értékkel.

11.4.2. feladat: Ebben a feladatban a 11.14. példa KOCKA(`Eladások`) adatkockát használjuk, amelyet a következő reláció alapján hoztunk létre:


```
Eladások(modell, szín, dátum, forgalmazó, összért, db)
```

Határozzuk meg a kocka azon sorait, amelyeket a következő lekérdezések megválaszolásához használnánk:

* a) Adjuk meg, hogy az egyes forgalmazóknak mennyi bevételük származott a kék autók eladásából!
b) Adjuk meg, hogy „Kedves Kunigunda" a zöld Tüskésekből összesen hányat adott el!
c) Adjuk meg, hogy 2000 márciusában naponta az egyes forgalmazók a különböző színű Tüskésekből átlagosan hány darabot adtak el!

*! **11.4.3. feladat:** A 11.3.1. feladatban egy kockába rendezett adatbázisról volt szó, amelyet számítógép-rendelések tárolására használtunk. Ha alkalmazni akarnánk a KOCKA műveletet, kényelmesebb lenne, ha néhány dimenziót tovább osztanánk. Például az egyetlen processzor dimenzió helyett lehetne egy dimenziónk a processzor típusára (AMD K–6 vagy Pentium–III), egy másik pedig a sebességére. Adjunk meg olyan dimenzió és függő attribútumokat, amelyek sokféle hasznos összesítő lekérdezés végrehajtását támogatják! Mi a szerepe a vásárlónak? A 11.3.1. feladatban az `ár` attribútum egyetlen egy számítógép árát adta meg, holott egyszerre több, ugyanarra a konfigurációra vonatkozó megrendelést is nyilvántarthatnánk ugyanabban a sorban. Melyek lennének a függő attribútumok?

11.4.4. feladat: Határozzuk meg a 11.4.3. feladatban szereplő kocka azon sorait, amelyeket a következő lekérdezések megválaszolásához használnánk:

a) Adjuk meg, hogy 2000-ben havonként összesen mennyit rendeltek a különböző sebességű gépekből!
b) Adjuk meg, hogy összesen hány számítógépet rendeltek, az egyes processzor- és merevlemeztípusok (például SCSI vagy IDE) kombinációira lebontva!
c) Adjuk meg 1999 januárjától kezdve havonként a 400 MHz-es számítógépek átlagárát!

! **11.4.5. feladat:** A 11.4.3. feladatban leírt adatkocka nem tartalmazza a monitorok adatait. Milyen dimenziók segítségével írhatnánk le őket? Feltehetjük, hogy a monitor ára benne van a számítógép árában.

11.4.6. feladat: Tegyük fel, hogy egy kockának 10 dimenziója van, és ezek mindegyikét 5-féle finomsági szinten oszthatjuk fel (beleszámítva a két triviális „minden", illetve „semmi" partíciót is). Hány egymástól eltérő nézettábla nyerhető a dimenziók különböző finomságú felosztásával?

11.4.7. feladat: Mutassuk meg, hogyan illeszthetők be a 11.20. ábrán látható hálóba a következő időegységek: óra, perc, másodperc, két hét, évtized, évszázad!

11.4.8. feladat: Hogyan illesztenénk be a 11.21. ábra forgalmazó hálójába a következő „régiókat", ha:

a) Egy régió az államok egy halmaza.

* b) A régiókat nem lehet államok szerint megadni, de minden várost csak egy régió tartalmaz.

c) A régiók az irányítószámokhoz hasonlóak[16]. Minden régió egy-egy államon belül helyezkedik el, néhány város két vagy több régiónak is része, és néhány régiónak sok város is része lehet.

! **11.4.9. feladat:** A 11.4.3. feladatban egy olyan kockát terveztünk, amelyre alkalmazható a KOCKA művelet. Azonban néhány dimenzióhoz megadható egy nemtriviális hálóstruktúra is. A processzortípust szervezhetnénk például gyártó (SUN, Intel, AMD, Motorola), sorozat (SUN UltraSparc, Intel Pentium vagy Celeron, AMD K-sorozat, Motorola G-sorozat) és modell (Pentium–III, AMD K–6) szerint.

a) A megadott példák alapján tervezzünk hálót a processzortípus dimenzióhoz!

b) Adjuk meg azt a nézettáblát, amely a processzorokat sorozat szerint, a merevlemezeket típus szerint, a CD-meghajtókat sebesség szerint csoportosítja, ezeken kívül pedig minden mást összevon!

c) Adjuk meg azt a nézettáblát, amely a processzorokat gyártó szerint, a merevlemezeket sebesség szerint csoportosítja, ezeken kívül pedig minden mást összevon, kivéve a memóriaméretet!

d) Adjunk példákat olyan lekérdezésekre, amelyekhez

   i) csak a b) nézettábla
   ii) csak a c) nézettábla
   iii) mindkét nézettábla
   iv) egyik nézettábla sem

használható!

*!! **11.4.10. feladat:** Amennyiben az F ténytáblázat ritka (azaz sokkal kevesebb sora van, mint a dimenziók lehetséges értékei számának a szorzata), akkor a KOCKA(F) és F táblázatok méretaránya nagyon nagy lehet. Mennyire?

11.5. Adatbányászat

Az adatbázis-alkalmazások egy családja, az *adatbányászat* (data mining) vagy *tudásbányászat* (knowledge discovery in databases) iránt jelentős érdeklődés mutatkozott az utóbbi időben, mert segítségükkel létező adatbázisok alapján meglepő megfigyelések birtokába juthatunk. Az adatbányászó lekérdezésre úgy is gondolhatunk, mint a

[16] Természetesen az Egyesült Államokban használt irányítószámokról van szó. *A fordító megjegyzése.*

döntéstámogató lekérdezések egy kiterjesztett formájára, bár a kettő között formálisan nincs különbség (lásd „Adatbányászati lekérdezések és döntéstámogató lekérdezések" című doboz). Az adatbányászatban a hagyományos adatbázisrendszereknek mind a lekérdezésoptimalizáló, mind az adatkezelő komponense hangsúlyos szerephez jut. Emellett nagyon fontos téma az adatbázisnyelvek kibővítése is. Itt elsősorban olyan nyelvi építőelemekről (nyelvprimitívekről) van szó, amelyek támogatják a hatékony mintavételt. Ebben a részben bemutatjuk, hogy milyen fő irányokat vett az adatbányász alkalmazások fejlődése. Kicsit részletesebben is foglalkozunk a „társítási szabályok" problémakörrel, amely adatbázisos szempontból az utóbbi időben a legtöbb figyelmet kapta.

11.5.1. Adatbányászati alkalmazások

Nagy vonalakban körülírva, az adatbányászó lekérdezések az adatok egy „hasznos" összefoglalását várják eredményül, gyakran a paraméterként a célnak legjobban megfelelő értékre vonatkozó mindenféle útmutató nélkül. Emiatt újra át kell gondolnunk, hogy hogyan nyerhetünk efféle bepillantást az adatok mélyére az adatbázis-kezelő rendszerek segítségével. Ebben a részben néhány olyan alkalmazást és problémát tekintünk át, amelyek nagyon nagy mennyiségű adattal kapcsolatban kerülnek elő. Mivel sok esetben még nyitott kérdés, hogy hogyan használhatók legjobban az ABKR-ek az adott problémával kapcsolatban, a megoldásokat itt nem tárgyaljuk, csupán néhány szóban utalunk arra, hogy miből áll a feladat nehézsége. A 11.5.2. részben viszont egy olyan probléma tárgyalásába kezdünk, amellyel kapcsolatban figyelemre méltó haladás következett be. Az utolsó részben bemutatunk egy nem triviális, adatbázis-orientált megoldást.

Döntési fa építése

Egy adatbázis felhasználói az adatok alapján egy fontos kérdést akarnak eldönteni. Az 5.7. példában egy olyan kérdést tettünk fel, amely akár egy érdekes adatbányászós probléma alapja is lehetne: „Ki vásárol arany ékszereket?". Abban a példában a vásárlóknak csak két tulajdonságával foglalkoztunk: az életkorukkal és a jövedelmükkel. A mai adatbázisok azonban sokkal több információt is felvehetnek a vásárlókról, közvetlenül vagy legitim források adatait integrálva egy tárházba. Ilyen lehet például a vásárló irányítószáma, családi állapota, lakáshelyzete vagy az általa beszerzett árucikkek különböző jellemzői.

Az 5.7. példával ellentétben, ahol az adatbázis csak olyan embereket tart nyilván, akik már vásároltak arany ékszereket, a *döntési fa* (decision tree) egy olyan eszköz, amely segítségével az adatok két részre bonthatók, az „igen-halmazra" illetve a „nem-halmazra". Az arany ékszerek esetében ezek az adatok emberekről nyilvántartott információkat jelentenének. Az igen-halmazba sorolnánk azokat, akikről valószínűnek tartjuk, hogy vásárolnának arany ékszereket, a nem-halmazba pedig azokat,

Adatbányászati lekérdezések és döntéstámogató lekérdezések

A döntéstámogató lekérdezéseknek lehet, hogy az adatbázis nagy részét kell megvizsgálniuk és összesíteniük, cserébe viszont a kérdést feltevő elemző pontosan megadja a végrehajtandó lekérdezést, azaz tudtára adja a rendszernek, hogy az adat mely részeivel foglalkozzon. Az adatbányászati lekérdezés még egy lépést tesz előre. A rendszerre hagyja annak az eldöntését, hogy a lekérdezés szempontjából az adatbázis melyik része lehet fontos. Egy döntéstámogató lekérdezés például „összesítsük a Süni gépkocsik eladását szín és év szerint" az adatbányászat nyelvén így hangzana: „mely tényezők befolyásolták legjobban a Süni gépkocsik eladását?". Az adatbányászati lekérdezések naiv megvalósításai nagyszámú döntéstámogató lekérdezés végrehajtását jelentik, és ezért olyan sok időbe telhet a válaszadás, hogy ez a fajta megközelítés teljesen használhatatlanná válik.

akikről nem tartjuk valószínűnek, hogy vásárolnának arany ékszereket. Ha az előrejelzésünk megbízható, máris rendelkezésünkre áll egy megfelelő célcsoport, amely tagjainak például közvetlen levél útján küldhetünk reklám-összeállítást az aranyékszer-kínálatunkról.

Maga a döntési fa valahogy úgy nézne ki, mint az 5.13. ábrán, azzal a különbséggel, hogy a levelek nem tartalmaznának számszerű értékeket. Minden belső csúcshoz tartozik egy attribútum és egy küszöbértékként szolgáló attribútumérték. Egy belső pont közvetlen leszármazottja vagy szintén belső pont vagy levél. A leveleket döntési csúcsnak nevezzük, és „igen" vagy „nem" érték tartozhat hozzájuk. Egy reláció adott sorában található adat kiértékelése úgy történik, hogy a döntési fa legtetejéről indulva minden lépésben a belső csúcsokhoz tartozó attribútumérték alapján hol a bal, hol a jobb oldali él mentén haladunk tovább, egészen addig, amíg egy döntési csúcshoz nem érkezünk.

A döntési fát az adatok egy olyan *mintahalmaza* (training set) alapján építjük, amelyekről tudjuk, hogy az igen-halmazba vagy a nem-halmazba tartoznak. Az arany ékszerek esetén vesszük a vásárlók adatbázisát, amely arról is tartalmaz információkat, hogy az egyes vevők vásároltak-e már arany ékszereket vagy sem. Adatbányászati probléma ezen adatok alapján egy olyan döntési fát létrehozni, amely a legnagyobb biztonsággal dönt egy adott tulajdonságokkal (életkor, jövedelem és így tovább) rendelkező új vásárló esetén annak a valószínűségéről, hogy fog-e arany ékszert venni vagy nem. Azaz meg kell határozni a legjobb A attribútumot és a hozzá tartozó legjobb v küszöbértéket, amit a gyökérre írhatunk, majd hasonlóan a gyökér bal- illetve jobboldali leszármazottjára írandó optimális attribútumot, és a küszöbértéket arra az esetre, ha a besorolandó új vásárlóhoz tartozó A attribútum értéke kisebb, illetve legalább akkora, mint v. Ugyanezt a problémát kell megoldanunk a döntési fa minden szintjén, egészen addig, amíg már nem érdemes újabb pontokat felvennünk a gráfba (mert a mintahalmazból túl kevés adat ér el egy adott csúcspontot, hogy ez alapján

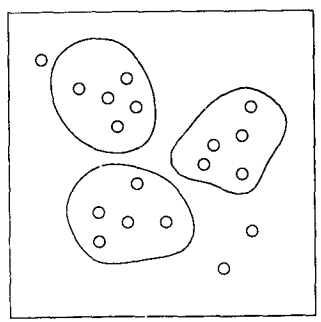
hasznos döntést hozhatnánk). A csúcspontok tervezése során használatos lekérdezések az adatok legnagyobb részét összesítik, így ezek alapján meg tudunk adni egy olyan attribútum-küszöbérték párosítást, amely a mintaadatok „igen" felének nagy részét elválasztja a mintaadatok „nem" felének nagy részétől.

Csoportosítás

Az adatbányászati problémák egy másik osztályába tartozik a „csoportosítás" kérdésköre. A feladat az, hogy az adatokat beosszuk néhány csoportba úgy, hogy az egyes csoportok elemeiben legyen valami lényeges vonás, ami közös. A 11.23. ábrán kétdimenziós adatok csoportosítása látható, a gyakorlatban természetesen a dimenziók száma sokkal nagyobb is lehet. Az ábrán a három legjobb csoport hozzávetőleges körvonalait rajzoltuk be, bár néhány pont mindegyik csoport központjától távol esik. Ezeket tekinthetjük „kívülállóknak", de az egyes pontokat hozzá is csaphatjuk a legközelebb eső csoporthoz.

Egy mintaalkalmazás kedvéért nézzük meg, hogyan működnek az internetes keresőprogramok. Ezek a programok gyakran százezrével találnak olyan weboldalakat, amelyek az adott keresési feltételnek megfelelnek. Hogy ezeket megfelelően lehessen rendszerezni, a keresőprogramok talán a használt szavak alapján csoportosítják a dokumentumokat. Vehetjük például azt a teret, amelyben minden szónak megfelel egy-egy dimenzió, talán a leggyakoribb szavakat, mint „és", „az", „a" *(leállító szavak)* kivéve, amelyek lényegében minden dokumentumban előfordulnak a tartalomtól függetlenül. Minden dokumentum elhelyezhető ebben a térben aszerint, hogy az egyes szavak milyen arányban fordulnak elő a szöveges részében. Ha például egy weboldalon 1000 szó található összesen és ebből kettő az „adatbázis", akkor ezt az „adatbázis" szónak megfelelő dimenzió mentén a 0,002 koordinátához igazítjuk. Ha ebben a térben csoportosítjuk az adott dokumentumokat, akkor a csoportok a dokumentumok témái szerint fognak szerveződni. Például az adatbázisokról szóló szövegekben előfordulhatnak olyan szavak, mint „adat", „lekérdezés", „zárolás" és így tovább, viszont a foci témájú szövegek aligha tartalmaznak ezekhez hasonlókat.

Itt az jelenti az adatbányászati problémát, hogy hogyan határozzuk meg a csoportok



11.23. ábra. *Kétdimenziós adatok három csoportja*

„középértékét" vagy középpontját az adott adatok alapján. Gyakran a csoportok száma előre rögzített, bár ezt az értéket az adatbányászati eljárás is megválaszthatja. Mindkét esetben azonban az a naiv algoritmus, amely úgy választja meg a csoportok középpontját, hogy egy tetszőleges pont átlagos távolsága a hozzá legközelebbi középponttól minimális legyen, számos bonyolult összesítést végrehajtó lekérdezést foglal magában.

11.5.2. Társítási szabály bányászat

Ebben a fejezetben egy olyan adatbányászati problémával ismerkedünk meg, amelyhez eredményesen fejlesztettek ki olyan algoritmusokat, amelyek a háttértárakat használják. A feladat, az úgynevezett „társítási szabályok" keresése, legkönnyebben a legfontosabb alkalmazásán, a *bevásárlókosár-adatok* elemzésén keresztül írható le. A mai áruházak gyakran tárolnak arról feljegyzéseket az adattárházban, hogy egyes vásárlók milyen árucikkeket vásároltak egyszerre. A vevő a teli „bevásárlókosarával" beáll a pénztárhoz, a pénztáros pedig az összes cikket, amit a kosárban talál, egyetlen tranzakció formájában rögzíti. Így aztán, ha magáról a vevőről nem is tudunk semmit, és azt sem tudjuk megmondani, hogy vásárol-e még valaha ebben az üzletben, bizonyos cikkekről *határozottan* tudjuk, hogy ezeket valaki egyszerre vásárolta.

Ha néhány cikk gyakrabban fordul elő együtt a kosarakban, mint az egyébként várható lenne, akkor az áruház ez alapján következtetéseket vonhat le arra vonatkozólag, hogy a vásárlók milyen útvonalon járják be az üzletet. Az árucikkek elrendezhetők úgy, hogy a vevők kénytelenek legyenek bizonyos útvonalakon végigmenni, és ezek mentén vonzó cikkeket lehet elhelyezni.

11.19. példa: Egy híres példa, sokak által megfigyelt jelenség, hogy azok az emberek, akik eldobható pelenkát vásárolnak, nagy valószínűséggel sört is vesznek. Hogy miért áll fenn ez a kapcsolat a két árucikk között, arra több elmélet is született. Többek között azzal is magyarázzák, hogy a pelenkavásárló emberek, mivel kicsi gyerekük van, ritkán járnak kocsmába, ezért otthon isszák a sört. Az áruházak aztán így vagy úgy kihasználhatják azt a tényt, hogy sok vásárló a pelenkáktól a sör felé veszi az útját, vagy fordítva. Az ötletes kereskedő burgonyaszirmot tesz a sör és a pelenka közé. Az állítás az, hogy ilyenkor mindhárom cikk iránt növekszik a kereslet. □

A bevásárlókosár-adatokat tárolhatjuk a következő relációban:

```
Kosarak(kosár, árucikk)
```

Az első attribútum egy „kosárazonosító", amely egyértelműen azonosít minden bevásárlókosarat, a második attribútum pedig a kosárban található cikk azonosítója. Vegyük észre, hogy a reláció szempontjából lényegtelen, hogy az adatok valódi bevásárlókosár-adatok legyenek. Bármilyen olyan adat megfelel, ahol a cikkek közötti kapcsolatokat akarjuk vizsgálni. A „kosarak" lehetnének például dokumentumok, a „cikkek" pedig szavak, ami azt jelenti, hogy valójában olyan szavakat keresünk, amelyek sok dokumentumban együtt fordulnak elő.

A *társítási szabály* legegyszerűbb formája bevásárlókosár-adatok esetén a cikkek egy halmaza. Az $\{i_1, i_2, \ldots, i_n\}$ cikkhalmazok jelentősége változó lehet. Egy halmaz legelemibb tulajdonsága, amelyet vizsgálhatunk, hogy azon kosarak száma, amelyekben a halmaz *minden* eleme szerepel, nagy. Egy cikkhalmaz *tartója* azon kosarak száma, amelyekben a halmaz minden eleme megtalálható. *Erős tartójú cikkhalmazok* felkutatása azt jelenti, hogy adott s küszöbértékhez meg kell találnunk az összes olyan cikkhalmazt, amely tartója legalább s.

Ha az adatbázisba felvett cikkek száma nagy, akkor még abban az esetben is, ha csak kis elemszámú cikkhalmazokkal, mondjuk cikkpárokkal foglalkozunk, az összes kérdéses cikkhalmaz tartóját kiszámítani nagyon sok időt vehet igénybe. Így a kézenfekvő megoldás az erős tartójú cikkhalmazok felkutatására még cikkpárok esetén sem működik. Ilyenkor, mint ahogy erre a 11.24. ábrán látható SQL-lekérdezés utal, minden $\{i, j\}$ árucikkpár tartóját ki kell számítanunk. A lekérdezés összekapcsolja a `Kosarak` relációt saját magával, az így kapott sorokat a bennük található két árucikk szerint csoportosítja, majd a csoportok közül kihagyja azokat, amelyekben a kosarak száma (azaz a tartó) nem éri el az s küszöbértéket. Érdemes végiggondolni, hogy a `WHERE` záradékban rögzített `I.árucikk < J.árucikk` feltétel mire használható. Ennek segítségével előzhető meg, hogy ugyanaz az $\{i, j\}$ pár fordított sorrendben, azaz $\{j, i\}$ alakban is előforduljon, továbbá, hogy az $\{i, i\}$ típusú, vagyis ugyanabból az egy elemből alkotott „párok" egyáltalán létrejöjjenek.

```
SELECT I.árucikk, J.árucikk, COUNT(I.kosár)
FROM Kosarak I, Kosarak J
WHERE I.kosár = J.kosár AND
      I.árucikk < J.árucikk
GROUP BY I.árucikk, J.árucikk
HAVING COUNT(I.kosár) >= s;
```

11.24. ábra. *Naiv megoldás az erős tartójú cikkpárok felkutatására*

11.5.3. Az előzetes algoritmus

A 11.24. ábrán látható lekérdezés futási ideje optimalizálható abban az esetben, ha az s küszöbérték elég magas ahhoz, hogy csak néhány pár feleljen meg a feltételnek. Ésszerű magas s értékkel dolgoznunk, hiszen cikkpárok százaival, ezreivel úgysem mennénk semmire. Az adatbányászati lekérdezéstől azt várjuk, hogy egy kis létszámú, de a lehető legjobb cikkpárokat tartalmazó halmazra hívja fel a figyelmünket. Az *előzetes* algoritmus a következő megfigyelésen alapul:

- Ha egy X cikkhalmaz tartója s, akkor minden $Y \subseteq X$ cikkhalmaz tartója is legalább s.

Például ha az $\{i, j\}$ cikkpár 1000 kosárban szerepel, akkor világos, hogy i is és j is feltűnik ugyanezekben a kosarakban, vagyis létezik legalább 1000 kosár az i cikkhez és 1000 kosár a j cikkhez is.

A fenti szabály megfordításából adódik, hogy ha legalább s tartójú cikkpárokat ke-

A társítási szabály más formái

A társítási szabály egy általánosabb formája a cikkhalmazt egy másik cikkel hozza kapcsolatba. A szabály $\{i_1, i_2, \ldots, i_n\} \Rightarrow j$ alakban írható. Ezzel az alakkal két tulajdonság adható meg:

1. *Bizonyosság:* Annak a valószínűsége, hogy egy $\{i_1, i_2, \ldots, i_n\}$ cikkeket tartalmazó kosárban j árucikk is megtalálható, egy bizonyos küszöbérték, például 50% felett van. Például „a pelenkavásárlók legalább 50%-a sört is vásárol".
2. *Érdekesség:* Annak a valószínűsége, hogy egy $\{i_1, i_2, \ldots, i_n\}$ cikkeket tartalmazó kosárban j árucikk is megtalálható, lényegesen magasabb vagy alacsonyabb annak a valószínűségénél, hogy j egy véletlenül választott kosárban előfordul. Statisztikai szóhasználattal j pozitívan vagy negatívan korrelál az $\{i_1, i_2, \ldots, i_n\}$ halmazzal. A 11.19. példa megfigyelése valójában az volt, hogy a {pelenka} ⇒ sör társítási szabály nagyon érdekes.

Vegyük észre, hogy egy nagy bizonyosságú vagy érdekességű szabály legtöbb esetben csak akkor lesz igazán használható, ha a kérdéses árucikkeknek is erős a tartójuk. Ez azért van így, mert ha a tartó nem erős, akkor a szabály előfordulásainak a száma sem magas, ami korlátozza az adott szabályt hasznosító stratégiával járó előnyöket.

resünk, akkor eleve kizárhatók azok a cikkek, amelyek – más cikkektől függetlenül – nem fordulnak elő legalább s kosárban. Ennek megfelelően az *előzetes algoritmus* a következő lépéseket hajtja végre:

1. Keressük meg a „jó" cikkeket – vagyis azokat, amelyek megtalálhatók elegendő számú kosárban, majd
2. Futtassuk csak a jó cikkeken a 11.24. ábra lekérdezését.

Az előzetes algoritmus az egymás után végrehajtott két SQL-lekérdezés formájában a 11.25. ábrán látható. Először feltölti a `JóKosarak` relációt az elég erős tartójú cikkekkel, azaz a `Kosarak` reláció egy megfelelő részhalmazával, majd a 11.24. ábra naiv algoritmusának mintájára összekapcsolja a `JóKosarak` relációt saját magával.

11.20. példa: Hogy érezzük, mennyit segít az előzetes algoritmus, nézzük meg, hogyan működik egy 10 000 árucikkes élelmiszer-áruház esetén. Tegyük fel, hogy átlagosan 20 cikk van egy-egy kosárban, és hogy az adatbázis 1 000 000 kosár adatait tárolja (a valósághoz képest még ez az érték is kicsiny). Ekkor a `Kosarak` reláció 20 000 000 sorból áll és a naiv algoritmus összekapcsolása után 190 000 000 pár jön létre, ugyanis az 1 000 000 kosárban $\binom{20}{2} = 190$-féle cikkpár szerepelhet összesen. A csoportosítást és a számlálást tehát 190 000 000 soron kell végrehajtani.


```
INSERT INTO JóKosarak
   SELECT *
   FROM Kosarak
   WHERE árucikk IN (
      SELECT árucikk
      FROM Kosarak
      GROUP BY árucikk
      HAVING COUNT(*) >= s
   );

SELECT I.árucikk, J.árucikk, COUNT(I.kosár)
FROM JóKosarak I, JóKosarak J
WHERE I.kosár = J.kosár AND
      I.árucikk < J.árucikk
GROUP BY I.árucikk, J.árucikk
HAVING COUNT(*) >= s;
```

11.25. ábra. *Az előzetes algoritmus az erős tartójú cikkpárok felkutatását az erős tartójú cikkek keresésével kezdi*

Tegyük fel azonban, hogy s = 10 000, azaz a kosarak számának 1%-a. Lehetetlen, hogy több, mint 20 000 000/10 000 = 2000 cikk legalább 10 000 kosárban szerepeljen, hiszen a `Kosarak` relációban csak 20 000 000 sor van, és minden cikk, amely 10 000 kosárban megjelenik, a reláció 10 000 sorában is feltűnik. Vagyis a 11.25. ábra előzetes algoritmusában az erős tartójú cikkeket kereső lekérdezés nem eredményezhet 2000-nél több cikket, de valószínűleg ennél sokkal kevesebbet fog találni.

Nem tudhatjuk előre, hogy a `JóKosarak` reláció mekkora lesz, a legrosszabb esetben ugyanis a `Kosarak` relációban előforduló *összes* árucikk megjelenik a kosarak 1%-ában is. Ha azonban s elég nagy, akkor a `JóKosarak` reláció a gyakorlatban lényegesen kisebb lesz, mint a `Kosarak` reláció. Tegyük fel például, hogy a `JóKosarak` relációban egy-egy kosár átlagosan 10 árucikket tartalmaz, azaz a reláció fele akkora, mint a `Kosarak` reláció. Ekkor a második lépésben az összekapcsolás után $1\,000\,000 \times \binom{10}{2} = 45\,000\,000$ sort kapunk, azaz kevesebb, mint a negyedét annak a sormennyiségnek, amit a `Kosarak` reláció önmagához kapcsolása eredményez. Ezek alapján az várható, hogy az előzetes algoritmus körülbelül negyedannyi időt vesz igénybe, mint a naiv algoritmus. Legtöbb esetben, ahol a `JóKosarak` reláció a `Kosarak` relációnak jóval kevesebb, mint a felét tartalmazza, a futási idő csökkenése még nagyobb mértékű. Általában, ha az összekapcsolásban részt vevő sorok számát n-ed részére csökkentjük, akkor a futási idő n^2-ed részére csökken. □

11.6. Összefoglalás

- *Információk egyesítése:* Gyakran előfordul, hogy többféle adatbázis vagy más információforrás egymással összefüggő adatokat tartalmaz. Megvan a lehetőség arra, hogy ezeket a forrásokat egyesítsük. Gyakran azonban heterogén forrásokkal van dolgunk. Az inkompatibilitás sokféle formában jelentkezhet: ugyanazokhoz az értékekhez különböző típus, kód, illetve konvenció tartozhat, azonos fogalmakhoz egymástól eltérő értelmezések adhatók, az egyes sémákban pedig megvannak a fogalmi különbségek.
- *Az információegyesítés megközelítési módjai:* A korai módszerek közé sorolható a „szövetség" létrehozása, ahol az egyes adatbázisok egymás nyelvén intézhetik egymáshoz a lekérdezéseket. Ennél újabb módszer az adattárház használata, ahol az adatokat egy globális sémának megfelelően alakítjuk át, és a tárházban tároljuk a másolatokat. Ennek egy alternatívája a közvetítő rendszer, ahol egy virtuális adattárháznak tehetők fel a globális sémának megfelelő lekérdezések, amelyeket az egyes adatforrások sémáihoz igazítva alakítunk át.
- *Adatkinyerő és borítékoló:* A tárház, illetve közvetítő rendszerek mindegyike tartalmaz egy, az adatforrásokhoz rendelt alkotóelemet, az adatkinyerőt illetve a borítékolót. Fő feladatuk a lekérdezések és az eredmények átalakítása a globális, illetve a lokális séma szerint.
- *Borítékoló generátor:* A borítékoló tervezésének egy módja a borítékoló sablonok használata, amelyek leírják, hogy egy speciális formájú globális sémának megfelelő lekérdezés hogyan alakítható át a lokális séma nyelvére. A sablonok táblázatba foglalása és értelmezése egy meghajtó feladata, amely majd az adott lekérdezéshez kiválasztja a neki megfelelő sablont, ha van ilyen. A meghajtó képes lehet arra, hogy a sablonokat különféle módon variálja, és/vagy összetettebb lekérdezések esetén további feladatokat is (például az adatok szűrését) elvégezzen.
- *OLAP:* Az adattárházak egy fontos alkalmazását jelenti az a lehetőség, hogy miközben az adatforrásokon a szokásos tranzakciófeldolgozó eljárások működnek, feltehetők a tárház egészét vagy nagy részét érintő bonyolult, általában összesítő jellegű lekérdezések. Ezek a lekérdezések az on-line analitikus adatfeldolgozó vagy OLAP-lekérdezések.
- *ROLAP és MOLAP:* Ha OLAP-alkalmazás céljából építünk adattárházat, gyakran hasznos, ha az adatra egy kocka képében gondolunk, ahol a kocka dimenziói mentén az adatot más és más megvilágításban látjuk. Az olyan rendszer, amely ezt az adatmodellt támogatja, vagy relációs szempontból tekint a kockára (ROLAP- vagy relációs OLAP-rendszerek), vagy az adatkockára specializálódik (MOLAP- vagy többdimenziós OLAP-rendszerek).
- *Csillag séma:* ROLAP-megközelítés esetén minden adatelemet (például egy árucikk eladását) egy relációban, a ténytáblázatban tárolunk, a dimenziók különböző értékeinek az értelmezéséhez (például az 1234 azonosítójú cikk milyen típusú termék?) szükséges információkat pedig az egyes dimenziókhoz tartozó dimenziótáblázatokban. Ezt a típusú adatbázissémát csillag sémának nevezzük. A ténytáblázat a csillag központja, a dimenziótáblázatok pedig a csillag csúcsai.

- *A KOCKA művelet:* MOLAP-megközelítés esetén hasznosnak bizonyul, ha a ténytáblázat dimenzióinak lehetséges részhalmazai mentén végrehajtunk egy előösszesítést. Az így kibővített táblázat kicsit több helyet foglal, mint az eredeti ténytáblázat, de segítségével nagymértékben csökkenthető az OLAP-lekérdezések futási ideje.
- *Dimenzióhálók és megvalósított nézettáblák:* Néhány adatkocka-megvalósítás a KOCKA műveletnél is hatékonyabb eszközt használ: minden dimenzióhoz létrehoz egy-egy hálót, amely az adott dimenzió – összesítésekhez használható – különböző finomsági fokait tartalmazza (például különböző időegységeket, mint nap, hónap, év). Az adattárházba aztán bizonyos megvalósított nézettáblák is bekerülnek, amelyek különféle módokon más és más dimenziók mentén összesítik az adatokat. Egy lekérdezés megválaszolására aztán a vele leginkább megegyező nézettábla használható.
- *Adatbányászat:* A tárházakon olyan tág értelmű lekérdezések is végrehajthatók, amelyek nem csak az előre meghatározott összesítéseket végzik el (miként az OLAP-lekérdezések), hanem keresik a „megfelelő" összesítést. Az adatbányászat gyakran előforduló típusai: az adatok hasonló csoportokba osztása; döntési fa tervezése, amely egy adott attribútum értékét jósolja meg a többi attribútum értéke alapján; sokféle érték között gyakran előforduló párokra vonatkozó társítási szabályok keresése.
- *Előzetes algoritmus:* Az előzetes algoritmus használatával hatékonyan tudunk társítási szabályok után kutatni. Ez a technika azt a tényt használja ki, hogy ha egy halmaz gyakran előfordul, akkor annak minden részhalmaza is gyakori.

11.7. Irodalomjegyzék

Az adattárház-rendszerekről és a kapcsolódó technológiákról ad áttekintést [10], [4] és [8]. Az adatbázis-szövetségekről [12]-ben van szó. A közvetítő fogalma [14]-ből származik. A közvetítő és a borítékoló megvalósításával, különös tekintettel a borítékoló generátor alkalmazására, [6] foglalkozik.

Manapság a legtöbb információegyesítő módszer a „felstrukturált" adatmodellen alapszik, amelynek segítségével megoldható a hiányzó értékek problémája, és a sémák közti egyéb különbségek is áthidalhatók. Az adatmodell ötlete [11]-ből származik; [1] és [13] áttekintést nyújt a témával kapcsolatban.

A KOCKA műveletet [7] vetette fel. A megvalósított nézettáblák segítségével megvalósított adatkockákról [9]-ben esik szó.

Adatbányászati technikák áttekintését adja [5]. Az előzetes algoritmussal [2] és [3] foglalkozik.

1. S. Abiteboul, „Querying semi-structured data," *Proc. Intl. Conf. on Database Theory* (1997), Lecture Notes in Computer Science 1187 (F. Afrati and P. Kolaitis, eds.), Springer-Verlag, Berlin, pp. 1–18.

2. R. Agrawal, T. Imielinski, and A. Swami, „Mining association rules between sets of items in large databases," *Proc. ACM SIGMOD Intl. Conf on Management of Data* (1993), pp. 207–216.
3. R. Agrawal, and R. Srikant, „Fast algorithms for mining association rule," *Proc. Intl. Conf. on Very Large Databases* (1994), pp. 487–499.
4. S. Chaudhuri and U. Dayal, „An overview of data warehousing and OLAP technology," *SIGMOD Record* **26**:1 (1997), pp. 65–74.
5. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining,* AAAI Press, Menlo Park CA, 1996.
6. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, V. Vassalos, J. D. Ullman, and J. Widom, The TSIMMIS approach to mediation: data models and languages, *J. Intelligent Information Systems* **8**:2 (1997), pp. 117–132.
7. J. N. Gray, A. Bosworth, A. Layman, and H. Pirahesh, „Data cube: a relational aggregation operator generalizing group-by, cross-tab, and subtotals," *Proc. Intl. Conf. on Data Engineering* (1996), pp. 152–159.
8. A. Gupta and I. S. Mumick, *Materialized Views: Techniques, Implementations, and Applications,* MIT Press, Cambridge MA (1999).
9. V. Harinarayan, A. Rajaraman, and J. D. Ullman, „Implementing data cubes efficiently," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1996), pp. 205–216.
10. D. Lomet and J. Widom (eds.), Special issue on materialized views and data warehouses, *IEEE Data Engineering Bulletin* **18**:2 (1995).
11. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, „Object exchange across heterogeneous information sources." *Proc. Intl. Conf. on Data Engineering* (1995), pp. 251–260.
12. A. P. Sheth and J. A. Larson, „Federated databases for managing distributed, heterogeneous, and autonomous databases," *Computing Surveys* **22**:3 (1990), pp. 183-236.
13. D. Suciu (ed.), Special issue on management of semistructured data, *SIGMOD Record* **26**:4 (1997).
14. G. Wiederhold, „Mediators in the architecture of future information systems," *IEEE Computer* **C-25**:1 (1992), pp. 38–49.

Index