Eötvös Lóránd Tudományegyetem
Informatikai Kar
Információs Rendszerek Tanszék

Osztott dokumentumok tranzakció-kezelése XML környezetben

Dr. Benczúr András
egyetemi tanár

Ádám József
nappali tagozat
programtervező matematikus

Budapest, 2009.

Témabejelentő helye

Tartalomjegyzék

1. Bevezetés

Az elmúlt évtized során fokozatosan egyre jelentősebb hangsúlyt kaptak a félig-struktúrált adatok az információcsere és tárolás kapcsán. A létrejött nagy mennyiségű különböző méretű, bonyolultságú, más-más célokat szolgáló, félig-struktúrált adat kezelésének problémája mostanra elég sürgetővé vált ahhoz, hogy az évek során lassan gyűlő tapasztalatokból (ugyan továbbra is elég lassan, de biztosan) gyakorlati megvalósítások is szülessenek. A kialakulóban lévő megoldások felvetettek egy fontos problémát a jelenlegi (objektum) relációs adatbázis kezelő rendszerekkel (O)RDBMS, illetve azok különböző kiegészítéseivel kapcsolatban, melyek nem az új struktúrát veszik-vették figyelembe, csupán a már meglévő megoldások felhasználásával biztosítottak korlátozott hozzáférést a tárolt dokumentumokhoz. A legfőbb problémát a kereskedelmi forgalomban használt (O)RDBMS-ek relációs volta jelenti, ezen belül is a relációs adatmodell kezeléséhez kialakult tranzakció-kezelési módszerek megvalósításai, melyek nem biztosítanak megfelelő finomságú megoldásokat a nagymértékű konkurens használathoz. A különböző RDBMS-ek általában legfeljebb a dokumentum szintű hozzáférést kezelik (megfelelően), a kialakult megoldások közül néhányról pár szó: a teljes dokumentumot gyakran egy CLOB mezőben tárolják, esélyt sem hagyva a dokumentum részeinek konkurens használatára (módosítására), ezen eset finomítása lehet a CLOB-on belüli tartomány zárolás (range locking) - ami természetesen teljes egészében figyelmen kívül hagyja az XML dokumentum szerkezetét - vagy a dokumentum alkotóelemeinek szétosztása valamilyen algoritmus alapján egy vagy több táblába (shredding),ami szintén nem a kívánt hatást eredményezi, hiszen a zárolási mechanizmusok a különböző problémák elkerülése végett, mint például a fantom probléma, a teljes táblákat zárolják, ezáltal akár több egyazon táblában tárolt dokumentum használatát is ellehetetlenítve. A fent említett - az (O)RDBMS-eket jellemző - hiányosságok és hátrányok nagyban segítették, hogy a témakörrel foglakozó szakemberek egy új - a relációs rendszereket leváltani nem hivatott - koncepcióval is sokat foglakozzanak, ez pedig a natív XML adatbázisok építése. Egy alapjaiban az XML struktúrájának kezelésére kitalált - és néhány esetben meg is valósított - adatbáziskezelő rendszer vagy legalább egy új tranzakciókezelő réteg, amely jobban fel van készítve az XML adatok lekérdezéseinek és módosítási kéréseinek kezelésére, új síkra helyezi ezt a tranzakció-kezelési problémát, a zárolások a dokumentum szintről csomópont szintre süllyednek bizonyos esetekben, illetve bizonyos implementációk változtatható vagy automatikusan változó zárolási mélységeket tesznek lehetővé - ezek hatékonysága és alkalmazhatósága természetesen górcső alá kerül -, az XML alapvető szerkezeti sajátosságaihoz több esetben a tárolási szinten elkezdődik az alkalmazkodás és van példa az XML DTD-je által nyújtott többletinformáció kihasználására is az ütköző zárolási kérések vizs-

gálatakor. Ennek folyományaként a továbbiakban a natív XML adatbázisok kutatásaiból született eredmények és a kimondottan az XML adatok kezeléséhez készített protokollok kerülnek előtérbe a dolgozatban, a relációs modellhez készült megvalósítások legfeljebb összehasonlítási alapul szolgáló példákként jelennek meg - ha azt például valamelyik forrás is említi -, külön az (O)RDBMS-ek XML-es felületével nem foglalkozunk, kivételt a tetszőleges rendszerhez felhasználható implementációk jelenthetnek, de ezeket sem vizsgáljuk a relációs adatokat is érintő kapcsolat hatékonysága szempontjából. Természetesen a natív XML adatbázis kezelő rendszerek közül sem mindegyik célozta meg a feladatok teljes spektrumának teljesítését, így ezek közt is léteznek olyanok, amelyek csak a hatékony keresés problémájával foglalkoznak. A teljesség igényének fényében, még egy fontos dolgot nem teljesítenek általában ezek a rendszerek illetve protokollok, mégpedig a különböző XML nyelvi modellek közül minél többnek a támogatását, azaz sok esetben csak egy nyelv vagy egy rész kerül lefedésre a támogatott nyelv biztosította lehetőségek közül.

A teljes képhez hozzátartozik, hogy az itt bemutatott protokollok, megvalósítások, új adattárolási elképzelések, indexelési technikák főként kutatási projektek eredményei és nem mindig foglalkoznak a problémakörrel minden részletében, ezáltal nem is hasonlíthatóak össze minden kívánt szempont alapján.

2. Kapcsolódó munkák

A konkurencia-kezelés a hagyományos relációs adatbázisok kapcsán nagy múlttal rendelkezik, a szakirodalomban [21] fellelhető gyakorlatilag az összes olyan protokoll, amely biztosítani tudja a tranzakciók ütemezését egészen a sorbarendezhetőség fogalmának megfelelő szintig. Ezen protokollok, illetve ezek kisebb-nagyobb módosításával, kombinálásával született megoldások alkotják a manapság használatos rendszerek konkurencia-kezelési megoldásait is. Mivel az XML adatok kezelésére is kialakultak olyan protokollok, amelyek nagyon erősen támaszkodnak a két-fázisú lock (2PL) protokollra, az időbélyegzős (timestamp-ordering) protokollokra, a többverziós (multiversion) konkurencia-kezelési protokollokra illetve a témakörhöz legközelebb álló, fa- vagy hierarchikus protokollokra (multi-granularity locking protocols), az új protokollok ismertetése előtt pár szóban kitérünk ezen alapkövek főbb tulajdonságaira is, hiszen ha nem is mindegyik ismertetésre kerülő újabb megoldás támaszkodik ugyanolyan mértékben a régebben kialakult konkurencia-kezelési megoldásokra, de mind magán hordozza a fentiek jegyeit.

Az alapokhoz legközelebb talán a Helmer, Kanne, Moerkotte [13, 16, 15, 14]hármas nevével fémjelzett kétfázisú (Node2PL, NO2PL, OO2PL) és időbélyegzős (XTO, XCO) protokollokon alapuló megoldások állnak, amelyek különböző finomságú zárolást és különböző megközelítéseket - csúcsok helyett csak mutatók zárolása, implicit többverziós működés az XTO és XCO esetén - alkalmaznak a kitűzött feladat eléréséhez. Az általuk ismertetett megoldások a DOM API által biztosított műveletekhez vannak igazítva és gyakran nagymennyiségű zárolást alkalmaznak, ezek nem főbenjáró bűnök és a többi ismertetett protokoll is általában rendelkezni fog valamilyen „hátrányos” tulajdonsággal, pusztán a lehetséges negatív hatások vázolása céljából kerülnek említésre. Szóval a DOM API és az intenzív lock használat egyrészt a felhasználhatóságot befolyásolhatja másrészt komoly teljesítménybeli korlátokat jelenthet nem megfelelő odafigyelés mellett.

Megpróbálva egy többé-kevésbé kronologikus sorrendet követni, a következő lépcsőfok két egymástól távol álló megközelítés megjelenésének ismertetése lenne. Mégpedig, az egyik oldalon kettő - Dekeyser és Hiddens nevéhez fűződő [3, 4]- ösvény zárolási (path locking) technikán alapuló - név szerint Path Lock Satisfiability és Path Lock Propagation - protokoll, a másik oldalon pedig egy Grabs, Böhm és Schek nevéhez fűződő[7] és a későbbiekben több protokoll alapjául szolgáló Dataguide[6] alapú protokoll (DGLOCK) szolgáltatja a vizsgálódás tárgyát. A két ösvény alapú protokoll közti legfőbb különbség a kiosztott zárak mennyisége és ezzel párhuzamosan az ütközések ellenőrzésére fordítandó idő alakulása a vizsgálandó ösvénykifejezések bonyolultságának függvényében. Hátulütőik közül, az IDREF-ek jelentette ugrások kezelésének kiforratlanságát és a lekérdezőnyelv (XPATH) szélesebb körű tá-

mogatásának kidolgozását maguk a szerzők is az elvégzendő feladatok közt említik. A másik oldal Dataguide alapú protokolljának legfőbb erénye, hogy a zárakat (szerkezeti és tartalmit egyaránt) nem közvetlenül a dokumentumra, hanem a háttérben karbantartott Dataguide-ra rakja és egy predikátum alapú zárolással figyeli a tartalmi ütközéseket. Ezekkel körülbelül egyidejűleg juthatott el még egy ösvény zárolásos protokoll arra a szintre, hogy megosztották a nagyközönséggel, ez Hye Choi és Kanai nevéhez fűződik [2] és szintén egy erősen különböző megközelítéssel bír. Itt a dokumentumoknak több verziója létezik, egy eredeti (D_{st}), egy amin minden módosítás látszik (D_{all}) és minden tranzakció számára egy-egy (D_i), amin a saját módosításait végzi és amelyet olvas az eredeti dokumentum helyett. Az ütközések megállapítása az XPath kifejezések kiértékelésekor - a precíziós zároláshoz hasonlóan - a különböző dokumentum verziók összehasonlításával történik.

A következő hullám első képviselője az MPX [20] kínából származik és Yuan Wang, Gang Chen és Jin-xiang Dong nevéhez fűződik. Amit kínál nekünk, az a sorbarendezhető ütemezések, a lépcsőzetes megszakítások elkerülése - amit az időbélyeg alapú protokollok nem tudnak garantálni - és többverziós voltából adódóan az író-olvasó és a csak-olvasó tranzakciók közti ütközések elkerülése. A Dataguide alapú protokollok is előrukkoltak aztán két újabb megoldással, az egyik a Pleshachkov, Chardin és Kuznetcov nevéhez fűződő XDGL [18], amely egy sorbarendezhetőséget biztosító XPath-alapú protokoll, ami a Dataguide-on elhelyezett hierarchikus és csúcs szintű zárakat ötvözve, valamint a DTD-t kihasználva kínál megoldást az XML adatok kezelésére. A másik megoldás [19] szintén ezen szerzőhármas két tagjának Pleshachkovnak és Kuznetcovnak a nevéhez fűződik. Az SXDGL - habár nem az XPath-t hanem az XQuery-t veszi alapul műveleteihez - az XDGL tulajdonságait mind magán hordozza, kiegészülve azon, a pillanatképek segítségével biztosított tulajdonsággal, hogy a csak-olvasó és az író-olvasó tranzakciók ütközéseit kiküszöböli, illetve néhány újabb zárolási móddal a módosító tranzakciók ütközéseit tovább csökkenti. Az utolsó bemutatásra kerülő csoport a Haustein és Härder nevéhez kapcsolódó taDOM protokoll család lesz [9, 11, 1, 12]. Ezeket a protokollokat kifejezetten a DOM API-hoz készítették és egy a DOM struktúrát kibővítő modellen (taDOM) alapulnak.

A taDOM protokollokkal szoros kapcsolatban áll - hiszen fejlesztőik megegyeznek - a szintén bemutatásra kerülő XTC(XML Transaction Coordinator) szerver [8], amit a szerzők az elkészített protokollok tesztelésére is használnak. Az XTC helyzetéből fakadóan korszerű megoldásokat [10] vonultat fel az adatbázis-kezelő minden rétegében, ezért részletesebben is tárgyaljuk majd megoldásait. A másik, ismertetett protokollokhoz kapcsolódó natív XML adatbáziskezelő az XDGL és SXDGL háza táján fejlesztett Sedna, ennek tárolási megoldása szóba került már az SXDGL ismertetésekor is, részletesen viszont csak ebben a szakaszban tárgyaljuk.

A kialakult megoldások ismertetése után - a megoldásokban alkalmazott technikák kínálta lehetőségek alapján - röviden néhány felhasználási lehetőség kerül vázolásra, törekedve az ismertetett megoldások minimális ráfordítással történő (újra)felhasználására.

3. A kialakult megoldások ismertetése

Az előző részben röviden összefoglaltuk a kialakult technikákat és kiemeltük néhány fontosabb tulajdonságukat. Ebben a részben a fent említett konkurencia-kezelési megoldások egy kicsit részletesebb bemutatásával foglalkozunk. Ebben részben nem foglakozunk a számszerű teljesítményadatokkal, amelyik esetben rendelkezésre áll valamilyen összehasonlítási alapul szolgáló eredmény azt a későbbiekben egy helyre összegyűjtve megmutatjuk. Továbbá nem szándékozunk a protokollok minden tulajdonságának formális bizonyítását az alábbiakban részletezni, ezek megtalálhatóak a hivatkozások közt felsorolt cikkek, könyv-fejezetek megfelelő részeiben.

3.1. A klasszikus megoldások áttekintése

Ebben a részben egy rövid visszatekintést teszünk az alapokat jelentő konkurencia-kezelési megoldásokra. Azok alapelveire, a hozzájuk kötődő alapfogalmakra, az általuk kínált lehetőségekre, amelyek a későbbiekben az újabb protokollokban sok helyütt megvalósulni látszhatnak.

3.1.1. Kétfázisú lock-protokoll

Egy zárolási protokoll kétfázisú, ha minden tranzakciónál megkülönböztethető egy fázis, amiben a zárakat megszerzi (growing phase) és egy ettől szigorúan elkülönülő fázis, amelyben a zárakat elengedi (shrinking phase). A kétfázisú lock-protokollok konfliktus sorbarendezhető ütemezéseket generálnak.

A protokollnak több változata is létezik, ezek közül kettőt kiemelnék, mivel olyan tulajdonságokkal rendelkeznek, amelyek a későbbiekben is elő fognak kerülni, mégpedig a szigorú (S2PL) és az erős (SS2PL) kétfázisú lock protokollok, amelyek annyiban különböznek az eredeti koncepciótól „mindössze", hogy a szigorú verzióban a tranzakció az általa megszerzett írási(exkluzív) zárakat a tranzakció végéig megtartja, illetve az erős variánsban mind a megszerzett írási mind az olvasási zárakat a tranzakció befejeződéséig megtartja.

3.1.2. Fa-protokollok (csak-író, író-olvasó)

Ezek a protokollok általános voltuk ellenére, csak bizonyos különleges körülmények közt rendelkeznek jó teljesítmény mutatókkal, mégpedig fa-szerű hozzáférési mintákkal - gyakorlatilag felülről-lefelé bejárások - rendelkező tranzakciók esetén. Ugyan ezek a protokollok nem kétfázisúak, de ha bizonyos szabályok teljesülnek a tranzakciókra, akkor azok helyettesítik a kétfázisú tulajdonságot.

A csak-író fa-protokoll (Write-Only Tree Locking) esetén a kiegészítő szabályok - a zárolás jól formáltságát jelentő szabályokon felül - a következők:

1. A gyökérelemtől különböző csúcson, csak akkor kaphat írási zárat egy tranzakció, ha a szülőcsúcson is van már egy zára.

2. Egy tranzakció, ha már feladta a zárát egy tetszőleges adatelemen, akkor nem kérhet rá új zárat.

A WTL általánosítása, az író/olvasó fa-protokoll (Read/write tree locking), amelyben különbséget teszünk írási és olvasási zárak közt. Ebben a protokollban egy újabb feltétel/szabály szükséges a konfliktus-sorolhatóság biztosításához, amit a tranzakciók által olvasott elemek halmazának komponensekre bontásával és ezen komponensek vagy más néven csapdák (pitfall) figyelésével lehet biztosítani. A szükséges szabály:

1. A tranzakció a csapda elemeinek zárolásait tekintve feleljen meg a kétfázisú tulajdonságnak.

Az RWTL protokoll általánosításával - és a WTL első szabályának módosításával - létrehozható a DAG locking protkoll.

3.1.3. Időbélyegzős protokollok

A záraktól történő megszabadulás egyik megközelítése az időbélyegek használata. Alapvetően arról van itt szó, hogy a tranzakció-kezelő minden tranzakcióhoz hozzárendel egy értéket - egy teljesen rendezett tartományból - például a tranzakció megkezdésekor az óra alapján vagy egy folyamatosan növelt számláló alapján. A tranzakció időbélyegét öröklik a tranzakció műveletei is. Az időbélyegeken alapuló ütemezők az ütköző műveleteket az időbélyegek alapján fogják rendezni, innen az időbélyegzős (**T**imestamp **O**rdering) protokoll elnevezés. Az időbélyegzős protokollok alapszabálya:

- Két ütköző művelet esetén annak kell először végrehajtódni, amelynek kisebb az időbélyege.

3.1.4. Sorolhatósági gráf tesztelés

Ez a protokoll-osztály a konfliktus-sorolhatóságot a konfliktus gráfban keletkező körök hiánya alapján hivatott biztosítani. A sorolhatósági gráf tesztelő protokoll (**S**erialization **G**raph **T**ester) egy ilyen konfliktus gráfot tart karban, amelyben a csomópontokat és éleket az ütemezőtől érkező műveletek alapján dinamikusan hozza létre és törli. A konfliktus-sorolhatósági tulajdonságot a gráf körmentesen tartásával lehet biztosítani. Az SGT egyetlen szépséghibája, hogy habár elméleti szempontból igen vonzó - generálja az összes konfliktus-sorolható ütemezést -, de a gráf helyigénye és az ellenőrzések okozta többletterhelés jó eséllyel elfogadhatatlan mértékű egy élő rendszeren.

3.1.5. Optimista protokollok

Az előzőekben vázolt protokollok abból a feltételezésből indultak ki, hogy az ütközések gyakori események és hogy ezeket az ütemezőnek, akkor és ott nyomban kezelnie kell és eldönteni, hogy az adott művelet végrehajtható, blokkolódik vagy vissza lesz utasítva. Az optimista protokollok épp az ellenkezőjét feltételezik, hogy az ütközések ritka események. Ami bizonyos alkalmazások esetén természetesen igaz is és ezen esetekben kár a rendszer erőforrásait - és idejét - a zárak kezelésére pazarolni.

Abból a feltételezésből kiindulva, hogy az ütközések ritkák, az ütemező az érkező műveleteket egyszerűen tovább engedi és időről-időre ellenőrzi, hogy az addig előállított ütemezés még mindig sorbarendezhető-e. Ez gyakorlatilag azt jelenti, hogy az ütemezők időnként validálják a kimenetüket, ezért nevezik még őket validáló protokolloknak (validating protocol esetleg certifier) is. Ezen protokolloknál a kritikus rész a committáló tranzakciók változtatásainak adatbázisba írása lehet, erre az implementáláskor fontos odafigyelni. Megjegyzem, már itt megjelenik a később terítékre kerülő több verzió karbantartása, ugyanis az egyes tranzakciók a saját munkaterületükön hozzák létre (olvasási fázis) azokat a verziókat, amelyek rögzítése bonyodalmakat okozhat (validáció és írási fázis). A bonyodalmak elkerüléséhez a kulcs, a tranzakciók validálásának visszavezetése a konfliktus gráf körmentesen tartására.

Két megközelítése a megoldásnak:

- A tranzakció validáláskor, a már committált tranzakciókkal szembeni konfliktusokat ellenőrzi.(**B**ackward-oriented **O**ptimistic **C**oncurrency **C**ontrol)
- Validáláskor a párhuzamosan futó, olvasási fázisban lévő tranzakciókkal történő ütközéseket vizsgáljuk.(**F**orward-oriented **O**ptimistic **C**oncurrency **C**ontrol)

Mindkét protokoll konfliktus-sorolható ütemezéseket generál, sőt az FOCC-ről belátható, hogy commit sorrend tartó ütemezéseket állít elő.

3.1.6. Hibrid protokollok

A hibrid protokollok alapja a konkurencia-kezelés problémájának két részre bontása:

1. *rw* (és *wr*) szinkronizáció: az olvasás műveleteket szinkronizáljuk az írás műveletekkel
2. *ww* szinkronizáció: az írás műveleteket szinkronizáljuk az írás műveletekkel, de az olvasás műveletekkel nem

Amennyiben ezeket a szinkronizációs feladatokat szétválasztjuk, az ütemezőt is szétválaszthatjuk és a feladatokat elláthatja két különálló komponens. A komponensek integrálásával kapott ütemezőket nevezzük *hibrid* ütemezőknek. A konfliktus fogalma is módosításra szorul ezen ütemezők esetén:

- Az *rw* (és *wr*) szinkronizáció esetén: két művelet akkor ütközik, ha egyazon adatelemet az egyik olvassa, a másik pedig írja
- A *ww* szinkronizáció esetén: két művelet ütközik, ha mindkettő írni próbálja ugyanazt az adatelemet

A hibrid ütemezők helyességének belátásához egy járható út a két komponens konfliktusgráfjának uniójából kapott gráf körmentességének fenntartása. Ehhez további kiegészítő lépésekre lehet szükség - újabb kör ellenőrzések, saját munkaterület a tranzakcióknak - és ezek a lépések akár jelentős többletterhelést is jelenthetnek, semlegesítve egy ilyen protokoll esetleges pozitív tulajdonságait. Például egy optimista és egy pesszimista protokoll kombinációja jól hangzik elméletben, hiszen mindkettő a maga hozzáférési mintáira kiemelkedően teljesít, de ha a kettő közti többletellenőrzések felemésztik a kapott protokoll részeinek tulajdonságaiból fakadó előnyöket, akkor javallott inkább egy egyszerűbben implementálható protokoll használata az adott környezetben.

3.1.7. Többverziós protokollok

Az eddig látott protokollokhoz és azok koncepciójához képest van itt egy fontos változás, mégpedig az adatelemek hozzáférhető változatainak számában. Míg az eddig vizsgált esetekben a műveletek egyazon elemet olvastak, írtak, felülírtak, a protokollok az ezekből a műveletekből adódó konfliktusokat vizsgálták, orvosolták, előzték meg, a többverziós konkurencia-kezelés alapvetően a több - a felhasználó számára általában láthatatlan - verzió létezéséből kovácsol előnyöket.

Az eddig ismertetett protokollok közül többnek létezik a gyakorlatban is használt többverziós változata (MVTO, MV2PL, MVSGT), ezeket akár hibrid protokollok komponenseiként is eredményesen használhatjuk. Általánosságként elmondható a több verziót kezelő megoldásokról, hogy bár a verziók kezelése odafigyelést igényel - főleg implementációs szinten - és sokszor meglehetősen nehéz feladatot jelent, de a megfelelő koncepciók megválasztásával jelentős nyereség könyvelhető el a párhuzamos hozzáférések terén, ami hatványozottan igaz a csak olvasó esetekre. A csak olvasó eset elméleti megvalósításának alappillére egy S2PL (módosító tranzakciókhoz), MVTO (olvasó tranzakciókhoz) hibrid protokoll (ROMV - Read-Only Multiversion Protocol), amely a konkurencia-kezelést végző komponenst terhelő, csak olvasó tranzakciók (nagyon kicsi arányú író tranzakció mellett) jelentette zárolási problémákat kiemelkedő mértékben tudja orvosolni, kihasználva a több verzió jelentette előnyöket.
A többverziós konkurencia-kezelésben egy fontos szempont - a fogalmi szinten alapértelmezetten nem hangsúlyozott - elérhető verziók száma. Az egyszerűség kedvéért alapvetően azt feltételezzük, hogy az elemek minden valaha keletkezett verzióját megtartjuk.

Ennek természetesen a valóságban nem tudunk eleget tenni, hiszen a szerverek rendelkezésére álló tárterületnek megvannak a maga korlátai. Ennek folyományaként vizsgálták azon eseteket, amikor csak k-verziót tartunk meg a keletkezett verziókból. A vizsgálatok azt mutatják, hogy tetszőleges $k > 0$-ra a több verziót kezelő ütemezők, a sorbarendezhető történetek egyre bővebb (szigorúan bővebb) halmazát állítják elő. A többverziós protokollok működéséről bővebben az új protokollok ismertetésénél fogunk még beszélni (pl.: MPX).

3.1.8. Hierarchikus protokollok

Más néven, többes finomságú zárolások (Multiple Granularity Locking). A zárak finomságának beállítása, a zárolásokat kezelő komponens által okozható nem elhanyagolható többletterhelés kapcsán egy igazán figyelemreméltó megoldás. A különböző, zárolásoknál alkalmazott struktúrák okozta terhelés ugyan általában csekély, de ha a megfelelő nagyságrendű terhelést vesszük figyelembe, akkor a szükségtelenül nagy finomságú zárolás-kezelés a zárakat nyilvántartó struktúrák számára szükséges helyigény megnövekedésével és ennek következményeképp válaszidő növekedéssel járhat. A zárak túl „durvára" hagyása ellenben a konkurens hozzáférések mértékének csökkenéséhez vezet. A jó megoldás többes finomságú zárolások támogatása, azaz a rekord vagy lapszintű és a tábla vagy akár táblatér szintű zárolási lehetőségek biztosítása - esetleg ezek közti dinamikus választási lehetőség vagy konverzió biztosítása a zár-kezelőnek. A különböző (finomságú) zárak konfliktusainak megállapításához kidolgoztak egy szándék zárakon (intention lock) alapuló megoldást, miszerint azok a tranzakciók, melyek nagyobb finomságú zárolást akarnak használni, a hierarchiában az összes náluknál durvább finomságú szinten ilyen szándék zárakat kell elhelyezzenek. Ez az általános módszer alkalmazható az adatbázis-kezelő rendszer alapvető struktúrájától függetlenül, a kutatások eleddig ugyan a relációs adatbázisokra fókuszáltak, de a félig-struktúrált modellel rendelkező XML-dokumentumok és a hozzájuk kialakított natív XML adatbázis-kezelők lesznek azok, amik igazán profitálnak ebből a hierarchikus szemléletmódból. Említésre került a fentiekben a zárolási finomság kapcsán a dinamikus választás lehetősége, ez ugyebár tipikusan futásidőben előálló probléma, amikor is egy tranzakció túl sok nagy finomságú zárat kap és ezáltal számottevő többletterhelést okoz a zár-kezelőnek - például ha elfogyasztja a zár-kezelő rendelkezésére álló memória jelentős részét vagy egy megadott küszöbértéket átlép a tranzakció zárainak száma. Ilyen esetekben jelenthet megoldást, ha a tranzakció meglévő zárait egy vagy több durvább zárra cseréljük. Ezt a konverziót nevezzük zár kiterjesztésnek (lock escalation).

3.1.9. Predikátum-orientált konkurencia-kezelés

A szemantikai ismeretek kihasználásának lehetőségeit vizsgálva különböző módszereket láttak napvilágot az évek során a relációs modell vonatkozásában. Ezen módszerek közül egy a predikátum-orientált konkurencia-kezelés - akár az előzőekben vázolt többes finomságú zárolást is lehet predikátum alapon végezni-, amely a lekérdező/módosító nyelv által meghatározott zárolási egységeken alapul. Például, egy SELECT kifejezés WHERE feltétele meghatározhat egy zárolási egységet.

A predikátum-zárolás nyújtotta hozzáférési lehetőségek azonban nemcsak nagyobb konkurenciát, hanem újabb problémaforrást is jelentenek. Azok a relációk (vagy másik struktúránál például csúcsok), amelyeket nem teljesen kizárásos alapon zárolnak és rajtuk módosító és/vagy lekérdező műveleteket hajtanak végre, egy olyan jelenséggel hajlamosak előállni, amit a szakirodalom fantom probléma (phantom problem) névvel illet. Az előálló jelenség ugyanis pontosan az, amit egy fantom is csinál, azaz hol látszik, hol nem. A fantomot észlelők nem mások, mint azok a tranzakciók, amelyek konkurensen műveleteket végeznek egy adott reláción/dokumentumon, a fantomok pedig az azonos célterületre (reláció,csúcs) beszúrt, ott módosított illetve onnan törölt rekordok (csúcsok, attribútumok). A probléma megoldását a kiadott zárak ütközéseinek ellenőrzései helyett, a predikátumok konfliktusainak ellenőrzései jelenthetik - a kiadott predikátum zárak akár szándék záraknak is tekinthetőek, hiszen az aktuális tartalomtól függetlenek.

Annak ellenére, hogy a predikátum-zárolás fogalmi szinten egy elegáns megoldásnak tűnik, a gyakorlatban a predikátum alapú zárak ütközéseinek ellenőrzését és a tranzakciók várakozási sorának karbantartását nem egyszerű megvalósítani, főleg nem a konkrét elemekre kiadott zárak kezeléséhez képest, így a gyakorlatban eddig nem nagyon találni erre épülő megoldásokat.

A predikátum zárolás egyik változata a precíziós zárolás (precision locking). Ebben a változatban az ütemező a zárolási kéréseket ellenőrzés nélkül engedélyezi és csak később, a konkrét olvasási/írási művelet végrehajtása előtt hajtja végre az ellenőrzést az adott művelet célelemének konkrét adatai és a predikátumok közt, tehát nem a művelet predikátumára, hanem az általa meghatározott halmaz minden elemére fog ellenőrzést végezni.

3.2. Az XML dokumentumok kezeléséhez kialakított új protokollok

3.2.1. Kétfázisú lock protokoll variánsok

A most ismertetésre kerülő 3 protokoll Helmer, Kanne és Moerkotte nevéhez fűződik [13, 14], alapjuk a (szigorú) kétfázisú lock protokoll és mindhárom egy kicsit más

observer	structure	firstChild lastChild previousSibling nextSibling getNodeById getElementByTagName
	contents	getTextContents nodeName getAttribute

mutator	structure	insertBefore replaceChild removeChild appendChild
	contents	appendData deleteData insertData replaceData setAttribute

1. ábra. DOM műveletek

megközelítést alkalmaz az XML dokumentumok struktúráján történő zárolások kivitelezéséhez.

Az XML dokumentumokon végezhető műveleteket itt négy kategóriába soroljuk, a tipikus API-k (mint a DOM) által biztosított műveletek alapján: vannak a csúcsok tartalmát lekérdező és módosító műveletek és a dokumentum struktúráját vizsgáló/bejáró és módosító műveletek (példák az 1. ábrán). A protokollok alapját a struktúrát módosító és bejáró műveletekre kidolgozott megoldások adják, amiket egy következő lépésben bővítünk a tartalmi módosítások kezeléséhez.

A tárgyalás leegyszerűsítése végett csak néhány egyszerű műveleten keresztül vizsgáljuk a problémakört. Kiindulásként feltesszük, hogy minden tranzakció első lépése a használni kívánt dokumentum kiválasztása **sd** (select document), ami a gyökérelemre mutató referenciát ad vissza. Ezután a következő műveleteket használja a struktúra bejárására és módosítására:

nthP az n-edik gyerek a gyerek lista elejétől számítva

nthM az n-edik gyerek a gyerek-lista végétől számítva

insA beszúrás egy adott csúcs mögé

insB beszúrás egy adott csúcs elé

del adott csúcs törlése

A különböző típusú (attribútum, elem) csúcsok közt nem tesz különbséget.

A szokásos S (shared) és X (exclusive) zárakhoz hasonlóan, melyeket a tranzakcióknak az olvasási és írási műveleteiket megelőzően kell megszerezniük, a dokumentum bejárásához két új zárolási módot vezetünk be, az osztott T (traversal/bejárás) és az exkluzív M (modification/módosítás) zárakat. Az új zárak kompatibilitási mátrixa a 2(a) ábrán látható.

A most következő protokollok mind csúcs szinten végzik a zárolásokat. A zárak elnevezései és a protokollok különbségei jól illusztrálhatók az alábbi 3. ábrával, ami az XML dokumentumot mutatókkal összekötött csúcsokként ábrázolja.

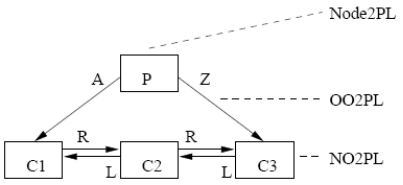
	T	M
T	+	-
M	-	-

(a)

	TL	TR	TA	TZ	ML	MR	MA	MZ
TL	+	+	+	+	-	+	+	+
TR	+	+	+	+	+	-	+	+
TA	+	+	+	+	+	+	-	+
TZ	+	+	+	+	+	+	+	-
ML	-	+	+	+	-	+	+	+
MR	+	-	+	+	+	-	+	+
MA	+	+	-	+	+	+	-	+
MZ	+	+	+	-	+	+	+	-

(b)

2. ábra. Kompatibilitási mátrixok



3. ábra. XML dokumentum reprezentáció mutatókkal

Az **Node2PL** névre keresztelt protokoll a szülő csúcsokon helyezi el a zárakat, azaz ha egy csúcs valamely gyerekét bejárjuk, akkor a szülőcsúcsra kerül egy T zár és ha egy új gyereket szúrunk be a csúcs alá, akkor a csúcsra kerül egy M zár.

Az **NO2PL** protokoll azon csúcsokat zárolja, amelyek mutatói bejárásra vagy módosításra kerülnek. Tehát egy gyerek beszúrása esetén (3.ábra alapján), ha egy C0 csúcsot beszúrunk, akkor a P szülőcsúcs és a C1 gyerek kerül exkluzív módú zárolásra (az első gyerek és a baloldali testvér mutató kapcsolatoknak megfelelően). Az újonnan beszúrt csúcsot nem kell zárolni, mivel nem érhető sem a szülőn, sem a jobboldali testvérén keresztül, mivel mindkettő exkluzíve zárolva van.

A harmadik protokoll, az **OO2PL** az előző kettőtől eltérően, nem a csúcsokat zárolja, hanem az azokat összekötő mutatókat (4.ábra), ennek következtében szükség van 4-4 új zárolási módra (első,utolsó gyerek,bal és jobb testvér), az új zárolási módok kompatibilitási mátrixa a 2(b) ábrán megtalálható.

A lehetséges hozzáférési minták vizsgálata alapján elmondható, hogy az OO2PL, NO2PL, Node2PL sorrend igaz a protokollok által támogatott konkurencia viszonylatában is. Igaz, ez a karbantartandó zárak számában is megjelenik, ugyanis az OO2PL csúcsonként akár 4 zárat is igényelhet, szemben az előző protokollok 1 zár/csúcsos igényével - ráadásul a Node2PL nem is rak zárakat a levelekre. A fenti protokollokra a szerzők készítettek egy tesztelési környezetet is, amelyen különböző teszteket végeztek a konkurens tranzakciók számának, a tranzakciók által végrehajtott műveletek számának és a dokumentumok számának függvényében. Az eredmények szintén a fenti sorrendet támasztják alá. Részletesebben [14]-ben olvashatunk a tesztekről.

A protokollok teljes értékűvé bővítéséhez szét kell választani a csúcsok tartalmára

vonatkozó módosításokat és lekérdezéseket illetve a strukturális bejárásokat és módosításokat. Ehhez bevezetjük a tartalomra vonatkozó S és X zárakat. A kibővített kompatibilitási mátrix:

	S	X	T	M
S	+	-	+	-
X	-	-	+	-
T	+	+	+	-
M	-	-	-	-

Ez alapján a Node2PL és NO2PL könnyedén kibővíthető, míg az OO2PL esetében nincs értelme a bővítésnek, hiszen abban a zárak a mutatókra vonatkoznak és nem a csúcsokra, ráadásul az S és X zárak gyakorlatilag kompatibilisek - mivel legalább egy Tx zárral együtt fordulhatnak csak elő - az Mx és Tx zárakkal.

Az ID, IDREF probléma kezelése. Mivel az XML dokumentumok lehetőséget biztosítanak egyedi azonosító alapján egy belső csúcshoz ugráshoz, ami eléggé összezavarná az eddig ismertetett - tipikusan fa-protokoll alapú - zárolási mechanizmusokat, további lépések szükségesek a protokollok teljes-értékűvé tételéhez.

Az azonnal felvetődő egyszerű megoldás, a csúcs leszármazottainak zárolása. Ez helyett a szerzők egy másik módot javasolnak. Dokumentumonként számontartott ID zárakat. Megkülönböztetve osztott(IDS) és exkluzív (IDX) zárakat. Ezeket a zárakat ugyanúgy a hozzáférések előtt kell megszereznie a műveletet végző tranzakciónak, mint a sima S és X zárakat, így például a problémát jelentő törlés esetében az összes ID-vel rendelkező leszármazottra is meg kell szereznie a zárakat.

DTD-ismeretének kihasználása. Egy felvetés a DTD ismeretéből fakadó többletinformáció kihasználására: amennyiben a DTD egy csúcs gyerekeit páronként diszjunkt halmazokra osztja fel, akkor az ezeken a halmazokon végzett műveletek kommutatívak - ez elérhető lenne halmazonként létrehozott ál-csúcsokkal(dummy node) . Például egy A*B*C* specifikációjú csúcs esetén a fenti protokollok nem engednek két konkurens tranzakciónak egyszerre beszúrni egy új A és egy új B gyereket, holott ez alapvetően semmilyen konfliktust nem kéne jelentsen. Ez a probléma például feloldható lenne a DTD-ből nyert információk felhasználásával.

3.2.2. Egy időbélyegzős megoldás - XTO

Az időbélyegzős protokoll ezen kibővítése szintén a fenti hármas Helmer, Kanne és Moerkotte nevéhez fűződik[13, 16]. A bővítés alapötlete a dokumentumok implicit többverziós kezelése - a törlések és beszúrások időbélyeggel ellátása és a csúcsok töröltnek illetve frissen beszúrtnak jelölése.

Az időbélyegzős protokolloktól örökölt működési jellemzők közül fontos, hogy mivel az ütemező a beérkező műveletek sorrendjét időbélyeg sorrendben határozza

op_1	op_2	$ts(T_1) < ts(T_2)$	$ts(T_1) > ts(T_2)$
T	T	no conflict[1]	no conflict[2]
D	T	ignore[3]	traverse[4]
I	T	traverse[5]	ignore[6]
T	D	mark as deleted[7]	abort T_2[8]
D	D	not possible[9]	abort T_2[10]
I	D	window[11]	not possible[12]
T	I	not possible[13]	ignore/abort T_2[14]
D	I	not possible[15]	ignore/abort T_2[16]
I	I	not possible[17]	ignore/abort T_2[18]

$ts(T_1) <>_{DCO} ts(T_2)$
no conflict[19]
choice[20]
choice[21]
$ts(T_1) <_{DCO} ts(T_2)$[22]
not possible[23]
not possible[24]
choice[25]
choice[26]
choice[27]

4. ábra. XTO és XCO protokoll akció tábla (a felsőindexben szereplő esetszámok részletesen lejjebb) forrás:[16]1.táblázat

meg, az időbélyeg sorrend fogja a konfliktusba keveredő tranzakciók commit sorrendjét is meghatározni és a később érkezett tranzakciók committálásának várakoztatása miatt a lépcsőzetes abortálás problémája fennáll. A több párhuzamosan létező verzió esetén a többverziós időbélyegzős protokollhoz (MVTO lásd: [21]) hasonlóan minden tranzakció az időbélyegének megfelelő verziót fogja látni. A tranzakciók által végezhető konkrét műveletek (nthP, nthM, del) helyett, a továbbiakban legtöbbször - a műveletek hatását jól mutató - **T**(raverse), **I**(nsert), **D**(elete) betűket használjuk. Első körben a strukturális műveleteket vizsgáljuk, a tartalom változtatására irányuló műveleteket az XTO és XCO protokollok strukturális műveleteinek vizsgálata után vesszük szemügyre.

Minden csúcshoz számon tartjuk az utolsó ütemezett bejárást, beszúrást, törlést. A bejárás(T) műveletek nem ütköznek egymással, míg a beszúrás(I) és törlés(D) műveletek minden művelettel ütköznek. A további vizsgálathoz néhány jelölés T_1,T_2tranzakciók , $op_1, op_2 \epsilon \{T, I, D\}$ a $T_1 és T_2$tranzakciók műveletei közül valók, ts(T_x) a tranzakció időbélyegét adja vissza. Továbbá feltételezzük, hogy op_1már ütemezésre került és op_2még ütemezésre vár, ekkor két esetet vizsgálunk, minden op_1,op_2kombinációra, mégpedig ts(T_1) < ts(T_2) és ts(T_1) > ts(T_2) eseteket (4.ábra).

Az XTO protokoll akciótáblájában lévő esetszámok leírása:

1. és 2. eset Két bejárási művelet sohasem ütközik

3. eset T_2 figyelmen kívül hagyhatja a T_1 által töröltnek jelölt csúcsot és a következő testvérre léphet nem számolva az n-edik csúcsot, de ha T_1 abortál, akkor T_2-nek is muszáj lesz,mert figyelmen kívül hagyott egy nem törölt csúcsot

4. eset Mivel a törlés a bejárás után történik, T_2 egyszerűen bejárja a (T_1 által töröltnek jelölt) csúcsot. Azért kell megtartani a törölt csúcsot egészen addig, amíg van a törlést végrehajtó tranzakciónál kisebb időbélyeggel rendelkező aktív tranzakció, mert különben előfordulhatna, hogy egy tranzakció kihagyna egy olyan csúcsot bejárás közben, amit be kellett

volna járnia.

5.eset T_2 egy olyan csúcsot jár be, amelyet T_1 szúrt be. Így ha T_1 abortál, akkor T_2-t is meg kell szakítani.

6.eset T_2 egyszerűen figyelmen kívül hagyja a csúcsot, mivel a beszúrás a bejárás után történik

7.eset T_2 töröltnek jelöli a csúcsot

8.eset T_2-t meg kell szakítani, mert a csúcsot már bejárta egy újabb időbélyegű tranzakció

9.eset Nem lehetséges: T_2-nek be kell járnia a csúcsot mielőtt törölhetné, de a bejárás művelet figyelmen kívül fogja hagyni a töröltnek jelölt csúcsot (lásd 3.eset)

10.eset T_2-nek abortálni kell, mert a csúcsot már törölte egy újabb időbélyegű tranzakció (a 8.esethez hasonlóan)

11.eset Az újonnan beszúrt csúcsot megjelöljük a ts(T_1) és a ts(T_2) időbélyeggel is. Egy harmadik T_3 tranzakció csak akkor fogja látni a csúcsot, ha ts(T_1) $<$ ts(T_3) $<$ ts(T_2) különben figyelmen kívül hagyja. (Ez lépcsőzetes abortáláshoz vezethet)

12.eset Nem lehetséges: (lásd 9.eset)

13.eset Nem lehetséges: T_1 nem tud bejárni egy olyan csúcsot, ami még nincs ott.

14.eset Ez egy igen problémás eset; ezért meg kell nézzük beszúráskor a két szomszédos csúcs utolsó bejárását és beszúrását és ha mindkettő újabb ts(T_2)-nél, akkor T_2-t meg kell szakítani, mert lehetséges, hogy egy bejárás egyikről a másikra látta a beszúrt csúcsot. Amennyiben legalább az egyik régebbi, úgy nem láthatta egyetlen ts(T_2)-nél újabb időbélyegű tranzakció sem az újonnan beszúrt csúcsot.

15.eset Nem lehtséges: nem törölhetünk olyan csúcsot, amit még be sem szúrtunk.

16.eset Ebben az esetben T_1 törölheti a T_2 által beszúrt csúcsot. Mivel nincsen olyan törlés, amit nem előz meg bejárás, ezért ezt az esetet lefedi a 14.eset egyik variációja. Azaz, ha a szomszédos csúcsok időbélyegei újabbak T_2 időbélyegénél, akkor T_2-t meg kell szakítani, különben pedig T_1 a megfelelő csúcsot törölte (ami nem egyezik meg a T_2 által beszúrttal)

17.eset Nem lehetséges: ugyanazt a csúcsot nem szúrhatjuk be kétszer

18.eset A 14. és 16.esettel analóg módon, meg kell nézzük, hogy T_1 figyelembe vette-e a T_2 által beszúrt csúcsot, és ha igen, akkor T_2-t meg kell szakítani.

3.2.3. Dinamikus commit rendezés - XCO

Továbbra is az előző protokollok alkotóinál maradva[13, 16], egy újabb XML struktúrára kialakított protokoll kerül bemutatásra. Az **X**ML Dynamic **C**ommit **O**rdering alapötlete, a tranzakciók committálási sorrendjének dinamikus megállapítása. Ez az előző időbélyeg alapú megoldáshoz képest is - ahol több esetben is a tranzakciók megszakítására kényszerültünk - jobb abortálási mutatókkal rendelkezik.

Az XCO protokoll nem rendel a tranzakciókhoz azonnal az indulásuk után időbélyeget, hanem kivár addig, amíg valamilyen ütközést nem tapasztal két tranzakció közt és csak ekkor rendezi őket. Ehhez karbantart a protokoll egy függőségi gráfot, ami dinamikus committálási sorrendet $<_{DCO}$ reprezentálja. A még nem rendezett tranzakciók sorrendjének meghatározását a 4.ábra jobboldali része és az itt található esetleírások részletezik. A sorrend meghatározása után az XCO ugyanúgy viselkedik, mint az XTO. Jöjjenek az esetleírások 19-től 27-ig:

19.eset Két bejárási művelet soha nem ütközik, így ebben az esetben még nem kell eldöntsük T_1 és T_2 sorrendjét.

20.eset Ebben az esetben rendezhetjük T_1-et és T_2-t mindkét irányba, visszajutván ezáltal a 3-as vagy 4-es esethez (és azok következményeihez, esetlegesen lépcsőzetes megszakításokhoz).

21.eset A 20.esethez hasonlóan, szintén két lehetőségünk van, döntésünk alapján az 5-ös vagy a 6-os esethez jutunk.

22.eset Ez az egyetlen olyan sorrend, amelyben nem kell megszakítani T_2-t.

23.eset Nem lehetséges: mivel egy törlés műveletet mindig meg kell előzzön egy megfelelő bejárás művelet, ezért T_1 és T_2 már rendezve vannak a 20.eset szerint

24.eset Nem lehetséges: Hasonlóan a 23.esethez, T_1 és T_2 már rendezve van a 21.eset szerint

25-27.eset Mivel a $\text{ts}(T_1) <_{DCO} \text{ts}(T_2)$ rendezés nem lehetséges, hiszen nem dolgozhatunk olyan csúccsal, amelyet még be sem szúrtak illetve nem szúrhatunk be egy csúcsot kétszer; ezért meg kell nézzük a frissen beszúrt csúcs szomszédos csúcsainak időbélyegeit. Amíg tudjuk garantálni, hogy $\text{ts}(T_2)$

újabb legalább az egyik szomszédos csúcs időbélyegénél, addig el tudjuk kerülni a konfliktusokat. Ezáltal az, hogy milyen sorrendet választunk (vagy nem döntünk még), azon múlik, hogy T_1 milyen időbélyegeket hagyott a szomszédos csúcsokon (ha hagyott egyáltalán).

A két protokoll teljesítményére vonatkozó tesztek igazolták az elején felvetett sejtést, miszerint a tranzakciók megszakítási aránya az XTO és XCO vonatkozásában, az XCO győzelmét hozta (részletesebben [16] 4.szakasz).

Az XCO és XTO teljes-értékűvé bővítése. A protokollok kiterjesztéséhez két újabb időbélyeg bevezetésére van szükség a csúcsok tartalmára vonatkozóan. Mégpedig az utolsó írás és az utolsó olvasás időbélyegére. Az olvasás műveletek alapvetően a bejárás(T) műveletekhez hasonlóan kezelendőek, így ütköznek a beszúrás(I) és törlés(D) műveletekkel és kompatibilisek a bejárás és olvasás műveletekkel, míg az írás műveletek alapvetően a beszúrás és törlés műveletekhez hasonlóak, egyetlen kivétellel, hogy a bejárás művelettel is kompatibilisek. A írás és olvasás műveletek közti konfliktusokat a klasszikus időbélyegzős protokoll szerint kezeljük.

ID, IDREF probléma. A probléma forrása az IDREF, IDREFS attribútumok nyújtotta ID-hez ugrás, ez természetesen nem felel meg a protokollok által eddig vizsgált felülről-lefelé bejárási elvet követő műveleteknek, így könnyedén elveszítenék a sorbarendezhető ütemezéseket generáló tulajdonságukat.

Az következő kiegészítésekkel kiküszöbölhető a probléma. Egyrészt megjelenik egy IDREF ugrás a bejáró műveletek közt, másrészt a törlés művelet pedig töröl fentről-lefelé egy-egy részfát, töröltnek jelölve a benne lévő csúcsokat. Ekkor az ugrás műveletnek mindössze ellenőriznie kell a célcsúcs őseit az ugrás előtt, hogy az esetleges ütközéseket észrevegyük. A beszúrás műveletek nem igazán érintettek az ID-hez ugrás problémakörben, de ha ID-vel rendelkező csúcsot szúrunk be, akkor ellenőriznünk kell, hogy nincs-e olyan IDREF ugrás, amely láthatta volna előtte az új csúcsot.

Az IDREF ugrásokat is figyelembe véve, a protokollok akciótáblája és az esetek leírása a következőképp alakul (5.ábra).

28-29.eset Két bejáró művelet nem ütközik.

30.eset T_2-t abortálni kell, mert a célcsúcsot törölték. A csúcs figyelmen kívül hagyása (mint a 3.esetben) nem lehetséges, mivel nincs alternatívája az ugrásnak, hacsak közben újra be nem szúrták máshová (ha igen, lásd 32. és 33. eset)

31.eset Mivel $\mathrm{ts}(T_1) > \mathrm{ts}(T_2)$, T_2 tranzakció bejárhatja a törölt részfát.

op_1	op_2	$ts(T_1) < ts(T_2)$	$ts(T_1) > ts(T_2)$
T	T	no conflict[28]	no conflict[29]
D	T	find newer version/abort T_2[30]	traverse[31]
I	T	traverse[32]	find older version/abort[33]
T	D	mark as deleted[34]	abort T_2[35]
D	D	abort T_2[36]	abort T_2[37]
I	D	window[38]	abort T_2[39]
T	I	not possible[40]	abort T_2[41]
D	I	not possible[42]	abort T_2[43]
I	I	not possible[44]	abort T_2[45]

5. ábra. IDREF ugrásokat is figyelembe véve az akció tábla (forrás [16]2.tábla)

32.eset T_2 bejárhatja az újonnan beszúrt csúcsot, de T_1 abortálása esetén T_2 is megszakad.

33.eset T_2 nem járhat be olyan csúcsot, ami még nem létezett, amikor T_2 megkapta az időbélyegét. Létezhetett ugyan egy csúcs korábban azonos ID-vel, ebben az esetben, T_2-nek ehhez a csúcshoz kell ugrani, különben pedig abortálnia kell.

34.eset T_2 töröltnek jelöli a csúcsot.

35.eset Ez egy olyan eset, amikor T_2 olyan csúcsot akar törölni, amire T_1 már ugrott. T_2-t meg kell szakítani.

36.eset T_2-nek be kellett járnia a csúcsot mielőtt törölni akarta, így ez a 30-as esettel megegyezik.

37.eset Nem törölhetjük ugyanazt a csúcsot kétszer, T_2-nek abortálnia kell.

38.eset Létezik egy olyan intervallum, amikor ez a csúcs érvényes (ts(T_1) és ts(T_2) közt). A bejáró tranzakcióknak ellenőrizni kell az intervallumba tartozásukat (mint 11.eset).

39.eset T_1 egy részfába ugrott és beszúrt egy csúcsot oda. T_2 törölte az részfa gyökerét és most járja be és törli az egész részfát, amikor is egy jövőben beszúrt csúcsot talál. T_2-t meg kell szakítani a konfliktus feloldásához.

40-es, 42-es és 44-es eset Ezek az esetek nem lehetségesek, mert nem járhatunk be vagy törölhetünk még be nem szúrt csúcsokat és kétszer be sem szúrhatunk egy csúcsot.

41.eset T_1 rossz csúcshoz ugorhatott, T_2-nek meg kell szakadni, mert T_1 láthatta az új csúcsot (mint 14.eset).

43.eset Olyankor fordulhat elő, amikor T_1 egy törlés művelete bejárt egy teljes részfát és az egészet töröltnek jelölte. T_2 pedig egy régebbi időbélyeggel

beleugrik az részfába és beszúr egy csúcsot - amit T_1-nek törölni kellett volna. A beszúrás során T_2 meg fogja nézni a szomszédos csúcsok időbélyegét és abortál (mint 14.esetben).

45.eset T_2 időbélyegéből az következik, hogy T_2 beszúrása T_1előtt történt. Ezáltal T_1-nek nem szabadott volna sikerüljön a beszúrás, de mivel már az megtörtént, egy lehetőség marad, T_2-t meg kell szakítani.

3.2.4. Ösvényzárolások 1. - Path Lock Satisfiability (Ösvényzár Kielégíthetőség)

Az ebben és a következő szakaszban ismertetésre kerülő protokollok Dekeyser és Hidders [4, 3]nevéhez fűződnek és mondhatni egyenértékűek. A köztük lévő különbségek egyben ellensúlyozzák is a keletkező előnyöket és hátrányokat, de gyakorlatilag azonosnak mondhatjuk a végeredményt. Mindkét protokoll sorbarendezhető ütemezéseket generál, amely tulajdonság bizonyítása megtalálható a referenciák közt lévő cikkekben, ezzel a továbbiakban itt részleteiben nem foglalkozunk. Mivel a cél csupán az ösvény alapú protokollok képviselőinek és azok különbségének megmutatása az eddig látottakhoz képest, próbálunk minél informálisabb hangnemben szólni a protokollokról. Ennek fényében a következzenek a szükséges alapok az ösvény alapú protokolljainkhoz.

Mivel most a dokumentum kezelését ösvénykifejezésekkel szeretnénk megoldani, pár szó az itt felhasználtakról. A végrehajtott lekérdezések egy-egy csúcs-ösvénykifejezés párt alkotnak, ahol a csúcs lehet a dokumentumot reprezentáló csúcs is és belső csúcsok is. Feltesszük továbbá, hogy a tranzakcióknak lehetőségük van tárolni változókban a lekérdezéseik eredményét vagy annak egy részét (például: egy belső csúcs azonosítóját későbbi felhasználás céljából vagy egy lekérdezés eredményét jelentő string listát). Szükség van ezen felül módosító műveletekre is, amelyek még sajnos nem kerültek be az XPath szabványába, most három csoportra bontjuk ezeket: attribútum, elem, szöveg-elem módosítások - a konkrét műveletek nem kimondottan fontosak a vizsgálat szempontjából és mivel viselkedésük is nagyon hasonló a teljeskörű vizsgálatnál is csak egy részüket szükséges vizsgálni. Paramétereik közt mindig van egy azonosító a módosítani kívánt csúcs meghatározásához és végrehajtásukat az eddigiekhez hasonlóan egy olvasás művelet előzi meg. A továbbiakban feltesszük még, hogy a megszerzett zárakat a tranzakciók befejeződésükig megtartják. Az XPath kifejezésekhez az alábbiakban a következő limitált nyelvtant használjuk:

P::= F |F/P |$F//P$

F::= .| T | * | @A |@* | γ | σ

Ahol T tag nevek halmaza, A attribútum nevek halmaza, γ a text() funkció,

ami egy csúcs összes text típusú gyerekét visszaadja és σ az attribútumok és string-csúcsok értékét visszaadó függvény.

Zárak. Az olvasási zárakat egy-egy (n,p) párossal ábrázoljuk, ahol n a csúcs azonosítója, amire végrehajtjuk a p ösvénykifejezést. Informálisan annyit tesz, hogy a tranzakció végrehajtotta a p lekérdezést az n csúcsra. A szükséges zárak (R_q) halmazának meghatározásához az alábbi szabályokat használjuk egy x_n= q lekérdezés esetén:

- ha q = x_m/p , akkor R_q= {(x,p) | x ϵx_m}
- ha q = x_m//p , akkor R_q= {(x,.//p) | x ϵx_m}
- ha q = /p , akkor R_q= {(r,p) }, ahol r a dokumentum csúcs
- ha q = //p , akkor R_q= {(r,.//p) }, ahol r a dokumentum csúcs

Az írási zárakat egy-egy (n,f) párossal reprezentáljuk, ahol n egy csúcs azonosítója, amire a zárolás vonatkozik és $f \epsilon F \backslash \{.\}$ a módosító nyelv nyelvtanának egy eleme, amely meghatározza, hogy a csúcson belül milyen módosítás történt (text eleme változott, gyerekeinek a listája változott). A különböző módosításokhoz meg van határozva, hogy milyen zára(ka)t szükséges elhelyezni (például: ha egy csúcshoz egy új attribútumot szúrunk be, akkor csúcsra egy @a zár kerül).

Most, hogy tudjuk mely műveletekhez, milyen zárak kiosztása szükséges, meg kell nézzük, hogy ezek a zárak mikor ütköznek. Ehhez 3 szabályt használ ez a protokoll:

- Két olvasási zár sohasem ütközik.
- Egy (n,p) olvasási zár ütközik egy (n',f) írási zárral akkor és csak akkor, ha n az n' őse és az ösvény(n,n')/f kielégíti p-t.
- Egy (n,f) írási zár mindig ütközik egy (n,f') írási zárral.

Ahol ösvény(n,n') az n csúcsból az n' csúcsba vezető úton található elemek neveit jelenti /-el elválasztva. Egy XPath kifejezés *kielégít* egy p kifejezést, ha az a p által reprezentált ösvények egyike.

A zárak megszerzésére vonatkozó szükséges és elégséges feltétel, hogy másik tranzakció nem rendelkezhet közben ütköző zárral.

A protokoll hely és időigényéről: a helyigény láthatóan nem jelent problémát, gyakorlatilag a dokumentum (fa) méretével egyenesen arányos, ellenben az idő (többletterhelés) szempontjából már más a helyzet, ugyanis annak meghatározása, hogy egy ösvénykifejezés kielégít-e egy általános XPath kifejezést, az leginkább ahhoz hasonlítható, hogy egy string-et elfogad-e egy nem determinisztikus automata, ez pedig $O(n^4)$-es probléma.

parent lock	child type	child name	child lock
$.//p$	element	-	$.//p$
t/p	element	t	p
$t//p$	element	t	$.//p$
$*/p$	element	-	p
$*//p$	element	-	$.//p$
$@a/p$	attribute	a	p
$@*/p$	attribute	-	p
τ/p	text	-	p

6. ábra. Ösvényzárak továbbterjesztési szabályai (forrás [3] 4.ábra)

3.2.5. Ösvényzárolások 2. - Path Lock Propagation (Ösvényzár Továbbterjesztés)

Ez a protokoll az előzőtől nem sokban különbözik, szerzőik megegyeznek, tudásuk szempontjából gyakorlatilag egyenértékűek, mindössze a zárak kiosztása és ellenőrzése követ egy másik megközelítést.

A zárak kiosztásánál alapvetően az előző protokollt követi, kiegészítve a kiosztandó olvasási zárak halmazát további *következmény* és *továbbterjesztett* zárakkal. Azaz, a kezdetben megszerzett olvasási zárak halmaza (R_q) megegyezik az ösvényzár kielégíthetőségnél megadott szabályokkal előállított halmazzal, majd ezen halmaz elemeire alkalmazzuk a következmény zárak és a továbbterjesztett zárak szabályait. A következmény zárakat (a szabály jobb oldalán lévő alakban) ugyanarra a csúcsra rakjuk, amely már rendelkezik egy az alábbi szabályok bal oldalán található alaknak megfelelő zárral:

- $.//p \Rightarrow p$
- $./p \Rightarrow p$

A *továbbterjesztett* zárak kiosztásakor mindig egy adott zárral rendelkező csúcs alatti rész-fa csúcsaira rakunk zárakat a 6.ábrának megfelelően.

A következmény és továbbterjesztési zárakat a kezdeti olvasási zárak halmazára (R_q) kell alkalmazni, mindaddig amíg ezáltal új zárak kerülnek be a halmazba. Az eredmény halmaz R_q^* a q lekérdezés által megszerzendő olvasási zárakat fogja tartalmazni és mivel ezek a zárak függnek a dokumentum szerkezetétől, minden az érintett részre vonatkozó módosítás után újra kell számítani őket. Megjegyzendő, hogy egy megfelelő alakú lekérdezés hatására jelentős mennyiségű zár kerül kiosztásra, ennek enyhítő körülményei, hogy a lekérdezés eredményének kiszámításakor bejárjuk az összes olyan csúcsot, amire zárat is kell rakjunk, tehát ez meglehetősen kevés többletmunkával megoldható és egy hash táblában hatékonyan tárolhatóak a kiosztott zárak.

A zárak kiosztása mindkét esetben (következtetés,továbbterjesztés) atomi utasításnak számít, tehát vagy mindet ki tudjuk osztani vagy egyiket sem.

Az írási zárakat ennél a protokollnál ugyanazon szabályok alapján helyezzük el, mint tettük azt az előző protokollnál. A következő különbséget a zárak ütközéseinél találjuk. A szabályok most a következők:

- Két olvasási zár sohasem ütközik
- Egy (n,p) olvasási zár akkor és csak akkor ütközik egy (n,f) írási zárral, ha $p{=}f$ vagy $p{=}*$.
- Egy (n,f) írási zár mindig ütközik egy (n,f') írási zárral.

A zárakat ennél a protokollnál is csak akkor szerezhetik meg a tranzakciók, ha másik tranzakció nincs már birtokában ütköző zárnak. A protokoll hely és időigényével kapcsolatban: itt most az írási és olvasási zárak ütközéseinek ellenőrzése lesz az egyszerűbb feladat, hiszen most az adott csúcson jelen kell lenni mindkét zárnak és csak azok ekvivalenciáját kell vizsgálni, ellenben a helyigény a fa méretéhez (n) és a lekérdezés hosszához (m) viszonyítva $O(nm)$-es lesz.

ID ugrás. A szokásos problémakör, amellyel az XML dokumentumokhoz készült protokolloknak szembesülniük kell. Ebben az esetben a szerzők ajánlata nem különösképp szép vagy bonyolult, pusztán egy lehetőség. Természetesen a kidolgozandó részletek közt említik egy jobb referenciakezelés megvalósítását. Konkrétan arról van itt szó, hogy a hivatkozásokat egyszerű attribútumokként kezelik, tehát egy lekérdezéssel lekérik az értékeket egy következővel pedig megkeresik a kívánt csúcsot. Ez meglehetősen szerény erőfeszítés és gyakorlatilag a teljes szükséges dokumentumrészlet bejárását hozza magával. (Mivel a forrásul szolgáló cikk nem túl friss 2002 és 2003 a két forrásanyag [3, 4] születésének éve, így mostanra, ha foglalkoztak a problémával azóta is, már lehet hogy született valami hatékonyabb megoldásuk is, most inkább hagyjuk ezt a részt a jövőbeni feladataik közt.)

3.2.6. Ösvényzárolások 3. - Egy másik megközelítés

Ez a megoldás Hye Choi és Kanai nevéhez fűződik [2], az alapok közül a precíziós zárolást emeli ki egy lehetséges jó megoldás alapjaként és ennek hatása alatt - a konfliktusok ellenőrzései a precíziós zárolásnál megszokott módon történnek - egy sorbarendezhetőséget garantáló protokollt mutat nekünk a megvalósításához szükséges adat karbantartási megoldással és a zárak ütközéseinek ellenőrzését végző algoritmussal.

Kicsit bővebben a megoldásról. Tehát az ütközés ellenőrzések az érkező írási és olvasási kérések alapján olvasó-író és író-olvasó ellenőrzésekként történnek, azaz az érkező írási zár kéréseket a már meglévő olvasási zárakkal hasonlítjuk össze és fordítva. A megoldás szerves részét képezi az adatok karbantartásához kitalált

megoldás, amely alapján három dokumentum fajtát tartunk számon: (1) az eredeti dokumentum jelölése: D_{st}, (2) minden tranzakció T_i számára egy másolat az eredeti dokumentumból D_i, és (3) az eredeti dokumentum egy másolata a konfliktusok ellenőrzéséhezD_{all}. A tranzakciók az eredeti dokumentum helyett a saját dokumentum verziójukat fogják látni és az írási műveleteiket mind a saját D_i mind a konfliktusok ellenőrzésére fenntartott D_{all} dokumentum verzión végrehajtják. Így D_{all} tükrözni fogja az összes módosítást és a tranzakciók módosításai csak akkor tükröződnek az eredeti dokumentumon, ha a tranzakció már committált.

A zárolásokat egy XPath alapú logikai zárolással oldják meg a szerzők, amely ugyan megnehezíti a konfliktusok ellenőrzését, de logikai volta miatt azonnal eltünteti a fantom probléma lehetőségét. Az ütközések vizsgálatához az imént említett adatmodellt és ezt az XPath alapú zárolást használják ki. Ennek fényében az olvasó-író ellenőrzéseket a D_i és D_{all} dokumentum verziók összehasonlításával kivitelezik, mivel a D_{all} dokumentum tárolja az összes változtatást elég ez az egy összehasonlítás, míg a komplikáltabb író-olvasó ellenőrzéseket a D_{all} dokumentum előző verzióinak felhasználásával oldják meg.

A felhasznált tranzakciós és adat-karbantartó modell. Az egy dokumentumhoz párhuzamosan hozzáférő aktív tranzakciók halmaza $T = \{T_1, T_2, ..., T_n\}$ (n$\geqq$2). Használjuk még a tranzakciók által végrehajtott olvasási és írási műveleteket csoportosító szekvenciákat (*R-access,W-access*), amelyek az egymás utáni azonos műveleteket jelölik és egy időrendi sorrendet mutató hozzáférési szekvenciát (*access sequence*), amely a tranzakciók írási és olvasási műveleteinek szekvenciáit mutatja időrendi sorrendben (AS_i). Definiálva vannak még különböző műveletek a szekvenciákra is:

- $WS_i(k)$ ($k > 0$): T_i k-adik írási szekvenciája
- $|WS_i(k)|$: a $WS_i(k)$-ban található írási műveletek száma
- wn_i: T_i aktuális írási szekvenciáinak száma
- $RS_i(k)$ (k$\leqq$0): ha k = 0, akkor az első olvasási szekvenciát jelenti, különben a $WS_i(k)$-t követő olvasási szekvenciát jelenti

Az olvasási műveletet *Read(path)* ként jelöli, ahol *path* egy XPath ösvénykifejezés. Az írási műveleteknél pedig feltüntetik a művelet cél csúcs(ai)t.

Az adat-karbantartásban a már ismertetett D_{st}, D_i, D_{all} dokumentum verziók veszek részt. Ezek mellett szükség van egy csúcsok közti ekvivalencia fogalomra és egy dokumentumok összeolvasztását lehetővé tévő műveletre (*Merge*(D_i, D_j)), ez az utóbbi művelet csak diszjunkt megváltoztatott csúcsokkal rendelkező dokumentumok esetén működik.

A zárak ellenőrzése. Az olvasó-író esetben ugyebár a probléma akkor áll fent, amikor egy lekérdezés eredménye nem azonos a tranzakció saját dokumentum verziójában és egy másik konkurens tranzakció dokumentum verzióján. Mivel a dokumentumhoz hozzáférő összes tranzakció verziójával az összehasonlítás túl sok erőforrást igényelne, ezért az összehasonlítást egy olyan dokumentumon (D'_i) végezzük el, amit a tranzakció saját D_i-jéből és a konkurens tranzakciók írási műveleteiből állítunk elő. A D'_i-n végzett ellenőrzések meg fogják találni az összes lehetséges ütközést (bizonyítás [2]-ben) és mivel az itt előállított D'_i kísértetiesen hasonlít a már említett minden változtatást tartalmazó D_{all} dokumentumra, az ellenőrzéseket természetesen ezen a központilag karbantartott verzión végezzük és nem építjük fel minden ellenőrzéshez. Az ellenőrzés feladatai közé tartozik továbbá az ütköző tranzakciók kezelése, ehhez először is meg kell határozza az ütköző műveletet végrehajtott tranzakciót (mivel D_{all}-ból nem derül ki, hogy ki végezte a módosítást), hogy annak végeztével újra megpróbálhassa az épp ellenőrzött és akadályokba botló művelet végrehajtását. Ezt a protokoll az ellenőrzés során meg is teszi, tehát nem kell további lépéseket tennie.

Az író-olvasó ellenőrzésekhez azonban szüksége van a protokollnak a D_{all} dokumentum előző verzióira. Ezek előállíthatóak a fentiekben definiált műveletek segítségével, tehát egy tetszőleges D_j-t előállíthatunk egy tetszőleges írási/olvasási szekvenciája előtti állapotában. Ezáltal lehetőségünk nyílik egy tranzakció dokumentum verziójának minden állapotára ellenőrzést futtatni és megállapítani, hogy mely műveletek végrehajtásából fakadt a probléma. A sok verzió tárolása természetesen felveti a szokásos többverziós protokolloknál előforduló problémát, azaz hány verzió tárolható az ésszerűség keretein belül illetve hogyan biztosítható a megfelelő verziók hozzáférhetősége. A D_{all} verziók mentési idejének meghatározásáról és az adott mentések törléseinek (megtartott verziók számának) hatásáról a protokoll teljesítményére részletes leírás áll rendelkezésre [2] 5.2-es szakaszában. Az író-olvasó ellenőrzésekhez, amikor egy T_i tranzakció egy W_i hozzáférést kér, a protokoll felhasználja a D_{all} verziókat végighaladva a tranzakció olvasási és írási műveletein mentett verziónként és kiértékeli a helyzetet lépésenként (részletes leírás és pszeudokód az ellenőrző algoritmushoz [2]-ben található). Közben szükség esetén tranzakciókat blokkol és újraindít műveleteket.

3.2.7. Egy többverziós megoldás - MPX

Tulajdonságait tekintve egy az eddigiekhez hasonló tulajdonságokkal rendelkező, sorbarendezhetőséget biztosító protokollról beszélünk. Amiben mégis más, az a többverziós voltából adódó lehetőségek hangsúlyozása, mégpedig a lépcsőzetes abortálás elkerülése illetve a csak-olvasó és az író-olvasó tranzakciók ütközéseinek elkerülése. A protokoll Yuan Wang, Gang Chen és Jin-xiang Dong munkássága nyomán látott

napvilágot és nevét a **M**ultiversion concurrency control **P**rotocol for **X**ML kifejezésből kapta.

A tárgyalás során használt adat és tranzakció modellről egy rövid áttekintéssel kezdünk. Az adatmodellünk most az XPath adatmodelljéből kerül kialakításra kiegészítve a többverziós működéshez. A dokumentumot d egy irányított gráfként képzeljük el, azaz d = (N,E,r) hármas,ahol N a csúcsok halmaza, E = N x N az élek halmaza és r ϵN a dokumentum gyökéreleme. A csúcsokat egy azonosítóhoz tartozó verziók sorozataként képzeljük el (id,<$n_{1,...}n_m$>), ahol n_k egy (l,v, τ, d) négyes, amiben l a címke, v az érték, τ a verziószám, d pedig egy logikai érték, ami n_k törölségét jelzi. A modell nem tesz különbséget az elemtípusok közt.

A protokoll fontos eleme a τ-val jelölt verziószám, ebből minden tranzakció kap egyet indulásakor ($\tau(t)$)és minden n_k csúcs-verzió is rendelkezik egy ilyennel ($\tau(n_k)$). A protokoll különbséget tesz csak-olvasó és módosító tranzakciók közt. Így ha egy csak-olvasó tranzakció olvas egy n csúcsot, akkor az az utolsó a tranzakciót megelőzően committált verziót (n_K) fogja olvasni, formálisan:

1. $\tau(n_k) \leq \tau(t)$
2. minden (j>k)-ra igaz: $\tau(n_j) > \tau(t)$
3. n_k nem törölt azaz $d(n_k)$= hamis

Ha a keresett csúcs-verzió nem létezik, akkor egyszerűen figyelmen kívül hagyja az olvasás műveletet.

Ha módosító tranzakció próbál olvasni egy csúcsot, akkor az először egy osztott (*shared*) zárat kell kérjen a csúcsra és csak miután azt megkapta olvashatja a megfelelő verziót ugyanúgy, mint a csak-olvasó tranzakció.

Ha egy tranzakció írás műveletet akar végrehajtani, akkor először egy kizáró vagy exkluzív (*exclusive*) zárat kell rá szerezzen, ha beszúrás, akkor a szülő csúcsra is kell kérjen egy ilyen zárat. Ha ütközés van, akkor blokkolódik a tranzakció amíg a rendszer fel nem oldja (befejeződik a blokkoló tranzakció vagy a holtpontot megtalálja a rendszer). A zár megszerzése után:

1. Ha beszúrás történik, akkor beszúrjuk a csúcsot és ∞-re állítjuk a verzióját.
2. Máskülönben megkeressük a megfelelő n_k-t, amit módosítani akarunk (ugyanúgy mint olvasásnál)
3. Ha $\tau(n_k)$=∞, akkor felülírja, mivel ezt a verziót ő maga szúrta be. Ha a verzió nem ∞, akkor létrehoz egy új verziót (n_j) ezzel a verziószámmal. Ha a művelet törlés, akkor $d(n_j)$-t igaz-ra állítja.

Amikor egy tranzakció committál, a következő műveleteket hajtja végre:

1. Minden általa létrehozott n_k verzión a verziószámot $\tau + 1$-re állítja (itt τ a globális verziószám, amiből a tranzakció is kapta indulásakor a verziószámát)

2. Megnöveli τ-t 1-gyel

3. Elengedi az általa tartott zárakat.

Ha egy módosító tranzakció abortál, egyszerűen elengedi a nála lévő zárakat, az általa létrehozott ∞ verziószámú verziókat pedig figyelmen kívül hagyja az MPX. A protokoll a szigorú kétfázisú lock protokoll szabályait követi a módosító tranzakcióknál, azaz a zárakat a tranzakció befejeződéséig megtartja.

A protokoll teljesítményéről részletesebb információt kaphatunk [20] 5.szakaszából, röviden csak annyit róla, hogy az OO2PL-lel az XCO-val történő összehasonlítások alapján (ezek voltak a cikk írásakor a versenyképes protokollok) az MPX legalább olyan jól teljesít mint az XCO, ez a csak-olvasó eseteknél jelentős különbséget jelent a 2PL alapú protokollok „csúcsmodelljéhez" képest. Amiben mindkettőt felülmúlja, az az alacsonyabb megszakítási arány, aminek a lehetőségét az elején hangsúlyoztuk is.

Sajnos a megtartott verziók számával és azok karbantartásával kapcsolatosan nem kapunk információt, ami egy fontos momentum lenne egy többverziós protokollnál, hiszen alapvetően befolyásolja az ütközések ellenőrzésének sebességét.

3.2.8. Dataguide és XML - DGLOCK

A Dataguide struktúrát felhasználó protokollok „úttörője" ez az XML adatok feldolgozásához kialakított általános DAG zárolási protokoll. A protokoll Grabs, Böhm és Schek nevéhez fűződik és bemutatását egy tetszőleges adatbáziskezelő fölé elhelyezhető XML tranzakció kezelővel (XMLTM) együtt tették meg. Az XMLTM működéséből fakadóan, ezen protokoll kapcsán sem találkozunk kiemelten az XML adatokhoz kialakított tárolási megoldással, ugyanis minden az XML dokumentumokkal kapcsolatos kérést ez a második tranzakciókezelő réteg fog továbbítani az adatbázisnak (tetszőleges tároló rétegnek).

A protokoll különbséget tesz tartalmi és strukturális műveletek közt - ezzel a megközelítéssel már találkozhattunk több eddig ismertetett protokolloknál is. A kérések tartalmi és strukturális részeit természetesen a tranzakciókezelő fogja meghatározni és ami a protokoll működésében az igazi érdekesség, hogy a zárakat nem a dokumentum csúcsaira rakja, hanem egy Dataguide-on [6] helyezi el, ami gyakorlatilag a dokumentum struktúrájának összegzése egy megfelelő alakú gráfba. A dokumentum használatának mellőzése a tranzakció-kezelés szempontjából gyakorlati megfontolásokon alapul, hiszen a legtöbb esetben nem férünk hozzá a teljes dokumentumhoz, mivel azt általában nem „konyhakész" állapotban tároljuk az adatbázisokban, így

	Granted					
Requested	None	*IS*	*IX*	*S*	*SIX*	*X*
IS	+	+	+	+	+	*P*
IX	+	+	+	*P*	*P*	*P*
S	+	+	*P*	+	*P*	*P*
SIX	+	+	*P*	*P*	*P*	*P*
X	+	*P*	*P*	*P*	*P*	*P*

7. ábra. DGLOCK kompatibilitási mátrixa (forrás: [7] 1.tábla)

itt most egy sokkal gyengébb feltételt támasztunk; elég a struktúrát összegző gráf. A DGLOCK protokoll egy (szigorú) két-fázisú lock protokollt alkalmaz a zárak kiosztására, azaz minden kérés után megpróbálja megszerezni a számára szükséges zárakat és ha megkapta azokat, akkor tranzakció végéig meg is tartja. A zárolásokhoz kizáró (X),osztott (S),kizáró-szándék (IX),osztott-szándék (IS) zárakat, valamint a tartalmi zárakhoz egyszerű $x\ \Theta const$, $\Theta\epsilon\{=,\epsilon,\neq,...\}$ alakú predikátumokat alkalmaz a protokoll. A zárak kompatibilitási táblázata (7.ábra), nem tartalmaz szigorú ütközéseket, mert azok csak a kiosztott predikátumoktól függnek. Ütközés akkor van, ha a kérelmezett predikátum zár ütközik a már meglévő predikátumokkal (ha a csúcson van egy predikátum nélküli ütköző zár, természetesen az is ütközést jelent).

A DGLOCK működése egy s kérés esetén:

1. Az összes s-ben található ε ösvénykifejezést kiértékelve meghatározza a strukturális és tartalmi zárakat és elhelyezi azokat az ösvénykifejezések alkotóelemein.

2. Meghatározza a Dataguide-ban azokat a csúcsokat (N), amik a kérésben előforduló ($e\epsilon\varepsilon$) elemeknek megfelelnek, különbséget téve a módosított és a csak olvasott csúcsok közt.

3. A kompatibilitási mátrixot felhasználva végrehajtja a következő lépéseket a Dataguide kiválasztott csúcsaira ($n\epsilon N$)

   (a) Ha a csúcsot módosítja a kérés s, akkor minden a gyökérből a csúcsba vezető úton található csúcsra IX zárat rak és csúcsra magára egy X zárat, mindeközben figyelembe veszi a csúcsokon és a kérésben érintett elemeken ($e\epsilon\varepsilon$) lévő annotációkat.

   (b) Ha s olvassa csak a csúcsot, akkor legalább egy a gyökérből a csúcsba vezető úton található csúcsokon IS zárakat helyez el és magán a csúcson egy S zárat és itt is természetesen figyelembe veszi a predikátumokat, mint a módosításnál.

Ha ütközést talál, akkor blokkolja tranzakciót. Ha olyan kérést dolgoz fel, ami holtpont kialakulásával járna, akkor a holtpont ellenőrzés megszakítja a tranzakciót.

A nagy mélységű Dataguide-ok esetén a kiosztott zárak mennyisége problémákat okozhat, ilyenkor alkalmazható durvább zárolás, tehát az S és X zárakat kioszthatjuk valamelyik ősre és nem a módosított/olvasott csúcsra. A teljesítmény és az eddigi protokolloknál is figyelembe vett tulajdonságok fényében meg kell még jegyezni, hogy a DGLOCK protokoll alapvetően nem biztosít sorbarendezhető ütemezéseket csak ismételhető-olvasást (*repeatable-read*) ezt természetesen a jobb teljesítmény érdekében teszi, ellenben ez életben hagyja a fantom problémát, amit az eddig ismertetett protokollok kezeltek. Továbbá nem esik szó az ID ugrásokról sem.

Összességében a DGLOCK protokoll nem a gyakorlati haszna miatt fontos számunkra, hanem a Dataguide felhasználásának első képviselőjeként játszik fontos szerepet a továbbiakban ismertetésre kerülő XDGL és SXDGL protokollok szempontjából. Megjegyezném, hogy a Dataguide-alapú lekérdezés a LORE [6] rendszerben jelent meg a 90-es évek végén, ahol félig-struktúrált adatok kezelésének hatékony megvalósításához használták fel, így igazából a DGLOCK az első XML adatokhoz igazított Dataguide-alapú protokoll, amivel az kapcsolódó munkákban gyakran találkozhatunk.

3.2.9. Egy újabb Dataguide-alapú protokoll - XDGL

A Dataguide struktúra felhasználásának ezen példája már sokkal jobb lehetőségeket kínál tulajdonságait tekintve, mint az előző alszakaszban ismertetett „alap" protokoll. A megoldás a Pleshachkov, Chardin, Kuznetcov hármas munkájának gyümölcse és a Dataguide-on elhelyezett zárakon felül a rendelkezésére álló DTD séma ismeretét kihasználja.

Az ütemezésre kerülő tranzakcióktól elvárt feltétel a szigorú két-fázisú lock protokollnak megfelelés (megszerzett zárak megtartása a befejezésig). A zárolásoknál alapvetően osztott (S) és kizáró (X) zárak és ezek szándék zárai (IS,IX) kerülnek felhasználásra. Az osztott és kizáró zárak megszerzésének itt is - mint a DGLOCK-nál - előfeltétele a csúcs ősein a megfelelő szándékzárak birtoklása. A zárak közt megjelennek még a rész-fák zárolását lehetővé tevő zárak (ST,XT) - az XT a törlés műveletnél biztosítja a csúcs alatti részfa zárolását, míg az ST minden lekérdezés, törlés és beszúrás esetén használható, ezek hasznos zárak, mivel az XPath lekérdezések eredményei gyakran rész-fák - és a fantom problémát kiküszöbölni hivatott logikai zárak. Megjegyzés: az osztott zárak alkalmazkodnak ebben a protokollban a módosító műveletekhez olyanformán, hogy - bár most nem a teljes XPath nyelvet lefedő megoldásról beszélünk - létezik külön mindhárom beszúráshoz (insert-into,-before,-after) egy osztott zár (SI, SA, SB), a művelet cél (vagy inkább eredmény) csúcsainak védésére ezek a zárak egyben a dokumentum csúcsainak sorrendjében bekövetkező konfliktusokat is megakadályozzák (például: két utolsó gyerek beszúrása).

Továbbá különbséget teszünk most is a strukturális és a tartalmi zárak közt, a

	granted							
requested	SI	SA	SB	X	ST	XT	IS	IX
SI	P	+	+	P	+	P	+	+
SA	+	P	+	P	+	P	+	+
SB	+	+	P	P	+	P	+	+
X	P	P	P	P	P	P	+	+
ST	+	+	+	P	+	P	+	P
XT	P	P	P	P	P	P	P	P
IS	+	+	+	+	+	P	+	+
IX	+	+	+	+	P	P	+	+

8. ábra. XDGL - kompatibilitási mátrix (forrás: [17] 2.ábra)

strukturális zárak az eddig említett csúcs és rész-fa zárak lesznek, míg a tartalmi zárolások megoldását a zárak annotálásával oldjuk meg, azaz kapnak a zárak egy-egy érték-alapú predikátumot (Value Based Predicate), az egyszerűség kedvéért például az IS és IX záraknál ez mindig #igaz értékű lesz.

Az XDGL protokoll kompatibilitási mátrixában (8.ábra) szintén nincsenek szigorú ütközések, hiszen itt is a predikátumok kiértékelése után derül csak ki, hogy valóban van-e ütközés. Szigorú kompatibilitás van ellenben az IX és X illetve IX és S zárak közt hiszen a szándék zárak nem a csúcsra jelentenek megkötést, hanem a csúcs egy leszármazottjára.

A fantom problémát megakadályozó logikai zárak gyakorlatilag tulajdonságok (logikai feltételek) halmazai, ezek a zárak akadályozzák meg a megfelelő tulajdonságokkal rendelkező csúcsok beszúrását. A másik logikai zár az IN (új beszúrás) zár amely egy beszúrandó csúcs tulajdonságait adja meg. A két most leírt zár természetesen a megfelelő tulajdonságok egyezése esetén (logikailag zárolt: csúcsok neve, neve-értéke, szülője-neve-értéke) ütközik.

Az XDGL ütemezője. Itt és most csak az ütemező lépéseit soroljuk fel az ütemező helyességének bizonyítása megtalálható [19] 4.3-as szakaszában.(Az ütemező lépésein is látszik hogy a protokoll a DGLOCK-ból alakult ki.)

Tehát az ütemező a következőképp jár el egy új művelet $a(op_i, t_j)$ esetén:

1. Meghatározza a Dataguide-on belül azokat az ösvényeket (DP = *data-path-set*), amelyek érintettek a lekérdezés/módosítás által.

2. Kiszámítja az érintett ösvényeket alkotó csúcsok predikátum halmazát NP (*node-predicate-set*) = $\{(n_j,p_j)\}$, ahol n_j egy érintett csúcs és p_j az op_i művelet által a csúcsra vonatkoztatott predikátum.

3. Kiszámítja a fantom halmazt PH=$\{(n_j, properties_j)\}$, ahol n_j a Dataguide egy olyan csúcsa, ahol fantom jelenhet meg és $properties_j$ a zárolandó csúcsok tulajdonságainak halmaza.

4. Ha a művelet egy beszúrás művelet és bővíti a Dataguide-ot, akkor kiszámítja a $properties_j$-t.

5. Megszerzi a művelet számára szükséges csúcs és fa zárakat:

- Ha lekérdezés, akkor $n_j \epsilon NP$-re szerez (ST,p_j) zárat és n_j minden ősére (IS, #igaz) zárat
- Ha I_I(csúcsba beszúrás)(I_B, I_A-re ugyanígy) a művelet, akkor minden $n_j \epsilon NP-re$: (1) ha n_j a beszúrás egyik célcsúcsa, akkor szerez rá (SI,p_j) zárat és (IS,#igaz) zárat az őseire, (2) ha n_j a művelet ösvénykifejezésének egy ága, akkor (ST, p_j) zárat állít be rá és (IS,#igaz) zárat az őseire, (3) ha $n_j$az egyik beszúrt csúcs, akkor (X, p_j) zárat rak rá és (IX,#igaz) zárat az őseire.
- Ha törlés műveletről van szó, akkor minden $n_j \epsilon NP - re$: (1)ha n_j a törlés egyik célcsúcsa, akkor szerez rá (XT,p_j) zárat és (IX,#igaz) zárat az őseire, (2) ha n_j a művelet ösvénykifejezésének egy ága, akkor (ST, p_j) zárat állít be rá és (IS,#igaz) zárat az őseire

1. Minden $n_j \epsilon PH$ csúcsra kioszt egy (L,$properties_j$) logikai zárat.

2. Ha a Dataguide-ot bővítő beszúrás művelet történt, akkor a Dataguide-ban az érintett csúcs őseire is szerez új beszúrás zárakat (IN,$properties_i$).

3. Ha ütközést talál, akkor késlelteti az új művelet végrehajtását.

Ez a Dataguide-alapú protokoll, már jobb lehetőségeket kínál ugyan a DGLOCK-nál, de sajnos nem közöltek róla kvantitatív mérési eredményeket a szerzők így jobbára csak az olvasó fantáziájára van bízva, hogy a zárolás áthelyezése a dokumentumról a háttérben karbantartott Dataguide-ra mennyire kifizetődő a logikai (predikátum jellegű) zárak kiértékelése jelentette garantált többletmunka mellett.

3.2.10. SXDGL - egy pillanatkép alapú megoldás

Az XDGL kutatógárdájának tagjai alkották meg ezt a protokollt is, név szerint Pleshachkov és Kuznetcov érintett létrehozásában. Az SXDGL az elődök alapján szintén különbséget tesz strukturális és tartalmi módosítások közt és figyelembe is veszi az ütközések elkerüléséhez a tartalmi tényeket. Amiben viszont különbözik elődjeitől, az a csak-olvasó és az író-olvasó tranzakciók közti ütközések és a lekérdezések zárolása okozta többletmunka kiküszöbölése a pillanatképekkel megvalósított többverziós működés következtében.

A többverziós működéshez szükséges verzióelérési problémához a szerzők implementációs megoldást is vázolnak, ez a Sedna XML-adatbáziskezelőben is megvalósított tárolómegoldás. Ebben a tárolórendszerben a központi elem a Dataguide,

a Dataguide csúcsainak vannak mutatói az XML dokumentum megfelelő csúcsait tároló lapokhoz. Az azonos Dataguide csúcshoz tartozó lapok egy kétirányú listává vannak összefűzve. A csúcsok strukturális és tartalmi (text) része szét van választva, a strukturális részt egy fix méretű leíró tartalmazza, a tartalmat pedig külön lapok tárolják. A csúcsok leírói közti kapcsolatokat direkt és indirekt mutatókkal (az indirekt mutatókat táblákban tárolva) valósítják meg, itt kiemelten fontos a lekérdezések és a módosítások által preferált mutatók (direkt,indirekt) közti egyensúly fenntartása a rendszer teljesítményének szempontjából.

A verziók kezeléséhez szintén a Sedna-ban megvalósított megoldást vázoljuk. Miszerint is a rendszer minden lapnak legfeljebb 4 verzióját tárolja, valamint a rendszer tárol két logikai pillanatképet (a tranzakciók konzisztenciáját megtartó képek) a verzionálás megvalósításához. A különböző verziók:

- WV (Working Version): munkapéldány, ez egy még nem committált tranzakció által létrehozott verzió.
- LCV (Last Committed Version): az utolsó committált verzió.
- CS: az aktuális pillanatképhez tartozó verzió.
- PS: az előző pillanatképhez tartozó verzió.

Az adatoknak lehet egyszerre több címkéjük is, de ezek nem tárolódnak, hanem dinamikusan állapítja meg őket a protokoll a segédadatokból (aktív tranzakciók listája: *ActList*, a verzió időbélyege, a verzió létrehozójának azonosítója, a pillanatkép időbélyege T_{CS}, T_{PS} és az aktív tranzakciók listájának a pillanatképpel egyidejű állapota $ActList_{CS}, ActList_{PS}$*)*. A verziók meghatározásához még egy (a Sednában a lapok fejlécében tárolt) listát (V) használ a protokoll. A listában egy adott adat verzió metaadatai (időbélyeg, létrehozó azonosítója) vannak csökkenő sorrendben. A verziók meghatározása:

- WV: Ez mindig a legfrissebb, tehát ha létezik, akkor a lista első eleme lesz. Akkor létezik csak, ha a létrehozó azonosítója benne van még az *ActList*-ben.
- LCV: Két lehetőség van: ha létezik a WV, akkor az LCV a lista rákövetkező eleme, különben az LCV a lista első eleme.
- CS: az aktuális pillanatképhez tartozó verzió meghatározásához először létre kell hozni V azon részlistáját V'-t, amely csak a T_{CS} időbélyeg előtti verziókat tartalmazza (ezek a pillanatkép létrehozásakor már létezett verziók). Mivel ebben a listában lehet WV is, így fel kell használni a pillanatkép *ActList*-jét ($ActList_{CS}$) és ez alapján meghatározni az LCV-t.
- PS: CS-hez hasonlóan.

Az adatok hatékony eléréséhez szintén a Sednában megvalósított memóriakezelési megoldást ajánlják a szerzők. Ez a megoldás (lapszintű finomsággal kezeli a verziókat) az adatbázis címterét (**D**atabase **A**ddress **S**pace) rétegekre bontja (a rétegeket lapok alkotják és azok tárolják az XML adatokat) és leképezi azt egy virtuális címtérbe (**V**irtual **A**ddress **S**pace). Így az adatelemek címét a réteg és a rétegen belüli cím fogja alkotni, a rétegen belüli cím lesz az elem címe a virtuális címtérben. Ha nincs bent egy cím a virtuális címtérben, akkor generálódik egy laphiba és betöltődik a lap. Mivel a lap verzióit az utolsó verzión keresztül érjük el, így a lap fejlécében található lista következtében a többi verzió kikeresése már gyors lesz. A megoldás hatására a drágának számító verziókeresés művelet relatíve ritka lesz, így a bemutatott verzionálási séma is kevesebb többletmunkát jelent a rendszer számára.

Az SXDGL protokoll. A protokoll különbséget tesz ugyebár csak-olvasó és módosító tranzakciók közt. Ez alapján szétválasztjuk ezek tárgyalását.

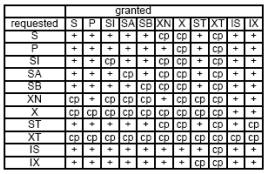
A csak-olvasó tranzakciók (lekérdezések) kezelése:. Mivel ezeket a tranzakciókat speciálisan kell kezelni, a tranzakció-kezelőnek tudnia kell, hogy egy tranzakció lekérdezés vagy sem. A lekérdezések kezelésének kulcsa a két logikai pillanatkép: az aktuális és az előző pillanatkép. Az új lekérdezések mindig az aktuális képet kezdik olvasni, nem zárolnak semmit, mindig a CS/PS verziókat olvassák és nem ütköznek a módosító tranzakciókkal.

A pillanatképek „elöregedése" miatt a tanzakció-kezelőnek rendszeresen új pillanatképeket kell készítenie, ehhez az előző pillanatképet használja fel. A pillanatkép készítésének idején nem dolgozhat lekérdezés az éppen frissített képen, így ha az aktuális pillanatképet használja egy másik lekérdezés, akkor az előző képet frissíti a rendszer az aktuális kép időbélyegével és aktív tranzakció listájával, hogy ezek a tranzakciók tovább futhassanak (ha azt is használja lekérdezés, akkor a pillanatkép létrehozását el kell halasztani). A pillanatkép frissítése amúgy az előbb említett két lépésből áll, az időbélyeg és az aktív tranzakciók listájának frissítéséből.

A módosító tranzakciók kezelése:. Az SXDGL két-fázisú zárolási szabályt alkalmaz a zárak kezelésére (szigorút,mivel a tranzakciók végéig megtartja a zárakat), valamint rövidtávú zárakat az adatok fizikai konzisztenciájának megőrzésére. A zárakat és a logikai predikátumokat a Dataguide csúcsaira helyezi, mint elődjei.

A strukturális zárakon belül itt is két fajta zárral találkozunk, a csúcs és a fa zárakkal. Ezek nevükhöz illően az XML dokumentum csúcsaira, illetve az azok alatt elhelyezkedő részfában található csúcsokra vonatkoznak, a zárak a Dataguide csúcsaira kerülnek és köthető hozzájuk predikátum. A felhasznált zárak:

- **P** (Pass - áthaladási) zár: ezeket a zárakat az ösvény-műveletek használják,



	granted									
requested	S	P	SI	SA	SB	XN	X	ST	XT	IS	IX
S	+	+	+	+	+	cp	cp	+	cp	+	+
P	+	+	+	+	+	cp	cp	+	cp	+	+
SI	+	+	cp	+	+	cp	cp	+	cp	+	+
SA	+	+	+	cp	+	cp	cp	+	cp	+	+
SB	+	+	+	+	cp	cp	cp	+	cp	+	+
XN	cp	+	cp	cp	cp	+	cp	cp	cp	+	+
X	cp	cp	cp	cp	cp	cp	cp	cp	cp	+	+
ST	+	+	+	+	+	cp	cp	+	cp	+	cp
XT	cp	cp	cp	cp	cp	cp	cp	cp	cp	cp	cp
IS	+	+	+	+	+	+	+	+	cp	+	+
IX	+	+	+	+	+	+	+	cp	cp	+	+

9. ábra. SXDGL kompatibilitási mátrix (forrás: [19] 3.ábra)

ilyen zárak kerülnek az ösvény köztes (*intermediate*) csúcsaira. A P zár megengedi a köztes csúcsok listájába tartozó csúcsok beszúrását, de tiltja azok törlését.

- **S** (Shared - osztott) zár: ezt a zárfajtát is az ösvény-műveletek használják, ilyen zár kerül az ösvény által meghatározott cél (*destination*) csúcsokra.

- **S**hared**I**nto (**S**hared**A**fter és **S**hared**B**efore hasonlóan) zár: Az *InsertInto* művelet célcsúcsaira kerül ez a zár, hatására nem lehet módosítani a beszúrt csúcsokat és beszúrni a zárolt *csúcsokba.*

- **XN** (e**X**clusive **N**ew - új kizáró) zár: az újonnan létrehozott csúcsokra kerül ez a zár, ami tiltja a zárolt csúcsok módosítását, de engedi az áthaladást ezeken a csúcsokon.

- **X** (exclusive - kizáró) zár: azok a műveletek használják ezt a zárat, amelyek a dokumentum belső csúcsait módosítják. A zár tiltja konkurens olvasást és módosítást egyaránt.

- **ST**(shared tree - osztott fa) és **XT** (exclusive tree - kizáró fa) zárak: mint nevük is mutatja ők a protokoll által használt fa zárak, azaz a teljes részfákat (a zárolt csúcs összes leszármazottját) zárolják osztott (minden módosítást gátló) vagy kizáró (minden olvasást és módosítást gátló) módban.

- **IS** (intention shared - osztott szándék), **IX** (intention exclusive - kizáró szándék) zárak: ezeket a zárakat az osztott és kizáró zárak kiosztásakor kell megszerezni a zárolandó csúcsok ősein, a durvább finomsági szinteken (a hierarchiában magasabban) elhelyezett zárakkal való ütközések elkerülése végett.

A zárak kompatibilitási táblázata (9.ábra) nem tartalmaz direkt ütközéseket, csak a létező predikátumok kielégíthetetlensége esetén áll fent ütközés.

A fantom problémát (olyan módosítások végrehajthatóságát, amelyek megváltoztatnák a már végrehajtott műveletek eredményét) kiküszöbölni hivatott logikai zárak az XDGL-nél is ismertetett L (logikai) és IN (insert new - új beszúrás) zárak. Ezek a zárak gyakorlatilag tulajdonságok halmazai, ahol a tulajdonságok logikai feltételek a csúcsra nézve. A logikai zárak tiltják a zárban rögzített tulajdonságokkal

rendelkező csúcsok beszúrását. Az IN zár a beszúrásra kerülő új csúcs tulajdonságait tartalmazza és a beszúrandó csúcs minden ősére meg kell szerezni, az ütközések elkerüléséhez. A módosító események csak akkor vezetnek fantomokhoz a Dataguide alapú protokolloknál, ha a Dataguide-ban új út jelenik meg, amire ugyebár előzőleg nem tehettünk zárakat, ezt a problémát orvosolja a két imént ismertetett zár. Az IN és L zárak ütközésének ellenőrzéséhez a következő tulajdonságokat kell vizsgálni: új-csúcs-szülőjének-neve,új-csúcs-neve,új-csúcs-értéke. Ütközés áll fenn ha:

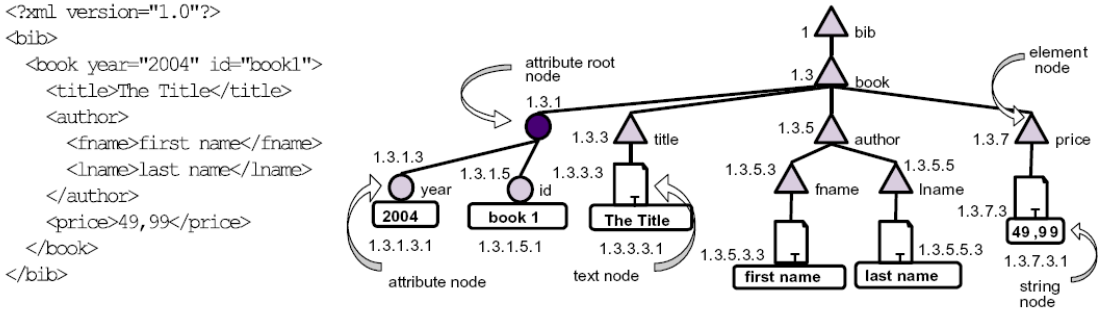
- Olyan nevű csúcsot akarunk beszúrni, amelyek neve szerepe már egy logikai zárban, aminek nincs más feltétele.
- A beszúrandó csúcs neve és értéke is egyezik egy már kiadott logikai zárban lévővel.
- A beszúrandó csúcs mindhárom tulajdonságára van egyezés egy létező logikai zárban.

A protokoll említést tesz még a DTD-ben szereplő sorrend kihasználásáról a feleslegesen szigorú fa zárak okozta ütközések elkerüléséhez, de ennek megoldását nem mutatja be részleteiben.

Az eddig ismertetett zárak és működési elvek még sajnos nem hozzák a kívánt eredményeket, ugyanis ha két aktív tranzakció ugyanazt az adatot módosítaná előfordulhatna, hogy egy adatelem legfrissebb verziója folyamatosan nem committált tranzakcióhoz kötődne. A probléma kezeléséhez bevezetjük a tranzakció függőségi gráfot (***T****ransaction* ***D****ependency* ***G****raph*), ami egy irányítatlan gráf, csúcsai a módosító tranzakciók, él pedig akkor vezet egyik csúcsból a másikba ha (1) mindkét tranzakció aktív és (2) az egyik olyan elemet akar módosítani, amelyet a másik már módosított. (1) és (2) hatékony ellenőrzéséhez a zár-kezelő segítségét vesszük igénybe. Mivel minden módosító tranzakciónak kell a módosítandó lapra egy X (rövid)zárat szerezni, miután az X zárat elengedi a módosító a tranzakció-kezelő átkonvertálja azt egy C zárrá, ami minden zárral kompatibilis és a C zárat a tranzakció committálásakor vesszük le a lapról. Ezt kihasználva (2)-est módosíthatjuk arra, hogy ellenőrizzük, hogy van-e C zár a módosítani kívánt lapon. Ezen kívül minden módosító tranzakciónak el kell végezni a következő műveleteket committálás előtt:

1. Ha benne van a tranzakció a TDG-ben, akkor a csúcsot P (prepared - kész) jelöléssel kell ellátni.
2. Ha a tranzakció nincs benne a TDG-ben, akkor el kell távolítani az *ActList*-ből.

Így a TDG-ben nem szerepő tranzakciók módosításai a befejeződésük után láthatóvá válnak a pillanatkép(ek)ben a lekérdezések számára. A TDG-ben szereplő tranzakciók módosításai továbbra sem elérhetőek. A TDG-ből viszont törölhetőek azok a



10. ábra. XML átalakítása taDOM fává (forrás: [9] 2.ábra)

részgráfok, amelyek csupa P-vel jelölt csúcsból állnak és ekkor már ezeket a csúcsokat is el lehet távolítani az aktív tranzakciók listájából. A TDG végtelen növekedésének megakadályozásához, ha a gráf túlnő egy határértéken, akkor a tranzakció-kezelő megváltoztathatja az X és C zárak kompatibilitását, aminek hatására a TDG egy idő után üressé válik és ezután visszaállíthatja a zárakat az alap kompatibilitásra. Ez biztosítja azt is, hogy a módosító új verziót fog létrehozni és nem az utolsó verzión hajt végre módosításokat.

A protokoll helyességbizonyítása és a DGLOCK-kal történt összehasonlítás mérési eredményei [19]-ben megtalálhatóak. A protokoll kapcsán megemlítendő, hogy a pillanatképek karbantartásának (frissítésének) következtében előfordulhat, hogy egyes lekérdező tranzakciók, nem a legfrissebb (de mivel a pillanatképek konzisztensek, ezért konzisztens) adatokat látják.

3.2.11. A taDOM protokollok

Ezek a protokollok a DOM API által biztosított műveletekhez lettek kialakítva, de az ezen műveletek támogatásához szükséges megoldások ötvözve a taDOM tárolási struktúrával és az XTC tranzakció-kezelővel, amit szintén a taDOM protokollok szerzőinek nevéhez köthetünk, a manapság létező XML tranzakció-kezelési megoldások között igazán jó pozíciót biztosítanak ennek az irányvonalnak.

A taDOM struktúra a szabványos DOM struktúrában létező elem, attribútum, text csúcsok mellé bevezet egy attribútum gyökér csúcsot és egy string csúcsot (10.ábra).

Ezek a csúcsok természetesen csak a zár-kezelő által látott logikai reprezentációban jelennek meg, a felhasználó számára nem jelent látható változást a DOM struktúrához képest. Az új attribútum gyökér csúcs megjelenik minden elemnél és az elem attribútumai nem közvetlenül az elemhez fognak ezután csatlakozni, hanem az attribútum gyökér csúcshoz, ez a megoldás a gyakorlati oldalon a *GetAttributes* hívásoknál jelent előrelépést, hiszen nem kel zárolni az összes attribútumot csak az attribútum gyökeret. A másik új csúcsfajta a *string* csúcs az XML dokumentumok-

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	+	+	+	+	+	+	+	-	-
NR	+	+	+	+	+	+	+	-	-
LR	+	+	+	+	+	+	-	-	-
SR	+	+	+	+	+	-	-	-	-
IX	+	+	+	+	-	+	+	-	-
CX	+	+	+	-	-	+	+	-	-
SU	+	+	+	+	+	-	-	-	-
SX	+	-	-	-	-	-	-	-	-

	-	IR	NR	LR	SR	IX	CX	SU	SX
IR	IR	IR	NR	LR	SR	IX	CX	SU	SX
NR	NR	NR	NR	LR	SR	IX	CX	SU	SX
LR	LR	LR	LR	LR	SR	IX_{NR}	CX_{NR}	SU	SX
SR	SR	SR	SR	SR	SR	IX_{SR}	CX_{SR}	SR	SX
IX	IX	IX	IX	IX_{NR}	IX_{SR}	IX	CX	SX	SX
CX	CX	CX	CX	CX_{NR}	CX_{SR}	CX	CX	SX	SX
SU	SU	SU	SU	SU	SU	SX	SX	SU	SX
SX	SX	SX	SX	SX	SX	SX	SX	SX	SX

11. ábra. taDOM2 kompatibilitási (jobb) és konverziós mátrix (bal) (forrás: [9] 3.ábra)

ban megszokott attribútum és text csúcsok értékeit hivatott tárolni és természetesen ez sem látható a felhasználók számára, ők továbbra is attribútum és text csúcsokat módosítanak, viszont a zár-kezelő implicit zárolja ezeket a csúcsokat ha az értékeiket olvassák vagy módosítják. Ez a megközelítés megfelel a DOM API-ban használatos elem (*GetElementById*,...) és értéklekéréseknek (*getValue*).

A protokoll működése során a zárak elhelyezését a taDOM fán egy zár-kezelőnek kell megoldani. A konkurens tranzakciók kiszolgálásához a protokoll navigációs és logikai zárakat is alkalmaz, a dinamikus zárolási finomság megvalósításához konverziós szabályokat is definiálunk. A taDOM protokollok nevei tükrözik az általuk támogatott DOM szabványt is, ezért létezik taDOM2 és taDOM3 protokoll is valamint mindkét alapprotokollnak létezik egy bővített taDOM*+ változata is.

A protokollok által használt zárolási módok a hierarchikus protokolloknál használt (R, X, U, IR, IX) zárakhoz hasonlóak, azok helyébe lépnek a megfelelő finomságú zárolások megvalósítása érdekében.

A taDOM2 protokoll által használt zárak kompatibilitási és konverziós mátrixa a 11.ábrán látható.

A zárak leírása:

- IR (*intention read* - olvasási szándék): jelentése, hogy valahol az alatta lévő részfában olvasni akarunk egy csúcsot (mint a hierarchikus protokollnál)

- NR (*node read* - csúcs olvasás): az olvasni szándékozott csúcsra kell ilyen zárat szereznünk. A csúcshoz vezető úton lévő őseire pedig IR zárat kell szerezni. Az NR zár csak azt a csúcsot zárolja, amelyre megszerezték, a leszármazottait nem.

- LR (*level read* - szint olvasás): egy csúcsot és a közvetlen gyerekeit zárolja. Hasznos a gyerekek,attribútumok lekérdezésekor, mert így nem kell mindre

külön NR zárat tenni.

- SR (*subtree read* - részfa olvasás): egy adott csúcsot és az alatta elhelyezkedő részfát zárol osztott hozzáféréssel.

- IX (*intention exclusive* - kizáró szándék): a hierachikus protokollhoz hasonlóan a zárolt csúcs alatti részfában végrehajtani szándékozott módosítást jelöli, de nem a közvetlen gyerek csúcsokon végrehajtandót (az a CX lesz).

- CX (*child exclusive* - gyerek kizáró): azt jelöli, hogy valamelyik közvetlen gyerek csúcson van már SX zár, ezáltal SR és LR már nem adható ki. Azonban másik CX zár még kiadható másik gyerekre.

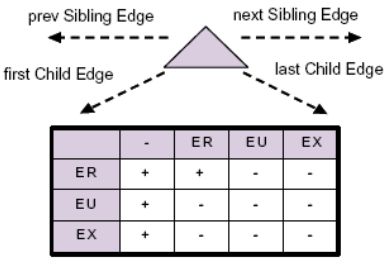
- SU (*subtree update option* - részfa módosítási lehetőség): lehetőséget biztosít a zárolt csúcs olvasására és ha szükséges a későbbiekben a módosítására is. Visszakonvertálható SR zárrá vagy ha szükséges és nincs több olvasási zár a csúcson, akkor kiterjeszthető SX zárrá. Az SU és (IR,NR,LR,SR) zárak kompatibilitása is úgy van kialakítva, hogy a kiéheztetést próbálja elkerülni.

- SX (*subtree exclusive* - részfa kizáró): a módosítani (tartalmi módosítás vagy törlés) kívánt csúcsra kell ilyen zárat rakni, szülőjére kell egy CX zár és a dokumentum gyökeréig felfelé kellenek az IX zárak.

Ezek a zárolási módok részben a hierarchikus protokoll zárolási módjaiból származtathatók, részben pedig a DOM-alapú konkurencia-kezelés támasztotta igényeket hivatottak kielégíteni. Egyes zárolási módok egymással való hasonlóságuk ellenére is szükségesek a finomabb zárolás megvalósításához, ezekre az esetekre magyarázó példák találhatóak [9]-ben.

A taDOM protokollok által használt zár átalakításokról pár szóban. A zárak konvertálásának szükségességét az egy tranzakció által, egy elemre igényelt zárak számának minimalizálása indokolja leginkább, mivel a nagyobb zár-listák bonyolultabb ellenőrzéseket és több erőforrást igényelnek a rendszertől. A zárak átalakításakor mindig megpróbáljuk a létező zárat olyanra cserélni, amely mind a létező zár által biztosítandó, mind az igényelt zár által biztosítandó tulajdonságokat biztosítani tudja. A konverziós szabályok a 11.ábra jobb oldalán található konverziós mátrixban találhatóak (sor: ami megvan, oszlop: amit igényel).

A csúcs zárakon felül szükség van az úgynevezett navigációs zárakra is. Ezek a zárak kell biztosítsák a tranzakcióknak, hogy az általuk bejárt utak nem változnak meg a tranzakció befejeződése előtt, azaz például az egyik csúcsból kiindulva bizonyos fix lépéseket végrehajtva a tranzakció teljes élete során azonos csúcsba jut (feltéve természetesen, hogy ő maga nem módosított az útvonalon). Ennek a feltétel-



	-	ER	EU	EX
ER	+	+	-	-
EU	+	-	-	-
EX	+	-	-	-

12. ábra. Virtuális élek a taDOM fában (forrás: [9] 5.ábra)

nek a biztosításához a taDOM fában csúcsokon kívül a hozzájuk tartozó virtuális navigációs éleket is zároljuk (12.ábra).

A tranzakcióknak a csúcsokra megszerzendő zárakon kívül, a csúcsokat összekötő virtuális élekre is zárakat kell igényelni és ezeket az éleket logikai voltuk miatt úgy kell zárolni, hogy a bejárást ne csak egy irányból védjék, hanem mindkét irányból és figyelni kell az első és utolsó gyerek mutatókat is. A bejárások támogatásához 3 zárolási módot használ a taDOM2:

- ER (*edge read*): olvasási célú bejárások esetén szükséges
- EX (*edge exclusive*): él módosítását teszi lehetővé (beszúrás, bővítés, törlés), a művelet által érintett minden élre meg kell szerezni.
- EU (*edge update option*): az SU zárhoz hasonlóan a holtpontok előfordulási esélyét csökkenteni hivatott a módosító tranzakciók közt.

A navigációs zárak segítenek a fantom probléma elkerülésében is, mivel védik a bejárt utakat az új beszúrásoktól.

Az eddig ismertetettek akalmassá teszik ugyan a taDOM2 protokollt a DOM 2 által meghatározott műveletek támogatására, de a csúcs átnevezése (renameNode) művelettel már meggyűlik a baja (ez DOM 3 szabvány). A kompatibilitás érdekében egy új csúcsfajtát vezettek be a szerzők, a virtuális név csúcsot (*virtual name node*), minden csúcshoz a taDOM fában csatoltak egy ilyet, ezzel oldva meg az átnevezett csúcs részfáját nem érintő kizárásos zárolást. Egy belső csúcs ilyenféleképp megoldott zárolásához egy SX zárat helyeznek a virtuális név csúcsra és egy-egy CX-et a módosítandó és a szülő csúcsra.

A taDOM2 protokollnak még egy hátránya van. A konverziós mátrixban alsóindexszel ellátott esetek olyan zárak kiosztását okozzák, amelyek kiosztásához az XML dokumentumot kell olvassuk a zárolandó csúcsok megállapításához. Ezt a problémát oldották meg a taDOM2+ protokollban további 4 zár bevezetésével, melyeket a kérdéses esetekben használ a zár-kezelő.

A taDOM3 protokoll tovább erősítette ezt a vonalat egy új zár bevezetésével, amely lehetővé teszi a belső csúcsok kizáró zárolását a részfa befolyásolása nélkül.

Ennek a zárnak a bevezetése módosulásokhoz vezet a kompatibilitási mátrixban is ugyan, de a taDOM2 és 2+ esetén bevezetett virtuális név csúcsok használatától megszabadítja rendszert, ami egyben azt is jelenti, hogy csúcsonként újra csak egy zárat kell fenntartani 2 helyett, ami a virtuális név csúcsok miatt alakult ki. A virtuális név csúcsoktól megszabadulva újra a konverziós mátrix felé fordulunk, amely a taDOM3 esetében is tartalmaz olyan konverziókat, amelyek miatt olvasnunk kéne az eredeti dokumentumot. A megoldás a 2+-hoz hasonlóan új zárolási módokkal érkezik, méghozzá 8 újabbal. (A 2+, 3, 3+ protokollok kompatibilitási és konverziós mátrixainak változásai [9] 7.-8.-9.-10.-11.ábráján megtalálhatóak.)

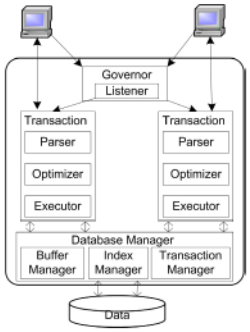
A taDOM család tagjainak komplexitása a fejlődésük során ugyan nőtt, de ez a teljesítményüket a megfelelő irányba mozdította el, ennek bizonyítéka a fejlesztői által az XTC-n futtatott tesztek eredménye, melyet a taDOM protokollok nyertek (részletek [9] 5.szakasz).

A taDOM család kapcsán nem említettük a hierarchikus protokolloknál megszokott zár kiterjesztési megoldást, de mivel ez a protokoll is hierarchikus alapokkal rendelkezik, természetesen itt is lehetőség van a zárolási mélység meghatározására, ami a legfontosabb befolyásoló tényezője ezen protokollok teljesítményének. A dinamikus zár kiterjesztések kezelését a tranzakció-kezelő feladata megoldani és a rendszert beállító szakemberek feladata a megfelelő zárolási mélységet (és ennek függvényében a lehetséges konkurencia mértékét) beállítani vagy a megfelelő határértékeket megadni (például: mekkora zármennyiség felett kell elkezdeni a zárak finomságának csökkentését) a zár- és tranzakció-kezelő optimális működéséhez.

3.3. Natív XML adatbáziskezelők

Az előző szakaszban ismertetésre került egy tucat kimondottan az XML adatok struktúrájához készült protokoll, némelyiknél meg is említettük, hogy a megoldás megvalósításra is került valamely natív XML adatbáziskezelőben. Most olyan minta implementációkról lesz szó, amelyek a fent említett protokollok valamelyikét használják és általában valamely kutatási projekt részeként, gyakran kimondottan az adott protokoll tesztelése céljából kerültek lefejlesztésre. Ezen implementációk előnye a ORDBMS-ekre készített XML kezelőkkel szemben, hogy egész a tárolási szintig az XML adatok szerkezetéhez vannak igazítva és nem csak a felhasználóknak biztosítanak valamely XML kezelési felületet.

A bemutatott protokollok sokszor csak valamiféle tesztkörnyezetben lettek kipróbálva és nem szerves részei adatbáziskezelőknek, sőt a létező implementációk közül sem mindegyik bír egy teljes értékű kereskedelmi rendszer erejével, így a bemutatásuk sem lehet olyan teljeskörű mint szeretnénk. Megjegyezném, hogy az XML adatbáziskezelőknek nem is célja a létező relációs adatbáziskezelők leváltása, céljuk alapvetően



13. ábra. A Sedna felépítése (forrás: [5] 1.ábra)

az XML adatok kezelésének terén felülmúlni a relációs megoldásokat.

A továbbiakban két erősen különböző megközelítést alkalmazó protokollhoz köthető megvalósítást nézünk meg részletesebben, a Dataguide alapú SXDGL-lel összefonódott Sedna [5] rendszert és az előző szakasz utolsó protokollcsaládjával „együttélő" XTC szervert.

3.3.1. Sedna

A Sedna rendszer Kuznetcov - ő részt vett az XDGL és SXDGL protokollok létrehozásában is - és munkatársai nevéhez fűződik. A Sedna az XQuery-re épít, tehát az itt bemutatásra kerülő megoldások, ennek a nyelvnek az adatmodelljéhez vannak kialakítva. Mivel a rendszert egy teljesértékű adatbáziskezelőnek szánják és az XQuery 1.0 szabványa nem tartalmaz módosító műveleteket, ezért az XUpdate módosító nyelvet használják kiegészítésként a megvalósításban.

A Sedna felépítése (13.ábra) eléggé szabványosnak mondható.

A *governor* (irányító) a rendszer „vezérlő pultja", ez folyamatosan tudja, hogy melyik adatbázis és azon milyen tranzakciók futnak és irányítja őket. A *listener* (hallgató) figyeli a klienseket és létrehoz számukra egy-egy példányt a *tranzakció* komponensből, valamint megteremti a közvetlen kapcsolatot a tranzakció komponens és a kliensek közt. Innentől kezdve a kliens folyamata (*session*) a tranzakciós komponens felügyelete alá kerül, amiben megtalálható a *parser* - szintaktikai elemző, az *optimizer* - optimalizáló és a végrehajtó egység (*executor*). A parser fordítja le a lekérdezést annak logikai reprezentációjára, ami egy műveleti fa. Az optimalizáló a logikai reprezentációból elkészíti a fizikai adatszerkezetre vonatkozó alacsony szintű műveletekkel az optimalizált végrehajtási terv műveleti fáját. A végrehajtási tervet a végrehajtó egység értelmezi. Az adatbázis-kezelő példányaiban egy adatbázis és a hozzá tartozó adatbázis-szolgáltatások találhatóak, a puffer-,index- és tranzakciókezelők.

A Sednában szabályalapú lekérdezés optimalizálások vannak egyenlőre csak megvalósítva, de ezeknek elég széles skálája, a költség-alapú optimalizálások még nem

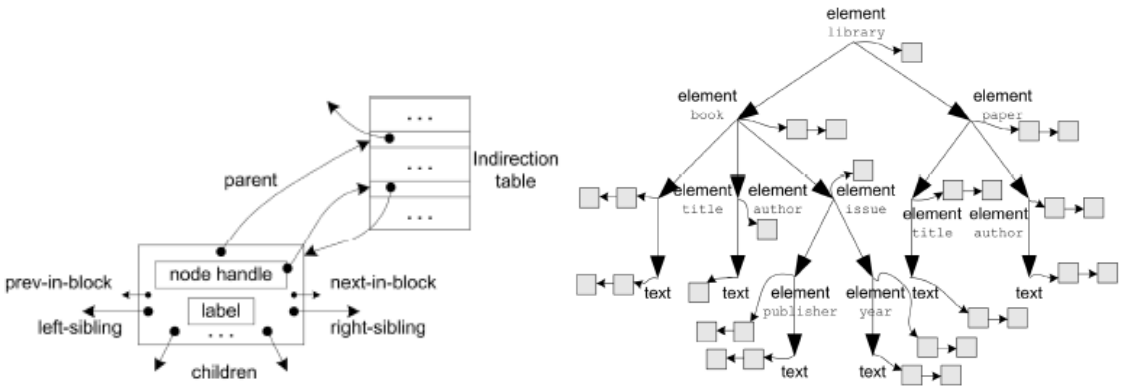
kerültek implementálásra. Itt ezeket nem részletezném, részletesebb információ [5] 2.3-as részében található.

A Sedna több felhasználó számára is hozzáférést biztosít a tárolt adatokhoz (ezt nem mindegyik minta implementáció teszi lehetővé, hiszen a tesztekhez csak párhuzamos tranzakciókra van szükség), viszont a forrásul szolgáló cikk megírásának idején még csak egy két-fázisú lock protokoll volt implementálva a konkurenciakezelő számára (az SXDGL-t 2007-ben publikálták és abban a cikkben már azt állítják, hogy a Sednában is implementálták, ez most azért nem kizáró információ számunkra, mert a tároló rendszer implementációja nem vagy csak kicsit változhatott és ez most fontosabb szempont számunkra a gyors hozzáférések és módosítások megvalósításának vizsgálatakor).

A Sedna tároló rendszere. A Sedna adattárolási megoldásának két fontos döntés szolgáltatja az alapját. Az egyik a csúcsok közti kapcsolatok megvalósítása mutatókkal, a másik a dokumentum csúcsainak tárolása a dokumentum tényleges tartalmának függvényében (tehát nem a DTD, hanem a Dataguide lesz a tárolás alapja). A mutatókhoz elöljáróban annyit, hogy az alapelképzelésben szereplő *direkt* mutatók nem alkalmasak a módosítások hatékony támogatására, így végleges megvalósítás a direkt mutatók minimalizálását tartja majd szem előtt és az XML dokumentumban lévő szülő, gyerek, testvér kapcsolatok közül csak a szülő kapcsolat indirekt mutatókkal lesz ábrázolva.

A Dataguide alapú tárolás mellett szólnak az így kapott séma tulajdonságai, úgy mint: pontosabb mint a DTD, minden út ami megtalálható a dokumentumban, ahhoz létezik pontosan egy út a Dataguide-ban és a Dataguide-ban lévő utak mindegyike egy a dokumentumban is jelenlévő utat reprezentál, a kapott Dataguide mindig fa lesz, valamint ez a tárolási forma alkalmazható olyan dokumentumokra is, amelyekhez nincs megadva a DTD. A tárolás központi eleme tehát a Dataguide lesz (14.ábra jobb oldal). A Dataguide csúcsait ellátjuk a megfelelő elemfajta címkével (elem, attribútum, szöveg,...), ahol szükséges ott név címkével is és az adott sémacsúcsnak lesz egy mutatója is a hozzá tartozó csúcsok adatait tartalmazó blokkokhoz. Az egy sémacsúcshoz tartozó adatblokkok mutatókkal egy kétirányú listába vannak szervezve. A blokklistában szereplő csúcsleírók részbenrendezettek a dokumentumsorrendnek megfelelően, azaz a blokkokon belül nincsenek sorbarendezve (fizikailag) a leírók, ez gyorsítást jelent a módosítások végrehajtásánál, mivel nem kell a blokkokon belüli rendezésekre annyi időt fordítani.

A tárolási modell szétválasztja a csúcsok szerkezeti és szöveges érték részét. A szöveges érték a szöveg csúcsok tartalma, az attribútumok értékei, stb. A szöveges értékek „szépségét” a változó hossz jelenti, ezt a *slotted-page* () szerkezeti megoldással kezelik, ami egy ismert és kimondottan a változó hosszúságú adatok kezelésére



14. ábra. Csúcsleírók (bal) és adatszervezés (jobb)

kidolgozott technika. A szerkezeti részt (a csúcsok kapcsolatait) a már említett csúcsleírók tartalmazzák (14.ábra bal oldal).

A csúcsleíróban található *label* - címke egy számozási séma szerinti címkét tartalmaz, amiből megállapítható a csúcsok egymáshoz való viszonya (pl.: szülő-gyerek kapcsolat) és az ilyen címkézési módszer lehetőséget ad a beszúrások egyszerűbb végrehajtására is.

A *next-in-block, prev-in-block* (következő, előző a blokkban) mutatók a dokumentum szerinti sorrend fenntartását biztosítják, a *left-,right-sibling* (bal-,jobb-testvér) mutatók a testvéreket, a *children* (gyerekek) mutatók közül nem tartalmazza mindet a leíró, hogy nehogy kinője a blokkméretet (a fix leíróméret döntő fontosságú a tárterület kezeléséhez), ezek közül minden gyerek-elem fajtából csak az első gyerekre mutató mutatót tartalmazza a leíró (a többi már úgyis láncolva van egymáshoz, ha több van).

A fix leíróméret azonban problémákat is okozhat új beszúrások alkalmával (pl.:amikor egy újonnan beszúrt gyerek mutatója már nem férne el a leíróban), ezért a feltételt úgy módosítjuk, hogy a leírók csak blokkon belül rendelkezzenek azonos mérettel (Ilyenkor a blokkban lévő leírók által tárolható gyerek-mutatók számát jelölik a blokk fejrészében). A direkt mutatók további problémát jelenthetnek az olyan esetekben, amikor egy beszúrás több adatátszervezéssel jár, ilyenkor túl sok direkt mutatót kéne módosítani, ennek enyhítésére vezették be, hogy a szülő mutatókat indirekt mutatókkal és indirekciós táblákkal valósítják meg.

A *node handle* (csúcs cím) egy szintén fontos része a koncepciónak. Ezzel a mutatóval lehet hivatkozni a csúcsra a teljes élettartama alatt. Ahhoz hogy ez hatékony legyen, ennek a mutatónak változatlanak kell maradni mindvégig. Így ez a mutató az indirekciós táblának azon sorára fog folyamatosan mutatni, ahol a csúcs címét tároljuk.

A direkt mutatók hatékony kezeléséhez a készítők létrehoztak egy 64bites címeket támogató memória-kezelést. A megoldás azon alapul, hogy az adatbázisban tárolt

adatokat (*Sedna Address Space*) rétegekre bontják (ezek a rétegek az adatokat tartalmazó lapokból állnak) és ezeket a rétegeket a folyamat rendelkezésére álló memóriaterületen (*Process Virtual Address Space*) belül maguk kezelik. A 64 bites címek első 32 bitje a réteget azonosítja be, a következő 32 pedig a rétegen belül az objektumot. A teljes címtér rétegekre bontása után nem használnak további cím leképezéseket, az objektumok beazonosításához egyszerűen ellenőrzik a réteg számát és a rétegen belüli címet (a virtuális címtér természetesen le van képezve az operációs rendszer által is használt memóriába, de ennek kezelése már a puffer-kezelő feladata). Az adatok elérését a szokványos lapozási technikáknak megfelelően oldották meg, azaz ha egy cím nincs benne a PVAS-ben, akkor laphiba generálódik és betölti a megfelelő lapot a rendszer.

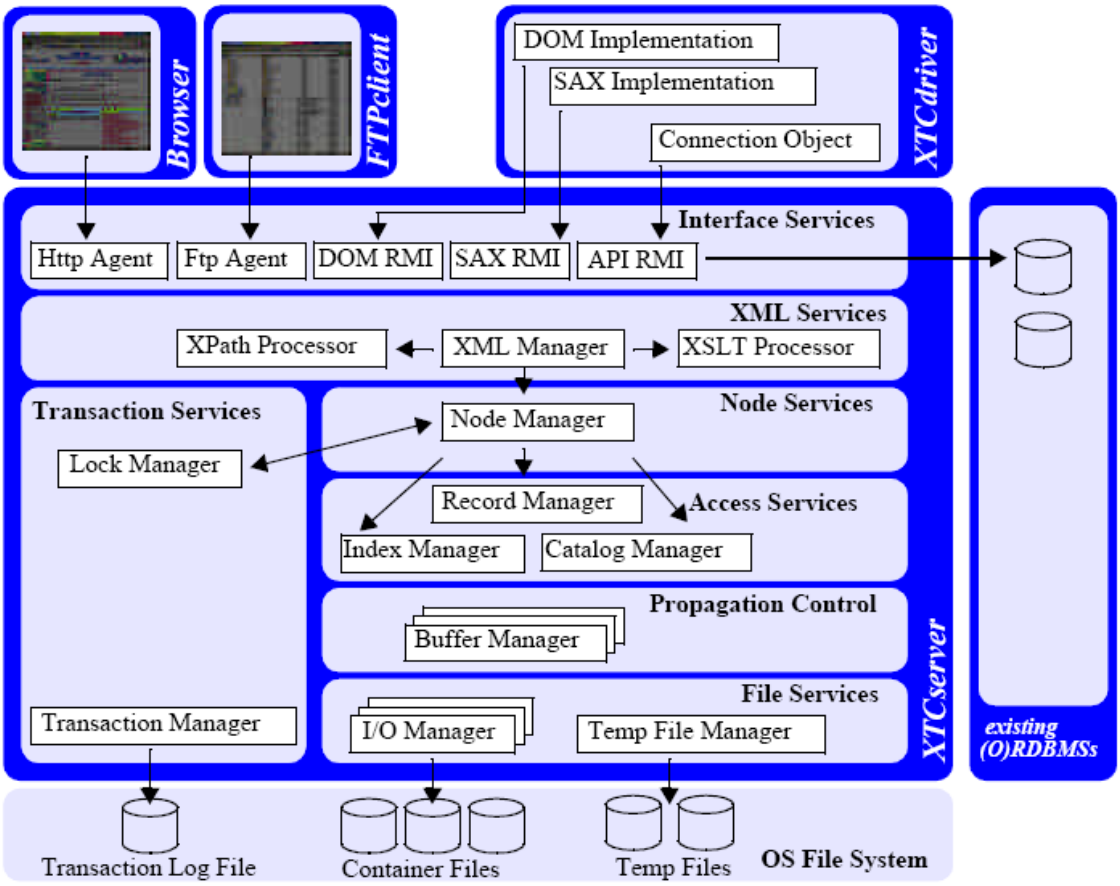
Ezzel a memória-kezelési módszerrel az adatbázis méretére vonatkozó korlátok „megszűnnek" és a mutató dereferálás egyszerűvé válik (nem jelent sokkal több munkát az egyszerű mutatókhoz szükségesnél).

A rendszer teljesítményét befolyásoló következő szint, a lekérdezések végrehajtása. Mivel a lekérdezések, eredményhalmazuktól függetlenül, alapvetően nagy mennyiségű adat megmozgatásával járnak, ezért létfontosságú a megfelelő technikák alkalmazása a beérkező lekérdezésekre, hogy a rendszer válaszideje elfogadható mértékű maradjon. Így a Sedna rendszer is különböző módszereket alkalmaz a lekérdezések kiértékelésére. Ezeket nem tárgyaljuk most részletesen, mindössze említés szintjén szólunk róluk pár szót.

Az elem konstruktorok kezelését, amely az XQuery egy kimondottan erőforrásigényes művelete, a szerzők egy *suspended* (felfüggesztett) konstruktor bevezetésével oldják meg. A felfüggesztett konstruktor hatására az olyan esetekben, amikor a lekérdezés által visszaadott elemek belső tartalmát végül nem vizsgálja semmilyen művelet, csak egy mutatót ad vissza a rendszer az eredeti csúcsra, így elkerülhető az adott elem intenzív adatelériseket igénylő teljes mélységű másolása.Az XPath lekérdezéseket a tárolási modell alapjául szolgáló Dataguide segítségével hatékonyan tudja támogatni a rendszer, ebben az esetben az érintett adathalmaz szűkítését célzó lépések sorrendjével tudnak leginkább kísérletezni. Végül az XQuery programozási nyelvként történő támogatásához a szerzők adaptáltak egy a relációs rendszerekhez kialakított általános iteratív lekérdezés végrehajtási modellt, amely megpróbálja minden lekérdezésnél megtalálni a megfelelő egyensúlyt a műveletek (függvények és más programegységek is szóba jöhetnek) eredményének és szövegének felhasználása közt.

3.3.2. Az XTC szerver

Az XTC szerver a taDOM struktúrával és a taDOM protokollokkal egybefonódva jelöl igazán egy fajsúlyos egységet az XML adatok kezelésére kialakított tesztkörnyezetek



15. ábra. Az XTC szerver felépítése (forrás: [8] 2.ábra)

és protokollok résztvevői közt. Eredetileg ez a megoldás is tesztkörnyezetként kezdte pályafutását a taDOM protokollokhoz, de mostanra kinőtte már ezt a fázist és szerkezete is komplexebb funkcionalitás elérését célozza meg. Alapvetően itt is az XML adatok hatékony (relációs megoldásokon túlmutató) kezelésének elérése a cél, de az XTC eredeti koncepciójában is benne van már a relációs adatbázis-kezelőkkel történő kapcsolat létrehozása az esetleges hibrid működés támogatásához. Most az XTC szerkezeti felépítése és elsősorban az XML adatok tárolásához kialakított megoldása lesz vizsgálatunk tárgya.

Az XTC-t a hagyományos ötrétegű adatbázis szerkezetnek megfelelően tervezték, felépítésének váza a 15.ábrán látható.

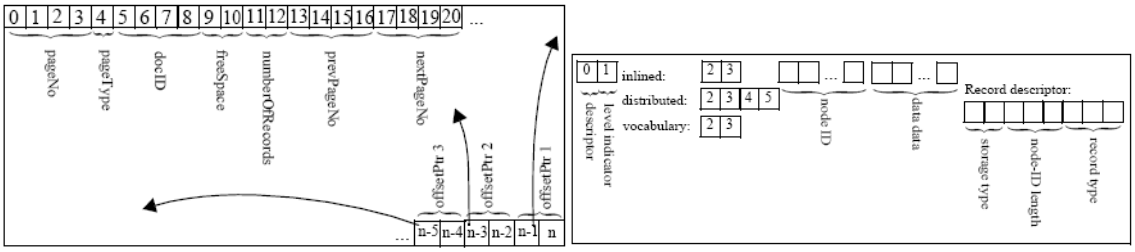
A szerverrel történő kommunikációért a *interface* réteg felelős, az *XTCdriver* felelős (ehhez fejlesztettek egy parancssor feldolgozót is, amire tekinthetünk akár az XTC API-jaként is) a kliensek kapcsolatainak kiépítéséért és az XTCdriver által továbbított lekérdezéseket az *API RMI* szál kezeli. Az API RMI elemzi a beérkező parancsokat és az SQL-es részt és az XML-es részt a megfelelő irányba továbbítja. Az *XTCengine* felelős a natív XML feldolgozásért és az interfész szolgáltatásokon keresztül kapcsolódó kliensek kiszolgálásért. Ezeken kívül az XTCserver lehetőséget

biztosít a szerveren tárolt XML dokumentumok elérésére ftp és/vagy http kapcsolaton keresztül (ehhez a megfelelő szolgáltatását kell csak aktiválni).

Az ötrétegű adatbáziskezelő rétegeinek bemutatásával folytatjuk az XTC-vel történő ismerkedést. Az első legyen is mindjárt az „alsó", tehát a fizikai bitsorozatokhoz legközelebb álló. Ennek a rétegnek a neve *file services* (fájl szolgáltatások, fájlkezelés). Ezt a réteget az *io-manager*, *temporary file manager* valósítja meg együttműködve az operációs rendszerrel. Minden io-vezérlőnek van egy *konténer fájlja*, ezen hajt végre fix blokkméretű olvasási és írási műveleteket egy felettes réteg számára. A blokkméretek konténerfájlonként különbözhetnek, ezt úgy állítja be a rendszer, hogy az XML objektumok lehetőleg minél kevesebb töredezettséget okozzanak, tehát se a kicsi se a nagy méretű objektumok ne okozzanak túl sok problémát. A konténer első blokkja (index blokk) tárolja az adott konténer blokkméretét 2 bájton - ebből ki tudja számolni a fájlt megnyitó io-vezérlő a benne lévő blokkok számát - és tartalmaz(hat) további adminisztrációs adatokat (AD) is. Ha a konténerfájl betelik, akkor dinamikusan bővíthető, így nincs szükség már lefoglalt blokkok áthelyezésére.

A következő réteg a *propagation control* (terjesztés irányítás), itt a *buffer manager* lesz a megvalósító, ami címtér - fizikai memóriára leképezett -fix méretű lapjait fogja szolgáltatni a felettes rétegeknek. Minden io-vezérlőhöz tartozik egy puffer kezelő is, ez inicializáláskor lekérdezi az io-vezérlőtől a blokkméretet és a beállított puffer méret alapján lefoglalja a puffer memóriát a folyamat számára. Az inicializálás után pedig ki tudja szolgálni a magasabb rétegekből érkező lap kéréseket. A lap igény egy 4 bájtos lapszám, az első bájt a konténer azonosítója, a többi 3 a blokkszám. A kérés hatására a puffer kezelő megpróbálja megtalálni a fájlt a memóriában - ebben segíti egy hash tábla,aminek kulcsai a lapszámok -, ha nem találja, akkor a szokásos lapozási technikákat használva - alapértelmezetten az LRD-V2 lapcsere algoritmust használja az írás-olvasás műveletek minimalizálásához, de ezt akár io-vezérlőnként is meg lehet adni, hogy az adott konténerfájlon milyen algoritmust használjon - betölti a kívánt lapot.

A következő réteg az *access services* (hozzáférési szolgáltatások), ennek megvalósítása már sokkal bonyolultabb, mivel itt kell kezelni az összes fizikai reprezentációt (adatrekordok, mezők, B- és B*-fák az elérési utakhoz, katalógusok). A hozzáférési szolgáltatásokat a *rekord-,index-* és *katalógus-kezelők* valósítják meg. Mivel a rekord-kezelő lesz a teljes XML feldolgozás alapja, így az kimondottan az XML adatokhoz van kialakítva. Minden csúcs egy fizikai rekordként tárolódik, a rekordok lapokon belül tárolódnak, az ilyen lapokat egy laptípus bájttal *rekordlapnak* jelöli. A csúcsok dokumentum szerinti sorrendjének megtartását a rekordok elhelyezkedése a lapon belül és a rekordokon lévő *szint-jelölő* biztosítja. A dokumentum adatainak fizikailag is egy helyen történő tárolása fontos az XML dokumentumokon gyakori



16. ábra. A rekordlap szerkezete (bal) és a rekord szerkezete (jobb) (forrás: [8] 4.-5. ábra)

részfa felépítési műveletek végrehajtása miatt. A teljes dokumentum összeállításához szükséges még a lapok sorrendjének ismerete, ezt egy előző- és egy következő-lap mutató biztosítja. A lapon belüli rekordok *offset* (eltolás) mutatókkal vannak megadva, amik a lap végén tárolódnak, ez lehetőséget biztosít a rekordok lapon belüli áthelyezésére úgy, hogy a külső címük nem változik meg, mivel a külső címet a lapszám és az eltolás mutató indexe alkotja (a rekordlap szerkezetét a 16.ábrán balra láthatjuk).

Mielőtt akár egyetlen rekord is rögzítésre kerülne, a puffer-kezelő létrehoz egy *dokumentum katalógus* típusú lapot, ami tárolja a dokumentum első rekordjára mutató mutatót. A dokumentum letárolása előtt a teljes dokumentumot analizálja a rendszer, így lehetőség nyílik a dokumentumnak legmegfelelőbb konténer fájl kiválasztására is. A fizikailag tárolásra kerülő rekordok felépítése a 16.ábra jobb oldali képén látható.

A rekord első bájtja egy rekord leíró (*descriptor*), a leíró az első két biten a tárolási módot kódolja, ez lehet *inlined* (beépített), *distributed* (szétosztott), *vocabulary* (szótár), a beépített módban az adatok a rekord-struktúra végén találhatóak (a 2. és 3. bájt adja meg az adatok hosszát), a szétosztott módban a rekord csak a hosszú rekord kezdő részletét tároló lap címét tárolja (a 2.-tól az 5. bájtig), ezt olyankor használja a rendszer, ha a rekord túl hosszú lenne és nem férne el egy lapon. A szótár módban csak egy referenciát tárol (a 2. és 3.bájton), ami egy XML szótárban tárolt adatra mutat. A szótár egy (helyettes, név) párokból álló rendezett lista, ahol a helyettes az adott attribútum vagy elemnév egy numerikus reprezentációja. Egy ilyen lista olyan XML dokumentumok esetén jelenthet kevésbé helyigényes tárolást, amelyek sok azonos nevű elemből állnak. Az ilyen jellegű tárolás egyben nagyobb dokumentumrészek gyorsítótárban tartásának lehetőségét is magában hordozza. A rekordleíró közepén 3 bit tartalmazza a csúcs azonosítójának - ez egy élethosszig tartó azonosító és minden csúcsnak van ilyen - tárolásához felhasznált bájtok számát. A rekordleíró utolsó 3 bitje az XML csúcs típusának megfelelő rekord típust tartalmazza. A rekord 2. bájtján tárolt szint-jelzés a rekord és az őt fizikailag közvetlenül megelőző rekord kapcsolatát jelöli (szülő-gyerek, testvér). A rekordban tárolt csúcsazonosító segítségével az index-kezelő hatékonyan be tudja azonosítani

a csúcsok fizikai rekordjait.

Mivel a lapokon belüli rekordsorrend határozza meg a dokumentumsorrendet, ezért a beszúrások túlcsordulást okozhatnak a lapokon, ilyenkor az index-kezelőnek is frissítenie kell a csúcsazonosítókhoz tartozó fizikai címeket. Az itt ismertetett tárolási módok hatékonyságára jó példa, hogy ha elem vagy attribútum típusú rekordokat tárolunk 3 bájt körüli azonosító hosszal, akkor a tárolt változat kisebb lesz mint a szöveges formátumú dokumentum.

A beszúrások következtében szükség lehet a rekordokat tároló lap felosztására, ilyenkor a lap adatainak megfelelő arányú elosztása a két keletkező új lap közt hatékony hely kihasználást és a későbbi beszúrások könnyű végrehajthatóságát eredményezi. A teljes dokumentumok betöltésekor természetesen nincs ilyen probléma, mert a betöltést megelőző analízis során a rendszer kiszámítja a dokumentum csúcsok várható méretét is és azok függvényében jó helykihasználási arányt ér el (konténer fájl paraméterezése).

A betöltési folyamat során a dokumentum elemzését követően a rendszer a rekordok beszúrásával párhuzamosan az indexeket,szótárakat is karbantartja. A beszúrt attribútum és szöveg csúcsok értékei mindig bépített *(inlined)* tárolási módban kerülnek letárolásra egy string típusú rekordba. A kapcsolatot meghatározó jelölő a dokumentum gyökerében 0 és minden elem nyitótaggal 1-gyel nő, gyakorlatilag a gyökérelemtől való távolságot jelöli, ezáltal a dokumentumrészek gyorsan újraépíthetők.

Az *index-kezelő* tartja karban a rendszer által használt szekvenciális listákat és B-, B*-fákat. A B-fák tárolják a csúcsok (csúcs-azonosítóihoz, fizikai cím) párjait is. A csúcsazonosítók élethosszig szólnak, a fizikai azonosítók pedig lapon belül fixek. Így ezen listákat csak akkor kell frissíteni, ha a rekordot másik lapra helyezi át a karbantartás például egy új beszúrás miatt.

A *katalógus-kezelő* hozza létre a szerver első indulásakor a szerver metaadatait tároló *mester-dokumentumot*, a csúcsok azonosítóit kezelő B-fát és ő kezeli a *strukturális-indexeket* is. A B-fa azonosítóját és a mester-dokumentumot egy katalógus lap tárolja a szerver elindulásától kezdve folyamatosan a memóriában.

A következő réteg a *node services* (csúcs szolgáltatások), ennek megvalósítása a *node manager* (csúcs-kezelő). Itt gyakorlatilag a navigációs műveletek, a tetszőleges helyre beszúrás, tetszőleges csúcs törlése és a DOM API műveleteinek nagy része megvalósításra kerül. Ez a réteg szolgálja ki a legfelső szinten lévő XML-kezelőt (a deklaratív lekérdezések feldolgozása csúcs-orientált szemlélettel). Mivel a csúcs-kezelő csak navigációs szemléletben dolgozik és ez nem hatékony a kapcsolatok feldolgozását igénylő lekérdezéseknél, a katalógus-kezelő által karbantartott strukturális-indexekhez kell fordulni a lekérdezések kiértékelésekor.

A strukturális-indexek négy B-fával vannak megvalósítva, amelyek a vizsgált (és

a B-fák méretének minimalizálása céljából csak a használt) csúcsok azonosítójához tárolják a szülő, utolsó-gyerek és az előző-, következő-testvérek azonosítóit. Ha egy csúcs azonosítójához nincsenek még betöltve az adatok ezekbe a fákba, akkor a rekord-kezelőnek gyakorlatilag „brute-force" meg kell keresni azokat kiindulva a csúcs azonosítójához tartozó fizikai azonosítóból és a szint-jelölőből. Ezeket az indexeket nem kell gyakran karbantartani a léthosszú csúcsazonosítók miatt és nincs is minden műveletbe beépítve, hogy töltse ezeket az indexeket. Olyankor töltődik ez index struktúra, amikor nagy részfák átnézésének kihagyását lehet elérni a megfelelő információkkal. Amikor egy helyen tárolt adatokat olvasunk ki, akkor felesleges tölteni. A kapcsolódó tesztek eredményei jelentős sebességbeli növekedést mutatnak az olyan lekérdezések esetében, amelyek már meglévő strukturális indexet használnak (az indexek nélküli esethez képest sebességnövekedést jelent az első futtatás is, de mivel közben készülnek az indexek így az első futtatás még nem a legjobb eredményeket hozza).

A legfelső réteg az *XML services* (XML szolgáltatások), ennek megvalósítója most az *XML manager* (XML-kezelő). Ez a réteg biztosítja a halmaz-orientált műveletek támogatását. Az XPath kifejezések feldolgozását a következő módon oldották meg: először is a kifejezést szétbontják egyéni keresési lépésekre (például: /könyv/tartalom/fejezet kifejezést /könyv, /tartalom, /fejezet részekre). Ezután egy iterátor a dokumentum gyökeréből kiindulva végrehajtja a lépéseket egyesével és folyamatosan szűkíti az eredménycsúcsok halmazát, így végül megkapja a keresési feltételnek megfelelő összes csúcsot.

A vizsgálat eredménye röviden . A két vizsgált natív XML adatbáziskezelő bemutatott részei jól mutatják, hogy a fejlesztőik mely részt hangsúlyozzák ki leginkább, ha a relációs adatbáziskezelők fő hátrányát akarjuk megállapítani. Nem kérdéses tehát, hogy az XML adatokhoz készült tárolási megoldások kulcsfontosságúak a hatékony XML módosítást is támogató rendszerekhez és ha a két ismertetett megoldást nézzük, akkor szembetűnő hasonlóságokat fedezhetünk fel az alapvető adatszervezési és csúcs-indexelési megoldásokban is. Mindezeket kombinálva az XML adatok szerkezetéhez kialakított bármely (!), az előző szakaszban bemutatott protokollal könnyedén felülmúlhatóak a jelenlegi (objektum)relációs adatbáziskezelő rendszerek, legalább a konkurencia és valószínűleg rövid időn belül a lekérdezések terén is (a lekérdezések sebessége itt most egyáltalán nem hangsúlyos, mivel a gyors válaszidejű alkalmazások úgyis preparált adatokon dolgoznak - jó esetben - vagy kellene hogy dolgozzanak, a kevésbé sebességkritikus alkalmazások nagy része pedig le tud nyelni egy bizonyos válaszidőt).

Természetesen ezen rendszerek nem kereskedelmi volta miatt a lekérdezés-optimalizálási, rendszer-helyreállítási és egyéb fontos rendszertulajdonságok nem a legfejlettebbek

(vagy egyszerűen hiányoznak) a bemutatott rendszerekben, de ezen dolgozat nem is szándékozott vizsgálni ezeket a tulajdonságokat, hiszen azok a tranzakciós rendszerek vizsgálata szempontjából másik, jól elkülöníthető részekhez tartoznak.

4. A látottak felhasználási lehetőségei

A következőkben az eddig ismertetett protokollok és rendszerek megoldásaiból fakadó ötletek vázolásáról lesz szó. A fenti megoldások rejtette lehetőségek számtalan területen és sokféle alkalmazásban eredményezhetnének a jelenlegieknél egyszerűbb komplex megoldásokat.

Az XML dokumentumok szerkezetét figyelembe vevő rendszerek és protokollok több szinten segíthetnek a hatékony programok előállításának kérdésében. A teljesen általános feladat a már meglévő alkalmazások egyszerű - akár transzparens - kisegítése a többfelhasználós környezetekben. Ezen a téren a natív XML adatbáziskezelők használata kétesélyes, ugyanis a jelenlegi relációs megoldások elég hatékony lekérdezési sebességet tudnak biztosítani bizonyos esetekben, azonban a számunkra fontosabb részen a módosító tranzakciókkal dolgozó alkalmazások terén, a bevezetőben is említett, az XML dokumentumokat különféle táblákba leképező megoldások, nem biztosítanak megfelelő mértékű konkurenciát. A módosító tranzakciók támogatásának kérdése volt az, amely a problémakör kutatásainak fő célpontjává lépett elő és amelynek nyomán a fenti megoldások is napvilágot láthattak. Az eredmény a natív XML adatbáziskezelők megjelenése lett, amelyek életrehívásukért cserébe képesek az alapprobléma kezelésére. Ezáltal megoldódni látszik az általános feladat, azaz a XML dokumentumok kezelése osztott környezetben, de ha már eljutottunk idáig, akkor miért ne próbálnánk a jövőben - ahogy tettük a relációs adatbáziskezelők esetében is - a programjainkat az ide vezető úton látott megoldások felhasználásával vagy azok által ösztönözve, oly módon kialakítani, hogy azok jobban alkalmazkodjanak a félig-struktúrált modellhez vagy ha nem akarunk ennyire mélyre menni, akkor a mai irányzatoknak megfelelően az XML adatbáziskezelőket ruházhatjuk fel további szolgáltatásokkal, amelyeket aztán valamilyen általános felületen keresztül kihasználhatnak a jövőben a fejlesztők.

A minta, amely kapcsán megvizsgáljuk ezeket a lehetőségeket egy Iratkezelő rendszer lesz. Itt és most nem fontos egy ilyen rendszer által nyújtott szolgáltatások teljes ismerete, ami számunkra fontos, hogy a példaprogramunk komplexitása megfelelő táptalajt biztosít újszerű megoldások kigondolására.

Első lépésként néhány szó az alkalmazás adatszerkezetéről. Az alkalmazás adatszerkezetén erőteljesen látszik, hogy egy - hosszadalmas evolúciós folyamatban kialakult - nyilvántartó rendszernek besorolható jellegű őssel rendelkezik. Ez alapvetően nem is lenne probléma, hiszen egy jól struktúrált adatszerkezet nem kéne rontson az alkalmazás mögé kerülő üzleti logika megvalósíthatóságán, de mintarendszerünk adatszerkezetén van mit javítani. Természetesen egy ilyen program esetén vitatható, hogy mi a megfelelő adatszerkezet, de maradjunk csak az alapvető oknál, ami miatt egyáltalán felhoztam az adatszerkezetet, a mi vizsgált rendszerünk nem kevesebb

mint 60 táblával és legalább ennyi nézettel oldja meg az adatok tárolását és az űrlapok kiszolgálását (plusz még jónéhány másik programmodul táblájához fér hozzá bizonyos metódusokon keresztül, mivel az iratkezelő is lényegében csak egy modul).

A fő probléma ebben az esetben az összetartozó adatok különböző jellegű hozzáféréseinek kiszolgálása. A két egymásnak homlokegyenest ellentmondó lekérdezés, ami ebben a rendszerben előfordul:

1. Bizonyos komplex objektumok áttekintő adatainak listázása (ez gyakorlatilag kötelező erővel a normalizált táblák összekapcsolását igényli különböző nézetekben, ami kétszámjegyű összekapcsolás esetén már messze nem lesz ideális).

2. Az objektumok összes kapcsolódó adatának megmutatása.

A fenti két lekérdezés fajtával mintha csak a tárolási megoldások XML adatokhoz alakításának szükségességét próbálnánk alátámasztani. Első ránézésre erősen úgy tűnik, hogy az Iratkezelő rendszer alatt elhelyezkedő adatszerkezet nem ideális. Ebbe a vitába nem megyünk bele, inkább választjuk azt a megoldást, hogy mi lenne ha az alatta lévő táblák közül többet összevonnánk és áttérnénk egy XML alapú adatszerkezetre. Ez természetesen jelentős változtatás és a rendszer lekérdezéseinek újraírásához vezetne, de a kapott gyorsulások kifizetődők lehetnek. A XML adatok félig-struktúráltsága lehetőséget biztosít számunkra egy a relációs modelltől erősen különböző szerkezet kialakítására, amit az XML adatbázisok a fizikai szinten is jól támogatnak(!).

Miért is lenne kifizetődő a szerkezet megváltoztatása . A kérdéses esetekben egy relációs modell esetén a lekérdezések feltételeit kielégítő sorok előállításához, több tábla, több fizikailag más helyen tárolt (!) tábla adatainak összegyűjtésére van szükség. Természetesen a relációs adatbáziskezelők is fel vannak ruházva a megfelelő gyorsítótárazási megoldásokkal az ilyen jellegű kérések kiszolgálására, de itt most arról az alapvető tényről van szó, hogy ha a redundáns adattárolást megelőzendő, normalizált táblákkal dolgozunk, akkor a tárolási réteg elkerülhetetlenül más helyen (akár más fizikai paraméterezéssel) fogja tárolni a más karakterisztikával rendelkező másik táblába rakott adatainkat (ami szekvenciális visszaolvasás szempontjából természetesen nagyon jó nekünk, de itt most nem erre van szükségünk), holott a rendszer logikai tervezésekor nekünk csak arra lenne szükségünk, hogy bizonyos adatokat valamilyen szempont alapján egy egységként kezeljünk (ez az XML-ben elég ha egy elembe kerül és nem kell neki külön tábla/dokumentum).

Ezt a problémát az XML-hez igazított rendszerek jól oldják meg azáltal, hogy a dokumentumok adatait fizikailag is egy helyen tárolják, azaz mindkét fenti lekérdezés számára optimális válaszidőt tudnak biztosítani. Itt megemlítendő, hogy egy ilyen

„táblaösszevonás” jellegű megoldás is csak bizonyos szintig kifizetődő, hiszen ha az eredeti struktúrát túl mélyen összevonjuk -azaz túl sokat denormalizálunk-, akkor végül elveszítjük a fizikai szinten egymáshoz közel elhelyezett adatokból származó előnyt, mert a visszaolvasáskor újra távol kerülnek fizikailag egymástól az adatok, például a testvércsúcsok (ezt még csak ellensúlyozná az indexelés), ami nagy valószínűséggel több lap elérésének szükségességét vonná magával, ami egyre növekvő többletterhelést jelentene az io-vezérlő szintjén.

Felvetődhet a kérdés, hogy ha egy ilyen jellegű „denormalizálást” csinálunk, akkor az hogyan hat a dokumentum méreteinek függvényében az adatelérés hatékonyságára. Erre a válasz ugyanaz mint a relációs adatbáziskezelőknél, a túl sok adatmozgatással járó lekérdezésekkel szemben itt sem támaszthatunk irreális követelményeket, hiszen nem „mesebeli gyógyulásról” beszélünk. A vázolt ötlet csupán a fizikai tárolás ismeretéből fakadó előnyök kihasználásának egy példája kívánt lenni és nem helyettesíti az adatok normalizált tárolásából fakadó előnyöket, sőt nem is érdemes olyan adatszerkezetre alkalmazni, amelynél nem ismerjük elég mélyrehatóan a felhasználók és az alkalmazás által végrehajtott lekérdezéseket, mert végül a normalizálatlan táblák lassúságával megegyező problémával találhatjuk szembe magunkat.

Probléma: verziókezelés. Ez köthető az iratkezelő egy specifikus folyamatához is, de alapvetően egy igen népszerű probléma egyszerűsített kezeléséről szól. A többfelhasználós rendszerek igen gyakori problémája ez és gyakran külső eszközökhöz (programokhoz) kell folyamodni a megfelelő dokumentumok hozzáféréseinek szinkronizálásához, verzionálásához. Ennek a problémának most azt a specifikus ágát vesszük szemügyre, amikor is kimondottan a témába vágó XML dokumentumok konkurens hozzáféréseit kellene kezeljük, például egy szövegszerkesztő jellegű alkalmazás kapcsán. A probléma adott, több felhasználó szeretne módosítani és olvasni egyidejűleg egy dokumentumot és mindenki szeretné egyértelműen tudni, hogy az ő általa látott dokumentum melyik/mikori állapotnak felel meg (mi a verziószáma). Ennek a problémának a kezelése teljesen szinkronban látszik lenni a konkurenciakezelési problémával és alapvetően ez igaz is. Tehát az XML adatbáziskezelő ki tudja szolgálni azokat az igényeket, amelyek a dokumentum egyes különálló részeinek módosítását célozzák meg a teljes dokumentum zárolása nélkül, ami előrelépés sok külső verziókezelő megoldáshoz képest, de ez még nem helyettesíti a verziókezelési szolgáltatásokat.

Verziókezelési szolgáltatások címszó alatt most a céldokumentum verziószámának karbantartását és mondjuk egy egyszerű, verziók közti különbség megmutatásához szükséges funkcionalitást értünk. Erre a problémára sok megoldás látott már napvilágot fájl szinten, sőt ha nagyon kellemetlenkedni akarunk, akkor a kereskedelmi forgalomban kapható adatbáziskezelők közt is van olyan, amely lehetőséget ad az adatok

időpillanathoz kapcsolódó értékeinek tárolására, de itt most olyan verzionálásról van szó, amelynek megvalósítása nem igényel jelentős többletmunkát, hiszen az érintett dokumentumok szerkezetileg megegyeznek az adatbázisban tárolt adatokkal és így a konkurenciakezelési szolgáltatást máris ingyen kapjuk (ahogyan szerették volna régebben a relációs adatbáziskezelőkre készített XML kiegészítések és XML dokumentumok esetében is). A verziókezelés ára a felvetésünkben leginkább az adatbáziskezelő megvalósításától függ.

Számunkra a legolcsóbb megoldást egy többverziós konkurencia-kezelési protokollokra is felkészített rendszer tudná szolgáltatni. A megoldás egyszerűsége abból fakadna, hogy az általunk explicit módon verzionálni kívánt dokumentumokat a rendszer a konkurencia-kezelés miatt a háttérben amúgy is csúcsszinten időbélyegekkel (verziószámokkal) látja el. Ebben az esetben a tranzakciók, azaz a felhasználói lekérdezések által látható verziók az aktuális időpillanatban a tranzakciókezelő által mindenképp karban vannak tartva. A többverziós protokollok hatékonyságát jelentősen befolyásoló megtartott verziók száma pedig elég olcsón kihasználható lenne a dokumentumverziók előállításához. Itt fontos megjegyezni, hogy a megtartott verziók számának egyszerűen „elég nagyra állítása" nem elég a verziók kezeléséhez, mivel a túl sok megtartott verzió kezelése a protokoll sebességcsökkenéséhez vezet.

A verziókezelés kapcsán több paramétert is bevezethetünk, amik fontosak a kezelés szempontjából. Ilyenek például: a megtartott verziók száma és a verziók létrehozásának, tárolásának szabályait befolyásoló paraméterek. A megtartott verziók száma befolyásolná a fizikailag tárolásra kerülő változatok számát, ez természetesen egy karbantartási mechanizmust is igényelne, amely az adott szám túllépése esetén a régebbi verziókat eldobja. Mivel túl nagy megtartott verziószám esetén a verziókat tároló lista nem férne el csúcs verziólistáját is tároló fix méretű header részben, ezért érdemes meggondolni, hogy a tárolásra kerülő verziók konkrétan hogyan fognak tárolódni, azaz minden verzionált csúcs maga fogja tárolni a megfelelő listát vagy verziónként létrehozunk egy listát a benne foglalt csúcsok verzióival és a csúcsok csak a konkurencia-kezeléshez szükséges verziólistát tartják a header részben vagy esetleg minden verzió fizikailag is letárolásra kerülne csak a dokumentum aktuális verziója ezt eltakarná és a külvilág felé mindig csak az aktuális verzió látszana és a régebbi verziókat csak bizonyos metódusok segítségével lehetne elérni. Ezek a kérdések összefüggésben vannak a felhasználással is, hiszen mindegyik említett lehetőség más működési módot jelent, más szempontok alapján értékelendő, más igényekhez alkalmazkodik.

A minta alkalmazásunk szempontjából gyakorlatilag bármely megoldás megfelelne, mivel tipikusan nem a teljes dokumentumot szeretnénk verzionálni, hanem annak csak bizonyos részeit (ez ugyanúgy megvalósítható az eddigiek alapján, mintha a teljes dokumentumot verzionálnánk, hiszen a tranzakciókezelő csúcs szinten fog verz-

iókat kezelni) és ezeken a részeken sem jellemző a konkurens módosítás, inkább csak a verziószámok létezésére és esetleg a verziók tartalmának különbségére van szükség. Tehát ebben az alkalmazásban elég lenne számunkra egy olyan működési mód, amely a csúcsokhoz tárolt verziólistákból elő tudja állítani a régi verziót és vissza tudja adni az adott dokumentumrész emberileg értelmezhető verziószámát (tehát nem az időbélyeget, ami a csúcslistában tárolva lesz hanem mondjuk egy folyamatosan növelt értéket). Ehhez a módhoz a legjobb megoldás talán a csúcsoknál tárolt verziólista (ami úgyis egy láncolt lista) folytatása lenne, de a lista folytatását valamely külön listába kéne helyezni, hogy a hosszabb lista ne lassítsa a tranzakciókezelés folyamatát és a verziókhoz létre is hozhatnánk egy-egy bejegyzést, amely tárolná az olvasható verziószámot és az abban a verzióban szereplő csúcsok verziószámait, így nem kéne minden csúcsot minden verzióhoz újra letárolni, akkor is ha a csúcs értéke változatlan vagy ha a csúcsleírót mindenképp létrehozzuk, akkor még mindig fennáll a lehetőség, hogy a csúcs adatai helyett csak egy mutatót tároljunk az ilyen változatlan pszeudó-verziókban.

További kérdéseket vet fel a nem aktuális verziók fizikai tárolása, ugyanis a tárolási modellek egy helyre tárolják a dokumentum csúcsait és ebbe „belekeverve" a verzionált csúcsokat jóeséllyel többletmunkát okozunk a kereséseknél. Ezt enyhítendő esetleg fizikailag is külön helyen kéne tárolni a verziókat, mintha egy teljesen különálló dokumentumrészt tárolnánk, így megmaradna a fizikai modellben is alkalmazott egy dokumentum egy helyen elv és a verziókeresésnél is könnyebb lenne beazonosítani az egy verzióhoz tartozó csúcsokat (a pszeudóverziók továbbra is tartalmazhatnának mutatókat a csúcsok adataira).

A fizikai meggondolások után visszatérve a működési paraméterekhez, a verziók létrehozásának szabályai többek közt megadhatnák azt, hogy egy dokumentum(rész)-ből mikor keletkezzen új, felhasználónak szánt verzió. Egy ilyen paraméterrel csökkenthetjük a létrejövő plusz verziók számát, méghozzá egészen addig a szintig, hogy csak akkor hozunk létre új verziót, ha erre a rendszer külön utasítást kap. Alapértelmezetten a módosító tranzakciók létrehoznak csúcs verziókat a committálásukkor, de ha ilyen részletes verzionálásra nincs szükség a felhasználói szinten, akkor ezt mi külön befolyásolhatjuk a megfelelő paraméterekkel.

Az itt vázolt verziókezelési megoldással, minimális többletmunkával, ötvözhetjük az adatbáziskezelő tranzakció-kezelési képességeit egy verzió-követő,-kezelő alkalmazás tulajdonságaival, mindezt teljesen natív módon és gyakorlatilag az adatbázisba bekerülő tetszőleges XML dokumentum tetszőleges részére. A verziókövetés bekapcsolásához tetszőleges elem típuson vagy akár tetszőleges elemen elég lenne egy jelölő elhelyezése, melynek hatására az adatbáziskezelő automatikusan végrehajtaná a fent vázolt akciókat. A verziók közti különbségek megállapítása szintén egyszerűen megtehető lenne a két dokumentum(részlet) verzió csúcsaihoz tartozó verziószámok

alapján.

A tranzakció-kezelés által érintett objektumok számának csökkentése. Azzal a számunkra kedvezőtlennek mondható állapot állunk szemben, hogy a nagy kiterjedésű XML dokumentumokban gyakran lát és olvas egy tranzakció olyan csúcsokat, amelyeket nem módosít. Ez az állapot természetesnek mondható a tranzakciós rendszerekben, tehát a legtöbb esetben így van. Azonban a konkurencia-kezelésben már az optimista protokolloknál felvetődött a zárak megszerzésének elodázási lehetősége a rendszer teljesítményének növelése céljából illetve az ismertetett protokollok közül a többverziós alapokra épülő MPX-ben is előjött a csak-olvasó tranzakciók és a módosító tranzakciók különválasztása. A zárak számának csökkentése a tranzakció-kezelés szempontjából kulcsfontosságú momentum lehet, ha a számszerű nyereséget a konfliktusok ellenőrzésénél el nem pazaroljuk. A zárak számának csökkentésére irányul a hierarchikus protokolloknál alkalmazott zár kiterjesztés, a részfákat zároló fa-zárak bevezetése, a bővíthető zárak alkalmazása, a dinamikusan változtatható zárolási mélység és ezekből a megoldásokból sohasem elég. Minél kevesebb a zár, annál hatékonyabbak a protokollok a konfliktus ellenőrzés terén.

A zárak számának csökkentéséhez választhatnánk akár egy kerülőutat is, azaz mi lenne ha, nem pusztán a zárak számát próbálnánk leszorítani, hanem lehetőséget adnánk az érintett csúcsok számának csökkentésére is, ami kényszerűen a zárak számának csökkenéséhez kéne vezessen. Ez a megközelítés azon zár-kezelőknél lehet egyszerűen megvalósítható, amelyek képesek azon protokollok kezelésére, amik nem a dokumentum csúcsaira helyezik a zárakat, hanem valamely segédstruktúrára - úgy mint a Dataguide alapú protokollok vagy a taDOM modellt használó protokollok. Innen már csak egy ugrás a felvetés központi részét jelentő elképzelés, csökkentsük a segédstruktúra méretét a zár-kezelő által figyelendő elemtípusok kijelölésével. Ha lehetőséget biztosítunk a felhasználóknak az XML dokumentum elemeinek (elemtípusainak) megjelölésére (például egy attribútummal) vagy felületet biztosítunk akár utólag a betöltött dokumentumok sémáján a figyelendő elemtípusok kijelölésére, akkor lehetőségünk nyílik a megfelelő ismeretek birtokában a háttérben keletkező segédstruktúra kialakításának befolyásolására is. Mivel a segédstruktúrák csúcsai mindenképp valamilyen módon össze vannak rendelve az XML dokumentum fizikai csúcsaival - Dataguide esetén a Dataguide csúcsához tartozik egy mutató, ami a csúcshoz tartozó konkrét csúcsok listájára mutat, a taDOM struktúra pedig minden csúcshoz direktben kell hozzárendelje a dokumentum csúcsait, mivel a teljes dokumentumot egy kiegészített modellel reprezentálja a zár-kezelő -, így ezzel az összerendelési kérdéssel nem kell kiemelten foglalkozni. Meggondolást igényel azonban az így kialakított dokumentumot reprezentáló segédstruktúra karbantartása. Annak ténye, hogy az új segédstruktúra nem tartalmazza feltétlenül a figyelni kívánt

csúcsokat összekötő utakat, két lehetőséget mutat számunkra:

1. Azon tranzakciókat, amelyek olyan csúcsokat módosítanak (főleg ha áthelyeznek, átneveznek, törölnek), amelyek valamely konkurens tranzakcióban figyelt csúcsokat kötnek össze, valamilyen központi megoldással szinkronizálni kell.

2. Egyszerűen figyelmen kívül hagyjuk az ilyen jellegű módosításokat, mivel az adott tranzakció szempontjából lényegtelenek.

Az 1. lehetőség támogatása esetén, akkor látszik kifizetődőnek a megoldás, ha az ütköző konkurens tranzakciók által lefedett dokumentumrész még mindig elég kicsi ahhoz, hogy érdemes legyen egy új segédstruktúrát létrehozni a teljes dokumentumot ábrázoló helyett, így nem kell további módszereket sem kidolgozni, elég ha az új csökkentett segédstruktúrát az érintett tranzakciók által figyelésre kijelölt csúcsokból alkotjuk meg. Mivel a zár-kezelő a tranzakciók kezeléséhez az esetek többségében csak 1 segédstruktúrát tart karban (és nem minden tranzakció számára egyet), meggondolást igényel az olyan tranzakciók kezelése, amelyek ugyan diszjunkt elemtípusokat figyelnek, de a dokumentumban található elemtípusok jelentős részét lefedik. Az ilyen esetek kezelése valószínűleg minimális hozamot eredményez, hacsak nem vezetünk be valamilyen módszert, a segédstruktúrában előforduló figyelt csúcsok tranzakciónkénti szétválasztására. A megoldás nem kézenfekvő, mivel mindennek ára van, méghozzá a zár-kezelőben, mindennek nagy ára lehet. Több megoldásból is meríthetünk a probléma kezeléséhez ihletet, ilyenek a következők:

- Ha a zárak kiosztásakor figyelembe vesszük, hogy az adott tranzakció mely elemtípusokat figyelteti, akkor egyrészt a nem figyeltetett csúcsokra eleve nem is fog zárakat szerezni, ez az ütközések ellenőrzésénél fontos. Másrészt, ha lopunk egy kis ötletet az optimista protokolloktól és mondjuk csak a módosító műveletek zárigényeit vezetjük csak rá a segédstruktúrára, akkor talán nem fognak számunkra a tranzakció számára felesleges, a (csökkentett) segédstruktúrában mégis szereplő, csúcsok jelentős gondot okozni.

- Egy másik megközelítés lehetne a figyelt csúcshalmazok diszjunktságából következően, hogy a zár-kezelő diszjunkt csúcsokat figyeltető tranzakciónként külön segédstruktúrát tart karban, hiszen minden érintett elemtípust így is csak egy helyen kéne karbantartania. Ebben az esetben felvetődik annak kérdése, hogy mi számít diszjunktnak egy fa-jellegű struktúrában. Ha egyszerűen csak a típusokat vesszük figyelembe, az kevésnek tűnik az XML dokumentumok navigációs műveleteihez képest, ha viszont figyelembe vesszük a szülő, gyerek, testvér kapcsolatokat is, akkor könnyedén jelentős megszorítások bevezetésére kényszerülhetünk a kijelölhető elemtípusok kapcsán.

- A segédstruktúra módosítása mellett, a zárak ellenőrzésénél is módosítások eszközölhetőek. Ha ugyanis annál a megoldásnál maradunk, hogy csak egy segédstruktúrát tartunk karban, akkor a zár-kezelő számára nyújthatunk még némi segítséget az által, ha a rendelkezésünkre álló információk alapján, nem a struktúrát bontjuk darabokra, mint azt az előző pontban tettük, hanem csak a központi zár-kezelő zárakat tároló struktúráját. Itt arról lenne szó, hogy az olyan tranzakciók zárainak kezelését, amelyekről tudjuk, hogy garantáltan nem ütközhetnek - ezt a típuskijelölés megadja -, egyszerűen külön hash táblákban tároljuk. Ez a technika alkalmazható lenne akár az alapmegoldás mellett is, hiszen kívülről nem befolyásolja a típuskijelölés sem.

A 2. lehetőség támogatása a fantom problémához hasonló problémát látszik előidézni. Ennek elkerüléséhez egyszerű, de meglehet hogy túlságosan nagy kötöttséget jelentő megoldásnak látszik az, ha olyan megkötéseket teszünk, amelyek a fantom probléma kezelésénél használatos logikai zárak módjára megelőzik (mint az 1.eset második pontjában) az olyan strukturális módosításokat, amelyek inkonzisztens vagy szimplán nehezen kezelhető állapotokhoz vezetnek (például: a figyelt csúcsot tartalmazó részfa törlése, dokumentumsorrendet felborító beszúrások végrehajtása). Ha teljesen figyelmen kívül akarjuk hagyni a módosítások pillanatában a szerkezeti kötöttségeket, akkor legjárhatóbbnak a committálást megelőző ellenőrzések tűnnek. Ilyen ellenőrzéseknek kéne figyelni a törlések és beszúrások, illetve átnevezések sorrendmódosítások által kialakítható inkonzisztenciákra (ennek helyességének belátásához az összes lehetséges eset vizsgálata lenne szükséges).

Ezeket a problémákat gyakorlatilag figyelmen kívül hagyhatjuk, ha ezt a kijelölési lehetőséget például csak a dokumentum szintjén engedjük meg (ekkor minden tranzakció ugyanazt a csökkentett segédstruktúrát használná, gyakorlatilag mintha a dokumentumnak csak egy kijelölt részét vagy egy másik dokumentumot módosítanának), vagy konkurens tranzakciók esetén csak akkor alkalmazzuk, ha a tranzakciók célcsúcsainak halmazai diszjunktak vagy a nem diszjunkt elemtípusok uniója a dokumentumban található típusok egy meghatározott százalékos határértékét (vagy valamely költség alapú megközelítés szerinti határértéket) nem lépi túl (ez nagyméretű XML dokumentumoknál - főleg ha a lekérdezést egy ezt az opciót ismerő alkalmazás hajtja végre - továbbra is gyakori esetként vehető számításba).

5. Összegzés

A kialakult megoldások áttekintése során képet kaphattunk az elmúlt évtized kutatómunkáinak nyomán kialakult, XML adatok kezeléséhez kialakított protokollokról és betekinthettünk két natív XML adatbázis legmélyebb rétegébe, amely a hatékony feldolgozás egyik kulcsát jelenti az XML adatok lekérdezése és módosítása szempontjából. Mivel nem az ismertetett megoldások számszerű mérési eredményeit helyeztük a vizsgálat középpontjába, így győztest sem hirdetünk a bemutatás végén. Mindenesetre tény, hogy azok a protokollok rendelkeznek a legvonzóbb eredményekkel, amelyekből több változat is készült, azaz a kutatóinak volt ideje a szükséges koncepciók újra átgondolására. Több megközelítés és sok hasonlóság mellett mindig akad egy-egy újszerű momentum a felbukkanó megoldásokban, sajnos a legtöbbjük soha nem jut el a teljesen kiforrott állapotig, de ezek a kiforratlan kezdeményezések alkotják általában az újabbak alapjait, így megalkotásuk és ismeretük nem számít elpocsékolt energiának. A vizsgált rendszerek jó példával szolgálnak egy manapság használatos és szükséges komplex rendszer tervezésének és fejlődésének megismeréséhez és ezen ismeretek birtokában lehetőségünk nyílik olyan megoldások kigondolására, amelyek a kellő odafigyelés mellett előrelépést jelenthetnek az XML adatok hatékony kezelése terén.

Hivatkozások

[1] Sebastian Bächle, Theo Härder, and Michael Peter Haustein. Implementing and optimizing fine-granular lock management for xml document trees. In Xiaofang Zhou, Haruo Yokota, Ke Deng, and Qing Liu, editors, *DASFAA*, volume 5463 of *Lecture Notes in Computer Science*, pages 631–645. Springer, 2009.

[2] Eun HYE CHOI and Tatsunori KANAI. Xpath-based concurrency control for xml data. 2003.

[3] Stijn Dekeyser and Jan Hidders. Path locks for xml document collaboration. *Web Information Systems Engineering, International Conference on*, 0:105, 2002.

[4] Stijn Dekeyser and Jan Hidders. A commit scheduler for xml databases. In Xiaofang Zhou, Yanchun Zhang, and Maria E. Orlowska, editors, *APWeb*, volume 2642 of *Lecture Notes in Computer Science*, pages 83–88. Springer, 2003.

[5] Andrey Fomichev, Maxim Grinev, and Sergey Kuznetsov. Sedna: A native xml dbms. 2006.

[6] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB*, pages 436–445. Morgan Kaufmann, 1997.

[7] Torsten Grabs, Klemens Böhm, and Hans-Jörg Schek. Xmltm: efficient transaction management for xml documents. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pages 142–152, New York, NY, USA, 2002. ACM.

[8] Michael Haustein and Theo Härder. Fine-grained management of natively stored xml documents. 2004.

[9] Michael Haustein and Theo Härder. Optimizing lock protocols for native xml processing. *Data Knowl. Eng.*, 65(1):147–173, 2008.

[10] Michael P. Haustein, Theo Härder, Christian Mathis, and Markus Wagner. Deweyids - the key to fine-grained management of xml documents. In *In Proc. 20th Brasilian Symposium on Databases*, pages 85–99, 2005.

[11] Michael Peter Haustein and Theo Härder. tadom: A tailored synchronization concept with tunable lock granularity for the dom api. In Leonid A. Kalinichenko, Rainer Manthey, Bernhard Thalheim, and Uwe Wloka, editors,

ADBIS, volume 2798 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2003.

[12] Michael Peter Haustein and Theo Härder. Adjustable transaction isolation in xml database management systems. In Zohra Bellahsene, Tova Milo, Michael Rys, Dan Suciu, and Rainer Unland, editors, *XSym*, volume 3186 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2004.

[13] Sven Helmer, Carl christian Kanne, Guido Moerkotte, S. Helmer, C. c. Kanne, G. Moerkotte, and Universität Mannheim. Isolation in xml bases. Technical report, Lehrstuhl fur Praktische Informatik III, Universitat, 2001.

[14] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Lock-based protocols for cooperation on xml documents. In *DEXA Workshops*, pages 230–234. IEEE Computer Society, 2003.

[15] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Evaluating lock-based protocols for cooperation on xml documents. *SIGMOD Record*, 33(1):58–63, 2004.

[16] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Timestamp-based protocols for synchronizing access on xml documents. In *Proc. DEXA Conference, Zaragoza, Spain*, pages 230–234, 2004.

[17] Peter Pleshachkov, Petr Chardin, and Sergei D. Kuznetsov. A dataguide-based concurrency control protocol for cooperation on xml data. In Johann Eder, Hele-Mai Haav, Ahto Kalja, and Jaan Penjam, editors, *ADBIS*, volume 3631 of *Lecture Notes in Computer Science*, pages 268–282. Springer, 2005.

[18] Peter Pleshachkov, Petr Chardin, and Sergei O. Kuznetsov. Xdgl: Xpath-based concurrency control protocol for xml data. In Mike Jackson, David Nelson, and Sue Stirk, editors, *BNCOD*, volume 3567 of *Lecture Notes in Computer Science*, pages 145–154. Springer, 2005.

[19] Peter Pleshachkov and Sergei Kuznetcov. Sxdgl: Snapshot based concurrency control protocol for xml data. In Denilson Barbosa, Angela Bonifati, Zohra Bellahsene, Ela Hunt, and Rainer Unland, editors, *XSym*, volume 4704 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2007.

[20] Yuan Wang, Gang Chen, and Jinxiang Dong. Mpx: A multiversion concurrency control protocol for xml documents. In Wenfei Fan, Zhaohui Wu, and Jun Yang, editors, *WAIM*, volume 3739 of *Lecture Notes in Computer Science*, pages 578–588. Springer, 2005.

[21] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.