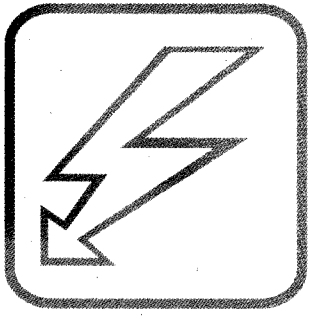
BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR

Gajdos Sándor

ADATBÁZISOK



Műegyetemi Kiadó

Előszó az első kiadáshoz

Az adatbázisok, adatbáziskezelés az utóbbi években különösen divatos fogalmak lettek. Ennek oka jelentős részben az, hogy megjelentek a személyi számítógépeken is működőképes adatbáziskezelők, amelyeknek teljesítménye megközelíti, néha túlszárnyalja több, 15 évvel korábbi "nagygépes" adatbáziskezelő teljesítményét. Azokra a speciális ismeretekre, amelyekre valaha csak néhány adatbázis szakértőnek volt szüksége, jelentős részben támaszkodnia kell mindenkinek, aki hatékony, megbízható adatbázist akar - akár az otthoni számítógépén - létrehozni. Így azután megnövekedett az igény olyan művek iránt, amelyekből az adatbáziskezelés alapjai elsajátíthatók.

Jelen jegyzet elsősorban a Budapesti Műszaki Egyetem Villamosmérnöki és Informatikai Kar Rendszerinformatika szak, valamint Műszaki informatika szak hallgatói számára készült. A jegyzet felépítése az ő előképzettségükhöz és igényeikhez igazodik. Ennek megfelelően célja nem naprakész gyakorlati ismeretek átadása - erre a célra az Adatbázisok-hoz kapcsolódó laboratóriumi gyakorlatok (Számítógép laboratórium V.) szolgálnak -, hanem olyan elméleti alapok nyújtása, amelyek egy-egy konkrét megvalósítástól függetlenek és olyan alapokat jelentenek, amelyek széles körben megkönnyítik az adatbáziskezelők működésének megértését, használatuknak a hatékony elsajátítását.

A jegyzet az alapfogalmak bevezetése után az adatok hatékony fizikai tárolásának kérdéseivel foglalkozik. Megismertet a hálós, a relációs és az objektum orientált adatmodellel. Gyakorlati jelentőségének megfelelően a legrészletesebben a relációs adatmodellt tárgyalja, ahol a relációs sématervezés módszerei kitüntetett szerepet kapnak. Külön fejezetek szólnak a többfelhasználós és a térben elosztott működés problémáiról.

Az adatbázisok gazdag és folyamatosan bővülő témaköreinek a fenti fejezetek természetesen csak egy (szűk) részét képezik. A bővítésnek számos feltétele és akadálya is van. Mindazonáltal a szerző nyitott minden javaslatra, amely a jegyzet tartalmával (esetleges sajtóhibákkal is) kapcsolatos. Különösen hálás vagyok érte, és előre köszönöm, ha az Olvasó a megjegyzéseit E-mail-ben elküldi a gajdos@db.bme.hu címre.

Budapest, 1999.

A szerző

Előszó a harmadik kiadáshoz

A jegyzet harmadik kiadása – néhány apróbb hiba kijavításán túlmenően – két területen bővült. Egyrészt megszületett gyakorló feladatoknak egy első gyűjteménye. Az utolsó fejezetben mintegy 70 feladatot gyűjtött össze Szatmári Zoltán korábbi zárthelyi- és vizsgafeladatokból, ez a bővítés tehát jelentős részben neki köszönhető. A jegyzet függelékében pedig a hagyományosnak tekinthető strukturált adatok kezelését kiegészítendő a szemistrukturált adatkezelés legfontosabb tudnivalóit foglaltuk össze.

Budapest, 2006. április

Bevezető

Az adatbáziskezelés az a terület, amelyen ma is talán a leggyakrabban alkalmazzák a számítógépet.
Adatok gyors, gépesített tárolásának és visszakeresésének igénye már akkor felmerült, amikor még csak (elektro-)mechanikus számológépek léteztek. A népesség-nyilvántartásban kezdetben "lyukkártyás tabulátorokat" alkalmaztak (→"Hollerith kártyák"), amelyek ősi adatbáziskezelőknek tekinthetők. Egy kártya egy rekord adatait - 80 karaktert - tárolta. Bár korukban óriási előrelépést jelentettek, mégsem nehéz elképzelni, hogy mennyi probléma lehetett ezekkel a szerkezetekkel. Ennek ellenére, amikor megjelentek az első elektronikus, mágnesszalagos háttértárat alkalmazó adatbáziskezelők, alapvetően a kártyás működést utánozták. Ma a soros helyett közvetlen hozzáférésű háttértárak (mágneslemez, optikai tár) dominálnak és az egykori kártyás adattárolóra a mai számítógépek már egyáltalán nem hasonlítanak, azonban az adatbáziskezelők dominánsan rekordorientált szemlélete máig megmaradt. Az elektronikus számítógépek elsősorban sebességükben hoztak újat, továbbá abban, hogy velük a rekordok, ill. rekord típusok között változatos kapcsolatok alakíthatók ki, más szavakkal: az adatbázis logikai struktúrája tetszőlegesen bonyolulttá tehető, ami változatosabb és bonyolultabb lekérdezéseket tesz lehetővé.

Az első nem szekvenciális hozzáférést biztosító fájlrendszert 1959-ben fejlesztették ki az IBM-nél. Az 1960-as években egy sor új, harmadik generációsnak nevezett programozási nyelv jelent meg, mint a Fortran, Basic, PL1, melyek között volt egy, a Cobol, amely egyenesen adatkezelés-orientált céllal jött létre. Egyes statisztikák szerint az adatbázis alkalmazások nagy része még pár évvel ezelőtt is ezen a nyelven készült, megelőzve a C, C++ nyelvet is, melyet inkább rendszerfejlesztésre használnak. Nem sokkal ezután megjelentek az első adatbáziskezelő rendszerek is, melyek a file-kezelőkből alakultak ki, azok szerves folytatásaként. 1961-ben dolgozták ki a hálós adatmodell alapjait, majd nem sokkal rá létrejött a hierarchikus adatmodell is. Az első hálózatos, konkurens hozzáférést biztosító adatbázis 1965-től működött az IBM-nél, és a SABRE nevet kapta. Az induló időszak hierarchikus, majd hálós adatmodelljei után az 1970-es években indult el hódító útjára a ma legelterjedtebb relációs adatbázis-kezelés. Az adatbázisokkal kapcsolatos elméleti kutatások is megszaporodtak, az 1970-es években indultak be a témához kapcsolódó neves konferenciák (VLDB, Very Large Data Bases és SIGMOD, Special Interest Group on Management of Data). Az 1980-as években a relációs adatbáziskezelők SQL kezelőfelülete is szabvánnyá vált, és megjelentek a relációs adatbázist kezelő alkalmazások hatékony fejlesztését szolgáló negyedik generációs, 4GL rendszerek is. A XX. század utolsó évtizedében az adatbázis-kezelés területén is tért hódítanak az új elvek, mint az objektumorientáltság vagy a logikai programozás, a hálózatok elterjedésével az elosztott adatbáziskezelők szerepe is növekszik. Emellett egyre nagyobb szerepet kapnak az adatszerű információkezeléstől eltérő felépítésű és funkciójú, szövegszerű kezelést megvalósító információs rendszerek is, melyek tágabb értelemben kapcsolhatók az adatbázis-kezelés területéhez.

1 Alapfogalmak

Adatbázisnak a valós világ egy részhalmazának leírásához használt adatok összefüggő, rendezett halmazát nevezzük. Ma ezek többé-kevésbé állandó formában egy számítógép háttértárolóján tárolódnak.
Azt a hardver-szoftver rendszert, amely egy vagy több személy számára magas szinten teszi lehetővé ezen adatok olvasását vagy módosítását, *adatbáziskezelő rendszernek* (Database Management System, DBMS) nevezzük. Az adatbáziskezelőt alapvetően jellemző tulajdonságok:

- nagy adatmennyiség,
- gazdag struktúra és
- hosszú életciklus.

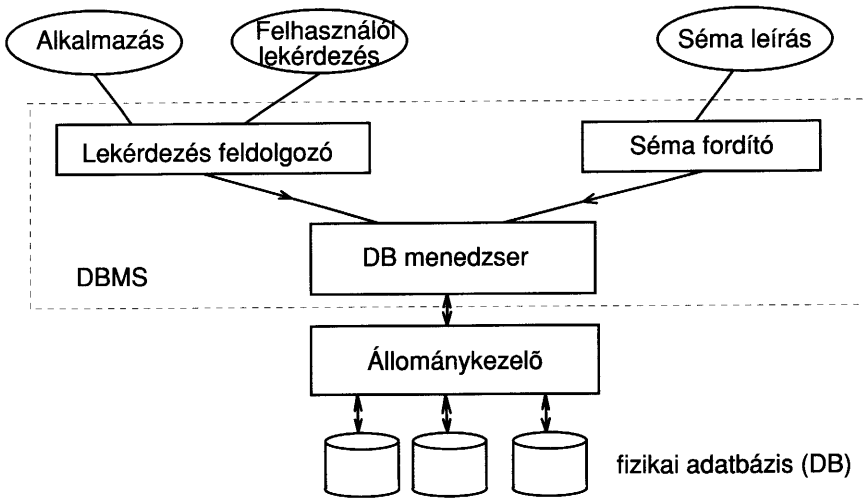
Az adatbáziskezelő rendszerek adatait ma túlnyomórészt merevlemezes mágneses tárolókon (winchester) tárolják, kisebb részben mágnesszalagon, optikai lemezen (CD-n). Így jelen jegyzetben a mágneslemezes háttértárak alkalmazására, sajátosságaira koncentrálunk. Az adatbáziskezelő gazdag struktúrája azt jelenti, hogy a tárolt rekordok között változatos logikai kapcsolatok hozhatók létre, amelyek meghatározott adatbázis-műveleteket megkönnyíthetnek (értsd: meggyorsíthatnak). Egyes adatbáziskezelők bizonyos logikai kapcsolatokat megengedhetnek, másokat nem, így különböző adatmodelleket (ld. 4.1. szakasz) realizálhatnak. Hosszú életciklusuk legjobban talán a népességi adatbázisok példájával szemléltethető, amelyeknek tetszőlegesen hosszú időt és igen sok technológiai váltást is túl kell élniük mindaddig, amíg népességi adatbázisokra egyáltalán szükség van.
Az adatok kezelésének magas szintje azt jelenti, hogy az adatbáziskezelő úgy működik (működhet), hogy a felhasználó anélkül tudja előírni a teendőket, hogy az adatbáziskezelő algoritmusairól vagy az adatok (fizikai) tárolási elvéről ismeretei lennének.

A továbbiakban megvizsgáljuk, hogy melyek a különböző adatbáziskezelők fontosabb közös vonásai.

1.1 A programozó és a felhasználó kapcsolata az adatbáziskezelő rendszerrel

Az 1.1. ábra egyesítve mutatja egy DBMS általános felépítését és környezetét.
Az adatbáziskezelő rendszerek használatának alapvetően két fázisa különül el. Először meg kell határozni az adatok majdani tárolásának logikai rendjét: ez az adatbázis (fogalmi) vázának, vagy más szóval sémájának megteremtését jelenti (1. fázis). Csak ezután van lehetőség az adatbázist adatok tárolására használatba venni, adatokkal feltölteni, majd az adatokat különböző szempontok szerint visszakeresni (2. fázis). Ez utóbbi történhet úgy, hogy egy (képzett) személy speciális adatbázis lekérdező nyelven kérdéseket fogalmaz meg, melyeket egy *interpreter* azonnal értelmez és az adatbáziskezelő válaszol is rá. Másrészt, a lekérdezések önálló programként vagy egy alkalmazás (applikáció) részeként is megjelenhetnek. Mindkét esetben egy interpreter, egy lekérdezés feldolgozó alakítja azokat az adatbázis menedzser által értelmezhető formába. Eközben általában a lekérdezés optimalizálása is megtörténik, hiszen egy-

egy végeredményhez gyakran több úton is eljuthatunk, amelyek számításigénye azonban akár nagyságrendileg is különbözhet.



1.1. ábra: A DBMS és környezete

Az 1. fázist egy speciális nyelv, az ú.n. *adatdefiníciós nyelv* (data definition language, DDL) támogatja, amellyel tehát megfogalmazhatjuk, hogy milyen adatokat milyen formában fogunk az adatbázisban tárolni. A *sémafordító* értelmezi az adatbázisnak ezt a *logikai (fogalmi) leírását* és külön fordítja le. Az adatbázis használatához, a 2. fázishoz a lefordított *séma* mindig rendelkezésre kell, hogy álljon. Ez olyan a DB menedzser számára, mint egy program deklarációs része. A 2. fázisnak is saját nyelve van: ez az *adatlekérdező és -manipulációs nyelv* (data manipulation language, DML). A DML és a DDL gyakran jelenik meg egy egységes nyelvként, mint pl. a szabványosított SQL nyelvben (ld. 5.3. szakasz).

A DBMS központi része az adatbázis menedzser, amely a lefordított séma alapján kezeli a felhasználói lekérdezéseket és olyan további feladatokat is ellát, mint az adatvédelem, adatbiztonság, szinkronizáció, integritás megteremtése (ld. 1.2. szakasz).

Az *állománykezelő* (file manager) biztosítja a hozzáférést a fizikai adatbázishoz. Az állománykezelő általában szoros kapcsolatban van az operációs rendszerrel. Egyszerűbb esetben annak számos szolgáltatását használhatja, de ennél többet is elvárhatunk tőle, ha figyelembe vesszük a későbbiekben megismerendő speciális állományszerkezeteket (ld. 3. szakasz). DB-környezetben az adatvesztés veszélye miatt nem megengedett a "nem fizikai" írás/olvasás.

Adatbázis (Data Base, DB) alatt általában csupán a fizikai adatbázist értjük.

1.2 Járulékos feladatok

Az adatbáziskezelőtől néhány egyéb feladat megoldását is elvárjuk. Az alábbiakban felsoroltakat elsősorban az 1.1. ábra adatbázis menedzsere valósítja meg.

1.2.1 Adatvédelem (privacy)

Nem minden felhasználó férhet hozzá minden tárolt adathoz. A hozzáférés módja is lehet különböző az egyes felhasználóknál: azok az adatok, amelyeket az egyik felhasználó kedvére módosíthat, egy másik számára esetleg csak olvasásra hozzáférhetőek. Gyakran jelszóhoz kötik az elérés jogának megszerzését, de bonyolultabb módszerek, pl. célhardver is támogathatja az adatok védelmét.

1.2.2 Adatbiztonság (security)

Bizonyos adatbázisokban a tárolt adatok igen nagy értéket képviselhetnek, így megsemmisülésük, vagy akár részleges megsérülésük semmiképpen nem megengedett, még szélsőséges körülmények esetén (elemi csapás, adathordozó ellopása, rendszerhiba, stb.) sem. Ennek biztosításához különleges eljárásokra van szükség, mint pl. naplózás, rendszeres mentések, kettőzött adatállományok, elosztott működés, stb.

1.2.3 Integritás

Fontos, hogy legyen olyan beépített szolgáltatás, amely segítségével az adatbázis adatainak "helyessége", ellentmondásmentessége - azaz *integritása* - ellenőrizhető, mivel a beszúrás, törlés, módosítás funkciók kényesek a sikeres végrehajtásra. Szerencsés, ha a DBMS már az adatbevitel során minél szélesebb körben képes az integritást sértő műveletek megakadályozására, gyakrabban azonban az adatbázis applikációkra hárul ennek a feladatnak egy része. Sőt - látni fogjuk -, az adatbázis logikai felépítése is jelentősen elősegítheti az integritás megőrzését. Az integritásnak számos foka és ennek megfelelő típusa létezik. Itt csak hármat említünk meg.
A formai ellenőrzés viszonylag egyszerűbb feladat. Ezalatt azt értjük, hogy egy adott mezőben valóban az ott engedélyezett érték áll-e. Árulkodó jel, ha egy családnév pontosvesszőt tartalmaz, vagy egy személy testmagassága három és fél méter (*domain sértés).*
Számos esetben kell annak a feltételnek teljesülnie, hogy az adatbázisból az egyik helyről kiolvasott adatelemnek meg kell egyeznie valamely más helyről kiolvasott másik adatelemmel (referenciális *integritás).*
Sokkal bonyolultabb kérdés a *strukturális integritás* ellenőrzése. Ezalatt azt kell értenünk és ellenőriznünk, hogy nem sérült-e meg valamely feltételezés, amelyre az adatbázis szerkezetének megtervezésekor építettünk. Leggyakrabban előforduló ilyen hiba az előzetesen feltételezett *egyértelműség* megszűnése. Például probléma lehet nem mohamedán országokban, ha egy férfiról egyidejűleg két érvényes bejegyzés van különböző feleségekkel. De ide tartozik az összes olyan *kényszer (constraint)* is, amelyek miatt az adatbázisban található adatok között bármiféle kapcsolat van. Ezek a kapcsolatok olykor nyilvánvalóak (mint pl. az előző példában), máskor jóval kevésbé azok. Az utóbbiak közé tartoznak a *függőségek* különböző fajtái, amikor egyes adatbázis-értékek meghatároznak más adatbázisbeli értékeket.

1.2.4 Szinkronitás

A ma használatos adatbáziskezelő rendszerek általában többfelhasználósak, és nagyon gyakran térben elosztott számítógéphálózaton üzemelnek.
Fontos, hogy az azonos adatokon közel egyidőben műveleteket végző felhasználók beavatkozásainak ne legyenek nemkívánatos mellékhatásai egymás tevékenységére,

illetve az adatbázis tartalmára. Ezt a *tranzakciókezelés* fogalmába tartozó módszerek képesek biztosítani jól bevált eszközök - pl. zárak (lock-ok) - rendszerével.

1.3 Az adatbázissal kapcsolatos tevékenységek szintjei

Az adatbázissal kapcsolatba kerülő személyek tevékenységük szerint négy jellegzetes csoportba tartozhatnak.

- Képzetlen felhasználó ("naiv user")

A felhasználók legszélesebb rétege, akik csak bizonyos betanult ismeretekkel rendelkeznek a rendszerről (pl. légitársaság alkalmazottja, amikor helyet foglal egy járatra), vagy még ennyivel sem (pl. áruházi katalógus lapozgatója).

- Alkalmazás programozó

Alkalmazás programozó az a szakember, aki a (képzetlen) felhasználó által használt programot készíti és karbantartja. Szaktudásánál fogva ismeri azt a nyelvet, amely lehetővé teszi az adatbázisban tárolt adatok elérését.
Ez olyan feladat, amely programozót igényel, de megoldásához nem szükséges, hogy az illető az adatbázis belső szerkezetébe is belelásson, vagy a szerkezetet (a tárolt adatok megőrzése mellett) módosítani tudja.

- Adatbázis adminisztrátor

Hagyományosan így nevezzük azt a személyt, aki az adatbázis felett gyakorlatilag korlátlan jogokkal bír. Vannak olyan kitüntetett tevékenységek, amelyeket kizárólag ő végezhet el az adatbázison, mint pl.:

Generálás:
: Az adatbázis létrehozása ("felállítása"), szerkezetének kijelölése, és annak meghatározása, hogy milyen állomány-szerkezetben tároljuk az adatokat.

Jogosítványok karbantartása:
: A hozzáférések jogának naprakészen tartása, módosítása.

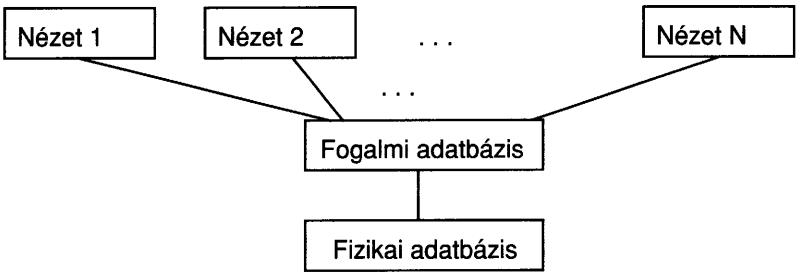
Szerkezetmódosítás:
: Az adatbázis eredeti szerkezetének módosítása. Ez feltételezi azt az alapvető igényt, hogy eközben egyetlen adat se semmisüljön meg azért, mert a régi adatok mellé újabbakat is felveszünk a tárolandók közé.

Mentés-visszaállítás:
: Célszerű lehet adatbiztonsági okokból időszakonként másolatot készíteni az adatbázisról. Ha az adatbázis megsérül, ez a másolat teszi lehetővé a visszaállítást a mentés időpontjának állapotába. A mentést alkalmas célprogram felhasználásával bárki elvégezheti, de a visszaállítás nagy körültekintést igénylő feladat.

DBMS tervező/programozó:
: Tudja, hogyan kell DBMS-t készíteni, ami különösen specializált tudást igényel.

2 Az adatbáziskezelők felépítése

A mai adatbáziskezelők bonyolult hardver-szoftver rendszerek. Komplexitásuk az operációs rendszerekével összemérhető, sőt, gyakran nagyobb annál. Egy ilyen rendszer megtervezése, implementálása, karbantartása nem egyszerű feladat, amelyre kifinomult módszerek léteznek. Ismertetésük túlmutat e jegyzet keretein, itt csak a legfontosabb modellezési, tervezést segítő elvek bemutatására van mód.
Mint a mérnöki gyakorlatban olyan sok más helyen, itt is eredményes a rétegezési koncepció alkalmazása. Az alapgondolat az, hogy az eredeti problémát több részre kell bontani úgy, hogy az egyes részek egymásra épüljenek, de egymással csak minél kisebb felületen érintkezzenek. Jól ismert példa minderre a számítógéphálózatok ISO OSI modellje [6]. Hasonló modell, sőt modellek léteznek az adatbáziskezelők számára is: a legegyszerűbb 3 rétegűtől kezdve a 7 rétegű modellig. Jelen jegyzetben részletesebben egy 3 rétegűvel ismerkedünk meg (2.1. ábra).

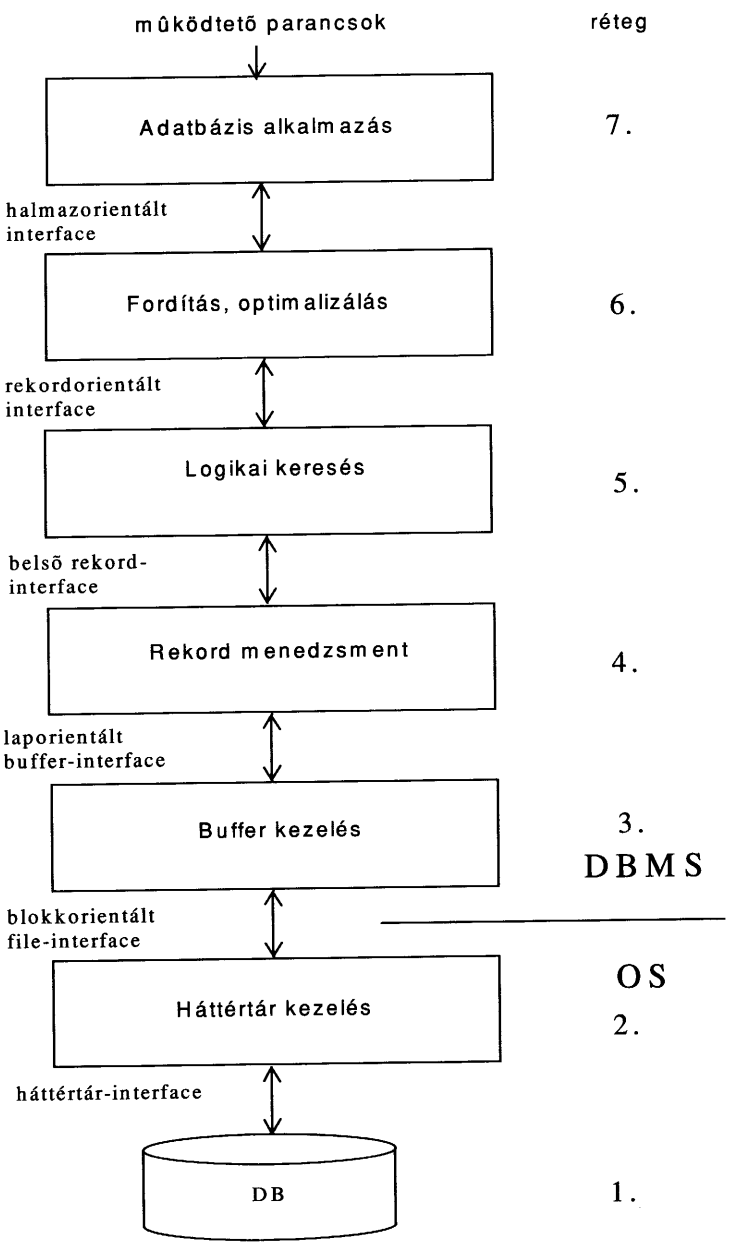


2.1. ábra: Adatbáziskezelők 3 rétegű architektúrája

A legalsó réteg a *fizikai adatbázis*. Itt valósul meg az adatbázis adatainak a fizikai tárolókon való elhelyezése. Ide érthetjük azokat az adatstruktúrákat is, amelyekben a fizikai tárolást megvalósítjuk (ld. 3. szakasz). Ehhez a réteghez tartozó fogalmak: *kötet, állomány, blokk, directory, vödrös hashing*, stb.
Középen helyezkedik el a *fogalmi (logikai) adatbázis*. Ez nem más, mint a való világ egy darabjának leképezése, az a modell, ahogy az adatbázis tükrözi a valóság egy részét. A fogalmi adatbázis határozza meg, hogy melyik adatot hogyan kell értelmezni. Pl. egy könyvtári adatbázisban ide tartozhat a kölcsönző személyek neve, kölcsönzőjegyének száma, egy kötet lelőhelye, ETO száma, példányszáma, címe, szerzője, kiadója, értéke, stb. A fogalmi adatbázishoz tartozó sémát gyakran *belső* vagy *logikai sémának* is nevezik.
Nézet (view, látvány) az, amit és ahogy a felhasználó az adatbázisból lát. Ha az adatbázisnak több felhasználási lehetősége van, ezek mindegyikéhez külön nézet tartozhat. Ez lehet a felhasználók jogosítványaihoz kötött is. (Pl. a légitársaság egységes nyilvántartásából más adatok érdekesek, ha a pilóták szabadságolási tervét készítjük, és az adatok másik körére van szükségünk, ha egy gép utaslistáját akarjuk megtekinteni.) A nézetekhez tartozó sémákat gyakran *külső sémáknak* is nevezik.

Minden jól megtervezett, a rétegezési koncepció alapján felépített rendszerben cél az, hogy a rétegek egymástól függetlenül megváltoztathatók, kicserélhetők legyenek, amennyiben a rétegek közötti interfészek közben változatlanok maradnak. Az adatbáziskezelés világában ezt az elvet az *adatfüggetlenség (data independence) elvének* nevezik.
Kétféle adatfüggetlenségről szokás beszélni: a fizikai és a fogalmi adatbázis között értelmezhető *fizikai adatfüggetlenségről,* ill. a fogalmi adatbázis és a nézetek között értelmezhető *logikai adatfüggetlenségről.*
A fizikai adatfüggetlenségen azt az elvárást értjük, hogy a fizikai szinten, a fizikai működés sémáiban véghezvitt változások ne érintsék a fogalmi (logikai) adatbázist. Ha ez teljesül (gyakorlatilag mindig), akkor a fizikai adathordozó egy teljesen más paraméterekkel rendelkezőre kicserélhető (pl. meghibásodás, technikai fejlődés, stb. miatt), vagy az állományszervezés módja megváltoztatható anélkül, hogy az adatbázisban bármilyen logikai változás lenne érzékelhető.
Logikai adatfüggetlenségről akkor beszélünk, ha a logikai adatbázis megváltozása nem jár az egyes felhasználásokhoz-felhasználókhoz tartozó nézetek megváltozásával. Ez az elvárás már nem teljesül minden esetben.

Illusztrációképpen bemutatjuk a 2.2. ábrán az adatbáziskezelőnek és környezetének egy tipikus, hétrétegű modelljét.



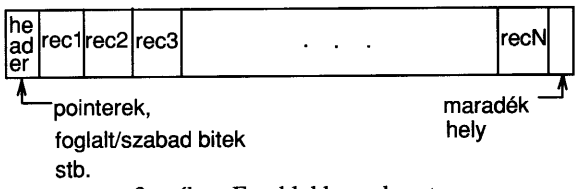
2.2. ábra: Adatbáziskezelő (és környezete) statikus 7 rétegű modellje

3 A fizikai adatbázis

A 2.1. ábrának megfelelően most az adatbáziskezelő rendszerünknek a legalsó szintjét vizsgáljuk meg, azt, hogyan lehet az adatrekordjainkat célszerűen tárolni a háttértárolón annak érdekében, hogy egy keresett rekordot vagy rekordok egy halmazát minél gyorsabban el tudjuk érni. Ennek eléréséhez az adott tárolóeszköz ismerete is szükséges. Ebben a fejezetben a mágneslemezes háttértáron való hatékony tárolás lehetőségeit vizsgáljuk meg. Mindez természetesen nem jelenti azt, hogy a diszkek szerepe kizárólagos. Különösen érdekes - és jelentősen eltérő - az olyan adatbáziskezelők felépítése és működése, ahol a teljes adatbázist memóriában tárolják. Ilyenek 2005-ben már kereskedelmi forgalomban is megjelentek.
A mi, diszkekre vonatkozó megállapításaink is csupán számos egyszerűsítő feltétel fennállása esetén igazak. Ezeket foglaljuk össze először.
Négy műveletet tervezünk megvalósítani, melyek a

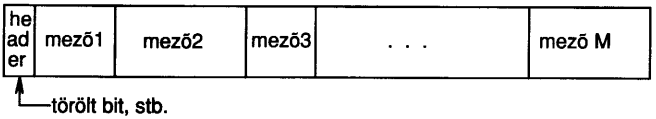
- keresés,
- beszúrás,
- törlés és a
- módosítás.

A számítógépen futó operációs rendszer az adatbázis adatait is állományokban (file-okban) tárolja. Az állományok azonos méretű blokkokból épülnek fel, a blokkméret általában 512-8192 byte. Az operációs rendszer tartja nyilván, hogy egy állományhoz mely blokkok tartoznak. Minden blokknak abszolút címe (is) van, egyetlen fejmozgatással és I/O művelettel a blokk elérhető, adatai az operatív tárba juttathatók. A szükséges adatokat minden esetben a mágneslemezről kell beolvasni, nincs lehetőségünk arra, hogy egyidőben több blokkot a számítógép memóriájában tároljuk.
Cél: a fent felsorolt négy művelet minél gyorsabb elvégzése.
Mivel a lemezműveletek időigénye több nagyságrenddel nagyobb ahhoz képest, mintha az adatokat a memóriában kellene megkeresni ugyanilyen szervezés mellett, ezért a fenti műveletek időigényét alapvetően a blokkelérések száma fogja meghatározni. A fizikai adatszervezés célját ezért úgy is átfogalmazhatjuk, hogy úgy kell az adatainkat a mágneslemezen tárolni, hogy a kért adat a lehető legkevesebb blokkművelettel legyen elérhető. A blokkművelet egyaránt jelentheti egy blokk írását vagy olvasását.
A blokkok tartalmazzák az adatrekordokat, általános struktúrájuk a 3.a. ábrán látható.



3.a. ábra: Egy blokk szerkezete

Lényeges, hogy a rekordok *blokkhatárt nem lépnek át*, így a blokkok általában nincsenek teljesen kihasználva.
A rekordok általános szerkezete a 3.b. ábrán látható.



3.b. ábra: Egy adatrekord szerkezete

A rekordokat megkülönböztethetjük, mint kötött vagy szabad rekordokat. Egy rekord kötött, ha mutató (pointer) mutat rá. Ekkor a rekordot a helyéről nem mozgathatjuk el anélkül, hogy a rá mutató pointert is meg ne változtatnánk. Szabadnak nevezzük a rekordot, ha mutató nem mutat rá. A szabad rekordok segíthetnek a háttértár hatékonyabb kihasználásában.

A rekordokat számos módon megcímezhetjük. Legegyszerűbb esetben minden rekordnak lehet egy abszolút fizikai címe. Gyakoribb ennél, hogy megadjuk annak a blokknak a fizikai címét, amely a rekordot tartalmazza, plusz egy offsetet, amely a blokkon belüli kezdőcímet adja meg. Ezen kívül logikailag is megcímezhető egy rekord, pl. ha megadjuk egy kulcsának az értékét.
Másrészről, a rekordok lehetnek rögzített vagy változó formátumúak is. Változó lehet pl. a formátumuk, ha változó hosszúságú mezőt vagy ismétlődő csoportot (tip. hálós adatbázisoknál, ld. 6. fejezet) tartalmaznak. A továbbiakban a változó hosszúságú rekordok problémájával csak a 3.4. szakaszban foglalkozunk.
Feltételezzük továbbá, hogy az adatállományok valamennyi rekordjában a mezők ugyanabban a sorrendben fordulnak elő és a hosszuk is minden rekordban azonos.
Tároljuk valahol a rekordokkal kapcsolatban azt az információt is, hogy az egyes mezőket hogyan kell értelmezni (integer, real, string, stb.). Így ha a blokk header után a rekordok tömörítve következnek, egyértelműen tudjuk a mezőket dekódolni. Ekkor már csak arról kell gondoskodni, hogyan különböztessük meg az "élő" rekordokat az üres helyektől. Egy lehetőség, ha a blokkheaderben egy számláló jelzi a blokkon belüli rekordok aktuális számát, de más megoldások is elképzelhetők.

A továbbiakban három fizikai szervezési módot vizsgálunk meg részletesen: a heap, a hash és az indexelt szervezést.

3.1 Heap szervezés

Ebben az esetben közelítjük meg legegyszerűbben a tárolás problémáját (heap: halom, kupac). Az adatokhoz nem rendelünk külön struktúrát.

3.1.1 Keresés

Mivel a már megismert struktúrákon kívül (blokk, rekord) új struktúrát nem hozunk létre, a tárolásban nincs rendszer. Egymás után beolvassuk a háttértárról a memóriába a kérdéses rekordokat tartalmazó állomány blokkjait, majd végigolvassuk a blokkokat mindaddig, amíg csak rá nem találunk a keresettre (lineáris keresés). Szerencsés esetben csak egyetlen blokkot, szerencsétlen esetben az állomány valamennyi blokkját végig kell olvasnunk. Így egy meghatározott rekord kiolvasásához átlagosan (összes blokkok száma+1)/2 számú blokkműveletet kell elvégezni.

3.1.2 Törlés

A törlendő rekordot megkeressük. A rekord headerben jelezzük, hogy a terület felszabadult, tehát ha a rekord nem kötött, akkor a terület felülírható. Időnként szükség lehet a gyakori törlések következtében szétszóródott lemezterületek összegyűjtésére és egyesítésére. Az ú.n. szemétgyűjtő programmodul (garbage collection) a törölt jelzés alapján tudja, hogy ezt a területet felül lehet írni.

3.1.3 Beszúrás

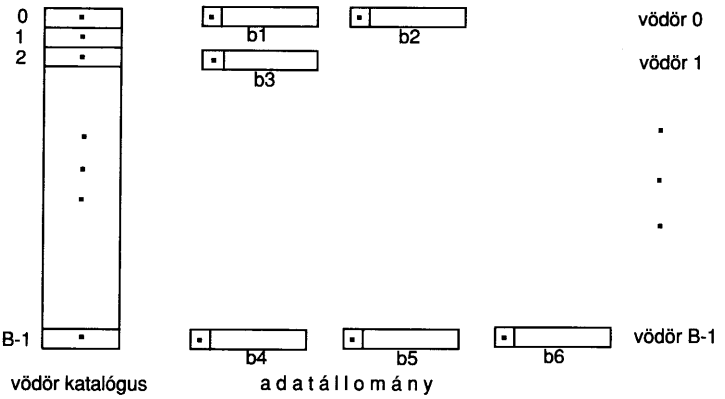
Először a törlés által felszabadított területeken próbálkozunk. Ha itt nem találunk helyet, akkor az állomány végén próbálkozunk. Ha itt sincs elég szabad terület, akkor az operációs rendszert kell kérni, hogy bővítse az állományt egy új blokkal.

3.1.4 Módosítás

A módosítás egyszerű, ha a módosított rekord nem hosszabb, mint az eredeti (ezt tételezzük fel). Ekkor a módosítás egy rekord megkeresését, a rekord felülírását majd a rekordot tartalmazó blokknak a háttértárra visszaírását jelenti.

3.2 Hash állományok

A hash címzés vagy csonkolásos címzés onnan kapta nevét, hogy elvileg a keresés kulcsának bitmintájából csonkolással is nyerhető egy cím, amely alapján a keresett rekord megtalálható.
A hashelés legegyszerűbb változatában minden rekordhoz egy egyértelmű címet rendelünk hozzá az ú.n. hash függvény segítségével. A hash függvény a rekord K (keresési) kulcsát egyértelműen képezi le egy intervallumra. Az intervallum legalább akkora, mint a szóba jöhető rekordok maximális száma. Így a rekord kulcsának ismeretében egy egyszerű számítással megkaphatjuk a rekord tárolási helyét, tehát egyetlen blokkműveletteI elérhetjük a rekordot.
Ez a megközelítés - bár igen gyors rekordhozzáférést tesz lehetővé - ebben a formájában gyakorlatban alig használható, mert a háttértárat igen rosszul használná ki.
Egy gyakorlati megoldást az ú.n. *vödrös hashelés* jelent, melynek alapgondolata és fogalmai a 3.2. ábrán követhetők.
Osszuk fel az állományt B (Bucket=vödör, bugyor) részre úgy, hogy minden rész legalább egy blokkból álljon! Hozzunk létre egy B számú mutatóból álló ú.n. vödör katalógust, amelyben minden mutató az állomány egy-egy blokkcsoportjának (vödörnek) a címét tartalmazza. Definiáljunk továbbá egy hash függvényt (h), amely a kulcsok szóbajöhető értékkészletét leképezi a [0, B-1] tartományra mégpedig lehetőleg egyenletesen, ha a kulcs befutja a szóba jöhető értékeit.
A hash szervezés lényege ezután az, hogy azt a rekordot, amelyiknek a kulcsa K értékű, mindig a h(K)-adik vödörben kell tárolni. A tárolás hatékonysága nagymértékben a hash függvény megalkotásán múlik, másrészt azon, hogy az adatállomány nagysága jól becsülhető, ill. közel állandó-e.
Egy gyakran használt hash függvény a h(K)=(c*K)mod B, ahol c egy alkalmasan megválasztott állandó.



3.2. ábra: Vödrös hash szervezés

3.2.1 Keresés

1. Meghatározzuk a rekord kulcsát: v
2. Kiszámítjuk h(v)-t.
3. Kiolvassuk a vödör katalógus h(v)-adik bejegyzését, ezen a címen kezdődő vödörben kell a rekordnak lennie, ha benne van egyáltalán.

Végigolvassuk a vödör első blokkjának valamennyi nem törölt és nem üres rekordhelyét. Ha valamely rekordnak v a kulcsa, akkor megtaláltuk a rekordot. Ha nem, akkor a vödör következő, ú.n. túlcsordulási blokkját vizsgáljuk végig hasonló módon. (Ezt a blokkot a blokk header-ben lévő mutató címzi.) Ha a vödör egyik blokkjában sem találtuk meg a v kulcsú rekordot, akkor az biztosan nincs az adatbázisban.

Vegyük észre, hogy egy vödrön belül lényegében lineáris keresést végeztünk. Ugyanakkor nem kellett a teljes adatállományt végigkeresni, hanem csak egy meghatározott vödörhöz tartozó blokkokat, átlagosan az állomány 1/(2B)-ed részét, amennyiben a vödrök egyforma hosszúak.

3.2.2 Beszúrás

Helyezzük el az állományban a K kulcsú rekordot! Ehhez kiszámítjuk h(K) értékét, és kiolvassuk a vödör katalógus h(K)-adik bejegyzését. Először végigkeressük az így meghatározott vödröt a K kulcsú rekord után. Ha megtaláljuk, akkor hibaüzenetet küldünk, mivel nincs értelme két különböző helyen azonos kulcsú rekordokat tárolni. Ha nem találjuk a K kulcsú rekordot, akkor az első szabad vagy törölt helyre beírjuk a rekordot miközben megfelelően beállítjuk a "törölt" bitet. Ha minden hely foglalt, akkor a vödörhöz egy új blokkot kell fűzni, és a rekordot itt kell elhelyezni.

3.2.3 Törlés

Megkeressük a kívánt rekordot a már jól ismert módon. A törölt bitet bebillentjük.

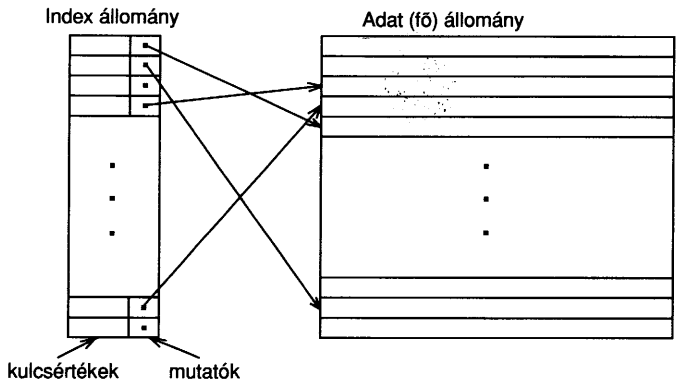
3.2.4 Módosítás

Ha a módosítás nem érint kulcsmezőt, akkor egyszerűen végrehajtható: megkeressük a módosítandó rekordot tartalmazó blokkot, és a módosítás elvégzése után a blokkot visszaírjuk a háttértárra. Ha a rekordot nem találtuk, akkor hibaüzenetet küldünk.
Ha a módosítás kulcsmezőt is érint, akkor a módosítás egy törlés és egy beszúrás egymás utáni végrehajtására vezet, mivel a módosított rekord feltehetően egy másik vödörbe fog kerülni.

Jól érzékelhető, hogy a hash állományszervezés igen gyors lehet, ha a vödrök hossza kicsi. Szélsőséges esetben, ha valamennyi vödör egyetlen blokkból áll, akkor minden rekord egyetlen blokkhozzáféréssel elérhető, feltéve, hogy a vödör katalógus elég kicsi ahhoz, hogy a memóriában lehessen tárolni. Ennek ára az, hogy a háttértár valószínűleg nincs jól kihasználva. Ellenkezőleg, ha a vödrök száma kicsi és emiatt a vödrök hosszúak, akkor a háttértár kihasználtsága javul, azonban a vödrökön belüli lineáris keresés miatt az egy rekord megtalálásához szükséges blokkelérések száma nő.
A hash állományszervezés másik jellegzetessége, hogy az ú.n. "tól-ig" kereséseket nem támogatja (ilyenkor mindazon rekordokat keressük az adatbázisban, amelyeknek kulcsa egy adott intervallumba esik). Ha ilyen feladat gyakran adódik, akkor valamilyen indexelt megoldás alkalmazása célszerű.

3.3 Indexelt állományok

Ha egy könyvtárban keresünk egy könyvet, nem nézzük végig a könyvtár raktárát és olvassuk végig valamennyi könyvcímet és/vagy szerzőt. Helyette a könyvtári katalógust lapozgatjuk, amelynek mérete töredéke a teljes könyvállományénak, így könnyebben kezelhető (katalóguscédulák, mikrofilm), ráadásul abc-be rendezett, szemben a könyvraktárral. Továbbá, a katalógus többféle szempont szerint is lehet rendezve: akár témakör, máskor könyvcím vagy szerző szerint. A keresés is lényegesen gyorsabb benne, hiszen a rendezettsége miatt a lineárisnál hatékonyabb keresési algoritmusokat alkalmazhatunk. A megtalált katalóguscédula azután megmutatja, hogy a raktárban melyik polcról lehet a keresett művet leemelni.
Ugyanez az *indexelt szervezés* alapgondolata is: a keresés kulcsát egy ú.n. index állományban (kb. katalógus) megismételjük, és a kulcshoz egy mutatót rendelünk hozzá, amely a tárolt adatrekord helyére mutat. Az indexelt állományszervezés alapstruktúráját és fontosabb fogalmait a 3.3.a. ábra mutatja.
A kulcsot és a mutatót is rögzített hosszúsággal ábrázoljuk. Az index állományt mindig rendezve tartjuk. Ha a kulcs numerikus, akkor a rendezés triviális. Ha a kulcs karakterfüzérből áll, akkor lexikografikus rendezést alkalmazhatunk. Összetett kulcs esetén, amikor a kulcs több mezőből áll, definiálnunk kell a rendezés módját, azt, hogy a kulcsmezők mely sorrendje alapján történjen a rendezés. Általában nincs akadálya, hogy több indexállományt is létrehozzunk ugyanazon adatállományhoz, különböző (vagy különbözően rendezett) kulcsmezőkkel, bár vannak nehézségek (ld. 3.3.4. szakasz). Vegyük észre, hogy az indexállomány (azonos hosszúságú) rekordjai szabadok, így könnyen mozgathatók, jól karbantarthatók. Ugyanakkor az adatállomány rekordjai valamilyen értelemben kötöttekké válnak.



3.3.a. ábra: Indexelt szervezés

Mindeddig nem volt szó arról, hogy az indexrekordokban található mutatók hogyan azonosítják a keresett rekordot. Két karakterisztikusan különböző megoldás lehetséges:

1. mutatót állíthatunk minden egyes adatrekordra, vagy
2. mutatót állíthatunk adatrekordok egy csoportjára, tipikusan az egy blokkban levőkre.

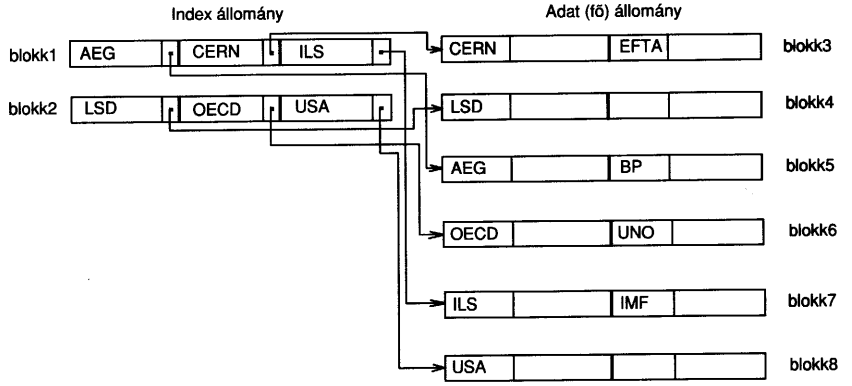
Az első esetben *sűrű indexekről*, a másodikban *ritka indexről* beszélünk.

3.3.1 Ritka indexek

A ritka indexelésnek megfelelő hétköznapi példa a szótárak lapjainak felső sarkába írott index, amely csak azt azonosítja, hogy a keresett címszó a szótár melyik oldalán található. Az adatbáziskezelésben használatos ritka indexelés esetén az indexrekordok azt határozzák meg, hogy az adatállomány rekordjai melyik blokkban találhatók. Ennek következtében *egy blokkon belül* az adatrekordok szabad rekordoknak tekinthetők. Ritka indexek esetén az *adatállományt is rendezetten kell tárolni* legalábbis abban az értelemben, hogy egy blokkban kell, hogy legyen valamennyi olyan adatrekord, amelyeknek a kulcsa egy adott intervallumba esik. Az adott blokkra mutató indexrekord a blokk címén kívül a legkisebb (vagy a legnagyobb) értékű kulcsot fogja tartalmazni.

3.3.1.1. Keresés

Tételezzük fel, hogy a k_1 kulcsú rekordra van szükségünk. Az indexállományban megkeressük azt a rekordot, amelyiknek k_2 kulcsa a legnagyobb azok közül, amelyek még kisebbek k_1-nél. A keresés lehet pl. bináris, hiszen az indexállomány kulcs szerint rendezett. A k_2 kulcsú indexrekord mutatója megcímzi azt a blokkot, amelyet végig kell keresni a k_1 kulcsú adatrekord után. A blokkon belüli keresés lehet lineáris is, annak időigénye még mindig jóval kisebb a blokk beolvasásának idejénél. De használhatunk bináris keresést itt is, ha a blokkon belül is rendezetten tároljuk az adatrekordokat - ami azonban nem szükségszerű.



3.3.1. ábra: Ritka index szervezés

3.3.1.2. Beszúrás

Tételezzük fel, hogy a k_1 kulcsú rekordot akarjuk tárolni. Ehhez először megkeressük azt a blokkot, amelyben a rekordnak lennie kellene, ha az adatállományban lenne. Legyen ez a B_i blokk. Ezután két eset lehetséges: vagy van elegendő hely a B_i blokkban a k_1 kulcsú rekord számára vagy nincs. Ha van, akkor a rekordot beírjuk a B_i blokkba. Ha nincs, akkor helyet kell számára csinálni. Egy lehetőség, hogy kérünk egy új, üres blokkot (B_n), majd a B_i blokk rekordjait (beleértve a k_1 kulcsút is) megfelezzük B_i és B_n között. Meghatározzuk mindkét blokkban a legkisebb kulcsú rekordot. A B_i-hez tartozó indexrekordban szükség esetén korrigáljuk a kulcsmező értékét. A B_n-hez tartozó legkisebb kulccsal és B_n címével új indexrekordot képezünk, amelyet a megfelelő pozícióban elhelyezünk az indexállományban. Ehhez esetleg az indexállományt is újra kell rendezni.

3.3.1.3. Törlés

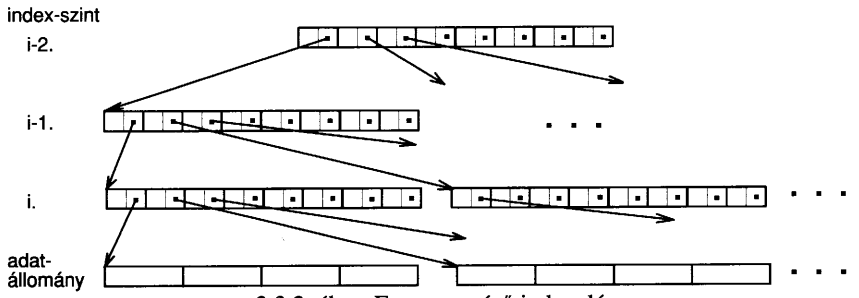
Tételezzük fel, hogy a k_1 kulcsú rekordot kívánjuk törölni. Ehhez először megkeressük azt a blokkot, amelyik a rekordot tartalmazza, legyen ez B_i. Ha a k_1 kulcs a blokkban nem a legkisebb, akkor a rekordot egyszerűen töröljük, a keletkező lyukat akár rögtön meg is szüntethetjük a rekordok blokkon belüli mozgatásával. Ha k_1 volt a legkisebb kulcs a blokkban, akkor az indexállományt is korrigálni kell B_i új, legkisebb kulcsának megfelelően.
Ha a B_i blokkban a k_1 kulcsú volt az egyetlen rekord, akkor a B_i-re mutató indexrekordot is törölni kell, az üres adatblokkot pedig fel kell szabadítani.

3.3.1.4. Módosítás

A módosítás egyszerű, ha nem érinti a rekord kulcsát. Ekkor meg kell keresni a szóbanforgó rekordot, a módosítást elvégezni majd az érintett adatblokkot visszaírni a háttértárra.
Ha a módosítás kulcsmezőt is érint, akkor egy törlést követő beszúrás valósíthatja meg egy rekord módosítását.

3.3.2 B*-fák, mint többszintes ritka indexek

Az indexelt szervezésnél $\log_2$N-nel (N az indexállomány blokkjainak száma) arányos átlagos keresési idő érhető el, amely lényegesen kisebb, mint a heap szervezésé (N-nel arányos), de elmarad a hash szervezésé (akár konstans 1 is lehet) mögött. Cserébe a háttértár kihasználtsága változó méretű adatállomány esetén is kézben tartható.
A szervezés bonyolítása árán lehetőség van a blokkelérések számát csökkenteni úgy, hogy $\log_k$N-nel arányos keresési időt érjünk el. Igen nagy méretű adatállományok esetén, ill., ha k elég nagy, jelentős az elérhető nyereség. Ennek ára, hogy az indexeket egy k-ágú fában kell tárolnunk és az adatállomány változása során az indexfát is gondosan karban kell tartanunk. Az ezzel járó többlet adminisztráció és az indexállomány valamelyest megnövekedett mérete áll szemben a gyorsabb blokkeléréssel.
Az alapgondolat az, hogy az indexállományban való keresést meggyorsíthatjuk, ha az indexállományhoz is (ritka) indexet készítünk hasonló szabályok szerint. Az eljárás mindaddig folytatható, ameddig az utolsó index egyetlen blokkba be nem fér. Az i-1. index tehát egyidejűleg ritka indexe az i. indexnek és adatállománya az i-2. indexnek. A legalsó szint mutatói az adatállomány egy-egy blokkjára mutatnak, a fölötte levő szintek mutatói pedig az index állomány egy-egy részfáját azonosítják (ld. 3.3.2. ábra).



3.3.2. ábra: Fa szervezésű indexelés

A fa szervezésű indexeknek számtalan változata képzelhető el. Itt a továbbiakban arról a változatról lesz csupán szó, amelynek minden levele és csomópontja pontosan blokkméretű és a gyökértől a levelekig vezető út mindig ugyanolyan hosszú, tehát a fa kiegyenlített ("balanced tree"). Szokásos még, hogy az egy csomópontban ábrázolt k mutatóhoz csak k-1 kulcsot tárolnak, mert a kulcs jelentése a kijelölt részfában tárolt legkisebb kulcsérték. Így az indexblokkok első kulcsérték bejegyzése nem hordozna információt. Az ilyen indexelést nevezik B*-fa ("bé-csillag fa") indexeknek.

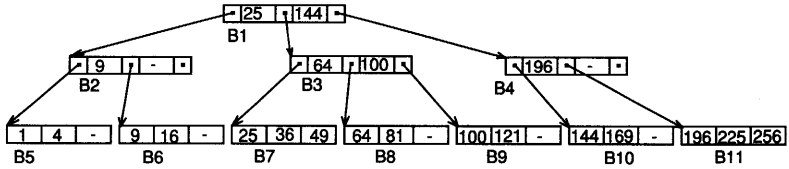
3.3.2.1. Keresés

Az eljárás hasonló ahhoz, mint amit az egyszintű ritka indexek esetén kellett végrehajtani, csupán az indexállományban keresést végezzük több lépésben.
Tételezzük fel, hogy a v_1 kulcsú rekordra van szükségünk. Az indexállomány csúcsán álló blokkban megkeressük azt a rekordot, amelyiknek v_2 kulcsa a legnagyobb azok közül, amelyek még kisebbek v_1-nél. Ennek a rekordnak a mutatója az eggyel alacsonyabb szintű indexben rámutat arra a blokkra, amelyben a keresést tovább kell folytatni egy olyan indexrekord után, amelyiknek v_3 kulcsa a legnagyobb azok közül,

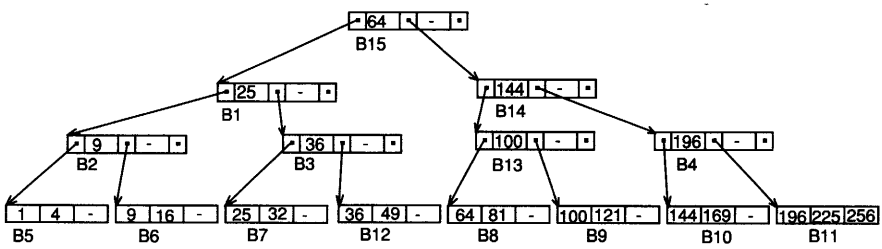
amelyek még kisebbek v_1-nél. Az eljárás mindaddig folytatandó, ameddig az utolsó mutató már az adatállomány egy blokkját azonosítja, amelyben a v_1 kulcsú rekordnak lennie kell.

3.3.2.2. Beszúrás

Az eljárás nagymértékben hasonló a 3.3.1.2. pontban leírtakhoz. Jelentős különbség csak az indexállomány karbantartásában van, amikor is gondosan ügyelni kell arra, hogy az eredeti fastruktúrát, annak kiegyenlítettségét fenntartsuk. Ez bizonyos esetekben az indexhierarchia valamennyi szintjén igényelheti néhány blokk megváltoztatását. A követendő eljárást a 3.3.2.2.a-b. ábra szemlélteti. Az ábrákon azt az egyszerűsítő feltételezést tettük, hogy az adatrekordoknak csupán egyetlen mezőjük van, ami egyben nyilván kulcsmezőül is szolgál.



3.3.2.2.a. ábra: Egy adatstruktúra B*-fa szervezés esetén



3.3.2.2.b. ábra: A B*-fa a 32 kulcsú rekord beszúrása után

3.3.2.3. Törlés

Megkeressük a kívánt adatot és töröljük. Az adatblokkokat lehetőség szerint összevonjuk. Összevonáskor, vagy ha egy adatblokk utolsó rekordját is töröltük, a megszűnt blokkhoz tartozó kulcsot is ki kell venni az index állomány érintett részfájából. Ehhez adott esetben a fa minden szintjén szükség lehet néhány blokk módosítására.

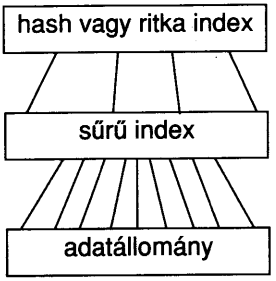
3.3.2.4. Módosítás

Elvben azonos a 3.3.1.4. pontban leírtakkal, a bonyolultabb indexstruktúrából adódó követelményeket értelemszerűen alkalmazva.

3.3.3 Sűrű indexek

A ritka index szervezésnél kihasználtuk azt, hogy egy rekord eléréséhez - a háttértároló fizikai működéséből adódó okok miatt - mindig egy teljes blokkot kell a

memóriába beolvasnunk. A blokkon belüli keresés már igen gyors, így elegendő az indexállományban a blokkcímeket tárolni a bennük található legkisebb kulcsértékkel együtt. Ennek ára viszont az, hogy az adatállományt is rendezetten kell tárolni, hacsak nem megengedhető az "egy adatblokk = egy adatrekord" tárolási sűrűség. Másrészről, az adatállomány rendezettsége miatt nincs mód arra, hogy egy-egy új rekordot tetszőleges szabad helyre szúrjunk be, ami a háttértár kihasználtságát csökkenti.
Mindkét problémára megoldást kínál, ha minden egyes adatrekordhoz tartozik indexrekord. Az indexrekord mutatója általában továbbra is csak a rekordot tartalmazó blokkot azonosítja, néha közvetlenül az adatrekordot. Ez utóbbi megoldással a blokkelérések számát természetesen nem lehet csökkenteni, legfeljebb a blokkon belüli keresés idejét.
Megj.: a "sűrű indexelés" önmagában nem állományszervezési módszer! A sűrű indexre mindig ráépül egy másik elérési mechanizmus is, ritka index vagy hash. A sűrű indexek elsősorban a fő állomány kezelését könnyítik meg, ill. több kulcs szerinti keresést teszik lehetővé (ld. 3.3.4. szakasz).
A sűrű indexek tipikus alkalmazása a 3.3.3. ábrán látható.



3.3.3. ábra: Sűrű index alkalmazása

A bemutatott megoldásnak a hátrányain kívül számos előnye is van:
Hátrányok:
- a sűrű indexnek plusz helyigénye van,
- eggyel több lapelérés kell egy rekord kiolvasásához,
- plusz adminisztrációval jár a sűrű index karbantartása.

Viszont:
- az adatállományt nem kell rendezetten tárolni, ezzel helyet takaríthatunk meg,
- meggyorsíthatja a rekordelérést, mert a ritka index mérete jóval kisebb is lehet, mint sűrű index nélkül,
- támogatja a több kulcs szerinti keresést,
- az adatállomány rekordjai szabadokká tehetők, ha minden további rekordhivatkozás a sűrű indexen keresztül történik (egyetlen mutatót kell megváltoztani).

3.3.3.1. Keresés a sűrű index segítségével

Az index állományban megkeressük a kulcsot, pl. bináris kereséssel. A hozzá tartozó mutatóval elérhetjük a tárolt rekordot.

3.3.3.2. Törlés

Megkeressük a kívánt rekordot. Foglaltsági jelzését szabadra állítjuk. A kulcsot kivesszük az index állományból, és az index állományt időnként - műveletigényessége miatt nem minden törlés után - tömörítjük.

3.3.3.3. Beszúrás

Keresünk egy üres helyet a tárolandó rekordnak. Ha nem találunk, akkor az állomány végére vesszük fel. Beállítjuk a foglaltsági jelzést, és beírjuk az adatot. A kulcsot és a tárolás helyére hivatkozó mutatót a kulcs szerint berendezzük az index állományba.

3.3.3.4. Módosítás

Sűrű indexelés esetén a módosítás viszonylag egyszerű: megkeressük a módosítandó rekordot tartalmazó adatblokkot, majd a módosított tartalommal visszaírjuk a háttértárra. Ha a módosítás kulcsmezőt is érintett, akkor az indexállományt újrarendezzük.

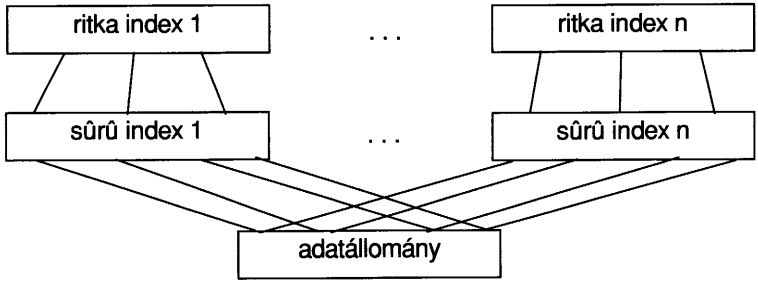
3.3.4 Másodlagos indexek, invertálás

Erősen korlátozott a használhatósága annak az adatbázisnak, amely pl. személyek adatait tartalmazza, de benne megtalálni valakinek az adatait csak akkor lehet, ha pontosan ismerjük az illető személyi számát. (Azért kellene a személyi számot ismerni, mert - mint mindenkit egyértelműen azonosító adatot - keresési kulcsnak tekintettük és ezért a személyi szám szerinti keresést támogattuk valamilyen állományszervezési módszerrel.) Gyakori az az igény, hogy csupán a név alapján is kereshessünk, vagy listát készíthessünk mindazokról, akik egy adott városban laknak. A név és a lakóhely nem kulcsmezők. Általában tehát több mező szerint is támogatni kell az adatrekordok megtalálását. Gyakran ilyenkor is kulcsról beszélnek azzal a mezővel kapcsolatban, amely szerint a keresés történik. Mivel a kulcs fogalma az adatbázisok logikai tervezése kapcsán is előkerül, a félreértéseket elkerülendő, célszerű hangsúlyozni, ha csak a fizikai keresés során tekintünk egy mezőt kulcsnak. Szélsőséges esetben akár minden mező lehet ú.n. *keresési kulcs.*
Az eddig tárgyalt módszerek különböző mértékben támogatják a fenti probléma megoldását. Itt csak annak néhány részletére térünk ki, ha az indexszervezés módosításával-kibővítésével támogatjuk a több kulcs szerinti keresést.
Az egyik kézenfekvő lehetőség, hogy több index állományt is létrehozunk, minden keresési kulcshoz egyet.

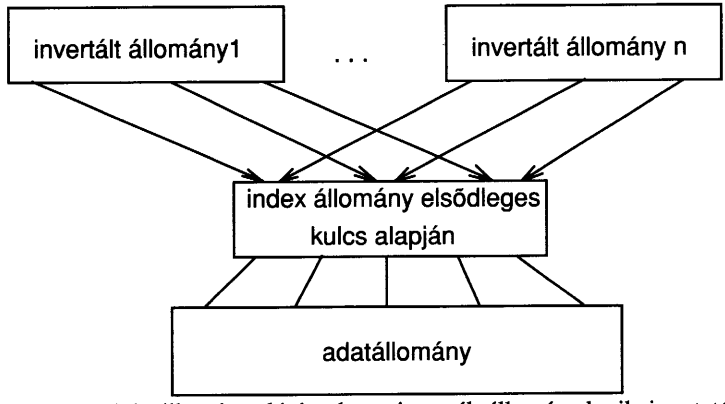
Definíció: Azt az index állományt, amely nem kulcsmezőre tartalmaz indexeket, *invertált állománynak* nevezzük. Az index neve ekkor *másodlagos index.*

Az invertált állomány mutatói
1. lehetnek fizikai mutatók, amelyek pl. mutathatnak
   a./ közvetlenül az adatállomány megfelelő blokkjára (esetleg közvetlenül a rekordra), vagy
   b./ az adatállomány elsődleges kulcsa szerinti (sűrű) indexállomány megfelelő rekordjára, ill.
2. lehetnek logikai mutatók, amelyek az adatállomány valamely kulcsának értékét tartalmazzák.

Az 1.a./ esetben az adatállomány rekordjai kötöttek és ráadásul csak egyetlen invertált állomány esetén használható.
Az 1.b./ esetben eggyel több indirekción keresztül érjük el a keresett rekordot, de az adatrekordok változtatásakor csak az érintett mezőt (mezőket) tartalmazó invertált állományokat és az elsődleges indexállományt kell módosítani.
Ha 2. megoldást választjuk, akkor az adatállomány rekordjai szabadok lehetnek, viszont nem ismerjük még a keresett rekord címét. Ennek megtalálását pl. hasheléssel vagy valamilyen indexeléses módszerrel támogathatjuk.
A 3.3.4.a. ábra azt mutatja be, hogyan lehet egy állományhoz az 1.b/ esetben sűrű index segítségével tetszőleges számú ritka indexet rendelni, a 3.3.4.b. ábra pedig ugyanennek a problémának a megoldását mutatja, ha az invertált állományokban logikai mutatót (az elsődleges kulcs értékeit) alkamazunk a rekordok azonosítására.



3.3.4.a. ábra: Adatállomány elérése több indexen keresztül, sűrű index támogatással



3.3.4.b. ábra: Adatállomány elérése, ha az invertált állomány logikai mutatókat tartalmaz

3.4 Változó hosszúságú rekordok kezelése

Egy rekord változó hosszúságát okozhatja, hogy
a.) egy mező hossza változó, vagy
b.) ismétlődő mező(csoport) van a rekordban (hálós adatbázisoknál gyakori, ld. 6. fejezet).
Az a.) esetben a leggyakoribb megoldás, hogy a változó hosszúságú mező helyett csak egy (fix hosszúságú) mutató van a rekordban, a mező tényleges tartalma egy külön

állományban tárolódik. Így biztosítható, hogy egy állomány csak egyféle rekordot tartalmaz, ami a karbantartást jelentősen megkönnyíti.
A b.) eset kezelésére három megoldás kínálkozik:

- lefoglalt hely módszer: ilyenkor a maximális számú ismétlődéshez elegendő nagyságú helyet foglalunk a rekordnak,
- mutatós módszer: a fenti a.) módszer megfelelője
- kombinált módszer: valamennyi helyet lefoglalunk, ha még több az ismétlődés, akkor a mutatós módszert alkalmazzuk.

3.5 Részleges információ alapján történő keresés

Igen gyakori az a szituáció, amikor egy rekord több mezőjének értékét ismerjük, és keressük azokat a rekordokat, amelyek ugyanezeket az értékeket tartalmazzák ugyanezen mezőikben. Feltételezzük, hogy a mezők egyike sem kulcs.
Egyik lehetőség, hogy több (pl. minden) mezőre felépítünk másodlagos indexeket. Minden specifikált mező-érték alapján előállítjuk a találati rekord-(vagy legalább mutató-) halmazt, majd ezeknek a metszetét képezzük. Nem igazán praktikus.
Másik lehetőség: *particionált* (feldarabolt) *hash függvény*ek alkalmazása:
A h(K) függvény a 3.2. szakaszban egy N hosszúságú címet állított elő, amely egy [0...B-1] intervallumba eső értéket jelentett. Most a hash függvény $h(m_1, m_2,..., m_k)= h_1(m_1)*h_2(m_2)*...*h_k(m_k)$ alakú, ahol m_i-k a rekord összesen k db, releváns mezőjének az értékeit jelentik, h_i az i-edik mezőre alkalmazott hash függvény komponens, * pedig a konkatenáció jele. A $h_i(m_i)$ függvényértékek x_i hosszúságon ábrázolható értéket állítanak elő. A h_i függvényeket tehát úgy kell megválasztani, hogy az $x_1+x_2+...+x_k$ érték, azaz a teljes cím hossza éppen N legyen. Az eljárás az x_i-k egyéb szempontok szerinti megválasztásával hangolható.
Használata: az ismert mezők értékei alapján meghatározhatjuk az N hosszúságú bitmintának az ismert értékű mezőkhöz tartozó darabjait, a többi nyilván tetszőleges lehet. Mindazon vödröket végig kell néznünk illeszkedő rekordok után, melyeknek a sorszáma illeszkedik a kapott bitmintára.

4 A fogalmi (logikai) adatbázis

Ebben a fejezetben a 2.1. ábrán megismert adatbázis-modell középső, logikai részét vizsgáljuk meg részletesebben.

4.1 Adatmodellek, modellezés

Amikor egy adatbázist létrehozunk, a cél az, hogy benne a való - vagy ritkábban egy kitalált - világ adatait tároljuk úgy, hogy belőle a való (kitalált) világról információkat nyerhessünk ahelyett, hogy a valóságból kelljen ugyanazt az információt megszerezni. Általában nincsen mód egy adott probléma- (téma-, jelenség-) körrel kapcsolatos valamennyi adat tárolására, így adatoknak csak meghatározott, szűk körét kezelhetjük. A tárolandó adatok kiválasztásánál klasszikus modellezési szempontok érvényesülnek, azaz a vizsgálat szempontjából fontosnak tartott jellemzőket tároljuk, a többit elhanyagoljuk. Jellemző alatt itt egyaránt értünk tulajdonságokat és kapcsolatokat is. Így az adatbázis a világ egy darabjának egy leegyszerűsített képét képes visszaadni.
Amikor ezt a képet elkezdjük kialakítani, követhetünk bizonyos konvenciókat, ami esetleg számos előnnyel jár. A konvenciók egy része arra vonatkozik, hogy milyen formában, milyen kapcsolatok kialakítását támogassunk az adataink között és hogy milyen műveleteket engedjünk meg az adatainkon. Így ú.n. adatmodelleket hozunk létre. Természetesen, a konvenciókhoz való alkalmazkodás járhat hátrányokkal is, ez esetben megfontolandó egy teljesen egyedi adatmodell megalkotása.
Egy adatmodell tehát hagyományosan két részből áll:

1. formalizált jelölésrendszer adatok, adatkapcsolatok leírására
2. műveletek az adatokon.

Az adatmodell tulajdonságai alapvetően meghatározzák az adatbázis tulajdonságait. A felhasználó számára pedig az adatbázisnak az egyik legfontosabb jellemzője az a forma, amelyben a tárolt adatok közötti összefüggések ábrázolva vannak. Az ábrázolás alapegysége a rekord, ill. a rekordtípus (vagy ezzel analóg, másképpen nevezett konstrukció). Mivel egy adatbázis struktúráját a rekordtípusok közötti kapcsolatok alkotják, ezért az adatmodelleket aszerint osztályozzuk, hogy a rekordtípusok között milyen kapcsolatok definiálása megengedett, azaz a felhasználó szempontjából miként valósul meg az adatok közötti kapcsolatok ábrázolása.
A hálós adatmodellnél (ld. 6. szakasz) a rekordtípusok között (pl. mutatók segítségével) tetszőleges kapcsolatokat szervezhetünk. A relációs adatmodellnél előre szervezett kapcsolat nincs, magukat a kapcsolatokat is relációkkal ábrázoljuk (ld. 5. szakasz). Az objektum-orientált adatmodell (ld. 7. szakasz) objektumokat tartalmaz, amelyek között változatos típusú kapcsolatokat hozhatunk létre. Ezért sok szempontból a hálós adatmodellhez hasonlatos.

Az adatmodell tehát meghatározza, hogy az adatbázisban az adatok milyen struktúrában tárolódnak és milyen mechanizmusokon keresztül lehet az adatokhoz hozzáférni. Így az adatbáziskezelő rendszer legalapvetőbb tulajdonságait rögzíti. Egy adatbáziskezelő rendszer ezért csaknem mindig egyetlen adatmodellnek megfelelően működik.

4.2 Egy majdnem-adatmodell: az egyed-kapcsolat modell

Az *egyed-kapcsolat* (entity-relationship, E-R) modell nem tekinthető a fenti értelemben adatmodellnek, mert benne nincsenek adatműveletek definiálva.

4.2.1 Az E-R modell elemei

Az E-R modell elemei:
- *egyed típusok*
- *tulajdonság típusok*
- *kapcsolat típusok*

Természetesen, a *típus*okhoz mindenütt tartoznak konkrét *esetek (példány, előfordulás)* is, de maga a modellezés a típusok szintjén történik. Összhangban azzal, amit általában típusnak nevezünk, a típus itt is a konkrétan létező egyedek, tulajdonságok, kapcsolatok absztrakciója. Az egyedek (tulajdonságok, kapcsolatok) bizonyos közös jegyek alapján halmazokba rendeződnek. Egy-egy halmaz neve az egyed (tulajdonság, kapcsolat) típusa, a halmazok elemei pedig a példányok (esetek, előfordulások).

4.2.1.1. Entitások

Definíció: *Egyed* (entitás): a valós világban létező, logikai vagy fizikai szempontból saját léttel rendelkező dolog, amelyről adatokat tárolunk.

Megj.: Ennek megfelelően az egyedek megkülönböztethetők kell, hogy legyenek.

Definíció: *Tulajdonság* az, ami az entitásokat jellemzi, amelyen vagy amelyeken keresztül az entitások megkülönböztethetők.

Pl.: entitás lehet(!) egy autó, egy személy, egy szerződés, de még a szeretet is, hiszen megfelelő attribútumok megválasztásával az autók, személyek, stb. megkülönböztethetővé tehetők, azaz "saját létet rendelhetünk hozzájuk". Ugyanakkor általában nem tekinthető entitásnak egy tojás vagy egy hangya, mivel a tojás- vagy a hangya-példányok rendszerint nem különböztethetők meg.
Megj.: Valójában az entitások definiálása modellezési kérdés. A modellalkotón múlik, hogy milyen tulajdonságokat rendel hozzá egy-egy entitáshoz, így biztosítja-e azok megkívánt szintű megkülönböztethetőségét. Elképzelhető, hogy egy tudományos adatbázisban éppen hangyák adatait kell tárolni, amelyekkel különböző kísérleteket végeztek. Ekkor a hangyák megkülönböztethetővé tehetők pl. az elkülönített tárolásuk segítségével, így a hangyák is entitásokká válhatnak.

Definíció: *Egyedek halmaza* (entity set): az azonos attribútum-típusokkal jellemzett egyedek összessége.

Az entitások közös attribútum-típusait zárójelben szokás az entitáshalmaz neve után felsorolni.

Pl.: EMBER(név, szül_dátum, anyja_neve, szeme_színe, személyi_szám)
SZERZŐDÉS(cég1, cég2, dátum, hely, szerződés_tárgya, érték, telj_határidő)

4.2.1.2. Kapcsolatok

Definíció: *Kapcsolat:* entitások névvel ellátott viszonya.

A valóságban az egyedek ritkán léteznek elszigetelten, egymástól függetlenül. Tipikus az, hogy valamilyen kapcsolatban állnak egymással: az emberek cégeknél *dolgoznak*, szerződéseket *írnak alá*, egymással rokoni kapcsolatban lehetnek (pl. *testvére valakinek*). Ezeket a tényeket kifejezhetjük, ha az entitáshalmazok között kapcsolat-típusokat definiálunk. Természetesen a kapcsolat típusok meghatározása szintén modellezési kérdés: az adott feladat dönti el, hogy egy konkrét adatbázisban milyen kapcsolat típusok definiálása szükséges.
Formálisan egy kapcsolat típus nem más, mint entitás típusok névvel ellátott sorozata.
Pl.: DOLGOZIK: EMBER, CÉG
Ez a bináris kapcsolat típus azt fejezheti ki, hogy valaki egy cégnél dolgozik.
ALÁÍR: EMBER, CÉG, SZERZŐDÉS
Ez a ternális (hármas) kapcsolat-típus azt fejezheti ki, hogy egy személy egy cég nevében egy szerződést aláírt.
TESTVÉRE: EMBER, EMBER
Ez a bináris kapcsolat típus azt fejezheti ki, hogy az egyik ember testvére egy másiknak. Ennek a kapcsolattípusnak egy példánya - egy konkrét kapcsolat - pl. azt fejezheti ki, hogy Kis Géza testvére Kis Antalnak.

A kapcsolatok igen sokfélék lehetnek. Fontos szempont, hogy hány entitás halmaz között teremtenek kapcsolatot, vagy hogy egy kiválasztott példány hány másikkal lehet kapcsolatban. Ezen belül érdekes lehet, hogy egy kiválasztott példányhoz mindig tartozik-e egy vagy több másik, ha igen, akkor mennyi a kapcsolódó egyedek minimális, maximális száma, stb. A kapcsolatok teljes mélységű jellemzése gyakran szükségtelen, mi sem törekszünk rá.

Megj.: a mindennapi gyakorlatban rendszerint elfeledkezünk a típusok (halmazok) és a konkrét példányok megkülönböztetéséről. Igen gyakran emlegetünk entitást, kapcsolatot akkor is, amikor valójában entitás- vagy kapcsolat típusról van szó. Ez megtehető általában, mert a szövegkörnyezet miatt többnyire nem okoz félreértést ez a pontatlanság. Engedve a szokásnak, a továbbiakban nem hangsúlyozzuk a különbséget, ha ez kétértelműséget nem okoz.

4.2.1.2.1. Kapcsolatok funkcionalitása (kardinalitás)

Említettük, hogy a kapcsolatok különbözhetnek pl. abban is, hogy egy entitáshalmaz egy eleméhez egy másik entitáshalmaznak hány elemét rendelik hozzá. A legegyszerűbb csoportosításban egy-egy, egy-több vagy több-több kapcsolatról beszélünk.

Definíció: *Egy-egy kapcsolat*: olyan (bináris) kapcsolat, amelyben a résztvevő entitáshalmazok példányaival egy másik entitáshalmaznak legfeljebb egy példánya van kapcsolatban.

Pl.: HÁZASSÁG: EMBER, EMBER
FŐNÖK: OSZTÁLY, EMBER

Megj.1.: Vegyük észre, hogy egy kapcsolat funkcionalitásának meghatározása is modellezési kérdés. Általában igaz ugyanis, hogy a HÁZASSÁG egy-egy kapcsolat, hiszen egy emberhez (egy időben) legfeljebb egy másik embert rendel hozzá. Elégtelen lenne azonban a valóságnak ez a szintű modellezése, ha az adatbázisunknak olyan iszlám országban is működnie kellene, ahol a többnejűség is előfordulhat!

Megj.2.: Egy-egy kapcsolatok még abban is különbözhetnek, hogy az egyik entitáshalmaz példányai minden esetben kapcsolatban vannak-e egy másik

entitáshalmaz egy példányával, vagy nem feltétlenül tartozik hozzá egy másik példány. Az előbbi helyzetre jellemző a FŐNÖK kapcsolat, hiszen általában minden osztálynak van pontosan egy főnöke. Ellenkező irányban mindez már nem feltétlenül igaz, hiszen az EMBER entitáshalmaznak nem minden példánya kell, hogy főnöke legyen valamely osztálynak. A HÁZASSÁG kapcsolatban szereplő entitáshalmazban lehetnek olyan személyek, akik egyedülállók, így egyik irányban sem teljesül, hogy egy kiválasztott példányhoz feltétlenül tartozik is egy másik példány. Ezek a megfontolások a kapcsolatok mélyebb analízisének lehetőségeire utalnak.

Definíció: *Több-egy kapcsolat*: Egy K: E1,E2 kapcsolat több-egy, ha E1 példányaihoz legfeljebb egy E2-beli példány tartozik.

Pl.: TANUL: DIÁK, OSZTÁLY

Definíció: egy kapcsolat több-több funkcionalitású, ha nem több-egy egyik irányban sem.

Pl.: TAN: DIÁK, TANÁR
Ez a kapcsolat azt fejezheti ki, hogy diákok és tanárok "tanítja - tanul nála" viszonyban lehetnek egymással. A kapcsolat több-több funkcionalitású, mert egy tanár több diákot is taníthat és egy diák több tanárnál is tanulhat.
Az EXPORT: ORSZÁG, TERMÉK kapcsolat azt fejezheti ki, hogy az országok termékeket exportálnak. Egy ország többféle terméket exportálhat és egy terméket több ország is exportálhat.
Az adatbáziskezelésben a több-egy (egy-több) kapcsolatok kitüntetett jelentőségűek, mert viszonylag egyszerűen ábrázolhatók, ugyanakkor elegendően általánosak, kifejezőek is.

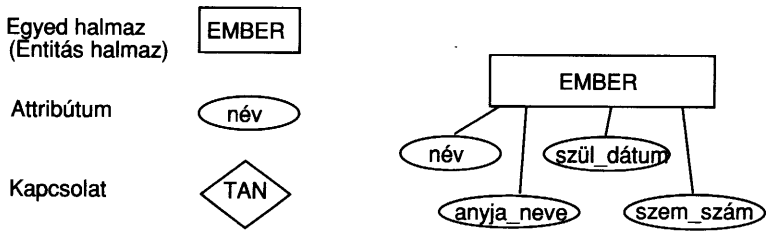
4.2.2 Kulcs

Definíció: Az E-R modellezésnél az attribútumoknak azt a halmazát, amely az entitás példányait egyértelműen azonosítja *kulcsnak* nevezzük.

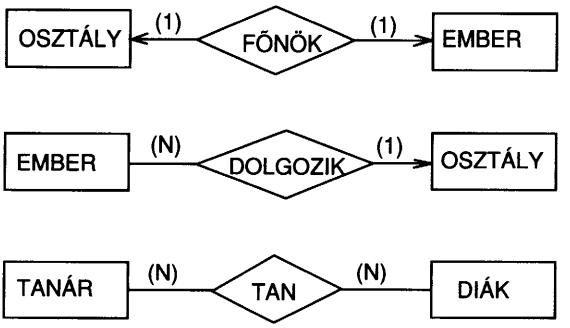
Pl. az EMBER entitáshalmaz elemeit egyértelműen azonosítja a (név, szül_dátum, anyja_neve) attribútumhármas, vagy a személyi_szám attribútum. Az EMBER entitáshalmaznak tehát két kulcsa is van.
Minden entitáshalmaznak legalább egy kulcsa mindig van, hiszen az egyedeknek megkülönböztethetőknek kell lenniük. Ehhez pedig az attribútumok teljes halmaza elegendő, tehát az attribútumok teljes halmaza mindig kulcs. A kulcs attribútumait hagyományosan aláhúzással jelöljük.

4.2.3 Az E-R modell grafikus ábrázolása: E-R diagram

Bár az előbbiekben bevezetett formális jelölésrendszer elegendő az E-R modell megadására, a gyakorlatban elterjedten használnak (különböző) grafikus megjelenítési formákat is. Mi most az eredeti jelölésrendszert mutatjuk be.



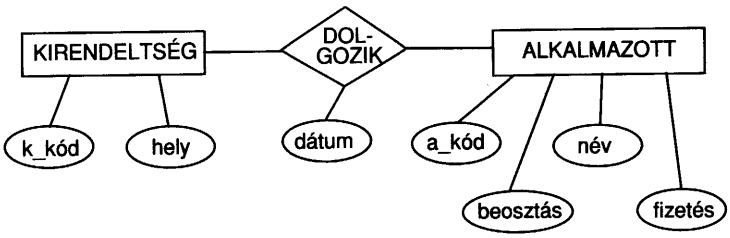
4.2.3. ábra: Az E-R diagram elemei



4.2.3.a. ábra: Kapcsolatok funkcionalitásának egy ábrázolása az E-R diagramoknál

Példa: A Nekeresdi Általános Biztosítónak számos kirendeltsége működik szerte Nekeresdország városaiban, néhányban több is. Minden kirendeltségnek külön kódszáma is van, amely egyértelműen azonosítja a kirendeltségeket. A kirendeltségeken többen is dolgoznak, de egy alkalmazott egy évben csak egy kirendeltségnél vállal munkát. A dolgozókat kódjuk egyértelműen meghatározza, de tárolni kell róluk még a nevüket, beosztásukat és fizetésüket is. Az alkalmazottak időnként munkahelyet változtatnak - de mindig csak jan. 1.-i dátummal -, és a Nekeresdi Általános Biztosítón belül másik kirendeltséghez mennek dolgozni.
A leírás alapján pl. az alábbi E-R modellt alkothatjuk:
entitások:
KIRENDELTSÉG(k_kód, hely)
ALKALMAZOTT(a_kód, név, beosztás, fizetés)
kapcsolat:
DOLGOZIK: KIRENDELTSÉG, ALKALMAZOTT, dátum
Megj.: A dátum attribútum - ami egy évszám, és azt fejezi ki, hogy egy adott évben mely alkalmazottak dolgoztak egy adott kirendeltségen - nem tartozik egyedül sem a kirendeltség sem az alkalmazott entitásokhoz, mindig csak a kettőhöz együtt. Az E-R modell azonban nem engedi meg ilyen attribútumok definiálását. Így kénytelenek vagyunk a "dátum"-ot önmagában entitáshalmazzá tenni és a két érintett entitáshalmaz között értelmezett DOLGOZIK kapcsolatba belevonni.

Mindezt a 4.2.3.b.ábra E-R diagramján is ábrázolhatjuk:



4.2.3.b. ábra: Alkalmazott-kirendeltség modell E-R diagramon

Megj.: Gyakori az a modellezési szituáció, amikor egy entitáshalmaz valamennyi eleme rendelkezik egy másik (általánosabb) entitáshalmaz attribútumaival, de azokon kívül még továbbiakkal is (specializáció). Ez a viszony a kapcsolatok egy speciális típusával az ú.n. "isa"[1] kapcsolattal írható le.

Pl.: SZEMÉLYZET(kódszám, beosztás, fizetés, végzettség)
PILÓTA(kódszám, beosztás, fizetés, végzettség, rep_eng_száma)
ISA: PILÓTA, SZEMÉLYZET

Az isa kapcsolatnak az objektum-orientált modelleknél kitüntetett szerepe van.

[1] is a: angol szóösszevonás

5 A relációs adatmodell

A relációs adatmodellen alapuló adatbáziskezelők ma a legelterjedtebbek, emiatt tanulmányozásuk kitüntetett figyelmet érdemel. Ezért a modell alapvető tulajdonságait ismertető jelen szakasz után a relációs adatbázisok logikai tervezésével a 8. fejezet külön foglalkozik.

A relációs adatmodell mögött a halmazelméleti relációk elmélete húzódik meg. A reláció szót ebben a szakaszban pontosan ebben az értelemben fogjuk használni.

Definíció: halmazok Descartes-szorzatának részhalmazát relációnak nevezzük.

Adott *n* (valódi, azaz azonos elemeket nem tartalmazó) halmaz. A halmazokban található értékek egy-egy ú.n. tartományból (domain) kerülnek ki. Legyenek ezek rendre $D_1, D_2,..., D_n$. A tartományok $D_1 \times D_2 \times .... \times D_n$ Descartes-szorzatában megtalálhatók mindazok a $(v_1, v_2,..., v_n)$ n-esek (*tuple, n-tuple*), amelyekre igaz, hogy $v_i \in D_i$, $\forall i=1, 2, ..., n$-re.

Példa:

$D_1=\{1,2,3\}$, $D_2=\{x,y,z\}$

$D_1 \times D_2 =$ {(1,x) (2,x) (3,x)
(1,y) (2,y) (3,y)
(1,z) (2,z) (3,z)}

Reláció lehet az így keletkezett n-eseknek tetszőleges részhalmaza. Magát a relációt névvel látjuk el.

Pl.: $R_1=\{(1,y) (1,z) (3,z)\}$

$R_2=\{(2,y) (1,z)\}$

Áttekinthetőbben ábrázolhatjuk relációnkat táblázatos formában. A táblázat oszlopai (amelyeknek szintén nevet adunk, ez az *attribútum*) jelentik a tartományokat, amelyekből az egyes oszlopokban található értékek kikerülnek. A táblázat sorai pedig a reláció elemei, az n-esek konkrét előfordulásai. A fejlécben az attribútumok megnevezése található.

Példák:

R_1:

D_1	D_2
1	y
1	z
3	z

SZEMÉLY:

NÉV	KOR	FOGLALKOZÁS.
Nagy Lajos	37	villamosmérnök
Kis Géza	4	informatikus
Közepes János	72	gépészmérnök

Igen gyakran fordul elő, hogy magára a relációra nincs szükségünk, csak arra az információra, hogy melyik relációban milyen attribútumok találhatók. Ezt *relációs sémának* nevezzük. Szokásos jelölése: $R(A_1, A_2, ..., A_k)$, ahol R a reláció neve, A_i-k az attribútumok nevei, ahol $1 \le i \le k$.

Pl.: SZEMÉLY (NÉV, KOR, FOGLALKOZÁS).

Bár általában nem okoz kétértelműséget, esetenként fontos, hogy a relációt, annak egy elemét, valamint a relációs sémát megkülönböztessük egymástól. *Általában* azt a jelölési konvenciót alkalmazzuk, hogy a relációt és az elemeit kis, a relációs sémát pedig nagy betűvel jelöljük.
Ha egy adatbázis több relációs sémát is tartalmaz, akkor a relációs sémák összességének neve *adatbázis séma.*
További elnevezések:

- A relációban lévő oszlopok (attribútumok, tartományok, tulajdonságok) számát a *reláció fokának* nevezzük (arity, aritás).
- A relációban lévő sorok számát (a konkrét előfordulások számát) a *reláció számosságának* nevezzük.

Az előbbiek következménye, hogy

- a reláció nem tartalmazhat két azonos sort,
- az n-esek (sorok) sorrendje nem számít,
- az oszlopoknak egyértelmű nevük van.

Igaz továbbá, hogy az oszlopok sorrendje nem számít akkor, ha az oszlopokra a nevükkel hivatkozunk, de természetesen számít akkor, ha csak a sorszámukkal hivatkozunk rájuk.

5.1 Műveletek relációkon

A relációs adatmodell a relációkon megengedett műveletek meghatározásával válik teljessé. Tekintve, hogy a relációk is halmazok, mégpedig valódi halmazok (ismétlődés nem engedhető meg bennük) néhány halmazalgebrai műveletet relációs műveletként is fel kívánunk használni.

5.1.1 Egyesítés (unió)

Az egyesítés feltétele, hogy az egyesítendő relációknak ugyanannyi attribútumból kell állniuk. Nem szükséges azonban, hogy ezek ténylegesen azonos attribútumokat jelentsenek. Tehát a műveletet ilyenkor mindig el tudjuk végezni, de nem biztos, hogy az eredmény attribútumait sorszámukon kívül nevükkel is azonosítani tudjuk.
Pl.:

R_1:

A	B	C
a	b	c
c	b	a
a	d	c

R_2:

D	E	F
a	c	d
a	d	c
b	b	c

$R_1 \cup R_2$:

1	2	3
a	b	c
c	b	a
a	d	c
a	c	d
b	b	c

5.1.2 Különbségképzés

Ugyanazok a megkötések érvényesek, mint az egyesítésnél.
Pl.:

R_1:

A	B	C
a	b	c
c	b	a

R_2:

D	E	F
a	c	d
a	d	c

$R_1 \setminus R_2$:

1	2	3
a	b	c
c	b	a

a	d	c

b	b	c

Bevezethetnénk a metszetképzés műveletét is, azonban ez szükségtelen, mert a különbségképzés segítségével a metszet kifejezhető: $A \cap B = A \setminus (A \setminus B)$

5.1.3 Descartes-szorzat

A reláció definíciójánál leírtaknak megfelelően az $R_1 \times R_2$ eredménye olyan (n_1+n_2)-esekből áll, amelyeknek első n_1 eleme az első operandusból, második n_2 eleme a második operandusból származik, ebben a rögzített sorrendben. Az operandusok szerkezetére ebben az esetben semmilyen megkötést nem kell tennünk.
Pl.:

R_1:

A	B
a	b
b	a

R_2:

A	D
c	d
a	c

$R_1 \times R_2$:

R_1.A	B	R_2.A	D
a	b	c	d
a	b	a	c
b	a	c	d
b	a	a	c

5.1.4 Vetítés (projekció)

A vetítés egyoperandusos művelete azt jelenti, hogy a reláció egyes attribútumait megtartjuk, a többit pedig törölve egy új relációt hozunk létre. Ennek során ki kell jelölnünk, hogy mely attribútumokat kívánjuk felhasználni, és az új relációban mi legyen a sorrendjük. A halmazalgebrában szokásos jelölésmód az eredeti attribútumokat sorszámukkal azonosítja. Ez megengedett a relációalgebrában is, de lehet az attribútumokat nevükkel is azonosítani. Pl.: $R(A_1, A_2, ..., A_k)$ esetén a $\Pi_{1,3,7,2}$ (R) jelölés azt írja elő, hogy vegyük az R reláció első, harmadik, hetedik és második attribútumát és ebben a sorrendben vegyük fel az új relációba az attribútumok értékeit. Az eredményreláció sémája tehát $S(A_1, A_3, A_7, A_2)$.
Hasonlóképpen, ha adott a
GÉPKOCSI(ÁR, RENDSZÁM, ÉVJÁRAT, ELSŐ_TULAJDONOS, VIZSGA_ÉRVÉNYESSÉGE, TÍPUS, FOGYASZTÁS)
reláció, akkor értelmezhető a
$\Pi_{\text{TÍPUS,ÉVJÁRAT,FOGYASZTÁS}}$(GÉPKOCSI) vetítés.
Amennyiben az eredményhalmazban ismétlődések fordulnának elő, azokat meg kell szüntetni.

5.1.5 Kiválasztás (szelekció)

A kiválasztás egyoperandusos művelete egy részhalmaz képzése az R reláción, amelynek vezérlésére egy logikai formula szolgál. Az R reláció valamennyi elemére kiértékeljük a formulát, és azokat az elemeket vesszük be az új relációba, amelyekre a formula igaz értéket vesz fel.
Jelölése: σ_F (R), ahol az F logikai formula a *szelekciós feltétel.*
A logikai formula kvantormentes, és a következő elemeket tartalmazhatja:
- konstansokat vagy R attribútumainak azonosítóit,

- aritmetikai összehasonlító operátorokat ($< = > \geq \leq$) és
- logikai operátorokat ($\wedge \vee \neg$).

Megjegyezzük, hogy az egyértelműség érdekében a numerikus konstansokat is aposztrófok közé írjuk, hogy meg lehessen különbözteni az attribútumok sorszámától. Ennek megfelelően $\sigma_{2>5}$ (R) az R reláció azon elemeinek halmazát jelenti, amelyekre igaz, hogy a második attribútum értéke nagyobb az ötödik attribútum értékénél, $\sigma_{KOR<'23' \wedge 1='Kovács'}$(NÉVSOR) pedig a NÉVSOR reláció azon elemeit, amelyeknek KOR azonosítójú attribútuma kisebb huszonháromnál, első attribútuma pedig Kovács.

Az 5.1.1.-5.1.5. szakaszokban definiált műveletek a relációs algebra *alapműveletei*. Segítségükkel változatos manipulációkat végezhetünk a relációinkon, néha akár többféle módon is. Mégis célszerű további, ú.n. *leszármaztatott műveleteket* is definiálni, mint pl. a különböző illesztések vagy a hányados műveletét, amelyekkel gyakori, bonyolult műveleteket lehet nagyon tömör formában leírni.

5.1.6 Természetes illesztés (natural join)

Ez a művelet különösen nagy jelentőséggel bír majd a relációs adatbázisok logikai tervezésénél (ld. 8. szakasz).

Adott két reláció, amelyeknek tipikusan van legalább egy, de akár több név szerint megegyező attribútuma. Vegyük sorra a két reláció valamennyi elemét, és válasszuk ki azokat, amelyeknek a megegyező nevű attribútumai érték szerint is megegyeznek. Egyesítsük ezeket olyan Descartes-szorzattá, amelyben a mindkét relációban szereplő, azonos értékű attribútumokat csak egyszer vesszük figyelembe.

Mivel leszármaztatott műveletről van szó, ki tudjuk fejezni a fentieket az alapműveleteink segítségével is:

Legyen $R(A_1, A_2,..., A_k)$ és $S(B_1, B_2,..., B_l)$ a két adott reláció sémája. Legyenek $R.A_{i1} = S.B_{j1}$, $R.A_{i2} = S.B_{j2}$,..., $R.A_{in} = S.B_{jn}$ az azonos nevű attribútumok, ahol R.A jelöli az R reláció A nevű attribútumát.

Ekkor

$$R \bowtie S = \Pi_{i_1, i_2,..., i_m} \sigma_{(R.A_{i_1}=S.B_{j_1}) \wedge ... \wedge (R.A_{i_n}=S.B_{j_n})}(R \times S),$$

ahol $i_1, i_2,..., i_m$ valamennyi attribútumot jelöli, kivéve $R.A_{i_1}, R.A_{i_2},..., R.A_{i_n}$-et.

Vegyük észre, hogy közös nevű attribútumok hiányában a természetes illesztés a Descartes-szorzatba megy át.

Kövessük végig mindezt egy egyszerű példán:

R

A	B
a	b
a	c
b	b

S

A	C
a	c
b	c

R×S

A	B	A'	C
a	b	a	c
a	b	b	c
a	c	a	c
a	c	b	c
b	b	a	c
b	b	b	c

$\sigma_{R.A=S.A}(R \times S)$

A	B	A'	C
a	b	a	c
a	c	a	c
b	b	b	c

$\Pi_{ABC}\sigma_{R.A=S.A}(R \times S) = R \bowtie S$

A	B	C
a	b	c
a	c	c
b	b	c

Pl. 2.:
Adott az OSZTÁLY nevű reláció, amely azt tartalmazza, hogy egy adott nevű személy melyik osztályon dolgozik, továbbá a SZEMÉLY reláció, amely megmondja, hogy egy adott nevű személy hol lakik és mikor született.
A példa két relációs sémája:
OSZTÁLY(NÉV, OSZT_NÉV)
SZEMÉLY(NÉV, LAKCÍM, SZÜL_DÁTUM)
Mindazok lakcímét, akik a Pénzügyi Osztályon dolgoznak nem fejezhetjük ki egyedül egyik reláció segítségével sem, csak úgy, ha a két relációt (a közös NÉV attribútumon keresztül, pl. a természetes illesztés műveletével) összekapcsoljuk. Az így kapott OSZTÁLY$\bowtie$SZEMÉLY reláció tartalmazza valamennyi olyan személy nevét, osztályának nevét, lakcímét és születési dátumát, akik neve mindkét relációban szerepelt. Így tehát már egyetlen relációban megtalálható valamennyi szükséges adat. Ebből kell kiválasztanunk azokat a sorokat, amelyekben az OSZT_NÉV pl. a Pénzügyi Osztálynak felel meg (PO), majd vetítenünk az eredményt a kérdéses attribútumokra: NÉV, LAKÓHELY.
Formálisan: $\Pi_{\text{NÉV, LAKÓHELY}}\sigma_{\text{OSZT\_NÉV="PO"}}(\text{OSZTÁLY} \bowtie \text{SZEMÉLY})$
Természetesen ugyanerre az eredményre más úton is eljuthatunk.
Felmerülhet olyan igény is, hogy az eredményreláció tartalmazza az osztály dolgozóinak nevét még akkor is, ha az illető neve nem szerepel a másik relációban, tehát lakcíme, születési dátuma nem ismert. Ilyenkor az ú.n. *külső illesztés* (outer join) műveletét alkalmazhatjuk. Ilyenkor a hiányzó adatok helyén NULL értékek fognak szerepelni (NULL-lal jelöljük, ha valamely attribútum értéke nem ismert, nem meghatározott).

5.1.7 Θ-illesztés (Θ-join)

Legyen R és S két reláció, Θ pedig aritmetikai összehasonlító operátor. R és S Θ-illesztésén az i,j pontban azt a relációt értjük, amely az R és S relációk Descartes-szorzatának az a részhalmaza, amelyre igaz, hogy az R-beli n-es i-edik attribútuma a Θ relációban áll az S-beli m-es j-edik attribútumával.
Jelölése: $R \bowtie_{i\Theta j} S$

Pl.: R

A	B	C
a	b	c
a	a	d
a	d	e
b	c	e
c	d	a

S

D	E	F
b	c	d
b	c	e
a	e	f
b	e	a
d	a	f

$R \bowtie S$

A	B	C	D	E	F
a	b	c	b	c	d
a	b	c	b	c	e
a	b	c	b	e	a
a	a	d	a	e	f
a	d	e	d	a	f
c	d	a	d	a	f

$R \bowtie_{2=1} S$

A	B	C	D	E	F
a	a	d	b	c	d
a	a	d	b	c	e
a	a	d	b	e	a
a	b	c	d	a	f
a	a	d	d	a	f
b	c	e	d	a	f

$R \bowtie_{2<1} S$

5.1.8 Hányados

Jelölje R÷S azt a relációt, amelyre igaz az, hogy az S-sel alkotott Descartes-szorzata a lehető legbővebb részhalmaza R-nek: $R \div S \times S \subseteq R$.

Pl.: R:

A	B	C	D
a	b	a	c
a	b	c	d
a	b	d	c
e	f	a	c
e	f	c	d
e	f	a	d

S:

A	B
a	b
e	f

R ÷ S :

C	D
a	c
c	d

5.1.9 Példák a relációalgebra alkalmazására

Egy boltban az alábbi adatokat gyűjtik:

záróösszeg (a pénztár tartalma a nap végén)	(ÖSSZEG)
azonosító	(DÁTUM)
az eladott áru neve	(ÁRUNÉV)
darabszáma	(DB)
kódja	(ÁRUKÓD)
egységára	(EGYSÁR)

Minden nap zárás után a pénztárban lévő pénzt a bankba szállítják, kivéve 4000 Ft-ot, amit a pénztárban hagynak másnapra váltópénznek. Így a bankba ÖSSZEG-4000 kerül (BEFIZ).

A 8.2.3.4. szakaszban meg fogjuk konstruálni ehhez a feladathoz az alábbi relációs sémákat, melyeket most megelőlegezünk:

ÁRU(ÁRUKÓD, ÁRUNÉV, EGYSÁR)
MENNYISÉG(DÁTUM, ÁRUKÓD, DB)
BEVÉTEL(DÁTUM,ÖSSZEG)
BEFIZ(ÖSSZEG, BEFIZ)[2]

Ezen sémákra illeszkedő relációk segítségével fejezzük ki az alábbi relációkat:

Az 1997. jan. 1. utáni napok bevételei a dátummal együtt:

$\sigma_{DÁTUM>'19970101'}(BEVÉTEL)$

Ha meg akarjuk cserélni az attribútumok sorrendjét:

[2]Megjegyzendő, hogy nem szerencsés az adatbázisban tárolni a BEFIZ attribútumot, amely származtatott értékeket tartalmaz, de egyéb szempontok miatt most tekintsünk el ettől.

$\Pi_{\text{ÖSSZEG,DÁTUM}}\sigma_{\text{DÁTUM}>'19970101'}(\text{BEVÉTEL})$

Az 1997. jan. 15-i bevétel és a befizetett összeg (első közelítésben):

$\Pi_{\text{ÖSSZEG,BEFIZ}}\sigma_{\text{DÁTUM}='19970115'}(\text{BEVÉTEL}\bowtie\text{BEFIZ})$

Ebben a formában az eredményrelációt költséges lehet előállítani, ha a BEVÉTEL és BEFIZ relációk nagyméretűek. Vegyük észre, hogy ugyanerre az eredményre jutunk, ha előbb a kiválasztás műveletét végezzük el, majd ezután (egyetlen sorral!) a természetes illesztést:

$\Pi_{\text{ÖSSZEG,BEFIZ}}((\sigma_{\text{DÁTUM}='19970115'}(\text{BEVÉTEL}))\bowtie\text{BEFIZ})$

Természetesen, a természetes illesztés helyett az alapműveleteket is használhatjuk:

$\Pi_{\text{ÖSSZEG,BEFIZ}}\sigma_{\text{DÁTUM}='19970115'\wedge\text{BEFIZ.ÖSSZEG}=\text{BEVÉTEL.ÖSSZEG}}(\text{BEVÉTEL}\times\text{BEFIZ})$

Hány darabot adtak el 1997. jan. 15-én az A1 kódú áruból, mi a neve és az ára?

$\Pi_{\text{DB,ÁRUNÉV,EGYSÁR}}\sigma_{\text{ÁRUKÓD}='A1'\wedge\text{DÁTUM}='19970115'}(\text{MENNY}\bowtie\text{ÁRU})$

5.2 Relációs lekérdező nyelvek

A kereskedelemben kapható relációs adatbáziskezelő rendszerek lekérdező nyelve alapvetően

- relációs algebra (pl. ISBL) vagy
- relációs sorkalkulus (pl. QUEL) vagy
- relációs oszlopkalkulus (pl. SQL, QBE)

jellegű. Mint az elnevezések is mutatják, a nyelvek részben *algebra*, részben logikai, *kalkulus* jellegűek. A relációs algebra és annak lehetőségei már ismertek az 5.1. szakaszból. Itt azt célszerű hangsúlyozni, hogy a relációs algebra segítségével megfogalmazott adatbázis lekérdezéseknél explicit módon elő kell írnunk, hogy mely reláción vagy relációkon milyen műveleteket milyen sorrendben kell elvégeznünk ahhoz, hogy a kívánt eredményt megkapjuk. Mint látni fogjuk, a logikai, kalkulus alapú nyelvek csak azt igénylik, hogy az eredményhalmazra vonatkozóan megfogalmazzuk az elvárásunkat. A lekérdezés ezután automatizálható, sőt, többféle lehetőség közül választva optimalizálható is annak érdekében, hogy minél kevesebb művelettel/leggyorsabban juthassunk el az eredményhalmazhoz. A relációs lekérdezéseknek ez a lehetősége volt az egyik legjelentősebb tényezője a relációs adatbáziskezelők sikerének[3].

5.2.1 Relációs sorkalkulus

A relációs sorkalkulus (a továbbiakban: sorkalkulus) egy elsőrendű nyelv, amely tehát kvantorokat is tartalmazhat és a kvantorok sorvektor változókat kvantifikálhatnak. Felépítése: a nyelv *szimbólum*aiból *atom*okat (alapformulákat, prímformulákat) hozhatunk létre, amelyek *formulá*kká építhetők össze, a formulák pedig egy *kifejezés*be építve alkalmasak arra, hogy segítségükkel relációkat írjunk le.
Szimbólumai:

[3] a lekérdezések deklaratív megfogalmazásának lehetőségén kívül

-zárójelek: ( )
-aritmetikai relációjelek: $< > = \leq \geq$
-logikai műveleti jelek: $\neg \wedge \vee$
-sorváltozók: $s^{(n)}$, n változós
-sorváltozók komponensei: $s^{(n)}[i]$, ahol $1 \leq i \leq n$
-(konstans) relációk: $R^{(m)}$, m változós
-konstansok: c

Az atomok felépítése:
$-R^{(m)}(s^{(m)})$
$-s^{(n)}[i]\ \Theta\ u^{(k)}[j]$, ahol $1 \leq i \leq n$, $1 \leq j \leq k$ és Θ aritmetikai relációjel
$-s^{(n)}[i]\ \Theta\ c$
$-R^{(n)}(c_1, c_2,...,c_n)$

A formulák felépítése:
-minden atom formula,
-ha Ψ_1 és Ψ_2 formulák, akkor $\Psi_1 \wedge \Psi_2$, $\Psi_1 \vee \Psi_2$, $\neg\Psi_1$ is formulák,
-ha Ψ formula és $s^{(n)}$ egy szabad sorváltozója, akkor $(\exists s^{(n)})\Psi$ és $(\forall s^{(n)})\Psi$ is formulák, amelyekben $s^{(n)}$ már kötött sorváltozó.

A kifejezések felépítése:
$\{s^{(m)} | \Psi(s^{(m)})\}$, ahol $s^{(m)}$ a Ψ formula egyetlen szabad sorváltozója

Példák:
atomok: $R^{(6)}(s^{(6)})$, $s^{(5)}[1] \leq u^{(4)}[2]$
formulák: $R^{(3)}(v^{(3)}) \wedge p^{(5)}[4] \geq c_1$, $\forall t^{(5)} R^{(5)}(t^{(5)}) \vee q^{(6)}[3] \leq c_2$

A bemutatott formalizmusnak készítsük el egy interpretációját rögzítve, hogy a változók és a konstansok milyen értékeket vehetnek fel.
Legyen pl. A tetszőleges, véges, számítógépben ábrázolható számhalmaz. Tételezzük fel, hogy $c \in A$, $s^{(n)} \in A^n$, $R^{(m)} \subseteq A^m$.
Ekkor minden formulához a sorváltozók egy rögzített értéke esetén egy igazságérték rendelhető.
Az igazságértékek meghatározása:
$-R^{(m)}(s^{(m)})$ pontosan akkor igaz, ha $s^{(m)} \in R^{(m)}$, $(s^{(m)} \in A^m)$,
$-s^{(n)}[i] \Theta u^{(k)}[j]$, továbbá $s^{(n)}[i] \Theta c$ pontosan akkor igaz, ha az értékekre fennáll a Θ aritmetikai reláció $(s^{(n)} \in A^n,\ u^{(k)} \in A^k,\ c \in A)$
$-R^{(n)}(c_1, c_2, ..., c_n)$ pontosan akkor igaz, ha $(c_1, c_2, ..., c_n) \in R^{(n)}$ $(c_i \in A)$,
-ha Ψ_1 szabad sorváltozói az $s_t^{(n_t)} \in A^{n_t}$, $t=1,2,....,r_1$ értékeket, míg Ψ_2 szabad sorváltozói a $v_j^{(k_j)} \in A^{k_j}$, $j=1,2,....,r_2$ értékeket veszik fel, akkor
$\Psi_1 \wedge \Psi_2$ pontosan akkor igaz, ha Ψ_1 és Ψ_2 is igaz,
$\Psi_1 \vee \Psi_2$ pontosan akkor igaz, ha Ψ_1 vagy Ψ_2 igaz,
$\neg\Psi_1$ pontosan akkor igaz, ha Ψ_1 hamis.
-ha Ψ_1 szabad sorváltozói $l^{(k)}$és $s_t^{(n_t)} \in A^{n_t}$, $t=1,2,...,r$, akkor
$\exists l^{(k)} \Psi(l^{(k)}, s_1^{(n_1)}, s_2^{(n_2)},...,s_r^{(n_r)})$ pontosan akkor igaz, ha van olyan $u^{(k)} \in A^k$, amelyre $\Psi(u^{(k)}, s_1^{(n_1)}, s_2^{(n_2)},...,s_r^{(n_r)})$ igaz, továbbá
$\forall l^{(k)} \Psi(l^{(k)}, s_1^{(n_1)}, s_2^{(n_2)},...,s_r^{(n_r)})$ pontosan akkor igaz, ha minden $u^{(k)} \in A^k$ esetén $\Psi(u^{(k)}, s_1^{(n_1)}, s_2^{(n_2)},...,s_r^{(n_r)})$ igaz.

A kifejezések interpretációja: $\{s^{(m)}|\Psi(s^{(m)})\}$ azoknak az $s^{(m)} \in A^m$-eknek a halmaza, amelyekre a Ψ formula igaz, tehát olyan relációkat határoznak meg, melyek attribútumértékei A elemei közül kerülnek ki. Ezt fogjuk a továbbiakban adatbázisok tartalmának lekérdezésére használni.
Az adatbázisok tartalma azonban időben változik, így egy interpretáció csak *egy adott időpillanatban* teszi lehetővé az adatbázis lekérdezését. Az idő múlásával változnia kellene az interpretációnak is. Ami nem változik, azok a formális nyelv elemei. Ezért ezt használhatjuk lekérdezésre különböző időpontokban.
A továbbiakban a formális jeleket és az interpretációjukat nem különböztetjük meg, nem tüntetjük fel továbbá mindenütt a sorváltozókban a változók darabszámát valamint a relációk fokszámát.

Példák:

Az 5.1.9. szakasz relációit fejezzük ki ismét.

Az 1997. jan. 1. utáni napok bevételei a dátummal együtt:

$\{t^{(2)}|\text{BEVÉTEL}(t)\wedge t[1]\geq 19970101\}$

Az 1997. jan. 15-i bevétel és a befizetett összeg:

$\{u^{(2)}|\text{BEFIZ}(u)\wedge(\exists v)\text{BEVÉTEL}(v)\wedge v[1]=19970115\wedge v[2]=u[1]\}$

Hány darabot adtak el 1997. jan. 15-én az A1 kódú áruból, mi a neve és az ára?

$\{s^{(3)}|(\exists u)\text{MENNYISÉG}(u)\wedge(\exists v)\text{ÁRU}(v)\wedge u[1]=19970115\wedge u[2]=\text{'A1'}\wedge v[1]=$
$\text{'A1'}\wedge s[1]=u[3]\wedge s[2]=v[2]\wedge s[3]=v[3]\}$

Természetes módon adódik a kérdés, hogy van-e olyan "jó" a sorkalkulus, mint a relációalgebra, azaz minden relációalgebrai kifejezéshez meg tudjuk-e konstruálni annak sorkalkulus megfelelőjét? A választ pontosabban az alábbi tétel adja meg.

Tétel: Minden, az $R_k^{(n_k)} \subseteq A^{n_k}$, (k=1, 2,...r) relációkból felépített E relációalgebrai kifejezéshez van olyan Ψ sorkalkulus formula, hogy Ψ csak az $R_k^{(n_k)}$-k közül tartalmaz relációkat és az E kifejezés megegyezik $\{s^{(m)}|\Psi(s^{(m)})\}$-vel.

Bizonyítás: az E kifejezésben található műveletek száma szerinti teljes indukcióval.
n=0, azaz nincs művelet E-ben, E csak egyetlen relációt tartalmazhat, pl. $R_k^{(n_k)}$-t, így $E=\{s^{(n_k)}|R_k^{(n_k)}(s^{(n_k)})\}$ formában teljesül az állítás.
T. f. h. E-ben n művelet van és az állítás még igaz. Igaz-e n+1 műveletre is? Vizsgáljuk meg mind az öt relációalgebrai alapműveletre.
Igaz volt tehát, hogy $E_1=\{t_1^{(n)}|\Psi_1(t_1^{(n)})\}$ és $E_2=\{t_2^{(m)}|\Psi_2(t_2^{(m)})\}$.

1. $E=E_1 \cup E_2$: itt n=m. Mivel $E:=\{t^{(n)}|\Psi_1(t^{(n)})\vee\Psi_2(t^{(n)})\}$ ezért létezik az uniónak megfelelő Ψ formula.
2. $E=E_1-E_2$: itt n=m. Mivel $E:=\{t^{(n)}|\Psi_1(t^{(n)})\wedge\neg\Psi_2(t^{(n)})\}$ ezért létezik a különbségnek megfelelő Ψ formula is.
3. $E=E_1\times E_2$: Mivel $E:=\{t^{(n+m)}|\exists t_1^{(n)}\exists t_2^{(m)}\Psi_1(t_1^{(n)})\wedge\Psi_2(t_2^{(m)})\wedge$
$\wedge t^{(n+m)}[1]=t_1^{(n)}[1]\wedge t^{(n+m)}[2]=t_1^{(n)}[2]\wedge\ldots\wedge t^{(n+m)}[n]=t_1^{(n)}[n]\wedge$
$\wedge t^{(n+m)}[n+1]=t_2^{(m)}[1]\wedge t^{(n+m)}[n+2]=t_2^{(m)}[2]\wedge\ldots\wedge t^{(n+m)}[n+m]=t_2^{(m)}[m]\}$
ezért létezik a Descartes-szorzatnak megfelelő Ψ formula is.
4. $E=\Pi_{i_1,i_2,\ldots,i_r}(E_1)$ sorkalkulus megfelelője:
$E:=\{t^{(r)}|\exists u^{(n)}\Psi_1(u^{(n)})\wedge t[1]=u[i_1]\wedge t[2]=u[i_2]\wedge\ldots\wedge t[r]=u[i_r]\}$

5. $E=\sigma_F(E_1)$ sorkalkulus megfelelője: $E:=\{t_1^{(n)}|\Psi_1(t_1^{(n)})\wedge F'\}$, ahol F' úgy kapható F-ből, hogy F i-edik komponensét átírjuk $t^{(n)}[i]$-re. Belátható, hogy F' is sorkalkulus formula lesz és csak olyan relációkat tartalmaz, amelyeket F is tartalmazott.

Összegezve tehát megállapítható, hogy a sorkalkulus kifejező ereje legalább akkora, mint a relációs algebráé.

A tétel megfordítása semmiképpen nem igaz, hiszen a

$$E=\{t^{(n)}|\neg R(t^{(n)})\}$$

relációt nem tudjuk a relációalgebra alapműveleteivel kifejezni. Tehát a sorkalkulus kifejező ereje nagyobb, mint a relációalgebráé.

Ez önmagában még nem lenne baj, azonban a sorkalkulusnál kiértékelési problémák is felmerülhetnek. Az előbbi relációnál pl. ha R-nek csak egyetlen eleme (sora) van, akkor E elemeinek száma $\#A^n-1$, ami a választott A mellett már kis n-ek esetén is ábrázolhatatlanná teszi az eredményhalmazt. A probléma úgy is jelentkezhet, hogy közbenső relációk nőnek kezelhetetlen méretűvé, maga a végeredmény már nem ilyen. Mivel a jelenség nem létezik a relációalgebránál, ezért célravezetőnek tűnik a sorkalkulus szűkítése.

A probléma megoldására vezették be az ú.n. *biztonságos (sorkalkulus) kifejezés*eket (safe expression). A cél az, hogy a sorkalkulus kifejezés kiértékelhető legyen számítógépben kezelhető méretű relációk/véges idő mellett is. Az alapgondolat pedig, hogy le kell szűkíteni azon változóértékek keresési halmazát, amelyek a sorkalkulus kifejezés formuláját igazzá tehetik egy olyan halmazra, amely magából az input relációkból és esetleges egyéb konstansokból áll. Ez a halmaz a *formula doménje*, DOM(Ψ) lesz.

Definíció: DOM(Ψ)≡{Ψ-beli alaprelációk valamennyi attribútumának értékei}∪{Ψ-ben előforduló konstansok}

DOM(Ψ) tehát egy egydimenziós halmaz. Pl. $\Psi(x^{(2)}):=x^{(2)}[1]=28\wedge R^{(2)}(x)$ esetén $DOM(\Psi)=\{28\}\cup\{\Pi_1(R^{(2)})\}\cup\{\Pi_2(R^{(2)})\}$.

Definíció: $\{t|\Psi(t)\}$ biztonságos, ha

a.) minden Ψ(t)-t kielégítő t minden komponense DOM (Ψ)-beli, és

b.) Ψ-nek minden $\exists u\omega(u)$ alakú részformulájára teljesül, hogy ha u kielégíti ω-t az ω-beli szabad változók valamely értéke mellett, akkor u minden komponense DOM (ω)-beli (a *részformula biztonságos*).

A definíció a.) része a formula szabad változójára, b.) része pedig a kötött változóira ír elő kényszert.

(A $\forall u\omega(u)$ alakú részformulák ekvivalensek $\neg\exists u(\neg\omega(u))$-val.)

Példák biztonságos formulákra:

- $\exists u(R(u)\wedge\ldots)$
- $\forall u(\neg R(u)\vee\ldots)$

Példák biztonságos kifejezésekre:

- $\{t|R(t)\wedge\Psi(t)\}$, feltéve, hogy Ψ(t) részformulái biztonságosak. U. is minden t, amely $R(t)\wedge\Psi(t)$-t igazzá teszi, eleme kell, hogy legyen R(t)-nek is, azaz eleme $DOM(R(t)\wedge\Psi(t))$-nek is.
- $\{t|R(t)\wedge S(t)\}$, $\{t|R(t)\wedge\neg S(t)\}$ hasonló okok miatt.
- $\{t|(R_1(t)\vee R_2(t)\vee\ldots\vee R_k(t))\wedge\Psi(t)\}$,

- $\{t^{(m)} | \exists u_1 \exists u_2 \ldots \exists u_k R_1(u_1) \wedge R_2(u_2) \wedge \ldots \wedge R_k(u_k) \wedge t[1]=u_{i_1}[j_1] \wedge t[2]=u_{i_2}[j_2] \wedge \ldots \wedge t[m]=u_{i_m}[j_m] \wedge \Psi(t, u_1, u_2, \ldots, u_k)\}$, mert minden t[n] komponens valamely R_i reláció egy attribútumának értékére van korlátozva.

Tétel: A relációs algebra és a biztonságos sorkalkulus ekvivalensek.

Bizonyítás: Relációs algebra⇒biztonságos sorkalkulus

Teljes indukcióval, az előző tétel alapján belátható. Csupán azt kell még feltételezni, hogy az n-edik lépésben $E_1=\{t_1^{(n)} | \Psi_1(t_1^{(n)})\}$ és $E_2=\{t_2^{(m)} | \Psi_2(t_2^{(m)})\}$ még biztonságos kifejezések. Ezután megvizsgálandó, hogy az öt alapműveletre adhatók-e biztonságos kifejezések.
Az előző tétel bizonyításánál alkalmazott jelölésekkel illusztrációképpen megmutatjuk az unióval ekvivalens sorkalkulus kifejezés biztonságosságát.
$E:=\{t | \Psi_1(t) \vee \Psi_2(t)\}$ biztonságos, ha $\Psi_1(t) \vee \Psi_2(t)$ csak olyan t-re igaz, amelyre $t \in DOM(\Psi_1 \vee \Psi_2)$.
Ez pontosan akkor igaz, ha $\Psi_1(t)$ és $\Psi_2(t)$ bármelyike igaz.
Mivel E_1 biztonságos, ezért ha $\Psi_1(t)$, akkor $t \in DOM(\Psi_1)$, ill.
mivel E_2 is biztonságos, így $t \in DOM(\Psi_2)$ is teljesül, tehát
$t \in DOM(\Psi_1) \cup DOM(\Psi_2)$.
Azonban $DOM(\Psi_1) \cup DOM(\Psi_2)=DOM(\Psi_1 \vee \Psi_2)$, tehát beláttuk, hogy a $\Psi_1(t) \vee \Psi_2(t)$-t igazzá tevő t-kre $t \in DOM(\Psi_1 \vee \Psi_2)$.
Másrészt E-nek esetleges kötött változói csak Ψ_1-ben vagy Ψ_2-ben lehetnek, márpedig az ezekkel felépített E_1 és E_2 kifejezések még biztonságosak, tehát a kötött változók a biztonságosságot nem ronthatják el.
A többi relációalgebrai alapműveletre a biztonságosság hasonlóképpen belátható.
Biztonságos sorkalkulus⇒relációs algebra
Nem bizonyítjuk.

5.2.2 Oszlopkalkulus

A relációs oszlopkalkulus elsősorban abban különbözik a sorkalkulustól, hogy sor-(vektor-)változók helyett egyszerű változók szerepelnek benne. A tapasztalat szerint bizonyos lekérdezések egyszerűbben fogalmazhatók meg ebben az esetben. Az oszlopkalkulus felépítése, interpretációja a sorkalkuluséhoz igen hasonló, ezért csak a különbségeket mutatjuk be.
Szimbólumai:
...
-oszlopváltozók: u_i,
...
Az atomok felépítése:
$-R^{(m)}(x_1, x_2, \ldots, x_m)$, ahol $x_1, x_2, \ldots, x_m$ konstansok vagy oszlopváltozók
$-x \Theta y$, ahol x és y konstansok vagy oszlopváltozók és Θ aritmetikai relációjel
...
A formulák felépítése:
...
Kifejezés felépítése:

$\{x_1, x_2,\ldots, x_m|\Psi(x_1, x_2,\ldots, x_m)\}$, ahol Ψ olyan formula, amelynek szabad változói csak $x_1, x_2,\ldots, x_m$.

Példák (azonosak a sorkalkulus szakasz példáival):

Az 1997. jan. 1. utáni napok bevételei a dátummal együtt:

$\{x, y|$ BEVÉTEL(y,x)$\wedge y \geq 19970101\}$

Az 1997. jan. 15-i bevétel és a befizetett összeg:

$\{x, y|$BEFIZ(x, y)$\wedge(\exists z)$BEVÉTEL(z, x)$\wedge z=19970115\}$

Hány darabot adtak el 1997. jan. 15-én az A1 kódú áruból, mi a neve és az ára?

$\{x, y, z|(\exists u)(\exists v)$MENNYISÉG(v, u, z)$\wedge$ÁRU(u, x, y)$\wedge v=19970115\wedge u='A1'\}$

Tétel: Rögzített A interpretációs halmaz és $R_k^{(n_k)} \subseteq A^{n_k}$ relációk esetén a sorkalkulus bármely kifejezéséhez létezik az oszlopkalkulusnak olyan kifejezése, amely az előzővel azonos relációt határoz meg.

Bizonyítás:

$\{s^{(m)}|\Psi(s^{(m)})\}$ ekvivalens $\{x_1, x_2,\ldots, x_m|\Psi'(x_1, x_2,\ldots, x_m)\}$-vel, hiszen

$s^{(m)}$ helyére $x_1, x_2,\ldots, x_m$-et írunk és Ψ-ből Ψ'-t úgy kapjuk, hogy

$R^{(n)}(t)$ helyére $R^{(n)}(x_1, x_2,\ldots, x_n)$-t,

$t^{(n)}[i]$ helyére pedig x_i-t írunk.

Annak, hogy az oszlopkalkulus kifejezésekben x_i-k pontosan megegyeznek valamely sorkalkulus kifejezés egyik sorváltozójának valamely komponensével, van egy további közvetlen következménye is: ha $\{s^{(m)}|\Psi(s^{(m)})\}$ biztonságos, akkor a neki megfelelő $\{x_1, x_2,\ldots, x_m|\Psi'(x_1, x_2,\ldots, x_m)\}$ is biztonságos lesz. Ezzel beláttuk, hogy a relációs algebra, a biztonságos sorkalkulus és a biztonságos oszlopkalkulus kifejező erejüket tekintve ekvivalensek egymással.

A kereskedelmi rendszerek konkrét lekérdező nyelvei ennél valamivel bővebbek, tipikusan aritmetikával, *aggregáció*val is kiegészítettek (képesek az aritmetikai alapműveleteken kívül átlag, szórás, minimum stb. képzésére is), továbbá támogatják a rekordok megváltoztatását, új rekordok bevitelét is. Amennyiben meg tudják mindazt jeleníteni, amit a megismert három ekvivalens absztrakt nyelv közül bármelyik, akkor *relációsan teljes*nek nevezzük a lekérdező nyelvet.

5.3 Az SQL nyelv[4]

- 1974-75-ben kezdték a kifejlesztését az IBM-nél, az "eredeti" neve SEQUEL (Structured English QUEry Language);
- 1979-től több cég (pl. IBM, ORACLE Corp.) kereskedelmi forgalomban kapható termékeiben;
- 1987-től ANSI szabvány.

5.3.1 Jelentősége

- szabvány, amelyet jelenleg csaknem minden relációs adatbáziskezelő alkalmaz (kisebb-nagyobb módosításokkal);

[4] Átvéve a [7] irodalomból, a szerző jóváhagyásával.

- tömör, felhasználó közeli nyelv, alkalmas hálózatokon adatbáziskezelő szerver és kliensek közötti kommunikációra;
- nem-procedurális programozási nyelv (legalábbis a lekérdezéseknél).

5.3.2 A példákban szereplő táblák

A leírás az ORACLE adatbáziskezelő SQL "dialektusát" ismerteti, ez többé-kevésbé megfelel az egyéb termékekben található nyelv variációknak. A nyelv termék- illetve hardverspecifikus elemeit nem, vagy csak futólag ismertetjük.
Az utasítások ismertetésénél a következő táblákat használjuk.
EMP tábla az alkalmazottak adatainak tárolására:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800.00		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1,600.00	300.00	30
7521	WARD	SALESMAN	7698	22-FEB-81	1,250.00	500.00	30
7566	JONES	MANAGER	7839	02-APR-81	2,975.00		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1,250.00	1,400.00	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2,850.00		30
7782	CLARK	MANAGER	7839	09-JUN-81	2,450.00		10
7788	SCOTT	ANALYST	7566	21-JUL-85	3,000.00		20
7839	KING	PRESIDENT		17-NOV-81	5,000.00		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1,500.00		30
7876	ADAMS	CLERK	7788	24-AUG-85	1,100.00		20
7900	JAMES	CLERK	7698	03-DEC-81	950.00		30
7902	FORD	ANALYST	7566	03-DEC-81	3,000.00		20
7934	MILLER	CLERK	7782	23-JAN-82	1,300.00		10

DEPT tábla a cég részlegeinek adataival:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

5.3.3 A nyelv definíciója

A nyelv utasításait a következő csoportokra oszthatjuk:

- adatleíró (DDS - Data Definition Statement)
- adatmódosító (DMS - Data Manipulation Statements)
- lekérdező (Queries)
- adatelérést vezérlő (DCS - Data Control Statements)

A nyelvben - a szöveg literálok kivételével - a kis- és nagybetűket nem különböztetjük meg. A megadott példáknál a könnyebb érthetőség miatt a nyelv alapszavait csupa nagy betűvel, míg a programozó saját neveit kis betűkkel írjuk.
A parancsok több sorba is átnyúlhatnak, a sorokra tördelésnek nincs szemantikai jelentősége. Az SQL parancsokat pontosvessző zárja le.

5.3.3.1. Lekérdezések

A lekérdezések általános szintaxisa a következő:

```
SELECT  <jellemzők>
FROM    <táblák>
```

```
[WHERE  <logikai kifejezés>]
[<csoportosítás>]
[<rendezés>];
```

A lekérdezés művelete eredményül egy újabb táblát állít elő - persze lehet, hogy az eredmény táblának csak 1 oszlopa és egy sora lesz. Az eredmény tábla a lekérdezés után megjelenik vagy a tábla felhasználható pl. az adatmódosító utasításokban.
A <jellemzők> definiálják az eredmény tábla oszlopait, <táblák> adják meg a lekérdezésben résztvevő táblák nevét, a <logikai kifejezés> segítségével "válogathatunk" az eredmény sorai között. A <csoportosítás> az eredmény tábla sorait rendezi egymás mellé, illetve a <rendezés> a megjelenő sorok sorrendjét határozza meg.
Nézzük meg, hogy a lekérdezés műveletével hogyan lehet megvalósítani a relációs algebra alapműveleteit.

5.3.3.1.1. Vetítés (projekció)

A vetítés művelete egy táblából adott oszlopokat válogat ki. A <jellemzők> között kell felsorolni a kívánt oszlopokat. Például

az alkalmazottak neve és fizetése:

```
SELECT ename, sal FROM emp;
```

minden oszlop kiválasztása:

```
SELECT * FROM emp;
```

A <jellemzők> közé nem csak a FROM mögött megadott tábla oszlopainak nevét lehet megadni, hanem használhatjuk az SQL beépített műveleteit, pl. egyszerű aritmetikai kifejezéseket új érték előállítására.

A dolgozók egész éves fizetése:

```
SELECT ename, 12 * sal FROM emp;
```

Amennyiben a dolgozó által kézhez kapott teljes összeget - fizetést és prémiumot - együtt akarjuk megkapni, hasonlóan járhatunk el. Az jelent csak problémát, hogy a comm érték nincs mindenhol kitöltve, az üres mezőket nem tudjuk hozzáadni a fizetéshez (az üres nem 0)! Ilyenkor használhatjuk az NVL(oszlop, érték) függvényt, amely az üres (NULL) mező helyett a megadott értéket adja vissza.

A dolgozó által felvett pénz:

```
SELECT ename, sal + NVL(comm, 0) FROM emp;
```

A kiválasztott oszlopokat tartalmazó táblákban lehetnek azonos sorok, ami ellentmond a relációs táblák egyik alapvető követelményének. Ennek ellenére a SELECT utasítás nem szűri ki automatikusan az azonos sorokat, mert ez túlságosan időigényes művelet. A programozónak kell tudnia, hogy az előállított táblában lehetnek-e (zavarnak-e) ilyen sorok. Ha kell, a következőképpen szűrhetjük ki ezeket:

Az összes különböző munka megnevezése:

```
SELECT DISTINCT job FROM emp;
```

5.3.3.1.2. Szelekció (restriction)

A szelekció műveleténél a tábla sorai közül válogatunk. A <logikai kifejezés> igaz értékeinek megfelelő sorok kerülnek be az eredmény táblába.

A 2000 dollárnál több fizetésű dolgozók:

```
SELECT ename, sal FROM emp WHERE sal > 2000;
```

Vizsgáljuk meg a logikai kifejezések szerkezetét. A kifejezések elemi összetevői:

- literálok különböző típusú értékekre: számok, szöveg, dátum;

- oszlopok nevei;
- a fenti elemekből elemi adatműveletekkel képzett kifejezések
  számoknál: aritmetikai műveletek,
  aritmetikai függvények;
  szövegeknél: SUBSTR(), INSTR(), UPPER(), LOWER(), SOUNDEX(), ...;
  dátumoknál: +, -, konverziók;
- halmazok: pl.: (10,20,30);
- zárójelek között egy teljes SELECT utasítás (ld. később).

A fenti műveletekkel képzett adatokból logikai értéket a következő műveletekkel állíthatunk elő:

- relációk: <, <=, =, !=, >=, >;
- intervallumba tartozás: BETWEEN .. AND ..;
- NULL érték vizsgálat: IS NULL, IS NOT NULL;
- halmaz eleme: IN <halmaz>;
- szöveg vizsgálat:
  mintával összevetés .. LIKE <minta>, ahol
  % a tetszőleges karaktersorozat,
  _ a tetszőleges karakter jelzése;

Végül a logikai értékeket a zárójelezéssel illetve az AND, OR és NOT műveletekkel lehet tovább kombinálni. Példák:

A 82 .. 89-es években felvett dolgozók:

```
SELECT  ename, hiredate FROM emp
WHERE   hiredate BETWEEN '01-JAN-82' AND '31-DEC-
89';
```

A 2000 dollárnál kevesebbet kereső és prémiumot nem kapó dolgozók:

```
SELECT  ename, sal FROM emp
WHERE   sal < 2000 AND comm IS NULL;
```

5.3.3.1.3. Összekapcsolás (join)

A természetes összekapcsolás műveleténél 2 vagy több tábla soraiból hozunk össze egy-egy új rekordot akkor, ha a két sor egy-egy mezőjének értéke megegyezik. A SELECT kifejezésben a <táblák>-ban kell megadni az érintett táblák neveit, a WHERE mögötti logikai kifejezés definiálja azokat az oszlopokat, amely értékei szerint történik meg az összekapcsolás.

Az egyes osztályok neve, székhelye és a hozzájuk tartozó dolgozók:

```
SELECT  dept.dname, dept.loc, emp.ename
FROM    dept, emp
WHERE   dept.deptno = emp.deptno;
```

Látható, hogy mivel mindkét felhasznált táblában azonos az összekapcsolást megvalósító oszlop neve, a WHERE-t követő logikai kifejezésben az oszlop neve mellé meg kell adni a tábla nevét is. Hasonló helyzet előfordulhat a SELECT-et követő <jellemzők> között is.

Ha megvizsgáljuk a fenti példában kapott eredmény, láthatjuk, hogy a 40-es osztály nem szerepel a listában. Ennek az az oka, hogy az osztálynak nincs egyetlen dolgozója sem, tehát az egyesítésnél nem találtunk az emp táblában egyetlen olyan sort sem, amelyet ehhez az osztályhoz kapcsolhattunk volna. Ez lehet kívánatos eredmény, azonban az SQL-ben lehetőség van arra is, hogy ezeket a sorokat is

egyesítsük, azaz az egyesítésben az emp táblához hozzáképzeljünk egy üres sort is. Ezt külső egyesítésnek hívjuk. A módosított példa a következőképpen néz ki:

```
SELECT  dept.dname, dept.loc, emp.ename
FROM    dept, emp
WHERE   dept.deptno = emp.deptno (+);
```

A (+) jel jelzi azt a táblát, amelyikhez az egyesítés előtt az üres mezőket tartalmazó sort hozzá kell venni.
Az egyesítésnél lehet ugyanarra a táblára többször is hivatkozni. Pl.

Azon dolgozók, akik többet keresnek a főnöküknél:

```
SELECT  e.ename, e.sal, m.ename, m.sal
FROM    emp e, emp m
WHERE   e.mgr = m.empno AND e.sal > m.sal;
```

A fenti példában ugyanazt a táblát kétszer is használjuk, a két tábla oszlopainak megkülönböztetésére a táblákat a FROM részben lokális névvel látjuk el. Lokális neveket természetesen különböző táblák esetén is használhatunk.
Látható, hogy az egyesítés mellett egyidejűleg más logikai kifejezéseket is használhatunk. A fizetések fenti vizsgálatát felfoghatjuk úgy is, mint a természetes egyesítés műveletének általánosított esetét, ahol nem csak az egyenlőség művelete használható (ami megengedett), valamint úgy is, hogy a már egyesített táblából zárjuk ki a fizetésekre szabott követelményeknek meg nem felelő sorokat. Az SQL programozónak - általában - fogalma sincs, hogyan hajtódik végre a fenti kifejezés. Itt fedezhető fel a nyelv nem algoritmikus jellege.

5.3.3.1.4. Oszlopfüggvények (aggregáció)

A lekérdezés eredményeként előálló táblák egyes oszlopaiban lévő értékeken végrehajthatunk szokásos nyelven ciklussal kifejezhető műveleteket, amelyek egyetlen értéket állítanak elő. Ilyenek:

AVG()	átlag
SUM()	összeg
COUNT()	darabszám
MAX()	maximális érték
MIN()	minimális érték

Az üzletkötők átlagfizetése:

```
SELECT AVG(sal) FROM emp WHERE job = 'SALESMAN';
```

Hány dolgozó van:

```
SELECT COUNT(*) FROM emp;
```

Hány különböző beosztás van:

```
SELECT COUNT(DISTINCT job) FROM emp;
```

Átlagos prémium:

```
SELECT AVG(NVL(comm, 0)) FROM emp;
```

Figyelem: az utolsó példa az NVL függvényt használva az összes dolgozóra átlagolja a kifizetett prémiumot, míg NVL nélkül csak azokra átlagolná, akik kaptak egyáltalán prémiumot. (Az oszlopfüggvények kiszámításánál a NULL értékű rekordok kimaradnak.)
Mivel az oszlopfüggvény eredménye egyetlen értéket állít elő, az oszlopfüggvény mellé vagy más oszlopfüggvényeket írhatunk, vagy olyan értéket írhatunk, amelyik az összes kiválasztott sorban azonos. Például írhatnánk:

```
SELECT job, AVG(sal) FROM emp WHERE job =
'SALESMAN';
SELECT COUNT(*), AVG(sal) FROM emp;
```

de hibás

```
SELECT COUNT(*), ename FROM emp;
```

Egymásba ágyazott lekérdezések

A WHERE mögött álló logikai kifejezésben állhat egy teljes SELECT utasítás is. Például

A nem New York-ban dolgozók listája:

```
SELECT ename FROM emp
WHERE deptno IN
    (SELECT deptno FROM dept WHERE loc != 'NEW
YORK')
```

Azaz először kiválasztjuk azon osztályok azonosítóját, amelyek nem New York-ban vannak, majd azt vizsgáljuk, hogy az ezekből képzett halmazban található-e az adott dolgozó osztályának azonosítója. Mellesleg ugyanezt a listát megkaphatnánk az egyesítés műveletével is:

```
SELECT  ename
FROM    dept, emp
WHERE   dept.deptno = emp.deptno AND dept.loc !=
'NEW YORK';
```

A beágyazott lekérdezés vagy egyetlen értéket - azért mert egyetlen megfelelő sor egyetlen oszlopát választottuk ki illetve oszlopfüggvényt használtunk -, vagy több értéket - több sort - állít elő. Az előbbi esetben a SELECT értékét az elemi értékekkel azonos módon használhatjuk. Több érték egy halmazt jelent, tehát a halmazműveleteket használhatjuk. A korábban megismert IN() - eleme - művelet mellett használható az ANY() illetve az ALL() művelet, ahol a kívánt reláció a halmaz legalább egy, illetve valamennyi értékére igaz.

Legmagasabb fizetésű dolgozók (lehet, hogy több van!):

```
SELECT  ename, sal FROM emp
WHERE   sal >= ALL (SELECT sal FROM emp);
```

illetve ugyanez a példa oszlopfüggvény felhasználásával:

```
SELECT  ename, sal FROM emp
WHERE   sal = (SELECT MAX(sal) FROM emp);
```

5.3.3.1.5. Csoportosítás

Az oszlopfüggvények a teljes kiválasztott táblára - minden sorra - lefutnak. Gyakran célszerű lenne a kiválasztott sorokat valamilyen szempont szerint csoportosítani és az oszlopfüggvényeket az egész tábla helyett ezekre a csoportokra alkalmazni.

Foglalkozásonkénti átlagfizetés:

```
SELECT  job, AVG(sal) FROM emp
GROUP BY job;
```

A fenti parancs az emp tábla sorait a job oszlop azonos értékei alapján csoportosítja. Természetesen az oszlopfüggvények korábbi használatához hasonlóan a SELECT <jellemző>-k között csak a csoportosítás alapját képező oszlop neve illetve a csoportokra alkalmazott oszlopfüggvények szerepelhetnek.
A csoportosítás után az eredményből bizonyos csoportok kihagyhatók.

Foglalkozásonkénti átlagfizetés a 1000 és 3000 dollár közötti tartományban:

```
SELECT job, AVG(sal) FROM emp
GROUP BY job
HAVING AVG(sal) BETWEEN 1000 AND 3000;
```

A HAVING mögötti logikai kifejezésben természetesen csak egy-egy csoport közös jellemzőire vonatkozó értékek - a csoportosítás alapját képező oszlop értéke, vagy oszlopfüggvények eredménye - szerepelhet. Természetesen a csoportosítás előtt azért a WHERE feltételek használhatók. Célszerű - gyorsabb - WHERE feltételeket használni mindenhol, ahol csak lehet, a HAVING szerkezetet csak akkor alkalmazni, amikor a teljes csoporttól függő értékeket akarjuk vizsgálni.

5.3.3.1.6. Rendezés

Az eddig tárgyalt lekérdezések eredményében a sorok "véletlenszerű" - a programozó által nem megadható - sorrendben szerepeltek. A sorrendet lehet az ORDER BY által megadott rendezéssel szabályozni. A rendezés több oszlop értékei szerint is történhet, ilyenkor az először megadott oszlop szerint rendezünk, majd az itt álló azonos értékek esetében használjuk a következőnek megadott oszlop(ok) értékét. Minden egyes oszlop esetében külön meg lehet adni a rendezés "irányát", amely alap esetben emelkedő, de a DESC módosítóval csökkenő rendezés írható elő.

A 30-as osztály dolgozói (kicsit rendezve):

```
SELECT * FROM emp WHERE deptno = 30
ORDER BY job, sal DESC;
```

Osztályok szerinti átlagfizetés növekvő sorrendben:

```
SELECT deptno, AVG(sal) FROM emp
GROUP BY deptno
ORDER BY AVG(sal);
```

5.3.3.1.7. Egyéb, nem részletezett műveletek

Az egyes lekérdezések által előállított táblák halmazként is felfoghatók, az SQL nyelv ezen táblák kombinálására tartalmaz szokásos halmazműveleteket is.

UNION	unió
INTERSECT	metszet
MINUS	különbség

A műveleteket két SELECT utasítás közé kell írni.
Relációs táblák segítségével le tudunk írni hierarchikus kapcsolatokat a különböző sorok között. Például az emp táblában az empno és az mgr oszlopok értékei alapján meg lehet konstruálni a vállalati hierarchiát. Erre a CONNECT BY PRIOR szerkezet szolgál.
Például a

```
SELECT  ename, empno, job, deptno, mgr
FROM    emp
CONNECT BY PRIOR empno = mgr
START WITH ename = 'KING'
```

```
ORDER BY deptno;
```

utasítás kiírja a dolgozók 'fáját', az elnöktől kezdődően.
Nem tartoznak szorosan az SQL nyelvhez, de a legtöbb rendszer tartalmaz utasításokat, amelyekkel a lekérdezések által előállított táblázatok megjelenését - pl. az oszlopok neveit, szélességét, adatformátumát, illesztését, tördelését - definiálhatjuk.

5.3.3.2. Táblák módosítása

A következő műveletek az adatbázis táblák módosítására szolgálnak.
új sorokat egy meglévő táblába az

```
INSERT INTO <táblanév> [(<oszlopnév>, ...)]
VALUES (<kif1>, ...)
```

illetve az

```
INSERT INTO <táblanév> [(<oszlopnév>, ...)]
<SELECT ...>;
```

utasításokkal lehetséges. Míg az első szerkezet egyetlen sort, addig a második a lekérdezés által előállított összes sort beilleszti. (Figyelem: a táblákban az egyes rekordok sorrendje tetszőleges, így a beillesztés sem feltétlenül a tábla "végére" történik.)
Amennyiben nem adtuk meg az oszlopok nevét, akkor a - tábla deklarálásánál megadott sorrendben - minden mezőnek értéket kell adni - esetleg NULL-t -, ha viszont megadtuk egyes oszlopok neveit, akkor csak azoknak adunk értéket, mégpedig a felsorolásuk sorrendjében, a többi mező NULL értékű lesz.
Az adatbáziskezelő ellenőrzi, hogy az egyes mezőkbe ne kerülhessen a tábla definíciójával ellentétben NULL érték.

Sorokat törölni a

```
DELETE FROM <táblanév>
[WHERE <logikai kifejezés>];
```

paranccsal lehet. Ha a WHERE hiányzik, a tábla valamennyi sorát, egyébként a logikai kifejezés által kiválasztott sorokat töröljük.

Sorokban mezőket módosítani az

```
UPDATE <táblanév>
SET <oszlopnév> = <kifejezés>, ...
[WHERE <logikai kifejezés>];
```

paranccsal lehet. Ha a WHERE hiányzik, a parancs a tábla valamennyi sorában módosít, egyébként csak a kiválasztott sorokban. Például:

az üzletkötőknek 20% fizetésemelés:

```
UPDATE emp SET sal = 1.2 * sal
WHERE job = 'SALESMAN';
```

A parancsban a kifejezés helyén a szokásos operátorokon és függvényeken felül egyes implementációk akár lekérdezést is megengednek.

5.3.3.3. Táblák, nézetek létrehozása

5.3.3.3.1. Táblák létrehozása

új táblákat a

```
CREATE TABLE <táblanév>
```

```
(<oszlopnév1> <tipus1> [NOT NULL], ...);
```

paranccsal lehet létrehozni. A lehetséges adattípusok implementációnként változhatnak, általában a következő adattípusok megtalálhatók:

CHAR(n)	max n. karakter hosszú szöveg;
LONG(n)	mint CHAR, de hosszára általában nincs (nagyon nagy) felső korlát;
NUMBER(w)	az előjellel együtt max w karakter széles egész szám;
NUMBER(w,d)	w a teljes szám, d a törtrész szélessége;
DATE	dátum (és általában időpont).

Ha valamelyik oszlop definíciója tartalmazza a NOT NULL módosítót, a megfelelő mezőben mindig érték kell, hogy szerepeljen.

A felhasznált táblák definíciója a következő lehet:

```
CREATE TABLE emp
      (empno    NUMBER(4) NOT NULL,
       ename    CHAR(10),
       job CHAR(9),
       mgr NUMBER(4),
       hiredate DATE,
       sal NUMBER(7,2),
       comm     NUMBER(7,2),
       deptno   NUMBER(2) NOT NULL);

CREATE TABLE dept
          (deptno NUMBER(2)     NOT NULL,
           dname  CHAR(14),
           loc    CHAR(13));
```

5.3.3.3.2. Nézet létrehozása

A nézetek olyan virtuális táblák, amelyek a fizikai táblákat felhasználva a tárolt adatok más és más logikai modelljét, csoportosítását tükrözik. Nézetet a

```
CREATE VIEW <nézetnév> [(<oszlopnév1>, ...)]
AS <lekérdezés>;
```

paranccsal hozhatuk létre. A lekérdezésre az egyedüli megkötés, hogy rendezést nem tartalmazhat. Amennyiben nem adunk meg oszlopneveket, a nézet oszlopai a SELECT után felsorolt oszlopok neveivel azonosak. Meg kell viszont adni az oszlopneveket, ha a SELECT számított értéket is előállít. Például:

```
CREATE VIEW dept_sal (deptno, avg_salary) AS
    SELECT deptno, AVG(sal) FROM emp
    GROUP BY deptno;
```

A nézetek a lekérdezésekben a táblákkal megegyező módon használhatók. Jelentőségük, hogy az adatok más modelljét fejezik ki, felhasználhatók a tárolt információ részeinek elrejtésére, pl. különböző felhasználók más nézeteken keresztül szemlélhetik az adatokat. A nézet általában csak olvasható, az adatmódosító műveletekben csak akkor szerepelhet, hagy egyetlen táblából keletkezett és nem tartalmaz számított értékeket.

5.3.3.3.3. Index létrehozása

Az indexek a táblákban történő keresést gyorsíthatják meg. Létrehozásuk:

```
CREATE [UNIQUE] INDEX <indexnév>
ON <táblanév> (<oszlopnév1>, ...);
```

Az indexeket az adatbáziskezelő a táblák minden módosításánál felfrissíti. Amennyiben valamelyik indexet a UNIQUE kulcsszóval definiáltuk, a rendszer biztosítja, hogy az adott oszlopban minden mező egyedi értéket tartalmaz. Lehetőség van több oszlopot egybefogó, kombinált indexek létrehozására.
A lekérdezésekben nem jelenik meg, hogy a táblához tartozik-e index. Az indexek a létrehozásuk után a felhasználó számára láthatatlanok, csak a lekérdezéseket gyorsítják. Indexeket azokra az oszlopokra érdemes definiálni, amelyek gyakran szerepelnek keresésekben. Szerencsés esetben az adattáblában nem kell egyáltalán keresni, a kívánt rekord az index alapján közvetlenül kiválasztható. Pl. a

```
SELECT * FROM emp WHERE ename = 'JONES';
```

az emp táblában keresés nélkül kiválaszthatja JONES rekordját, ha az ename oszlopra definiáltunk indexet.

5.3.3.3.4. Törlések

A fenti adatbázis objektumokat a DROP paranccsal lehet törölni.

```
DROP [TABLE | VIEW | INDEX] <név>;
```

Tábla definíciók módosítása

Már létező táblákat módosítani az

```
ALTER TABLE <táblanév>
[ADD | MODIFY] <oszlopnév> <típus>;
```

paranccsal lehet, ahol ADD egy új, NULL értékű oszlopot illeszt a táblához, míg a MODIFY paranccsal egy létező oszlop szélességét növelhejük.

5.3.3.4. Adatelérések szabályozása

Az adatbáziskezelő rendszereket tipikusan több felhasználó használja, ezzel kapcsolatban újabb problémák merülnek fel.

5.3.3.4.1. Jogosultságok definiálása

A egyes felhasználók részint az adatbáziskezelő rendszerrel, részint az egyes objektumaival különböző műveleteket végezhetnek. Ezeknek a megadására szolgálnak a GRANT utasítások. A

```
GRANT [DBA | CONNECT | RESOURCES]
TO <felhasználó>, ...
IDENTIFIED BY <jelszó>, ...;
```

parancs az egyes felhasználóknak az adatbázishoz való hozzáférési jogát szabályozza. A DBA jogosultság az adatbázis adminisztrátorokat (DataBase Administrator) definiálja, akiknek korlátlan jogai vannak az összes adatbázis objektum felett, nem csak létrehozhatja, módosíthatja illetve törölheti, de befolyásolhatja az objektumok tárolásával, hozzáférésével kapcsolatos paramétereket is. A RESOURCE jogosultsággal rendelkező felhasználók létrehozhatnak, módosíthatnak ill. törölhetnek új objektumokat, míg a CONNECT jogosultság csak az adatbáziskezelőbe belépésre jogosít.

Az egyes objektumokhoz - táblák ill. nézetek - a hozzáférést a

```
GRANT <jogosultság>, ...
ON <tábla vagy nézetnév>
TO <felhasználó>
[WITH GRANT OPTION];
```

parancs határozza meg. A <jogosultág> az objektumon végezhető műveleteket adja meg, lehetséges értékei:

ALL
SELECT
INSERT
UPDATE <oszlopnév>, ...
DELETE
ALTER
INDEX

Az utolsó két művelet nézetekre nem alkalmazható. A felhasználó neve helyett PUBLIC is megadható, amely bármelyik felhasználóra vonatkozik. A WITH GRANT OPTION-nal megkapott jogosultságokat a felhasználók tovább is adhatják.

5.3.3.4.2. Tranzakciók

Az adatbázisok módosítása általában nem történhet meg egyetlen lépésben, hiszen legtöbbször egy módosítás során több táblában tárolt információn is változtatni akarunk, illetve egyszerre több rekordban akarunk módosítani, több rekordot akarunk beilleszteni. Előfordulhat, hogy módosítás közben meggondoljuk magunkat, vagy ami még súlyosabb következményekkel jár, hogy az adatbáziskezelő leáll. Ilyenkor a tárolt adatok inkonzisztens állapotba kerülhetnének, hiszen egyes módosításokat már elvégeztünk, ehhez szorosan hozzátartozó másokat viszont még nem.

A tranzakció az adatbázis módosításainak olyan sorozata, amelyet vagy teljes egészében kell végrehajtani, vagy egyetlen lépését sem. Az adatbáziskezelők biztosítják, hogy mindig vissza lehessen térni az utolsó teljes egészében végrehajtott tranzakció utáni állapothoz.

Egy folyamatban lévő tranzakciót vagy a COMMIT utasítással zárhatjuk le, amely a korábbi COMMIT óta végrehajtott összes módosítást véglegesíti, vagy a ROLLBACK utasítással törölhetjük a hatásukat, visszatérve a megelőző COMMIT kiadásakor érvényes állapotba.

Beállítható, hogy bizonyos műveletek automatikusan COMMIT műveletet hajtsanak végre.

```
SET AUTOCOMMIT [IMM | OFF];
```

Az IMM állapotban az ALTER, CREATE, DROP, GRANT és EXIT utasítások sikeres végrehajtása COMMIT-ot is jelent.

A rendszer hardver hiba utáni újraindulásakor illetve hibás INSERT, UPDATE vagy DELETE parancs hatására automatikusan ROLLBACK-et hajt végre. Érdemes hát

biztonságos helyeken COMMIT parancsot kiadni, nehogy egy hibásan kiadott parancs visszavonja a korábbi módosításokat.

5.3.3.4.3. Párhuzamos hozzáférések szabályozása

Az adatbázist több felhasználó egyidejűleg is használhatja. Ilyenkor az egyes táblákhoz a párhuzamos hozzáférést külön-külön lehet szabályozni.

```
LOCK TABLE <táblanév>, ...
IN [SHARE | SHARED UPDATE| EXCLUSIVE] MODE [NOWAIT];
```

A *LOCK paranccsal egy felhasználó megadhatja, hogy az egyes táblákhoz más felhasználóknak milyen egyidejű hozzáférést engedélyez. Az utasítás végrehajtásánál a rendszer ellenőrzi, hogy a LOCK utasításban igényelt felhasználási mód* kompatibilis-e a táblára érvényben lévő kizárással. Amennyiben megfelelő, az utasítás visszatér és egyéb utasításokat lehet kiadni. Ha az igényelt kizárás nem engedélyezett, az utasítás várakozik amíg az érvényes kizárást megszüntetik, ha csak a parancs nem tartalmazza a NOWAIT módosítót. Ebben az esetben a LOCK utasítás mindig azonnal visszatér, de visszaadhat hibajelzést is.
A táblához hozzáférést az első sikeres LOCK utasítás definiálja.
A SHARE módban megnyitott táblákat mások olvashatják, a SHARE UPDATE módban mások módosíthatják is, ilyenkor automatikusan kölcsönös kizárás teljesül egy-egy sorhoz hozzáférésnél. Az EXCLUSIVE mód kizárólagos hozzáférést biztosít.

5.3.4 Bővítések

5.3.4.1. Konzisztencia feltételek

A táblák definíciójánál eddig csak azt adhattuk meg, hogy milyen adattípusba tartozó értékeket lehet az egyes oszlopokban használni, illetve mely oszlopokban kell feltétlenül értéknek szerepelnie. Célszerű lenne a táblákhoz olyan feltételeket rendelni, amelyek szigorúbb feltételeket definiálnak az egyes adatokra, amelyeket aztán a rendszer a tábla minden módosításánál ellenőriz. Ilyenek például:

- az egyes adatok értékkészletének az általános adattípusnál pontosabb definíciója (pl. adott intervallumba tartozás, adott halmazba tartozás, ahol a halmaz lehet egy másik tábla egyik oszlopának értékei);
- az oszlop elsődleges kulcs, azaz a tábla soraiban minden értéke különböző (hasonló hatás elérhető a UNIQUE index-szel is);
- az oszlop idegen kulcs, azaz értéke meg kell, hogy egyezzen egy másik tábla elsődleges kulcs oszlopának valamelyik létező elemével;

Amennyiben a táblák módosításánál valamelyik feltételt megsértenénk, a rendszer egy kivétel jelet (exception) generál és lefuttatja a hibajelhez tartozó kiszolgáló utasítást, ha van ilyen.

6 A hálós adatmodell

6.1 Története

A hálós adatmodellre épülő adatbázisok mintapéldája a COBOL nyelv szabványosításáról ismert Conference on Data System Languages (CODASYL) Data Base Task Group (DBTG) nevű csoportjához fűződik. Az általuk kidolgozott ajánlásnak (1971-1981) két eleme van. Az adatdefiníciós formalizmus Subschema Data Definition Language (Subschema DDL) néven, az alkalmazói programok írására alkalmas formalizmus Data Manipulation Language (DML) néven vált ismertté. Bár a XXI. századra az elterjedtsége marginálissá vált, még mindig vannak üzemben nagyméretű rendszerek bankokban, államigazgatásban.

6.2 Alaptulajdonságok

Alapvető struktúraegysége a rekord, amely számos atomi komponensből (mezők) tevődhet össze. A következő struktúraegység lehetővé teszi a rekordok összetartozásának megjelenítését láncolás formájában, így születnek meg a CODASYL terminológia szerinti *set*-ek. Egy rekord egyidejűleg több set-hez is tartozhat, így a rekordok változatos - első pillantásra akár kusza - módon kapcsolódhatnak össze. Innen az adatmodell elnevezése.
Egy hálós adatmodell szerint felépült adatbázisban feltehetően nem minden rekord különböző, hanem egyesek azonos séma szerint épülnek fel. Ezek a rekordok egy közös típushoz tartoznak.

Definíció: Egy R *rekord típus* egy olyan A_1, A_2, ..., A_n, n-es (tupel) ahol A_i-k az attribútumnevek és minden A_i-hez egy D_i halmaz, az attribútum domain-je is hozzátartozik, amely halmazból az A_i attribútum értéket vehet fel.

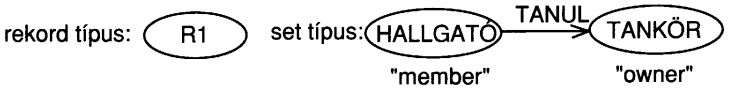
Egy R típusú, n-elemű r rekord a $D_1 \times D_2 \times ... \times D_n$ Descartes-szorzatnak egy eleme, szintén egy n-es: $r(d_1, d_2, ..., d_n)$.
Valamilyen szempontból összetartozó rekordok rendezett összefogása céljából vezették be (a példányok szintjén!) a set fogalmát, amely kétféle rekordból áll:

az egyenrangú *member-rekord*oknak egy (akár üres) halmazából, és
pontosan egy *owner-rekord*ból, aminek a member-rekordok alárendeltek.

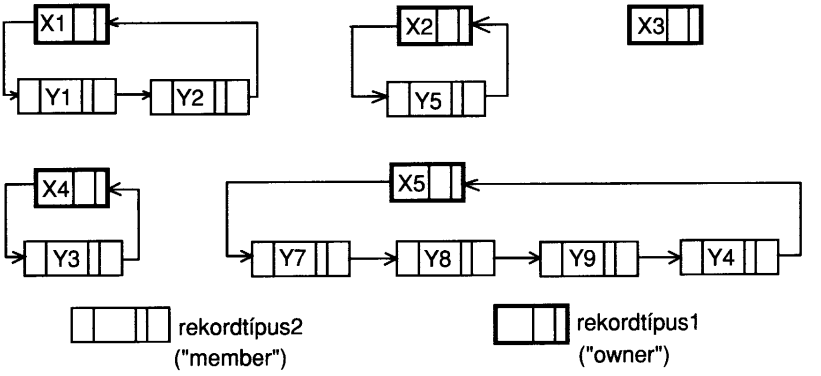
Az összerendelt (tipikusan összeláncolt, ld. később) rekordok ugyanannak a kapcsolatnak a példányait (eseteit) valósítják meg.
Ha a 6.2.b. ábrán rekordtípus1-nek pl. TANKÖR, rekordtípus2-nek HALLGATÓ felel meg, akkor a kapcsolatukat pl. TANUL-nak nevezhetnénk. Mivel TANUL minden példányában egy TANKÖR-rekord számos HALLGATÓ-rekorddal kapcsolódik össze, célszerű ezeket az azonosan strukturált kapcsolatokat a típusok szintjén is leírni.

Definíció: Legyen R_1 és R_2 két rekord-típus és legyenek $\Im(R_1)$ és $\Im(R_2)$ a konkrét eseteik halmazai. Ekkor az S *set-típus*t az $S:=R_1 \times R_2$ művelettel definiálhatjuk, ami egy $\Im(R_2) \rightarrow \Im(R_1)$ függvényszerű kapcsolatot ír le. R_1 az owner-típus, R_2 pedig a member-típus.

A set-típusokat grafikus ábrázolásban hagyományosan a member-típustól az owner-típushoz (a függvényszerűség irányába mutató) irányított nyilakkal jelezzük.

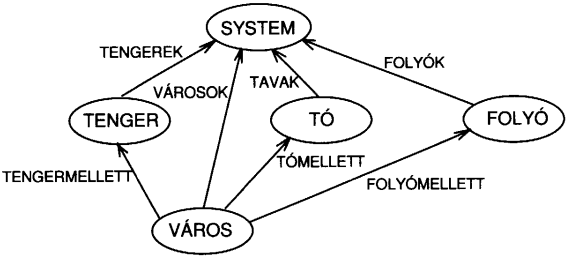


6.2.a. ábra: A hálós adatmodell egy grafikai jelölésrendszerének elemei



6.2.b. ábra: Példa set-ek ugyanazon set-típusból

Egy hálós adatbázis tehát a legmagasabb szinten set-típusok összességéből áll. Gyakran létezik egy "SYSTEM" rekord-típus is, amelynek egyetlen példánya van csupán, azonban owner-évé tehető akár valamennyi, az adatbázisban található rekord-típusnak. Ezzel az azonos típusú rekordok elérését könnyítik meg, amelyek általában csak több set-ből lennének összegyűjthetők (ld. a hálós DML-nél). A 6.2.c. ábra egy egyszerű kartográfiai adatbázis set-típusait mutatja be. A példában feltételeztük, hogy egy város mindig csak egyetlen tenger (tó, folyó) mellett fekszik.



6.2.c. ábra: Egy egyszerű kartográfiai adatbázis hálós modellje

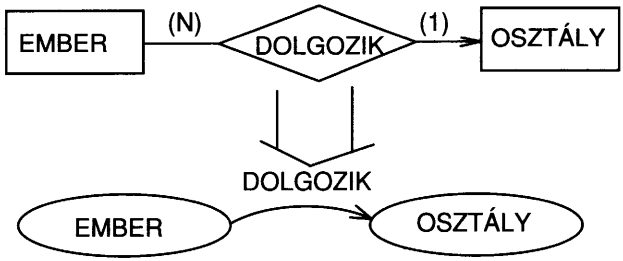
A SYSTEM rekord-típushoz tartozó set-típusokat a grafikus ábrázolásokon általában nem tüntetik fel.

6.3 Implementációs kérdések

Elsősorban a set-ek megvalósítása említésre méltó.
Kézenfekvőnek tűnik, ha az egy owner-hez tartozó membereket egy változó hosszúságú rekordban ábrázoljuk. Ha R_1 a member típus és R_2 az owner típus, akkor az $R_2(R_1)^*$ formájú rekordok egy set-nek felelnek meg. A változó hosszúságú rekordok implementálására a fizikai szervezésnél megismert módszerek alkalmazhatók.
Probléma akkor adódik, ha R_1 member-e más set-típusnak is, pl. R_3 is owner-e. Ekkor u. is R_1 példányait fel kellene R_2 és R_3 után is sorolni, ami közvetlenül nyilván nem tehető meg.
Jobbnak tűnik egy olyan megoldás, amelyben nem kell különböző típusú rekordoknak szomszédoknak lenniük. Ilyen tulajdonságúak a láncolt listák. Legegyszerűbb formáját a 6.2.a. ábra már bemutatta: ilyenkor rekordonként egy-egy mutatót kell a rekordokba járulékosan elhelyeznünk, amelyek a set (logikailag) következő tagjára mutatnak. A mutatók körbe érnek, így tetszőleges, a set-hez tartozó rekordról tetszőleges másik (akár owner, akár member) elérhető. Az owner ill. member rekordok pl. a típusuk alapján különböztethetők meg.
Ha a set-en belüli keresés gyorsasága ezt indokolja, akkor a rekordok ellenkező irányban is összeláncolhatók, sőt, az owner gyors eléréséhez egy-egy további mutató is beépíthető a member rekordokba, amelyek közvetlenül az owner-re mutatnak. Mindezek természetesen a set-ek karbantartását teszik költségesebbé.

6.4 Hálós adatbázis logikai tervezése E-R diagramból

Ha egy adott problémakörnek már elkészült E-R diagramja, átalakíthatjuk azt hálós modellbe. Az entitás halmazoknak a rekord-típusokat, a bináris egy-több (több-egy) kapcsolatoknak (a kapcsolatban résztvevő entitás halmazokkal együtt) pedig egy-egy set-típust feleltetünk meg.



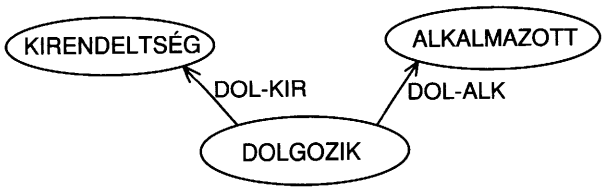
6.4.a. ábra: Egy bináris több-egy kapcsolat megfelelője egy set típus

A hálós adatmodellben azonban a rekordok közötti kapcsolatok csak egy-több (több-egy) típusú bináris kapcsolatok lehetnek - ez a tulajdonság teszi lehetővé, hogy adatainkat egyszerű irányított gráffal jellemezzük -, így nem képezhetők le közvetlenül az E-R modell olyan részletei, amelyek

- nem követik szigorúan az egy-több szemantikát, vagy
- nem bináris kapcsolatot ábrázolnak, vagy
- a kapcsolat attribútummal van ellátva.

A megoldás ilyen esetekben új rekord típus bevezetése, amelynek az az egyetlen szerepe, hogy segítségével ezeket a kapcsolatokat is néhány bináris, több-egy kapcsolatba transzformáljuk. Az újonnan bevezetett rekord típus elnevezése: *virtuális rekord* (máshol láncrekord, kapcsoló rekord, stb.).

A teljesség igénye nélkül, példaképpen nézzük meg, hogyan alakítható át a 4.2.3.b. ábra E-R diagramja hálós modellbe.



6.4.b. ábra: Hálós adatbázis séma a 4.2.3.b. ábra E-R diagramjából

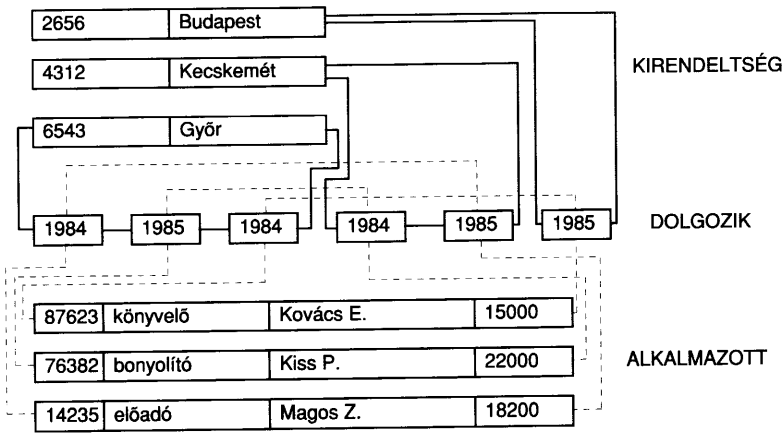
Az egyes rekord típusokban az alábbi mezők fognak megjelenni a set-ek kialakításához szükséges mutatókon kívül:

KIRENDELTSÉG: k_kód, hely

ALKALMAZOTT: a_kód, beosztás, név, fizetés

DOLGOZIK: dátum.

Végül álljon itt ennek az adatbázisnak néhány mintarekordja.



6.4.c. ábra: Néhány rekord és set a 6.4.b. ábra modelljéhez

6.5 Adatkezelés lehetőségei a hálós adatmodellben

Természetesen nincs lehetőség a részletes tárgyalásra (és az értelme is megkérdőjelezhető lenne), emiatt számos részlet maradt ki az alábbi leírásból. A cél elsődlegesen a hálós adatkezelés bemutatása volt.

6.5.1 A hálós sémaleíró nyelv (DDL) elemei

A hálós DDL két nyilvánvaló része egyrészt a
rekordtípusok, másrészt a
set típusok
deklarálását teszi lehetővé.

Egy rekord típus leírásának általános formája:
`RECORD <rek. típ. név> <tárolási inf.> <mezők leírása> <kényszerek>`
A <tárolási inf.> mezőben a rekord típushoz tartozó rekordok fizikai tárolásának módját lehet befolyásolni. Az azonos típus rekordok tipikusan egyetlen állományban tárolódnak, és ezen belül az itt meghatározott mechanizmuson keresztül érhetők el.

1. `LOCATION MODE IS CALC <procedure név> USING <mező lista>`
   esetén tipikusan a rendszerbe beépített procedurák közül választhatunk, amelyek pl. egy hash függvényt határozva meg a rekordok hashing mechanizmuson keresztüli elérést teszik lehetővé.
2. `LOCATION MODE IS DIRECT`
   esetén a rekordok csak az adatbázisbeli kulcsukon (mutató) keresztül érhetők el, sorrendjük tetszőleges lehet.
3. `LOCATION MODE IS VIA <set típus név> SET`
   esetén feltételezhető, hogy a <rek. típ. név> típusú rekordok a <set típus név> típusú set-ek member rekordjai, ezért azok a specifikált set típus owner rekordja mellett kerülnek fizikai tárolásra.

Az 1. tárolási mód esetén a rekordok elérése a <mező lista> értékei alapján igen gyors lesz, nem hatékony viszont az ú.n. *navigáció* (ld. hálós DML) támogatása szempontjából.
A 2. tárolási mód elsősorban a háttértár hatékony kihasználása szempontjából előnyös, a keresés viszont jóval lassúbb lesz, mint az előző esetben.
A 3. tárolási mód esetén viszont támogatott a navigáció egy set-en belül, de lassú a keresés, ha nem ismerjük a kérdéses rekord owner-ét.
A <mezők leírása> mezőben kell felsorolni a rekord típus mezőit és meghatározni a típusukat, hosszukkal együtt. Lehetőség van többértékű mezők (ú.n. vektormezők) deklarálására és ismétlődő mezők (sőt ismétlődő csoportok) deklarálására is.
A redundancia kezelésére virtuális mezőket is létre tudunk hozni. Ha két rekord típusban is szerepelnie kell ugyanazon mezőknek, akkor a két helyen történő tárolás az adatbázis potenciális inkonzisztenciájának forrása lehet. Ezt elkerülendő a mezőt csak egyetlen helyen hozzuk létre, a további helyeken virtuálisnak deklaráljuk.
A <kényszerek> mezőben egyszerű előírások (constraints) fogalmazhatók meg a rekord mezőinek értékeire vonatkozóan.
Pl. CHECK IS HALLGATÓ.ÖSZTÖNDÍJ<100000

Egy set típus leírásának általános alakja:
`SET <set típ. név> <rendezési inf.> OWNER IS <rek. típ. név1> MEMBER IS <rek. típus név2> <insert opciók> <retention opciók>`
A <rendezési inf.> mezőben azt írhatjuk elő, hogy egy set-ben a member rekordok milyen (logikai) sorrendben legyenek. Ez a sorrend úgy valósul meg, hogy az új rekordok a <rendezési inf.>-nak megfelelően kerülnek bele a <set típ. név> típusú set-be.
Pl.

- `ORDER IS FIRST|LAST|NEXT` esetén az újonnan felvett rekord a set-ben az első|utolsó|következő lesz. A következő az ú.n. kurrencia mutatóhoz képest értelmezett (ld. a hálós DML-ről szóló szakaszban).
- `ORDER IS SORTED ASCENDING|DESCENDING BY KEYS` esetén a kulcsmezők által meghatározott sorrendnek megfelelően kerülnek bele a set-be a rekordok.

Az <insert opciók> helyén arról rendelkezhetünk, hogy egy, az adatbázisba újonnan felvett rekord hogyan kerüljön be egy kapcsolatba (set-be) is, és ha bekerül, akkor melyikbe. (Ez a lépés nem nyilvánvaló, a hálós adatbázisban létezhetnek rekordok set-ektől függetlenül is.)

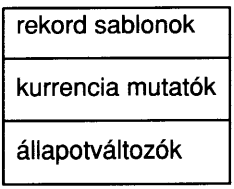
- `MANUAL` esetén "kézzel", esetleg a rekord felvétele után jóval később kell rendelkezni a set-be rendezésről.
- `AUTOMATIC` mód esetén a rekord valamely mező értékétől függően automatikusan bekerül meghatározott set-be.

A <retention opciók> helyén egy rekord "egyedüli" létezését határozhatjuk meg:

- `OPTIONAL` esetén, ha a rekord kikerül egy set-ből, akkor létezhet önállóan is,
- `MANDATORY` esetén ha kikerül egy set-ből, akkor egyúttal egy másikba be kell kerülnie. Pl.: DOLGOZÓ-OSZTÁLY mellett praktikus.
- `FIXED` esetén a rekord owner-e rögzített, így egyáltalán nem kerülhet ki adott set-ből Pl.: ANYA-GYEREK

6.5.2 Hálós DML

A hálós adatlekérdezés alapvetően *rekordorientált*, azaz egyszerre egyetlen rekord kiválasztása (majd kiolvasása) támogatott. (V.ö. a relációs lekérdezések (ld. 5.2. szakasz) ezzel szemben *halmazorientáltak*).
Rekordcsoportok eléréséhez a DML elemeit valamely procedurális *gazdanyelv*be (host language) kell beágyazni (COBOL, PL/I, PASCAL), amely - többek között - alkalmas ciklusok szervezésére is.
A hálós lekérdezések megvalósításának fontos kérdése ezután a csatolás megvalósítása a lekérdező nyelv és a gazdanyelv között. Ennek eszköze az ú.n. User Work Area (UWA), amely egy pontosan definiált adatstruktúra. Felépítése a 6.5.2. ábrán látható.



6.5.2. ábra: A hálós adatbázis alkalmazások környezete: UWA

A "rekord sablonok" adott típusú rekordok egy-egy példányának tárolására alkalmas terület. Egyaránt elérhető a gazdanyelvből és a lekérdező nyelvből is. Az itteni rekordok, mezők neveit célszerű az adatbázisbeli nevekkel azonosra választani, bár ez nem előírás.
Azért, hogy egyik rekordot a másik után elérhessük, a megvalósított adathálóban "mozognunk" kell - szigorúan a struktúra által meghatározott módon -, amit *navigáció*nak neveznek. Így egy adott rekordból kiindulva minden más rekord elérhető, ami az adott rekorddal közvetlenül vagy közvetetten kapcsolatban van, a set-ek által meghatározva. A navigációt ú.n. "kurrencia mutatók" (currency indicator) támogatják. Értelmük a "hol vagyok most, ill. hol voltam legutóbb?" kérdések megválaszolása. A kurrencia mutatók egy-egy adatbázis kulcsot tartalmaznak, melyek segítségével egy-egy rekord egyértelműen azonosítható. A kurrencia mutatók értékét az adatbáziskezelő rendszer automatikusan aktualizálja.
A kurrencia mutatók a következők:

- *current of run-unit* (CRU): a legutóbb elért rekordra mutat
- *current of record type*: a rekord típusban legutóbb elért rekordra mutat
- *current of set type*: az adott set típusban legutóbb elért set azonosítója: owner vagy member rekord adatbázis kulcsa.

Az utóbbi két mutatóból annyi van, ahány rekord, ill. set típus előfordul.
A UWA harmadik területén lévő állapotváltozók információkat hordoznak arról, hogy az adatbázis műveletek sikeresek voltak-e. Értékük minden adatbázis művelet után aktualizálódik.
Pl.: End_of_set értéke igaz lesz, ha a set-ben nincs több rekord.
FAIL hamis értéket vesz fel, ha a parancs sikeres volt.

Áttekintés a parancsokról:

- rekord beolvasás: `GET`
- navigálás: `FIND`
- rekord módosítása: `STORE, ERASE, DELETE, MODIFY`
- set módosítása: `CONNECT, DISCONNECT, RECONNECT, REMOVE`

Egy példa hálós adatlekérdezésre Pascal gazdanyelv mellett:

```
{a UWA változói és a rekord típusok nevei azonosnak
feltételezettek}
ALKALMAZOTT.nev='Kovács E.';
$FIND ANY ALKALMAZOTT USING nev          /kurrencia beállítása
if FAIL=0 then
begin
   $GET ALKALMAZOTT;                     /rekord kiolvasása
   writeln (ALKALMAZOTT.beosztás);       /Kovács E. beosztásának
end                                      /kiírása
else writeln ('Nem talált');
```

A GET parancs

```
GET <rek. típus> <mező lista>
```

A CRU által azonosított rekord tartalmát beolvassa a <rek. típus> által meghatározott sablonba. Ha a két típus nem egyezik meg, akkor hibaüzenetet kapunk. Amennyiben <mező lista>-t is kitöltjük, akkor csak az ott felsorolt mezők értéke aktualizálódik.

A FIND parancsok

Több formája is létezik, céljuk a navigáció támogatása azzal, hogy valamennyi kurrencia mutatók értékét - a találat függvényében - beállítják.

- keresés adatbázis kulcs alapján

```
FIND <rek. típus> RECORD BY DATABASE KEY <változó>
```

ahol <változó> egy a munkaterületen levő, DB kulcsot tartalmazó változó

Pl.:
```
x=CURRENT OF ALKALMAZOTT
FIND ALKALMAZOTT RECORD BY DATABASE KEY X
GET ALKALMAZOTT; BEOSZTÁS
```

kiolvassa az ALKALMAZOTT típus legutóbb elért rekordjának BEOSZTÁS mezőjét.

- keresés CALC kulcs szerint

```
FIND <rek. típus> RECORD BY CALC-KEY
```

Feltéve, hogy ALKALMAZOTT-nak a LOCATION MODE-ja "CALC" (ráadásul a NÉV mező alapján), az alábbiak szerint olvashatjuk ki Kovács Ernő fizetését:

```
ALKALMAZOTT.NEV='Kovács E.'
FIND ALKALMAZOTT RECORD BY CALC-KEY  /beállítja az összes
                                     /kurrencia
                                     /mutatót Kovács
                                     /E. rekordjára
GET ALKALMAZOTT; FIZETÉS             /beolvassa a UWA-ba a
                                     /rekordot
```

Ha több Kovács Ernő is létezik az ALKALMAZOTT rekordok között, akkor azok a

```
FIND DUPLICATE ALKALMAZOTT RECORD BY CALC-KEY
```

formában találhatók meg.

- owner rekord keresés egy set-ben

Egy set végigkereséséhez - egyik módszerrel - először az owner-t kell megkeresni a

```
FIND OWNER OF CURRENT <set típus> SET
```

paranccsal, amely beállítja a kurrencia mutatókat: a CRU a <set típus> kurrens set-ének ownere lesz, úgyszintén a kurrens set típus is erre az owner rekordra fog mutatni.

- a set végigkeresése ezután a

```
FIND FIRST|LAST|NEXT <rek.típus> RECORD IN CURRENT <set típus> SET
```

paranccsal történhet, aminek hatására NEXT esetén körbelépkedhetünk a set-ben a "current of set type" kurrencia mutató által azonosított rekordtól kiindulva (ezt az előbb az owner-ra állítottuk be).

Példa navigációra egyik set-ről egy másikra. Mondjuk meg, hogy Kovács Ernőnek melyik évben hol voltak a cégen belül a munkahelyei (ld. 6.4.b-c.ábrák):

```
ALKALMAZOTT.NEV='Kovács E.'
FIND ALKALMAZOTT RECORD BY CALC-KEY
FIND FIRST DOLGOZIK RECORD IN CURRENT DOL_ALK SET
GET DOLGOZIK
while FAIL=0 do
     FIND OWNER OF CURRENT DOL_KIR SET
     GET KIRENDELTSÉG
     print KIRENDELTSÉG.hely,DOLGOZIK.dátum
     FIND NEXT DOLGOZIK RECORD IN CURRENT DOL_ALK SET
     GET DOLGOZIK
end
```

- a set végigkeresése adott értékű mezők után

`FIND (DUPLICATE) <rek. típus> RECORD IN CURRENT <set típus> SET USING <mező lista>`

hasonló az előzőhöz, de a <mező lista>-ban felsorolt mezők előzetesen beállított értékeivel a talált rekordoknak egyeznie kell.

Végül nézzünk meg néhány rekord-, ill. set-módosítási lehetőséget!
Rekordok beírása az adatbázisba a STORE paranccsal lehetséges. Egy rekord önállóan is megjelenhet az adatbázisban, minden set-től, kapcsolattól függetlenül. Pl.

```
SZALLITO.NEV='Kiss'
SZALLITO.CIM='Vas u. 34, Budapest'
STORE SZALLITO
```

hatására egy új SZALLITO típusú rekordot írunk be. Egyúttal beállítódik a CRU erre a rekordra, a SZALLITO rekord típusnak ez a rekord lesz a kurrense és minden olyan set típusnak is kurrense lesz, amelynek SZALLITO owner-e vagy member-e.
Annak a módja, hogy hogyan kerül be az új rekord valamely set-be, az a set típus deklarációjától függ (insert options: AUTOMATIC vs. MANUAL)
A CRU által azonosított rekord kivétele a <set típus> kapcsolatból a
REMOVE <rek.típus> FROM <set típus>
paranccsal lehetséges, amennyiben a set típus deklarációja nem volt MANDATORY vagy FIXED.
A rekord tényleges törlése a DELETE <rek.típus> paranccsal történik, ha <rek. típus> member típus, ill. owner, de nincs membere.
Vigyázat, DELETE <rek.típus> ALL rekurzívan törli valamennyi membert, így szerencsétlen esetben akár az egész adatbázist.

7 Objektum-orientált adatbáziskezelő rendszerek

A programozási nyelvek, módszertanok területén az objektum-orientált paradigma nem új, jól bevált. Ugyanakkor az objektum-orientált (OO) adatbáziskezelés területén hosszú ideje nem történik meg az az áttörés, ami annak idején a relációs rendszerek megjelenését jellemezte. Bevezetésül vizsgáljuk azt meg, hogy milyen előnyöket nyújthat egyáltalán az OO szemlélet az adatbáziskezelés területén, ill. milyen hátrányokra kell számítanunk.

- Az OO adatbáziskezelőktől a OO tervezési, fejlesztési módszerek összes előnye elvárható: jól megtervezett struktúrák, osztályszerkezetek kialakításával robosztus, könnyen továbbfejleszthető rendszereket, könnyen újrafelhasználható részegységeket kapunk, a fejlesztés produktívabbá válik, a létrejövő szoftver minősége javul. Mindezek a tulajdonságok - mint minden nagy szoftverprojektben - az adatbáziskezelésben is nagyon fontosak.
- Az OO szemlélet szakítva az évtizedeken át uralkodó algoritmusközpontúsággal, az absztrakciós szint finomításának gyakorlatával, az adatokat és a rajtuk végezhető műveleteket állította a középpontba. Az ilyen típusú struktúrák eltárolására a relációs modell nem bizonyult hatékonynak, ezért az OO szemlélet bevezetése új *adatmodell* szükségességét vetette fel.

Az elsőként említett jellemzők minden OO metodológiával készülő projekttel szemben elvárható alaptulajdonságok. Itt az utóbbi, adatbázis specifikus kérdésekre koncentrálunk.

7.1 A relációs adatmodell gyengeségei

Az OO koncepciók megjelenése az adatmodellek - adatok és rajtuk végezhető műveletek - területén elsősorban a relációs adatmodell számára jelent konkurenciát, mivel egyéb rendszert ma már csak elvétve találunk a piacon.

A relációs adatmodelltől alapjaiban különböző, rugalmasabb, ugyanakkor komplexebb adatmodell kialakítására már régen jelentkezett az igény. Ennek oka, hogy vannak olyan problémák, melyek nehézkesen képezhetők le a relációs adatmodellre. Erre néhány egyszerű példát mutatunk be.

Példa 1.: Hogyan írhatjuk le objektum-orientált módon, hogy egy autót garázsban tárolunk?

Objektumok: autó, garázs
Műveletek: tárolás

Ezek után ha bármikor az autóra hivatkozunk, annak összes alkatrészét (is) ért(het)jük alatta, az objektumok összetettsége könnyen leírható. Relációs adatmodell használata esetén az autót érintő információt szét kell bontani relációkba; az autó alkatrészeit esetenként akár külön táblákba is szét kell osztanunk: kerekek, gyertyák, motor, stb. Ezek után bármikor, amikor az autóra, mint egységre hivatkozunk, a táblákba szétosztott adatokat az adatbáziskezelőnek kell összeválogatnia. Ez nemcsak időigényes, de logikailag összeillő adatok szétszabdalására is kényszeríti a rendszer tervezőjét.

Újabb nehézség merül fel, ha menet közben derül ki, hogy az adatstruktúrán változtatni kell. Objektum-orientált esetben ehhez csak az érintett osztály(ok) belső szerkezetét kell átalakítani, melynek - ha a metódusok változatlanok maradnak - nincs hatása a program többi részére. Relációs esetben komolyabb változtatásokra lehet szükség.

Példa 2.: A relációs lekérdező nyelvek nem támogatják a *rekurzív lekérdezéseket.* Vegyünk példának egy rekurzív relációt. Célunk egy - fa analógiájú - egyszerűsített vállalati struktúra leírása, ahol *főnökök* és *beosztottak* vannak. A beosztottaknak legfeljebb egy főnökük van (több-egy kapcsolat). Ez azt jelenti, hogy a vállalatnál mindenki *dolgozó*, és más dolgozókkal való kapcsolata alapján lehet főnök és/vagy beosztott. A relációs modellben ezt egy olyan relációval írhatjuk le, ahol nyilvántartjuk a dolgozó adatait, majd egy olyan speciális külső kulccsal hivatkozunk a főnökére, amelyik ugyanannak a relációnak egy sorára mutat. Tehát főnök és beosztott ugyanabban a relációban foglal helyet.

Ha meg akarjuk tudni, kik egy adott főnök akárhányadik szintű beosztottjai, nehézségeink lesznek, hiszen a megismert és elterjedt relációs lekérdező nyelvek nem támogatják a rekurzív kérdések megválaszolását.

Példa 3.: Grafikák, folyamatábrák, CAD tervek eltárolásakor gyakran merül fel az igény tetszőleges számú töréspontot tartalmazó vonalak eltárolására. Ezt a struktúrát egy relációban pl. a következő módon tárolhatjuk:

```
VonalPontok={VonalID, PosX, PosY, Pozicio}
```

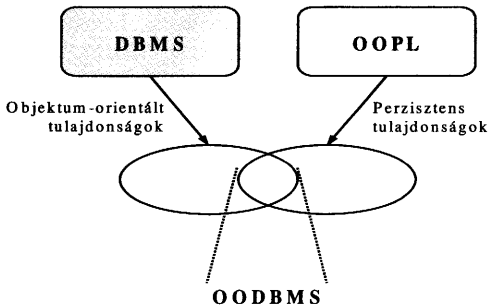
ahol `VonalID` azonosítja, hogy a `PosX` és `PosY` koordinátákkal megadott pontok melyik vonalhoz is tartoznak. A `Pozicio` attribútum pedig azt adja meg, hogy a vonalon az adott pont hányadik helyen található; nem mindegy ugyanis, hogy milyen sorrendben kötjük össze a pontokat. Még ha az eltárolás meg is oldható, kényelmesnek, kézenfekvőnek nem nevezhető. Megállapíthatjuk tehát, hogy a relációs adatmodell nem támogatja a lista jellegű információ tárolását sem.

7.2 Objektum-orientált adatbáziskezelők

Az OO adatbáziskezelő rendszerek történelmileg két irányból fejlődtek. Az egyik az OO programozási nyelvek (Object-oriented Programming Languages - OOPL), a másik az adatbáziskezelők iránya. Ahhoz, hogy adattárolásra képesek legyenek, az OO programozási nyelveket a *perzisztencia* tulajdonságával kell felruházni; azaz az objektumok ne csak a program futása alatt létezzenek, hanem a futás befejeződése után is. A hagyományos adatbáziskezelőket OO tulajdonságokkal (osztályhierarchiák, öröklődés, polimorfizmus, egységbezárás, stb.) kell ellátni[5] (7.2.a. ábra).

Az OO adatbáziskezelők területén nem létezik egységes, szabványosított, de akár még csak hallgatólagosan elfogadott adatmodell sem. Szándékok léteznek ilyen létrehozására, ám ez már csak azért sem egyszerű mert majdnem "ízlés kérdése", hogy egy adatbáziskezelő mely OO tulajdonságokat és milyen formában valósít meg. Azonban vannak olyan alapvető jellemzők, melyek a legtöbb adatbáziskezelőben megtalálhatók. Mi ezekre összpontosítunk.

[5] Ezeket a rendszereket legtöbbször objektum-relációs adatbáziskezelőknek nevezik. A továbbiakban ahol objektum-relációs rendszerekről van szó, azt külön hangsúlyozzuk.



DBMS: Data Base Management System
OOPL: Object-oriented Programming Language
OODBMS: Object-oriented Database Management System

7.2.a. ábra: Az OODBMS-ek származtatása

Az OO adatábrázolás és a relációs adatmodell közötti különbségek ellenére bizonyos analógiákat nevezhetünk meg a könnyebb megértés érdekében. Az OO rendszerek *objektuma* egy *n-esnek* (a reláció egy sorának), az *osztály* pedig a *relációs séma*nak feleltethető meg. Az OO modellben ezután *sémadefiníció* alatt az osztályok és a köztük fennálló kapcsolatok leírását értjük. Az OO adatmodell ezen a ponton sokkal inkább a hálós modellhez hasonlít, hiszen a relációs modellben direkt kapcsolatok nem léteznek. A különböző osztályokhoz tartozó objektumok és a köztük lévő kapcsolatok (*asszociációk*) összessége képezi magát az OO adatbázist.

7.2.1 Típuskonstruktorok

Minden programozási nyelv és adatbáziskezelő rendszer ismer bizonyos alaptípusokat (Integer, Real Character, stb.). Ezeknek létezhetnek bizonyos előre definiált kiterjesztéseik (Date, Time, stb.) is. Azonban a felhasználó által definiált tetszőlegesen bonyolult típusokat (struktúrákat) a relációs modellben nem tudunk leírni. Erre a problémára megoldásként az OO adatbáziskezelők egy része közvetlenül vagy közvetve a *típuskonstruktorokat* nyújtja. Példaként lássunk három alap típuskonstruktort:

Halmazkonstruktor:

T_{set}=**SET** OF (A:T), ahol A attribútum, T pedig az attribútum típusa. A halmaz típus legfontosabb jellemzője, hogy elemei rendezetlenek.

Listakonstruktor:

T_{list}=**LIST** OF (A:T), ahol A attribútum, T pedig az attribútum típusa. A lista típusra jellemző, hogy elemei szekvenciálisan rendezettek. A programozási nyelvek láncolt listájával analóg struktúra kialakítására ad lehetőséget.

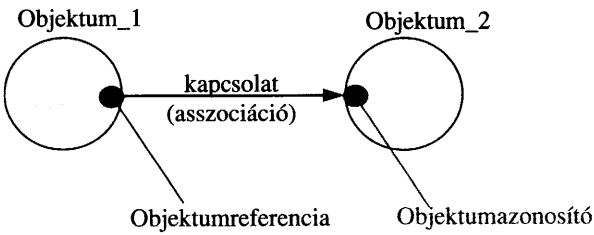
Tuplekonstruktor:

T_{tuple}=**TUPLE** OF (A_1:T_1, ..., A_n:T_n), ahol A_i attribútum, T_i pedig az A_i attribútum típusa. A tupel típus a relációs modellben a reláció egy sorának felel meg.

A típuskonstruktorok tetszőlegesen bonyolult - akár hierarchikus módon is történő - felhasználásával kapott típusok komplexitásukban hasonlítanak a Pascal nyelv *record* illetve a C nyelv *struct* fogalmához.

7.2.2 Kapcsolatok - asszociációk

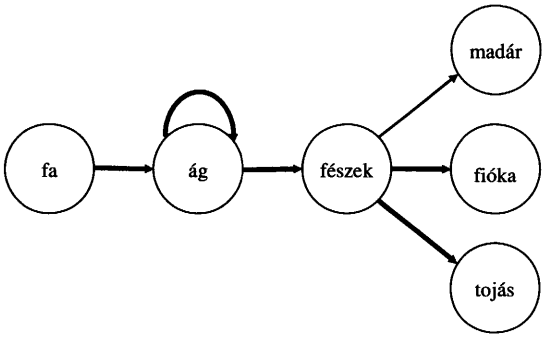
Az OO adatbáziskezelők nagy újítása, hogy tetszőlegesen bonyolult kapcsolatokat is átláthatóan tudnak kezelni. Ezeket a kapcsolatokat a szakirodalom - a más "kapcsolatoktól" való megkülönböztetés érdekében - gyakran *asszociációnak* nevezi (7.2.2.a. ábra). Az asszociációkat osztályokra definiáljuk, így az asszociációk az osztályok példányai, az objektumok között teremtenek kapcsolatot.



7.2.2.a. ábra: Az objektumazonosító és az objektumreferencia

Ha egy A objektum kapcsolatban áll egy B objektummal, ez azt jelenti, hogy A-ból B elérhető. Ha B-ből A is elérhető, az asszociáció *kétirányú*. Asszociációk segítségével könnyedén leírhatjuk a *több-több* típusú (Pl.: tanár-diák) *kapcsolatokat* is. A kapcsolatok mentén történő objektumról objektumra lépkedést itt is *navigációnak* nevezik. A navigációnak lekérdezésekkor van elsősorban jelentősége. Egy objektumból kifelé irányított asszociáció mentén elérhetünk egy újabb objektumot, melynek nyilvános adatmezőihez így hozzáférhetünk. Az asszociáció kiindulási oldalán *objektumreferencia*, azaz a célobjektum *objektumazonosítója* található. Az objektumazonosító szolgál arra, hogy - még ha az objektumok tartalma teljesen azonos is - az objektumokat meg tudjuk különböztetni.
Az asszociációk mentén *terjedési tulajdonságok*at is meghatározhatunk. Asszociációk mentén terjedő tulajdonságok lehetnek például az objektum törlése, zárolása és a zár felszabadítása. Ez azt jelenti, hogy ha egy olyan A objektumot törlünk le, melynek egy másik B objektum felé törléssel terjedő asszociációja volt, akkor ez a bizonyos másik B objektum is letörlődik. Ha B objektumnak is van ilyen asszociációja C objektum felé, akkor ugyanez történik tovább korlátlan mélységben. Mivel az esetek túlnyomó részében olyankor alkalmazzuk ezt a tulajdonságot, amikor egy objektum részekből épül fel (ld. Példa 1. autója), azt az objektumot, amelyből terjedő asszociáció indul ki *tartalmazó objektumnak* (composite object) nevezik.
Példa 4.: Erdész megbízónk arra kért, hogy egy speciális erdőrészletet vegyünk nyilvántartásba a fák főbb ágaival és az azokon fészkelő madarakkal együtt. A fáknak vannak ágaik; az ágak további ágakra bomolhatnak és így tovább. Az ágakon helyenként fészkek találhatók. A fészkekben lehetnek madarak, fiókák, ill. tojások. Mivel a fák nagyon öregek, előfordul, hogy villám csap beléjük, és kidőlnek. Ilyenkor a tojások és a fiókák odavesznek, a szülőmadarak pedig elrepülnek.
Hogyan ábrázoljuk ezt a struktúrát az asszociációk és a terjedési tulajdonságok felhasználásával? (7.2.2.b. ábra)
Az ábrán vastaggal jelöltük azokat a kapcsolatokat, amelyek mentén terjed a törlési tulajdonság. Visszafelé, jobbról balra haladva értelmezzük az ábrát. A tojások és a fiókák megsemmisülnek ha a fészek megsemmisül. A szülők tovább élnek attól még,

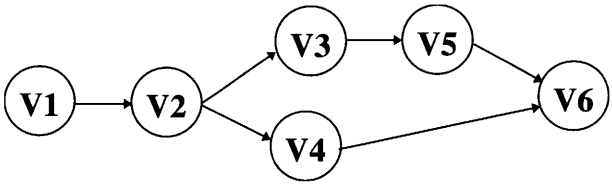
hogy a fészek elpusztul. Tovább lépve balra: a fészek elpusztul, ha az ág, amire építették letörik, elég, stb. Az ág rekurzív módon akkor semmisül meg, ha bármelyik olyan ág elpusztul, amelyik neki "őse". Ezt fejezi ki az önmagába visszatérő rekurzív asszociáció. Ha pedig a fa[6] elpusztul, az összes ága is elpusztul.



7.2.2.b. ábra: Terjedő asszociációk

7.2.3 Verziókezelés

Különösen nagy rendszerek esetén jól jöhet, ha az objektumok állapotát nem csak az adott pillanatban ismerjük, hanem korábbi állapotaikat is előhívhatjuk. Erre szolgálnak a *verziók*, melyek kezelését a legtöbb OO adatbáziskezelő rendszer támogatja (ld. még 9.8.3. szakasz).
A verziók fejlődése lehet lineáris és elágazó. Nem kizárt különböző verziók újbóli "egymásra találása" sem.
Példa 5.: Tekintsük egy autómodell fejlesztését (7.2.3. ábra). Kezdetben gyártottak egy modellt (V1), amit évről évre fejlesztgettek (V2). Amikor jól kezdett menni a cégnek, kínálatszélesítésként több almodellt is elkezdtek gyártani (V3, V4) (nyitható tető, kombi, stb.), melyek a maguk útján fejlődtek (V5). Később költségkímélés miatt beszüntették a széles skála gyártását, ismét csak az alapmodellel foglalkoztak (V6), amiben viszont megtalálható volt az összes eddig külön fejlesztett ág jónéhány tulajdonsága.



7.2.3. ábra: Verziók

[6] Didaktikailag talán helyesebb lenne ebben a kontextusban fa helyett fatörzsről beszélni.

7.2.4 Nyelvek

Hasonlóan az OO adatbáziskezelőkhöz, a lekérdező nyelvek is két irányból fejlődtek. Az egyik út - melyet elsősorban a relációs adatbáziskezelők gyártói járnak - az SQL OO kiterjesztésének megalkotása. A legjelentősebb ezek közül az SQL3 szabvány (ISO/IEC 9075-2:1999). Az SQL3 a hagyományos relációs lekérdezéseken felül ismeri és kezeli az absztrakt adattípust annak minden tulajdonságával (metódusok, öröklés, objektumazonosság, polimorfizmus, stb.). Az SQL3 könnyebben integrálható más programozási nyelvekkel, mint elődei voltak (*ld. objektum-relációs technológia*).
A másik út, ami előttünk áll, a C++, mint legelterjedtebb OO nyelv perzisztens tulajdonságokkal való ellátása. A legtöbb napjainkban működő OO adatbáziskezelő ezt az utat választotta. Nem elhanyagolható azonban az egyre nagyobb teret nyerő SUN által kifejlesztett Java nyelv, melyhez egyre több rendszer kínál elérési felületet.
Míg a relációs adatbáziskezelőknél az esetek többségében külön kellett foglalkoznunk adatdefiníciós, adatmanipulációs és host (gazda) nyelvekkel, OO esetben ezek egységes egészként jelenhetnek meg. Az adatdefiníció például ugyanúgy C++-ban készülhet, mint a lekérdezések. Az adatbáziskezelő nyelvének más nyelvbe történő beágyazására pedig értelemszerűen nincs szükség, hiszen például az SQL-t legtöbbször éppen C-be ágyazzák be. Azzal, hogy nem csak az adatbázis-alkalmazás fejlesztése, hanem maga az adatbáziskezelés is egy OO programozási nyelven történik, az absztrakciós távolság a két terület között lecsökkent, ezáltal a teljes rendszer implementációjának a hatékonysága nőtt.

7.3 Az objektum-relációs technológia

Az OO szemlélet hosszabb idő alatt, fokozatosan hatolt be a relációs adatbáziskezelés világába. Először az OO alkalmazásfejlesztő eszközök jelentek meg, amelyekkel klasszikus relációs adatbázis alkalmazásokat lehet hatékonyan fejleszteni. Az objektum-relációs (OR) alapgondolat szerint a relációs rendszerek összes előnyét megtartva (lekérdezés optimalizálási lehetőség, kiforrott technológia, stb.), OO tulajdonságokkal látják el a relációs adatbáziskezelőt. Ezt a szemléletet követi 1997-ben a relációs adatbáziskezelők gyártóinak egy - piaci részesedés alapján mindenképpen - jelentős hányada.
Lehetőségeiben hasonló a két rendszer: polimorfizmus, komplex kapcsolatkialakítás, automatikus objektum-hierarchia tárolás, stb. Az objektum-hierarchia relációs adatmodellre képzése hathatós rendszertámogatással történik. A lekérdezések nemcsak objektumok szerint hajthatók végre, hanem a deklaratív SQL segítségével a valóságos tárolás helyéül szolgáló táblákból közvetlenül is megtehető.

7.4 Összegzés

Az OO és OR adatbázis koncepciók megítélése napjainkban még nem egyértelmű. Bizonyos tekintetben még visszalépés is történt a relációs adatbáziskezelőkhöz képest.

- Mivel az OO adatbáziskezelők mögött nem áll olyan hatékony matematika, mint ami a relációs rendszereket támogatja, nincs, vagy korlátozott lehetőségek vannak csak a lekérdezések optimalizációjára, a séma - nagyrészt - automatikus egyszerűsítésére (normalizálás). Mivel OR esetben az objektum-hierarchia

leképezése táblákra automatikusan történik, az bonyolult esetekben pazarló, észszerűtlen lehet.

- Míg a relációs modell alapkoncepciója teljesen egységes, nem létezik ilyen egységes OO, ill. OR modell.
- A relációs lekérdezések deklarativitása forradalmi újítás volt megjelenésükkor. Az OO lekérdezések pedig jellemzően mégiscsak navigáción alapulnak. Az OR rendszerek egyebek mellett éppen a relációs lekérdezések lehetőségét nem akarják feladni.

Másrészről az OO adatbáziskoncepciók újítása a hagyományos - elsősorban relációs - adatbáziskezelőkhöz képest az, hogy az emberi gondolatok számítógép számára történő leképzésénél alkalmazott absztrakciós lépcső kisebb, azaz elképzeléseinket sokkal természetesebben vetíthetjük le ilyen rendszerekre. Említésre méltó még, hogy az OO rendszerek - bizonyos esetekben jóval - gyorsabbak lehetnek relációs társaiknál. OR adatbáziskezelők esetén erre a sebességnövekedésre - a háttérben lévő relációs motor miatt nem számíthatunk.

Felmerül tehát a kérdés: Mikor érdemes OO adatbáziskezelőt használnunk? Általában olyan esetekben, amikor az objektumok közötti kapcsolatok és az objektumok struktúrája komplex és nem feltétlenül az adatok változatos szempontok szerinti hatékony lekérdezése a legfontosabb. Tipikusan ilyen alkalmazások lehetnek a telekommunikáció, a CAD, a CASE, a térképészet-geográfia, a multimédia, stb. területének problémái. Ezeken a területeken az OO, ill. OR adatbáziskezelő rendszerek egyre intenzívebb előretörése várható.

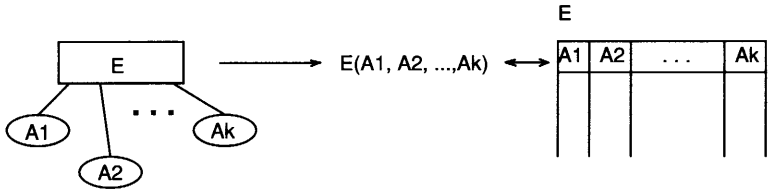
8 Relációs adatbázisok logikai tervezése

Bár a relációs adatmodell nem is az egyedüli, nem is a legjobb minden szempontból, mégis, a legelterjedtebb adatbáziskezelő rendszerek még ma (2006.) is a relációs adatmodellen alapulnak. Emiatt kitüntetett jelentősége van a relációs adatbázisok tervezésének. Jelen jegyzetben ezért szántunk a problémának külön fejezetet.
A tervezés az esetek túlnyomó többségében az adatbázis logikai tervezésére terjed ki, hiszen konkrét feladat megoldásához általában valamely megvásárolható relációs adatbáziskezelő rendszert használják fel. Ilyenkor nincsen sem mód sem szükség az adatbáziskezelő működésének számos részletét megváltoztatni. A tervezés ekkor tehát arra korlátozódik, hogy meg kell határozni az adatbázis logikai szerkezetét (a relációs sémákat, definiálni kell az egyes adattípusokat), majd meg kell írni magát az adatokat manipuláló programot/programokat. Ez utóbbiakat nevezzük *adatbázis alkalmazás*oknak. Az alkalmazások alapulhatnak pl. a szabványosított SQL nyelven, amelyet gyakran ágyaznak be valamely magas szintű procedurális nyelvbe. Az alkalmazásfejlesztésnek természetesen más, fejlettebb módszerei is léteznek.
Bárhogyan hozzuk is létre az alkalmazásainkat, az első lépés mindig az adatbázis logikai (koncepcionális) megtervezése. A tervezésnek két karakterisztikusan különböző módszere ismert, amelyek azonban kombinálhatók is, és adott esetben jól kiegészítik egymást.

8.1 Tervezés E-R diagramból

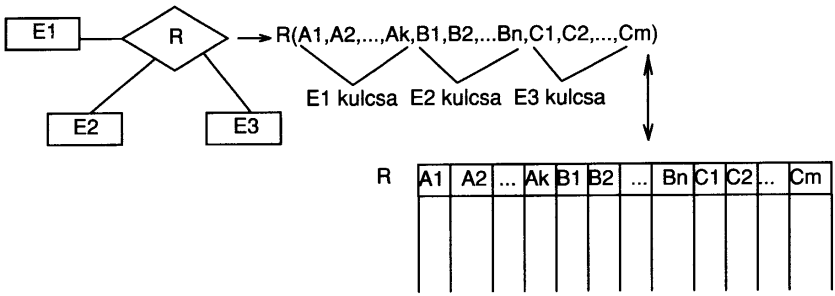
A 4.2. szakaszban megismerkedtünk az E-R diagrammal, amely szemléletes ábrázolásmódja következtében hatékonyan támogatja a valóság modellezésének folyamatát. Az 5. szakaszban megismerkedtünk a relációs adatmodellel. Természetes az az igény, hogy E-R diagramokat relációs sémákká kíséreljünk meg átalakítani, így alapozva meg a valóságot jól modellező relációs sémák kialakítását. Az átalakítás teljes, ha megmondjuk, hogy az E-R diagram elemeit hogyan kell relációs modellben megengedett adatstruktúrába (értsd: relációkba) transzformálni.

- Az entitáshalmazokat olyan relációs sémával ábrázoljuk, amely tartalmazza az entitáshalmaz valamennyi attribútumát. A reláció valamennyi n-ese az entitáshalmaznak pontosan egy példányát fogja azonosítani (ld. 8.1.a. ábra).
  Ha az entitáshalmazok között olyan is van, amelynek egyes attribútumait egy (általánosabb) entitáshalmaz egy "isa" kapcsolaton keresztül meghatározza, akkor a specializált entitáshalmazhoz rendelt relációs sémába az általánosabb entitáshalmaz attribútumait is fel kell venni.



8.1.a. ábra: Entitáshalmaz transzformációja relációs sémába

- A kapcsolatokat olyan relációs sémákká alakítjuk, amelyek attribútumai között szerepel a kapcsolatban résztvevő valamennyi entitáshalmaz kulcsa is (ld. 8.1.b. ábra). Feltételezzük, hogy két entitáshalmaz valamely kulcsattribútuma nem azonos nevű még akkor sem, ha az entitáshalmazok megegyeznek (mint pl. a HÁZASSÁG: EMBER, EMBER kapcsolatban). Névkonfliktus esetén az attribútumokat átnevezéssel kell megkülönböztetni. Az így kapott relációban minden egyes n-es olyan entitáspéldányokat rendel egymáshoz, amelyek a szóbanforgó kapcsolatban vannak egymással.



8.1.b. ábra: Kapcsolat transzformációja relációs sémába

A kapcsolatok relációs sémákba átalakítására a kapcsolat funkcionalitása és egyéb tulajdonságai (pl. specializáció kifejezése esetén) függvényében számos más lehetőség is van, amelyek adott esetben jobbak is lehetnek a bemutatott, egészen általános módszertől.
Vegyük észre, hogy az E-R diagram relációs sémákba alakításával elveszítettük az egyedek és kapcsolatok formális megkülönböztethetőségét.
Példa: Transzformáljuk relációs sémákká a 4.2.3.b. ábra E-R diagramját!
Sémák az entitáshalmazokból:

KIRENDELTSÉG(KKÓD, HELY)
ALKALMAZOTT(AKÓD, NÉV, BEOSZTÁS, FIZETÉS)

Séma az egyetlen kapcsolatból:

DOLGOZIK(KKÓD, AKÓD, DÁTUM)

Azokat az attribútumokat, amelyek egy adott relációban nem, de egy másikban kulcsattribútumok, szokás szerint kétszeres aláhúzással jelöljük. Ennek az elnevezése: *idegen (vagy külső) kulcs* (ld. 8.2.2.1. szakasz).

8.2 Tervezés sémadekompozícióval

Ha az E-R modellezés segítségével jutunk el a relációs sémáinkhoz, akkor a sémákban található attribútumokat és a relációk további tulajdonságait az fogja meghatározni, hogy milyen "ügyesek" voltunk az E-R diagram megalkotása során. A relációk azonban tetszőleges számú attribútumot tartalmazhatnak. Így akár a rendszerben található valamennyi adatot beépíthetjük egyetlen relációba (ú.n. *univerzális relációba*), és ekkor egyetlen tábla írja le az egész rendszert. Ez a felhasználó számára roppant kényelmes, hiszen nem kell tudnia, hogy melyik adatot melyik reláció

tartalmazza, így bizonyos lekérdezésekhez nem kell a relációk összekapcsolásával sem vesződnie, csupán ki kell válogatnia a számára érdekes adatokat.
Másrészről - tárolási hatékonyság szempontjából - ez a megközelítés nem előnyös, mert általában sok "felesleges" adatot is tartalmaz. Ezek kezelése lassítja a rendszer működését, a háttértárat és a memóriát feleslegesen foglalja, az adatbázist következetlenné/ellentmondásossá teheti.

A többször tárolt adatok egy adatbázisban, ill. realációban feleslegesek lehetnek. Nem minden többször tárolt adat felesleges. Ha egy reláció azt tartalmazza, hogy kinek milyen színű a szeme, akkor a "barna" (vagy annak a kódja) igen sokszor is előfordulhat ugyanabban a relációban, mégsem érezzük ezt feleslegesnek. Ugyanakkor feleslegesnek érezzük, ha ugyanannak a személynek a szeme színe többször is előfordul a relációban. Ezt a hétköznapok során is redundanciának nevezzük. Általánosabban:
Definíció: ha egy relációban valamely attribútum értékét a relációban található más attribútum(ok) értékéből ki tudjuk következtetni valamely ismert következtetési szabály segítségével, akkor a **relációt redundáns**nak nevezzük.
A kikövetkeztethető adatokra gyakran azt mondjuk, hogy *származtatott* adatok.
Például:
Az alábbi reláció szállítók adatait tartalmazza, ki milyen árut mennyiért szállít és a szállító hol lakik.

NÉV	CÍM	TÉTEL	ÁR
Tóth István	Bp. Fa u. 5.	tégla	30
Tóth István	Bp. Fa u. 5.	vas	200
Kis János	Baja Ó u. 9.	tégla	40
Kis János	Baja Ó u. 9.	pala	20
Nagy Géza	Ózd Petőfi u. 11	cement	350

Ebben a relációban a címek többszörös tárolása teljesen felesleges, redundanciát okoz. Vegyük észre, hogy az egyes relációk redundanciáját csökkenthetjük, ha az adatbázist több, egyenként kevesebb attribútumot tartalmazó relációból alakítjuk ki.

8.2.1 Anomáliák

A redundáns relációknak megfelelő adattárolással kapcsolatban egy sor kellemetlen jelenség fordulhat elő. Ezeket hagyományosan *anomáliáknak* nevezik.

8.2.1.1. Módosítási (update) anomália

Tételezzük fel, hogy Tóth István címe megváltozik. A változást elvileg annyi helyen kell átvezetni, ahány helyen Tóth István címe szerepel. Ha csak egy helyen is ezt elmulasztjuk (pl. rendszerösszeomlás miatt), később különböző helyekről többféle címet is kiolvashatunk. Az ilyen szituáció tehát az inkonzisztencia lehetőségét hordozza magában.

8.2.1.2. Beszúrási anomália

Ennek lényege, hogy nem tudunk tetszőleges adatokat nyilvántartásba venni, ha nem ismert egy másik adat, amivel a tárolandó adat meghatározott kapcsolatban áll. Más szavakkal: nem tudunk a relációba olyan sort felvenni, amelynek olyan mezője

kitöltetlen (NULL), amely a reláció definíciója miatt nem lehet kitöltetlen. Ez a helyzet egy reláció kulcsmezőivel.
Pl.: egy új szállítót nem tudunk felvenni, ha még nem szállított semmit.

8.2.1.3. Törlési anomália

Mivel csak egész sorok törölhetők, elveszíthetünk olyan adatokat is, amelyekre még szükségünk lehet.
Ha pl. a cement tételt töröljük, elveszítjük Nagy Géza címét is.

A fenti problémákat megoldhatja a relációk függőleges felbontása (vertikális dekompozíciója). Az építési anyag szállítók relációját például két részre célszerű felbontani:

SZÁLLÍTÓ

NÉV	CÍM
Tóth István	Bp. Fa u. 5.
Kis János	Baja Ó u. 9.
Nagy Géza	Ózd Petőfi u. 11.

ANYAGOK

NÉV	TÉTEL	ÁR
Tóth István	tégla	30
Tóth István	vas	200
Kis János	tégla	40
Kis János	pala	20
Nagy Géza	cement	350

A felbontás módja most nyilvánvalónak tűnik, de a valóságban ez ritkán van így. Ezen kívül számos probléma is felmerülhet. Nem világos pl., hogyan lehet biztosítani, hogy az eredeti reláció mindig helyreállítható legyen. Továbbá: hogyan lehet jó felbontásokat készíteni? Milyen értelemben jobb az egyik felbontás a másiknál?
Ahhoz, hogy ezekre a kérdésekre válaszolni tudjunk a függőségek, kulcsok és normál formák mélyebb ismerete szükséges.

8.2.2 Funkcionális függőségek

Láttuk, hogy a relációs adatbázisok hatékony működtetésének egyik központi kérdése a relációkon belüli redundancia csökkentése. Hogyan is keletkezett a redundancia? Legegyszerűbb formájára a SZÁLLÍTÓ relációban láttunk példát. A redundancia ott két okból keletkezett:

1. a szállító nevét több sorban is fel kell használnunk az általa szállított *különböző* tételek azonosításához.
2. ha feltételezzük, hogy a valóság úgy 'működik', hogy nincs két azonos nevű, különböző lakhelyű szállító, akkor minden olyan sorban, ahol megjelent egy szállító neve, *törvényszerűen* megjelent *ugyanaz* a lakcím is, ami nyilván felesleges.

Ez utóbbi tényt úgy is kifejezhetjük, hogy azt mondjuk: a NÉV attribútum értéke meghatározza a CÍM attribútum értékét, tehát minden olyan két sorban, amelyekben a NÉV értéke megegyezik, megegyezik a CÍM értéke is. A jelenségnek megfelelő matematikai konstrukciót *funkcionális (függvényszerű) függőség*nek (vagy függésnek) nevezik, melynek pontos definíciója a következő:

Definíció: Legyen adott az $R(A_1, A_2, ..., A_n)$ relációs séma, ahol A_i-k, i=1, 2,...,n (alap)attribútumok. Legyen X és Y a reláció attribútumainak két részhalmaza: $X \subseteq R$ és $Y \subseteq R$. Ha a reláció bármely két t, $t' \in r(R)$ sorára bármely időpillanatban fennáll az, hogy ha t[X]=t'[X] akkor t[Y]=t'[Y] (ahol t[Z] jelenti: $\Pi_Z(t)$-t, azaz a t n-es Z attribútumhalmazra eső vetületét), akkor azt mondjuk, hogy az Y attribútumok *funkcionálisan függenek* az X attribútumoktól. Azt is mondhatjuk, hogy az X értékei meghatározzák az Y értékeit. Mindezt így jelöljük: $X \rightarrow Y$.

Megjegyzések:

1. A definíció nem követeli meg, hogy bármikor is valóban legyen két olyan $t, t' \in r(R)$ sor, melyre fennáll, hogy t[X]=t'[X]. Ekkor nyilván nem lehet ellenőrizni, hogy t[Y]=t'[Y] fennáll-e. Ha soha sincsen ilyen két t, t' sor, az $X \rightarrow Y$ függőség akkor is fennállhat! A definíció a logikából ismert implikáció műveletével tökéletes összhangban van.
2. Érdemes felfigyelni a definíciónak arra a feltételére, hogy 'bármely időpillanatban'. Gyakori az, hogy egy R sémára illeszkedő *adott* r relációban (tehát pl. egy kiválasztott időpillanatban) minden olyan t, t' sorra, melyre t[X]= t'[X] fennáll t[Y]=t'[Y] is. Ilyenkor *eseti* funkcionális függőségről beszélünk. Megkülönböztetésül ezért néha az eredeti (fenti) definícióval kapcsolatban *érdemi* funkcionális függőséget emlegetünk. Amikor a valóságos viszonyokat kívánjuk modellezni (pl. egy információs rendszerben), az természetesen az érdemi függőségek segítségével történhet. Az eseti függőségek összessége csak bővebb lehet, mint az érdemi függőségeké. *A továbbiakban, ha nem hangsúlyozzuk külön, akkor érdemi függőségekről lesz szó.*
3. Az előzőekből az is következik, hogy a funkcionális függőségek meghatározása modellezési kérdés. Ez azt is jelenti, hogy egy reláció egyetlen pillanatnyi állapotából sohasem dönthető el egy függőség fennállása, legfeljebb arra következtethetünk, hogy mely függőségek nem állnak fenn!
4. Láthatóan a funkcionális függések egy relációban redundanciát okozhatnak, amennyiben valamely $X \rightarrow Y$ funkcionális függés mellett a relációnak van legalább két olyan eleme, amelyek X-ben azonosak.

Példa.: Az S(NÉV, CÍM, VÁROS, IRÁNYÍTÓSZÁM, TELEFON) relációs sémában a valóságot "elég jól" modellező funkcionális függőségek pl. az alábbiak:

CÍM,VÁROS→IRÁNYÍTÓSZÁM	(ha ismerjük a címet és a város nevét, akkor ehhez egyértelműen tartozik egy irányítószám)
IRÁNYÍTÓSZÁM→VÁROS	(egy irányítószámhoz csak egy város tartozik)
NÉV→CÍM	
NÉV→VÁROS	
NÉV→IRÁNYÍTÓSZÁM	
NÉV→TELEFON	(ha nincsenek azonos nevek, akkor egy névhez egyértelműen tartozik egy lakcím, városnév, irányítószám és telefonszám)

Adott R sémán értelmezett (megadott, ismert) funkcionális függőségeket gyakran egyetlen halmazba gyűjtjük össze: ezt a halmazt jelöljük pl. F_R-rel.

Megadott funkcionális függőségekből kiindulva a továbbiakban számos új fogalmat definiálunk, melyekre a későbbiekben hivatkozni fogunk.
Definíció: Ha X,Y⊂R és X→Y de Y nem függ funkcionálisan X egyetlen valódi részhalmazától sem, akkor X-et Y *determinánsának* nevezzük. Azt is mondjuk, hogy Y *teljesen függ* (funkcionálisan) X-től. Ha van X'⊂X, hogy X'→Y, akkor Y *részlegesen függ* X-től.

8.2.2.1. Relációk kulcsai

Az E-R modellezésnél már használtuk a kulcs fogalmát. A fizikai szervezésnél is előkerült a (keresési) kulcs fogalma. Ott azt mondtuk, kulcs minden, ami szerint keresni akarunk.
Most megadjuk egy reláción értelmezett kulcs matematikai definícióját is. Az előző szakaszban bevezetett jelöléseket alkalmazzuk.

Definíció: X-et pontosan akkor nevezzük *kulcsnak* az R reláción, ha
1. X→R, és
2. X-nek nincs olyan X' valódi részhalmaza, hogy X'→R.

Szuperkulcs, kulcs

X-et szuperkulcsnak nevezzük, ha igaz, hogy X→R. Más szavakkal akkor, ha tartalmaz kulcsot.
Néha hangsúlyozandó egy kulcs minimális voltát *minimális kulcs*ról beszélünk.
Ha egy kulcs csak egy attribútumból áll, akkor *egyszerű kulcs*, egyébként *összetett kulcs* a neve.
Pl.:
Az építőanyagok (8.2. szakasz) relációjának a {NÉV,TÉTEL} attribútumhalmaz (összetett) kulcsa, ha az ottani funkcionális függőségeket értelmezzük a relációs sémán.
A jelen szakaszban definiált S relációnak {NÉV} (egyszerű) kulcsa. Egy szuperkulcsa lehet pl. a {NÉV,TELEFON} attribútumhalmaz.
A kulcsok néhány fontos tulajdonsága:
- a kulcs a relációnak egy és csakis egy sorát határozza meg,
- egy kulcs attribútumai nem lehetnek NULL-értékűek (azaz értékük meghatározatlan).

Tétel: Minden relációnak van kulcsa.

Válasszuk ugyanis az attribútumok teljes halmazát. Ez a kulcsokra vonatkozó első feltételnek eleget tesz, hiszen nincs olyan attribútum, amit ne vettünk volna figyelembe. Tehát meghatározza a reláció valamennyi attribútumának értékét. Ha a második feltétel is teljesül, akkor kulcs, ha pedig nem, akkor szuperkulcs, tehát tartalmaz kulcsot.

Elsődleges kulcs

Ha X és Z az R relációnak egyaránt kulcsai, miközben X≠Z, akkor az R relációnak több kulcsa is van.
Ezek közül kiválasztunk egyet, amelyet *elsődleges* kulcsnak (primary key) nevezünk. A többi kulcsot *kulcsjelöltnek* (candidate key) hívjuk.

Idegen (külső) kulcs.

Adott egy R és egy R' relációs séma. Tételezzük fel, hogy R'≠R. Ha ∃D⊆(RïR'), hogy D→R' és minimális - azaz R' kulcsa -, akkor D neve az R sémával kapcsolatban *idegen kulcs*. Más szavakkal: egy sémában lehetnek olyan attribútumok, amelyek egy másik sémára illeszkedő relációban a sorokat egyértelműen azonosítják, tehát ott kulcsok. Ezeket idegen kulcsoknak nevezzük.

8.2.2.2. Funkcionális függőségek további tulajdonságai

A korábbiak alapján már sejthetjük, hogy a megadottakon kívül más funkcionális függőségek is igazak lehetnek. Egy jó példa erre a *triviális függőség*, amit ugyan ritkán hangsúlyozunk, amikor egy valóságos helyzetet írunk le funkcionális függőségekkel, bár attól még fennáll. Ha egy R séma tartalmazza az A és B attribútumokat, akkor mindig fennállnak ezen a sémán az AUB→B vagy az AUB→A függőségek is. U. is minden R sémára illeszkedő r relációban, ha ∀t, t'∈r(R) sorára ha t[AUB]=t'[AUB] akkor t[B]=t'[B] valamint t[A]=t'[A] egyaránt fennáll.

Jelölés: a továbbiakban az AUB helyett egyszerűen AB-t írunk.

Ha azonban több függőség is ismert, vagy a sémának számos attribútuma van, akkor már nem nyilvánvaló annak a megválaszolása, hogy mi az adott F_R függőségek mellett még fennálló függőségek teljes rendszere. Mivel ennek a kérdésnek pl. a relációs sématervezésnél nagy jelentősége lesz, ezért a cél az, hogy az összes függőség előállítására a gyakorlatban is jól használható 'módszert' adjunk. A 'módszer' következtetési szabályok, ú.n. axiómák alkalmazása lesz. Ezektől az axiómáktól a következő két alapvető tulajdonságot várjuk el:

- csak olyan függőségeket lehessen velük előállítani, amelyek 'igazak', ugyanakkor
- 'meg lehessen kapni' valamennyi 'igaz' (funkcionális) függőséget, amelyek egy adott függőséghalmazban található függőségekkel nincsenek ellentmondásban.

Az 'igaz' és a 'meg lehet kapni' kifejezések alatt pontosan az alábbiakat értjük:

Definíció: (igaz) egy adott R sémán az attribútumain értelmezett F_R függéshalmaz mellett egy X→Y függőség pontosan akkor igaz, ha minden olyan r(R) reláción fennáll, amelyeken F_R valamennyi függősége is fennáll. Jelölése: $F_R \models X \rightarrow Y$.

Definíció: ('meg lehet kapni', azaz *levezethető*) egy W→Z funkcionális függőség pontosan akkor vezethető le adott F_R függőségekből, ha az axiómák ismételt alkalmazásával F_R-ből kiindulva megkaphatjuk W→Z-t. Jelölése: $F_R \vdash W \rightarrow Z$.

Az imént definiált tulajdonságokkal bíró következtetési szabályok *Armstrong axiómái* néven váltak ismertté.

Armstrong axiómái a funkcionális függőségekről

Adottak az R sémán az X, Y, Z attribútumhalmazok.

a./ Ha X⊆Y, akkor Y→X (reflexivitás vagy triviális függőség).
b./ Ha X→Y és Y→Z, akkor X→Z (tranzitivitás).
c./ Ha X→Y, akkor XZ→YZ (bővíthetőség).

Tétel: Az Armstrong axiómák igazak, alkalmazásukkal csak igaz függőségek állíthatók elő (*igazság tétel*).

Formálisan: $F_R \vdash X \rightarrow Y \Rightarrow F_R \models X \rightarrow Y$

Bizonyítás: Lássuk be egyenként, hogy az axiómák igazak, akkor nyilván igaz lesz minden olyan függőség is, amelyet az axiómák (véges számban) ismételt alkalmazásával kapunk (azaz le tudunk vezetni). Lássuk be pl., hogy c./ igaz! Ehhez tételezzük fel, hogy c./ nem igaz, bár X→Y fennáll. Ez azt jelenti, hogy ha ∃ t, t' n-esek valamely r(R) relációban, hogy t[XZ]=t'[XZ], akkor t[YZ]≠t'[YZ]. A Z-hez tartozó attribútumok nyilván megegyeznek a t és t' sorokban, hiszen különben t[XZ] és t'[XZ] nem lehetne azonos. Tehát az Y-beli attribútumok értékében különbözik t[YZ] és t'[YZ], azaz t[Y]≠t'[Y]. De ez nem lehetséges, mert a kiindulási X→Y függőség miatt ha t[X]=t'[X], akkor t[Y]=t'[Y]. Tehát ellentmondásra jutottunk c./ tagadásával, így c./ igaz kell, hogy legyen.

Az axiómák tömören:
a./ $X \subseteq Y \models Y \rightarrow X$
b./ $X \rightarrow Y$ és $Y \rightarrow Z \models X \rightarrow Z$
c./ $X \rightarrow Y \models XZ \rightarrow YZ$

8.2.2.3. Az axiómák következményei

d./ $X \rightarrow Y$ és $X \rightarrow Z \models X \rightarrow YZ$ (egyesítési szabály).
e./ $X \rightarrow Y$ és $WY \rightarrow Z \models XW \rightarrow Z$ (pszeudotranzitivitás).
f./ $X \rightarrow Y$ és $Z \subseteq Y \models X \rightarrow Z$ (dekompozíciós/felbontási szabály).

Fentiek mostmár az Armstrong axiómák felhasználásával is bizonyíthatóak, példaképpen nézzük meg d./ bizonyítását!
$X \rightarrow Y \models X \rightarrow XY$ (c./ miatt)
$X \rightarrow Z \models XY \rightarrow ZY$ (c./ miatt)
$X \rightarrow XY$ és $XY \rightarrow ZY \models X \rightarrow ZY$ (b./ miatt), ami éppen a bizonyítandó állítás.

Példa: Vegyük elő újra az R(NÉV, TÉTEL, CÍM, ÁR) sémát a rajta értelmezett (NÉV, TÉTEL)→ÁR és NÉV→CÍM függőségekkel! Keressük meg R kulcsát!
NÉV,TÉTEL→ÁR ⊨ NÉV,TÉTEL→ÁR,NÉV, TÉTEL
NÉV→CÍM⊨ NÉV,TÉTEL→CÍM,TÉTEL.
Az egyesítési szabályt alkalmazva adódik, hogy NÉV,TÉTEL→ ÁR,NÉV,TÉTEL,CÍM. Mivel NÉV,TÉTEL együttesen meghatározza R valamennyi attribútumát, ezért NÉV,TÉTEL (szuper)kulcs. Láthatóan bármelyik attribútumot is hagyjuk el, már nem fogja R valamennyi attribútumát meghatározni, tehát a { NÉV,TÉTEL} attribútumhalmaz minimális is, azaz kulcs.

8.2.2.4. Attribútumhalmaz lezárása

Gyakran felmerülő kérdés, hogy bizonyos attribútumok értékeinek ismeretében esetleg milyen más attribútumok értékeit tekinthetjük még ismertnek azt kihasználva, hogy a (funkcionális) függőségek rendszerén keresztül egyes attribútumok meghatározzák más attribútumok értékeit. Lényegében ezt fejezi ki az alábbi definíció:

Definíció: Az X *attribútumhalmaz lezárása* adott F függéshalmaz mellett az a legbővebb $A \subseteq R$ halmaz, amelyre az X→A függőség adott függéshalmaz mellett fennáll. Jelölése: $X^+(F)$.

Formálisan: $X^+(F) = \{A | A \in R$ és $F \models X \rightarrow A\}$

Tehát $\forall$Y-ra, amelyre $X \rightarrow Y$ igaz, hogy $Y \subseteq X^+$, és megfordítva: $\forall$Y-ra, amelyre $Y \subseteq X^+$ igaz, hogy $X \rightarrow Y$.

Algoritmus: Egy attribútumhalmaz lezárása csaknem lineáris időben meghatározható az alábbi rekurzív algoritmus segítségével:

$X^{(0)} = X$

:

$X^{(i+1)} = X^{(i)} \cup \{A \mid \exists V \subseteq X^{(i)}, V \rightarrow U \in F \text{ és } A \in U\}$

Példa: Adott az R(A, B, C, D) séma és a funkcionális függőségek F halmaza: $F=\{A \rightarrow B, B \rightarrow D\}$. Mi az X=AC attribútumhalmaz lezárása, X^+?

$X^{(0)} = AC$

$X^{(1)} = AC \cup \{B\}$ ($A \rightarrow B$ miatt)

$X^{(2)} = ACB \cup \{D\}$ ($B \rightarrow D$ miatt)

Mivel $X^{(2)} = ABCD$, vagyis az R séma attribútumainak teljes halmaza, így az eljárás véget ért, $X^+=ABCD$, amiből az is következik, hogy AC szuperkulcsa a relációnak. (Mivel AC minimális is, ezért valójában kulcs.)

Tétel: Az Armstrong axiómák teljesek, azaz belőlük minden függőség levezethető. (nem bizonyítjuk)

Mostantól kezdve a $\models$ és a $\vdash$ jeleket egyenértékűeknek, felcserélhetőknek tekinthetjük.

8.2.2.5. Függéshalmaz lezárása

Láttuk, hogy (funkcionális) függőségeket definiálva az Armstrong axiómák segítségével továbbiakat is meg tudunk határozni, sőt valamennyit, amelyek logikailag a megadottakból következnek. Gyakran hasznos, ha ezekre a függőségekre közösen tudunk hivatkozni.

Definíció: Az F *függéshalmaz lezárása* mindazon függőségek halmaza, amelyek az F függéshalmaz elemeiből az Armstrong axiómák alapján következnek.

Formálisan: $F^+ = \{X \rightarrow Y \mid F \models X \rightarrow Y\}$.

Még ha F viszonylag kicsi is, F^+ igen nagy lehet. Ha pl. $F=\{A \rightarrow B, B \rightarrow C\}$, akkor F^+ elemei: $F^+=\{A \rightarrow B, B \rightarrow C, A \rightarrow AB, AB \rightarrow B, B \rightarrow BC, BC \rightarrow C, A \rightarrow C, AB \rightarrow C, A \rightarrow 0, A \rightarrow A, B \rightarrow 0, B \rightarrow B, ...\text{stb.}\}$ (0 az üres halmaz jele)

Tanulság: az adott függőségek szerkezetétől függően a lezárt halmaznak akár 2^n számú eleme is lehet, így a lezárt meghatározása esetenként igen költséges művelet. Ez az oka annak, hogy a függéshalmaz lezártjára elméleti megfontolásokban gyakran fogunk hivatkozni, gyakorlatban használható algoritmusaink ellenben lehetőleg nem támaszkodnak a meghatározására.

Gyakori az a feladat, hogy el kell dönteni egy függőségről - legyen ez $A \rightarrow B$ -, hogy következik-e adott F függőséghalmazból. A feladat direkt megoldása lehetne, hogy kiszámítjuk F^+-et és megvizsgáljuk, hogy elemei között van-e $A \rightarrow B$. Függéshalmaz lezártjának költséges számítása helyett gyorsabban eredményre jutunk, ha (lineáris időben!) A^+-t határozzuk meg a megismert algoritmussal. Amennyiben $B \in A^+$, akkor $A \rightarrow B \in F^+$. Ez az attribútumhalmaz lezártja definíciójának közvetlen következménye.

Ha függőségeknek egy bonyolult rendszere adott, gyakran szeretnénk könnyebben áttekintető, egyszerűbb formába alakítani - nyilván úgy, hogy közben az új alak ugyanazt az 'információt' hordozza, mint az eredeti. Ez alatt azt értjük, hogy a módosított függéshalmaz segítségével pontosan u. azokat a függőségeket lehessen előállítani. A függéshalmaz lezárása segítségével lehetőségünk van arra, hogy egyszerű definíciót adjunk két függéshalmaz egyenlőségére.

Definíció: Két függéshalmaz pontosan akkor *ekvivalens*, ha lezártjaik megegyeznek.

Ezt így jelöljük: $F=G \Leftrightarrow F^+=G^+$ és azt is mondjuk, hogy F lefedi G-t, ill. G lefedi F-et.
Láttuk, hogy az ekvivalencia eldöntése a definíció alapján igen költséges lehet, így a gyakorlatban használhatóbb az alábbi algoritmus:
Vizsgáljuk meg, hogy $F \subseteq G^+$ és $G \subseteq F^+$ egyaránt teljesül-e. Ha igen, akkor ekvivalensek.
Először az $F \subseteq G^+$ teljesülését vizsgáljuk. Minden $X \rightarrow Y \in F$ függőségre a tartalmazás hatékonyan eldönthető $X^+(G)$ kiszámításával: ha $Y \subseteq X^+(G)$, akkor $X \rightarrow Y \in G^+$.
$G \subseteq F^+$ eldöntése nyilván azonos elven történhet.

Definíció: F *minimális függéshalmaz* (beszélünk F *minimális fedés*éről is) akkor, ha
1. a függőségek jobb oldalán csak egyetlen attribútum van,
2. nincs olyan függőség, amely elhagyható,
3. a függőségek bal oldaláról nem hagyható el attribútum.

Tétel: Adott függéshalmazzal ekvivalens minimális függéshalmaz mindig előállítható.

Bizonyítás: konstruktív.
Adott egy F függéshalmaz.
1. Minden $X \rightarrow Y \in F$ függőség helyettesíthető a $X \rightarrow A_1$, $X \rightarrow A_2,...,X \rightarrow A_n$ függőségekkel (ahol $Y=\{A_1, A_2,...,A_n\}$), így egy F' függéshalmazt kapunk. Nyilvánvaló, hogy F és F' ekvivalensek.
2. Vizsgáljuk meg $\forall Z \rightarrow B \in F'$ függésre, hogy elhagyható-e[7]. Definíció szerint ehhez az kell, hogy $(F' \setminus \{Z \rightarrow B\})^+=(F')^+$ fennálljon, aminek az eldöntése költséges. Célszerűbb ezért azt vizsgálni, hogy $B \in Z^+(F' \setminus \{Z \rightarrow B\})$ vajon teljesül-e - amint már korábban beláttuk, ez ekvivalens, ugyanakkor hatékonyabb. Eredményül egy F'' függéshalmazt kapunk, amely továbbra is ekvivalens F-fel.
3. Vizsgáljuk meg $\forall S \rightarrow C \in F''$ függőségre, ahol $S = \{D_1, D_2,...,D_n\}$, hogy elhagyható-e valamely D_i attribútum. Definíció szerint ehhez az kell, hogy $(F''')^+=(F'')^+$ fennálljon (ahol
   $F''' = F'' \setminus \{S \rightarrow C\} \cup \{(D_1, D_2,...,D_{i-1}, D_{i+1},...,D_n) \rightarrow C\}$), ami ekvivalens az
   $F'' \subseteq (F''')^+$ és $F''' \subseteq (F'')^+$ egyidejű fennállásával. Előbbi ekvivalens
   $C \in S^+(F' \setminus \{S \rightarrow C\} \cup \{D_1,D_2,...,D_{i-1},D_{i+1},...,D_n \rightarrow C\})$ vizsgálatával, utóbbi pedig
   $C \in (D_1,D_2,...,D_{i-1},D_{i+1},...,D_n)^+(F'')$ vizsgálatával. Mivel
   $C \in S^+(F' \setminus \{S \rightarrow C\} \cup \{D_1,D_2,...,D_{i-1},D_{i+1},...,D_n \rightarrow C\})$ triviálisan teljesül, elegendő csupán a másik irányt megvizsgálni. Végül egy olyan F* függéshalmazt kapunk, amely továbbra is ekvivalens F-fel, de a minimális függéshalmaz valamennyi tulajdonságának megfelel.

[7] Egyes szerzők a függéshalmaz *redundanciamentes fedés*éről beszélnek, ha eddig eljutottunk a minimalizálás során.

Megjegyzés: vegyük észre, hogy a 2. és 3. lépésekben az egyes függőségek, ill. attribútumok elhagyásának sorrendje tetszőleges. Ennek következtében a végeredményül kapott minimális függéshalmazok különbözőek is lehetnek. Következmény: egy adott függéshalmazzal ekvivalens minimális függéshalmaz nem feltétlenül egyértelmű!

Példa:
Adott: F={AB→CD, AC→BD, C→AB}, mi egy minimális fedése?

F'={AB→C, AB→D, AC→B, AC→D, C→A, C→B}
Ebből C→B miatt AC→B elhagyható, így
F''={AB→C, AB→D, AC→D, C→A, C→B}
Az AC→D függőségben C→A miatt az A attribútum felesleges, így elhagyható.
Továbbá, az {AB→C, AB→D, C→D, C→A, C→B} halmazban AB→C, C→D függésekből következik az AB→D, így az elhagyható. Végül is
F'''={AB→C, C→D, C→A, C→B} F-nek egy minimális fedése.

8.2.3 Relációk normál formái

Ahhoz, hogy az 8.2.1. szakaszban ismertetett anomáliákat elkerülhessük, a relációink sémái meghatározott feltételeket kell, hogy teljesítsenek. Ezeket a feltételeket *normál formák*nak nevezik (Codd, 1970). A normál formák tehát megszorítások a relációs séma tulajdonságaira vonatkozóan annak érdekében, hogy a sémákra illeszkedő relációkkal végzett műveletek során egyes nemkívánatos jelenségeket elkerülhessünk.

8.2.3.1. A nulladik normál forma (0NF)

Ilyen alakúnak tekintünk minden olyan relációt/relációs sémát, amelyben legalább egy attribútum nem atomi abban az értelemben, hogy az attribútum értéke nem tekinthető egyetlen egységnek, azaz egyes részeihez külön is hozzá akarunk férni.
0NF-ben van az a reláció, amelyik pl. ismétlődő csoportot tartalmaz az attribútumai között.
Példa 0NF alakra:

EGYETEM_NÉV
REKTOR
KAR
DÉKÁN
TANSZÉK
VEZETŐ

UNI(EGYETEM_NÉV,REKTOR(KAR,DÉKÁN(TANSZÉK,VEZETŐ)*)*)
A * jellel az ismétlődő csoportokat jelöltük.

8.2.3.2. Az első normál forma (1NF)

Definíció: Egy reláció 1NF alakú (vagy más szóval *normalizált*), ha csak atomi attribútum-értékek szerepelnek benne.

Megj.: Az 1NF alak definíciója nyilvánvalóan nem a redundancia csökkentést szolgálja. Egyszerűen csak kiindulási alapot teremt a további normalizálás számára.

8.2.3.3. A második normál forma

Definíció: Egy R reláció $A \in R$ attribútuma *elsődleges*, ha A eleme a reláció valamely K kulcsának. Egyébként A *másodlagos attribútum.*

Ezek szerint egy reláció kulcsai az attribútumokat két diszjunkt halmazba sorolják:
Ha R kulcsainak halmaza=$\{K_1, K_2, ..., K_n\}$, ahol $K_i \subseteq R$, akkor UK_i $\forall i$-re az elsődleges attribútumok, $R\text{-}UK_i$ pedig a másodlagos attribútumok halmaza.

Definíció: Egy 1NF relációs séma 2NF alakú, ha benne minden másodlagos attribútum a reláció bármely kulcsától teljesen függ.
Más szavakkal: másodlagos attribútum nem függ egyetlen kulcs egyetlen valódi részhalmazától sem.

A 2NF definíciójának célja már nyilvánvalóan a redundanciacsökkentés. Ha ugyanis egy attribútumhalmaznak (értsd: kulcs) már egy része is meghatároz egy másodlagos attribútumot, akkor annak a másodlagos attribútumnak a meghatározásához elegendő csupán a kulcs adott része attribútumértékeit tárolni.

A definíció közvetlen következményei:
- Ha minden kulcs egyszerű, akkor 1NF⇒2NF.
- Ha nincsenek másodlagos attribútumok, akkor 1NF⇒2NF.

Tétel: Minden 1NF reláció felbontható 2NF relációkba úgy, hogy azokból az eredeti reláció torzulás nélkül helyreállítható
(ld. a veszteségmentes dekompozícióról szóló 8.2.4. szakaszt).

Példa:
Az 5.1.9. szakaszban ismertetett feladathoz a relációs sémák megtervezéséhez a NAPI_HELYZET relációból indulunk ki, amely valamennyi attribútumot tartalmazza:
NAPI_HELYZET(DÁTUM, ÖSSZEG, ÁRUNÉV, DB, ÁRUKÓD, EGYSÁR, BEFIZ)
Ez a séma 1NF alakú, mivel benne minden attribútum atomi.
A sémán az alábbi funkcionális függőségeket definiáljuk (amelyek a valóságos viszonyokkal is jó összhangban vannak):

ÁRUKÓD→ÁRUNÉV, EGYSÁR	(az áru kódja meghatározza az áru nevét és egységárát)
DÁTUM, ÁRUKÓD→DB	(minden nap minden árucikkből meghatározott darabszámút adtak el)
DÁTUM→ÖSSZEG, BEFIZ	(minden nap egy jól meghatározott összeget visznek a bankba)
ÖSSZEG→BEFIZ	(a BEFIZ=ÖSSZEG-4000 törvényszerűség összekapcsolja BEFIZ és ÖSSZEG egy-egy értékét)
BEFIZ→ÖSSZEG	

Más funkcionális függőséget nem definiálunk.
Láthatóan a NAPI_HELYZET egy kulcsa a {DÁTUM, ÁRUKÓD} attribútumhalmaz, amely összetett kulcs. Nem nyilvánvaló ugyan, de a relációnak ez az egy kulcsa van.

Elsődleges attribútumok: DÁTUM, ÁRUKÓD
Másodlagos attribútumok: DB, ÖSSZEG, BEFIZ, ÁRUNÉV, EGYSÁR
A NAPI_HELYZET reláció nincsen 2NF-ben, mert pl. a DÁTUM → ÖSSZEG, BEFIZ függőség miatt van olyan attribútum (pl. ÖSSZEG), amelyet már a kulcs egy része is meghatároz (DÁTUM).
Bontsuk fel ezért a NAPI_HELYZET-et több relációra az alábbiak szerint:
ÁRU(ÁRUKÓD, ÁRUNÉV, EGYSÁR)
MENNYISÉG(DÁTUM, ÁRUKÓD, DB)
BEVÉTEL(DÁTUM, ÖSSZEG, BEFIZ)
Ahhoz, hogy ezen sémák normál formáiról mondhassunk valamit, szükségünk van a sémákon értelmezett függőségekre. (Ehhez részletesen ld. a vetített függőségekről szóló szakaszt). Itt csak a végeredményt közöljük:
$F_{ÁRU}$={ÁRUKÓD→ÁRUNÉV, EGYSÁR}
$F_{MENNYISÉG}$={DÁTUM, ÁRUKÓD→DB}
$F_{BEVÉTEL}$={DÁTUM→ÖSSZEG, BEFIZ, ÖSSZEG→BEFIZ, BEFIZ→ÖSSZEG}
Könnyen belátható, hogy most már mindhárom reláció 2NF-ben van: az első és utolsó azért, mert kulcsuk egyszerű kulcs, MENNYISÉG pedig azért, mert DB-t az összetett kulcs egyik része sem határozza meg.

8.2.3.4. A harmadik normál forma (3NF)

A definícióhoz szükségünk lesz néhány új fogalomra.

Definíció: (triviális függőség) Ha az X, Y attribútumhalmazokra igaz, hogy $Y \subseteq X$, akkor az $X \rightarrow Y$ függőséget *triviális függőségnek* nevezzük, egyébként a függőség *nemtriviális*.

Definíció: (tranzitív függés) Adott egy R séma, a sémán értelmezett funkcionális függőségek F halmaza, $X \subseteq R$, $A \in R$. A *tranzitíven függ* X-től, ha $\exists Y \subset R$, hogy $X \rightarrow Y$, $Y \not\rightarrow X$, $Y \rightarrow A$ és $A \notin Y$.

Pl.: adott R(Járat, Dátum, Pilótakód, Név)=R(J, D, P, N),
F={JD→P, P→N, N→P}.
Az N attribútum tranzitívan függ (J,D)-től, mert (J,D)→P, P$\not\rightarrow$JD, P→N.

Definíció 1: (3NF) Egy 1NF R séma 3NF, ha $\forall A \in R$ másodlagos attribútum és $\forall X \subseteq$ R kulcs esetén $\nexists Y$, hogy $X \rightarrow Y$, $Y \not\rightarrow X$, $Y \rightarrow A$ és $A \notin Y$.

Szavakkal: ha egyetlen másodlagos attribútuma sem függ tranzitívan egyetlen kulcstól sem.

Definíció 2: (3NF) Egy 1NF R séma 3NF, ha $\forall X \rightarrow A$, $X \subseteq R$, $A \in R$ nemtriviális függőség esetén

- vagy X szuperkulcs,
- vagy A elsődleges attribútum.

A 3NF első definíciója szemléletesebb, ha a redundanciacsökkentést tartjuk szem előtt. Felesleges u. i. egyazon relációban tárolni az X, Y, A attribútumokat. Hiszen minden olyan sorban, amelyben X és Y értékeit rendeljük egymáshoz, meg kell adnunk A értékét is, amelyek azonosak kell, hogy legyenek minden olyan sorban, amelyben Y értéke azonos. Márpedig Y értéke azonos lehet különböző X értékeke is.

Ilyenkor az Y→A függőségnek megfelelő (y,a) értékeket többszörösen tároljuk, nyilvánvaló redundanciát hagyva a relációban.
Egyes szerzők jobban kedvelik a 3NF második definícióját, talán azért, mert könnyebben ellenőrizhető ebben a formában, hogy egy séma teljesíti-e a 3NF kritériumait. A definíció szemléletes tartalma ugyanakkor itt már nem nyilvánvaló.

Tétel: Def. 1 ⇔ Def. 2.

Bizonyítás: Def. 1⇒ Def. 2.

Indirekt: Def. 1. feltételei mellett t.f.h. ∃Z→B∈F nemtriviális függőség, hogy sem nem szuperkulcs Z, sem nem elsődleges attribútum B.
Viszont minden relációnak létezik kulcsa, legyen ez X. Igaz tehát, hogy X→Z, Z↛X (mert akkor Z szuperkulcs lenne), Z→B, B∉Z (mert különben Z→B triviális függőség lenne). Ez pedig éppen egy másodlagos attribútum kulcstól való tranzitív függése, ellentmondásban Def. 1. feltételeivel. Tehát Def. 1⇒Def. 2.
Visszafelé: Def. 2⇒ Def. 1.
Indirekt: Def. 2. feltételei mellett t.f.h. ∃Y⊂R, ∃X kulcs és ∃A másodlagos attribútum, hogy X→Y, Y↛X, Y→A és A∉Y.
X→Y: mivel X kulcs, ezért nincs ellentmondásban Def. 2.-vel,
Y↛X, tehát Y nem lehet szuperkulcs,
Y→A és A∉Y miatt tehát létezik egy nemtriviális függőség, melyben Y nem szuperkulcs, A nem elsődleges attribútum, ellentmondásban Def. 2 feltételeivel. Tehát Def. 2⇒Def. 1.

Tétel: Minden legalább 1NF relációs séma felbontható 3NF sémákba úgy, hogy azokból az eredeti reláció információveszteség nélkül helyreállítható.

Bizonyítás: ld. 8.2.6. szakasz: veszteségmentes dekompozíció 3NF-be.

Példa:
Az előző szakasz ÁRU, MENNYISÉG és BEVÉTEL relációit vizsgáljuk meg, hogy teljesítik-e a 3NF kritériumait:
Az ÁRU(ÁRUKÓD, ÁRUNÉV, EGYSÁR) reláció nemtriviális függősége csupán ÁRUKÓD→ÁRUNÉV, EGYSÁR, tehát ÁRUKÓD kulcs, így ÁRU 3NF alakú.
A MENNYISÉG(DÁTUM, ÁRUKÓD, DB) reláció egyetlen nemtriviális függősége DÁTUM, ÁRUKÓD→DB, tehát (DÁTUM, ÁRUKÓD) kulcs, így MENNYISÉG is 3NF alakú.
A BEVÉTEL(DÁTUM,ÖSSZEG, BEFIZ) reláció nemtriviális függőségei:
DÁTUM→ÖSSZEG, BEFIZ
ÖSSZEG→BEFIZ
BEFIZ→ÖSSZEG
A BEVÉTEL egyetlen kulcsa tehát DÁTUM, másodlagos attribútumai ÖSSZEG és BEFIZ. Az utóbbi két függőségben a determináns nem szuperkulcs, és az sem teljesül, hogy ilyenkor a függőségek jobb oldalán elsődleges attribútum áll. BEVÉTEL így *nem* 3NF alakú. Bontsuk fel ezért az alábbi formában:
BEVÉT(DÁTUM,ÖSSZEG)
BEFIZ(DÁTUM, BEFIZETÉS)
Könnyen belátható, hogy mindkettő 3NF alakú.

Tétel: Ha egy séma 3NF alakú, akkor 2NF is egyben.

Bizonyítás: Indirekt. T. f. h. az R séma nem 2NF, ekkor ellentmondásra kell jutnunk a 3NF definíciójával. Ezzel ekvivalens, ha belátjuk azt, hogy másodlagos attribútum részkulcstól való függéséből következik kulcstól való tranzitív függése.

Legyen $A \in R$ másodlagos attribútum, K egy kulcs, amelyekre $K \rightarrow A$, továbbá $\exists K' \subset K$, hogy $K' \rightarrow A$ is igaz. $K' \not\rightarrow K$, hiszen ekkor K nem lehetne minimális, tehát nem lehetne kulcs. Továbbá $A \notin K$, mert A másodlagos attribútum, tehát egyetlen kulcsnak sem lehet eleme. Tehát: $K \rightarrow K'$, $K' \not\rightarrow K$, $K' \rightarrow A$, $A \notin K'$, ami éppen azt jelenti, hogy az A attribútum tranzitívan függ a K kulcstól, ellentmondásban a 3NF definíciójával.

8.2.3.5. A Boyce-Codd normál forma (BCNF)

Vegyük észre, hogy a 3NF definíciója csak a másodlagos attribútumok kulcstól való tranzitív függését zárja ki. Lehetséges tehát elsődleges attribútum kulcstól való tranzitív függése 3NF sémákban. Ez viszont azt jelenti, hogy - a már ismert okfejtést követve - a 3NF relációk is tartalmazhatnak még redundanciát. Indokolt tehát további normál forma, a *Boyce-Codd normál forma (BCNF)* bevezetése. Be fogjuk látni, hogy a BCNF sémákra illeszkedő relációk már redundanciamentesek.

Definícó 1: (BCNF) Egy 1NF R séma BCNF, ha $\forall A \in R$ attribútum és $\forall X \subseteq R$ kulcs esetén $\nexists Y$, hogy $X \rightarrow Y$, $Y \not\rightarrow X$, $Y \rightarrow A$ és $A \notin Y$.

Szavakkal: egyáltalán nincs tranzitív függőség kulcstól.

Definíció 2: (BCNF) Egy 1NF R séma BCNF, ha $\forall X \rightarrow A$, $X \subseteq R$, $A \in R$ nemtriviális függőség esetén X szuperkulcs.

Vegyük észre, hogy minden séma, amely legfeljebb két attribútumot tartalmaz, törvényszerűen BCNF.

Tétel: Def. 1 $\Leftrightarrow$ Def. 2.

Bizonyítás: Hasonló a 3NF definícióinál leírtakhoz

Def. 1$\Rightarrow$ Def. 2.

Indirekt: Def. 1. feltételei mellett t.f.h. $\exists Z \rightarrow B \in F$ nemtriviális függőség, hogy Z nem szuperkulcs.

Viszont minden relációnak létezik kulcsa, legyen ez X. Igaz tehát, hogy $X \rightarrow Z$, $Z \not\rightarrow X$, $Z \rightarrow B$, $B \notin Z$. Ez pedig éppen a B attribútum X kulcstól való tranzitív függése, ellentmondásban Def. 1. feltételeivel. Tehát Def. 1$\Rightarrow$Def. 2.

Visszafelé: Def. 2$\Rightarrow$ Def. 1.

Indirekt: Def. 2. feltételei mellett t.f.h. $\exists Y \subset R$, $\exists X$ kulcs és $\exists A$ attribútum, hogy $X \rightarrow Y$, $Y \not\rightarrow X$, $Y \rightarrow A$ és $A \notin Y$.

$X \rightarrow Y$: mivel X kulcs, ezért nincs ellentmondásban Def. 2.-vel,

$Y \not\rightarrow X$, tehát Y nem lehet szuperkulcs,

$Y \rightarrow A$ és $A \notin Y$ miatt tehát létezik egy nemtriviális függőség, melyben Y nem szuperkulcs, ellentmondásban Def. 2 feltételeivel. Tehát Def. 2$\Rightarrow$Def. 1.

Példa: Az előző szakasz ÁRU, MENNYISÉG, BEVÉT, BEFIZ sémái mind BCNF alakúak, ami a definíció alapján könnyen ellenőrizhető.

Tétel: Ha egy séma BCNF alakú, akkor 3NF is.

Bizonyítás: A két definíció közvetlen következménye.

Tétel: A BCNF sémákra illeszkedő relációk nem tartalmaznak redundanciát (legalábbis a funkcionális függőségek következtében).

Következmény: Emiatt egyetlen attribútum értékét sem lehet kikövetkeztetni más attribútumok értékeinek ismeretében, ismert funkcionális függőség alapján.

Bizonyítás: Indirekt. T. f. h. még van benne redundancia. Ez azt jelenti, hogy lehet benne olyan két sor, t és t', hogy egy A attribútum értékét a t sor értékei és a sémán értelmezett funkcionális függőségek alapján a t' sorban nem írhatjuk be tetszőlegesen, és Y ráadásul nem üres. Vizsgáljuk meg az alábbi relációt:

	X	Y	A
t	x	y1	a
t'	x	y2	?

Az X és az Y attribútumhalmazokat jelölje ki az a tény, hogy t és t'-ben az X értékei mind azonosak, míg léteznek olyan attribútumok is, amelyek t-n és t'-n különböznek. Ez utóbbiak az Y attribútumok, tehát $y1 \neq y2$. T. f. h. az attribútumok között definiált függőségi kapcsolatok miatt a ? helyére kötelezően a-t kell írnunk. Ez azt jelenti, hogy léteznie kell egy $Z \rightarrow A$ függőségnek, ahol nyilván $Z \subseteq X$. Viszont Z nem lehet szuperkulcs, mert akkor t és t' soroknak azonosaknak kellene lenniük, ami ellentmond annak a feltételezésnek, hogy $y1 \neq y2$. Ha pedig Z nem szuperkulcs, akkor a $Z \rightarrow A$ függőség ellentmond a BCNF definíciójának.

Ez utóbbi tétel azt sugallja, hogy a relációs adatbázisok tervezése során célszerű BCNF sémákat kialakítani. Hiszen ekkor - ha az attribútumaink között csak funkcionális függőségekkel leírható kapcsolatok vannak - a relációink redundanciamentesek lesznek, ami lényegesen megkönnyíti az adatokat kezelő alkalmazások megírását. T. f. u. is, hogy nem bízhatunk a redundanciamentességben: ekkor minden egyes új elem bevitele előtt ellenőrizni kell(ene), hogy a relációban már meglévő elemek és az új elem együttesen nincs-e ellentmondásban valamely ismert kényszerrel (pl. funkcionális függőséggel). Ez igen költséges lehet. Ezzel szemben BCNF sémák esetén elég csak a kulcsattribútumok értékeinek egyediségét biztosítani (ami pl. indexeléssel hatékonyan támogatható is), a többi attribútum értékét már tetszőlegesen vihetjük be a relációba. A redundancia minél alacsonyabb szinten tartása tehát kritikus az ú.n. tranzakciókezelő (manapság leginkább On-Line Transaction Processing, OLTP) rendszereknél[8]. Ennek tipikus példája a repülőtéri helyfoglaló rendszer.
A valóság ezzel szemben az, hogy még további szempontok is léteznek a relációs sématervezésnél és az adatbázisok üzemeltetésénél, amelyeket - bár még nem ismerünk - majd figyelembe kell venni. Emiatt nem lesz mód arra, hogy mindig BCNF sémákat alakítsunk ki.

A gyakorlatban ritkán dolgozunk egyetlen sémával, így esetenként egész sor, egy adott adatbázishoz tartozó sémáról kell megállapítani, hogy milyen normál formában található. Nem meglepő módon:

[8]Az ú.n. döntéstámogató rendszerekre a ritkán módosuló adatok, nagy tömegû, bonyolult és változatos lekérdezések jellemzőek. Ilyen esetekben a lekérdezés sebességére kell optimalizálni és emiatt kifejezetten ellenjavallt lehet a normalizált sémák alkalmazása. Ld. dimenziós modellezés, adattárházak.

Definíció: Egy adatbázis BCNF (3NF, 2NF, 1NF) alakú, ha a benne található valamennyi relációs séma rendre legalább BCNF (3NF, 2NF, 1NF).

8.2.4 Veszteségmentes felbontás (lossless decomposition)

Definíció: Egy R relációs sémának $\rho=\{R_1, R_2, ..., R_k\}$ egy felbontása, ha $R_1 \cup R_2 \cup ... \cup R_k = R$.

A felbontásnak a célja általában az, hogy ezáltal a rész-sémák valamely magasabb normál formába kerüljenek. Egy séma felbontásával nyilván a sémára illeszkedő relációk is felbomlanak. Itt azonban körültekintően kell eljárnunk, mert egy reláció ötletszerű felbontásakor információt is veszíthetünk. Ez abban nyilvánul meg, hogy később nem tudjuk a rész-relációkból az eredeti relációt helyreállítani. A helyreállítás itt a rész-relációk természetes illesztését jelenti, ha ez nem adja vissza az eredetit, akkor más mód nincsen rá.

Példa: Legyen az R(A,B,C) sémán egyetlen funkcionális függőség értelmezve: C→A. Vizsgáljuk meg a ρ_1 = (AB, BC) és a ρ_2 = (AC, BC) felbontásokat az alábbi reláción:

R(A,B,C)

A	B	C
a	c	e
a	d	f
b	c	g
b	d	h

$R'_1(A,B)$

A	B
a	c
a	d
b	c
b	d

$R'_2(B,C)$

B	C
c	e
d	f
c	g
d	h

$R'(A,B,C)=R'_1 \bowtie R'_2$

A	B	C
a	c	e
a	c	g
a	d	f
a	d	h
b	c	e
b	c	g
b	d	f
b	d	h

R(A,B,C)

A	B	C
a	c	e
a	d	f
b	c	g
b	d	h

$R''_1(A,C)$

A	C
a	e
a	f
b	g
b	h

$R''_2(B,C)$

B	C
c	e
d	f
c	g
d	h

$R''(A,B,C)=R''_1 \bowtie R''_2$

A	B	C
a	c	e
a	d	f
b	c	g
b	d	h

Láthatóan R'≠R, miközben R''=R. Tehát az R'-höz tartozó ρ_1 felbontásnak gyakorlati haszna aligha van, hiszen következtében az eredeti relációt többé nem tudjuk helyreállítani. Mivel ezzel információt veszítettünk, ezért ezt úgy fejezzük ki, hogy ρ_1 *veszteséges*. A ρ_2 felbontás viszont használható lehet, ha bebizonyosodik, hogy minden r(R,F) esetén hasonlóan viselkedik. Ekkor ρ_2-re azt mondjuk, hogy *veszteségmentes*.

Tanulság: amikor relációs adatbázis sémákat tervezünk, nem elegendő csupán a minél magasabb normál formákra, azaz minél kevesebb redundanciára törekedni. Egyidejűleg a veszteségmentesség biztosítása feltétlen követelmény, hiszen semmit sem ér egy kevés redundanciát tartalmazó adatbázis, amelyből hamis adatokat is ki lehet olvasni!

Definíció: Az R relációs séma egy $\rho(R_1, R_2, ..., R_n)$ *felbontását veszteségmentesnek* mondjuk, ha $\forall r(R)$ relációra

$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie ... \bowtie \Pi_{Rn}(r) = r.$

Legyen $m_\rho(r) \equiv \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie ... \bowtie \Pi_{Rn}(r)$ (*project-join mapping*).

Tétel: Adott egy R séma és egy $\rho(R_1, R_2, ..., R_n)$ felbontása. $\forall r(R)$ relációra $r \subseteq m_\rho(r)$.

Bizonyítás: Vegyünk egy tetszőleges $t \in r$ sort. Képezzük t vetületeit az R_i rész-sémákra, legyen ez $t[R_i]$, amely nyilván eleme az i-edik rész-relációnak. Ez a vetület nem változik $m_\rho(r)$-ben sem, és a természetes illesztés tulajdonságai miatt $m_\rho(r)$ valamely sorában valamennyi $t[R_i]$ megjelenik. Így $t \in m_\rho(r)$.

A tétel állítása más szavakkal: tetszőleges (tehát nemcsak veszteségmentes!) felbontás esetén sorok nem tűnhetnek el, csak újak keletkezhetnek.

Tétel: Adott egy R séma és R-nek egy $\rho(R_1, R_2, ..., R_n)$ veszteségmentes felbontása. Legyen $\sigma(S_1, S_2, ..., S_m)$ valamely $R_i \in \rho$ részsémának szintén veszteségmentes felbontása. Ekkor a $\tau(R_1, R_2,...,R_{i-1}, R_{i+1},...,R_n, S_1, S_2,...,S_m)$ R-nek szintén veszteségmentes felbontása.

Bizonyítás: A join asszociativitását kihasználva triviális.

Tétel: Adott egy R séma és R-nek egy $\rho(R_1, R_2, ...,R_n)$ veszteségmentes felbontása. Tetszőleges $\tau \supseteq \rho$ esetén τ is veszteségmentes.

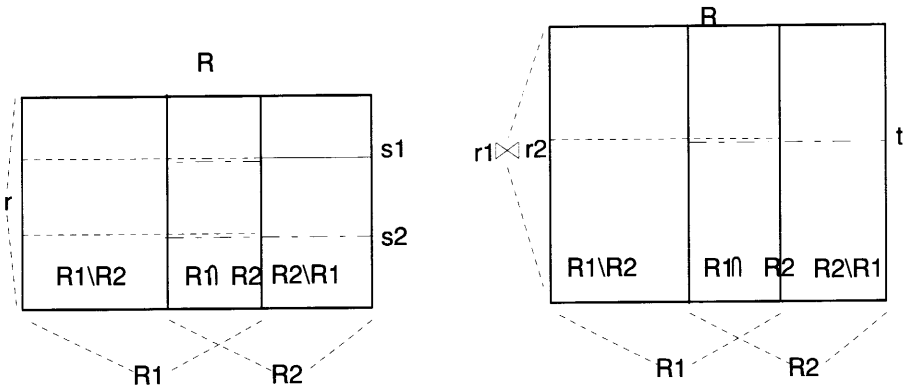
Bizonyítás: Ismét a join asszociativitását használjuk fel.
τ veszteségmentes, ha $\forall r(R)$ relációra $r = m_\tau(r)$.
$m_\tau(r) = m_\rho(r) \bowtie \Pi_{R_k}(r) \bowtie \Pi_{R_l}(r) \bowtie ... \bowtie \Pi_{R_m}(r)$, ahol $R_k, R_l, ..., R_m \in \tau$, de $\notin \rho$. Készítsük el ezért $m_\rho(r)$-et, melyre nyilván $m_\rho(r) = r$. Vegyünk ezután egy olyan R_i sémát, melyre $R_i \in \tau$, de $R_i \notin \rho$. Képezzük $m_\rho(r) \bowtie \Pi_{R_i}(r)$-t. Mivel $m_\rho(r) = r$, ezért $m_\rho(r) \bowtie \Pi_{R_i}(r) \equiv r \bowtie \Pi_{R_i}(r) \equiv r$. E miatt $m_\rho(r) \bowtie \Pi_{R_k}(r) \bowtie \Pi_{R_l}(r) \bowtie ... \bowtie \Pi_{R_m}(r) = r$, tehát $m_\tau(r) = r$, azaz τ veszteségmentes.

Tudjuk mostmár, hogy vannak "jó" és "rossz" sémafelbontások, de egyáltalán nem világos, hogy ennek az oka a reláció elemeiben vagy a séma szerkezetében (értsd: a sémán értelmezett függőségek rendszerében) keresendő. Konkrét esetben természetesen mindig kipróbálható, hogy egy felbontás melyik kategóriába esik - mint a fenti példában -, a gyakorlatban azonban ez a megközelítés nyilván használhatatlan. Szerencsére erre nincs szükség, megmutatható, hogy egy felbontás veszteségmentessége vagy veszteségessége kizárólag a relációs sémán és a sémán értelmezett függőségeken múlik. Természetesen ez csak akkor igaz, ha a reláció elemei nincsenek ellentmondásban a sémán értelmezett függőségekkel! Ebben az esetben a séma vizsgálata választ ad egy felbontás veszteségességére. Erre szolgál a következő tétel.

Tétel: Adott az R séma, a séma attribútumain értelmezett F függőséghalmaz és egy $\rho = (R_1, R_2)$ felbontás. ρ veszteségmentes $\Leftrightarrow$ $(R_1 \cap R_2) \rightarrow (R_1 \setminus R_2) \in F^+$ vagy $(R_1 \cap R_2) \rightarrow (R_2 \setminus R_1) \in F^+$.

Bizonyítás:

Visszafelé: Azt akarjuk belátni $(R_1 \cap R_2) \rightarrow (R_1 \backslash R_2) \in F^+$ vagy $(R_1 \cap R_2) \rightarrow (R_2 \backslash R_1) \in F^+$ alapján, hogy tetszőleges $t \in r_1 \bowtie r_2$ egyúttal r-nek is eleme.



Biztos, hogy $\exists s_1 \in r$, hogy $s_1[R_1] = t[R_1]$ (pontozás), továbbá

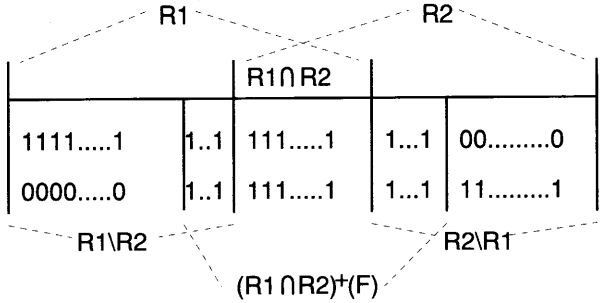
$\exists s_2 \in r$, hogy $s_2[R_2] = t[R_2]$ (pont-vonal).

Viszont $t[R_1 \cap R_2] = s_1[R_1 \cap R_2] = s_2[R_1 \cap R_2]$.

T. f. h. a kétféle lehetőség közül $(R_1 \cap R_2) \rightarrow (R_1 \backslash R_2)$ áll fenn, amely szerint, ha $s_1[R_1 \cap R_2] = s_2[R_1 \cap R_2]$, akkor kötelezően $s_1[R_1 \backslash R_2] = s_2[R_1 \backslash R_2]$. Emiatt - mint a fenti ábrán is látható - $s_2 = t$, azaz megtaláltuk azt az r-beli sort, amelyik egy tetszőleges $t \in r_1 \bowtie r_2$-vel megegyezik.

Előre: Indirekt módon bizonyítjuk. T. f. h. sem $(R_1 \cap R_2) \rightarrow (R_1 \backslash R_2) \in F^+$ sem $(R_1 \cap R_2) \rightarrow (R_2 \backslash R_1) \in F^+$ nem áll fenn. Ekkor ellentmondásra kell jutnunk a feltétellel. Ehhez elegendő egyetlen olyan, az adott sémára és függőségekre illeszkedő relációt találni, amely veszteségesen bontható fel.

Az alábbi ábrán látható reláció ilyen tulajdonságú.



Látható, hogy sem $(R_1 \cap R_2) \rightarrow (R_1 \backslash R_2)$ sem $(R_1 \cap R_2) \rightarrow (R_2 \backslash R_1)$ nem áll fenn. Azt kell még belátni, hogy kielégíti F valamennyi függőségét. Valamely $V \rightarrow W \in F$-re:

- ha V "kilóg" $(R_1 \cap R_2)^+$-ből, akkor nincs két azonos sor, tehát $V \rightarrow W$ teljesül bármely W-re.

- ha $V \subseteq (R_1 \cap R_2)^+$, akkor $(R_1 \cap R_2) \rightarrow V$, $V \rightarrow W$ miatt $(R_1 \cap R_2) \rightarrow W$, melynek feltétele, hogy $W \subseteq (R_1 \cap R_2)^+$. Így W értékei is azonosak, tehát $V \rightarrow W$ ekkor is teljesül.

Utolsó lépésként ellenőrizhető, hogy $r_1 \bowtie r_2 \neq r$, tehát a felbontás valóban veszteséges.

Példa: Tekintsük újra az R(A,B,C) sémát, amelynek F függéshalmaza csak a $C \rightarrow A$ függőséget tartalmazza. Vizsgáljuk meg rendre a $\rho_1=(AB, BC)$ és a $\rho_2=(AC, BC)$ felbontásokat a fenti tétel alapján!

ρ_1: $AB \cap BC=B$, $AB \setminus BC=A$, $B \rightarrow A \notin F^+$, hiszen nyilván nem következik $C \rightarrow A$-ból.

$AB \cap BC=B$, $BC \setminus AB=C$, $B \rightarrow C \notin F^+$, hiszen nyilván nem következik $C \rightarrow A$-ból. Tehát ρ_1 veszteséges.

ρ_2: $AC \cap BC=C$, $AC \setminus BC=A$, $C \rightarrow A \in F^+$ (triviális), tehát ρ_2 veszteségmentes.
($AC \cap BC=C$, $BC \setminus AC=B$, $C \rightarrow B \notin F^+$)

Megjegyzés: a tételt arra is felhasználhatjuk, hogy segítségével veszteségmentes felbontásokat készítsünk $\rho=(R_1, R_2)$ formába. Ehhez bármely $X \rightarrow Y \in F$ nemtriviális (vagy akár $X \rightarrow Y \in F^+$) függés alapján, ahol X és Y diszjunktak, legyen

- $R_1=XY$, ill.
- $R_2=R \setminus Y$.

Könnyen ellenőrizhető, hogy ekkor az $(R_1 \cap R_2) \rightarrow (R_1 \setminus R_2)$ pontosan $X \rightarrow Y$ formában fog teljesülni, azaz ρ veszteségmentes lesz.

Tétel: Adott R(XYZ) és funkcionális függőségek F halmaza esetén, ahol X, Y és Z (páronként diszjunkt) attribútumhalmazok is lehetnek $\rho=(XY, XZ)$ pontosan akkor veszteségmentes, ha $X \rightarrow Y \in F^+$ vagy $X \rightarrow Z \in F^+$.

Bizonyítás: az előző tétel közvetlen következménye.

Az előző tételek hatékony analízis és tervezési módszert adtak arra, hogyan lehet egy sémát veszteségmentesen két részre bontani. Az alábbi módszer tetszőleges számú részre bontott részsémáról lehetővé teszi a veszteségmentesség eldöntését.

Adott egy $R(A_1, A_2, ..., A_k)$ séma, a sémán értelmezett F függőségek és R-nek egy ρ $(R_1, R_2, ..., R_n)$ felbontása. Táblázatot készítünk n sorral és k oszloppal. Az oszlopokat a séma attribútumainak, a sorait a részsémáknak feleltetjük meg. Kiindulási állapotként úgy töltjük ki a táblázatot, hogy az i-edik sor j-edik oszlopába

- a-t írunk, ha $A_j \in R_i$
- b_i-t írunk, ha $A_j \notin R_i$.

Ezután mindaddig módosítjuk a táblázat elemeit az F függőségeinek figyelembe vételével, az alábbiak szerint, ameddig a táblázat változik:

Vegyük egy tetszőleges $X \rightarrow Y \in F$ függést. Ha létezik két olyan sor a táblázatban, amely X-en azonos, akkor Y-on is egyenlővé tesszük őket, mégpedig

- ha valahol a-t találunk, akkor a másik sor azonos oszlopának eleme is legyen a,
- ha nem a egyik sem, akkor b_i-t és b_j-t tegyük egyenlővé tetszőlegesen.

Példa: Legyen R(S, A, I, P), $F=\{S \rightarrow A, SI \rightarrow P, P \rightarrow S\}$, ρ(SA, SI, IP, PS).
Kezdőállapot:

	S	A	I	P
SA	a	a	b_1	b_1
SI	a	b_2	a	b_2
IP	b_3	b_3	a	a
PS	a	b_4	b_4	a

Végállapot:

	S	A	I	P
SA	a	a	b_1	b_1
SI	a	a	a	a
IP	a	a	a	a
PS	a	a	b_4	a

Tétel: a ρ felbontás veszteségmentes ⇔ van csupa 'a'-ból álló sor.

Bizonyítás előre:
A táblázatot tekintsük olyan r(R) relációnak, ahol a-ket és b_i-ket DOM(A_j)-ből választottuk. A végállapotban r kielégíti F valamennyi függőségét, hiszen az algoritmus pontosan az F-beli függőségek sértéseit korrigálja. Bontsuk fel r-et is ρ-nak megfelelően, ekkor a táblázat kezdőállapotának konstrukciója miatt igaz, hogy $\Pi_{R_i}(r)$-ben lesz olyan sor, amely csupa 'a'-ból áll, minden i-re. Tekintve, hogy $m_\rho(r) \equiv \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie ... \bowtie \Pi_{R_n}(r)$, így $m_\rho(r)$-ben biztosan keletkezik olyan sor, amely csak 'a'-kat tartalmaz. Ha ρ veszteségmentes, akkor $m_\rho(r)=r$, minden r-re, tehát a végállapot-beli táblázat $\equiv m_\rho(r)$, azaz tartalmazza a csupa 'a'-ból álló sort.
Bizonyítás visszafelé: nem bizonyítjuk.

8.2.5 Függőségőrző felbontások

Vizsgáljuk meg a következő példát! Adott az R(VÁROS, ÚT, IR_SZÁM) = (V, U, I) séma, amelyen - a valósággal jó összhangban - az F={VU→I, I→V} függőségeket értelmeztük. Készítsük el a séma ρ=(UI, VI) felbontását! (Könnyen ellenőrizhető az előző tétel segítségével, hogy ρ veszteségmentes, tehát tetszőleges, (R,F)-re illeszkedő r reláció esetén r helyreállítható természetes join-nal a rész-sémákra vonatkozó vetületeiből.)
Egy adatrögzítő a rész-sémákba az alábbi adatokat vitte be:

r_1

ÚT	IR_SZÁM
Kossuth	2142
Kossuth	2144

r_2

VÁROS	IR_SZÁM
Baja	2142
Baja	2144

Helyreállítva az eredeti relációt az alábbi sorokat kapjuk:

ÚT	IR_SZÁM	VÁROS
Kossuth	2142	Baja
Kossuth	2144	Baja

Az eredménnyel az a probléma, hogy nyilvánvalóan nincs összhangban a feltételezett VU→I függőséggel, amely szerint egy városban egy utcának csak egyetlen irányítószáma lehet. Tehát a rész-relációk egyesítése "hamis" eredményre vezetett. A

hiba azonban nem az egyesítésnél van, hiszen a felbontás veszteségmentes, tehát az eredeti relációt kell visszakapnunk. A valódi ok az, hogy *nem is volt* 'eredeti' reláció, azaz nem tudunk olyan r relációt konstruálni, amely illeszkedik (R,F)-re és ugyanakkor a fenti r_1 és r_2 vetületei lennének.

Mindaddig, amíg a séma egyben van, minden egyes sor bevitele előtt elvileg lehetőség van ellenőrizni, hogy a bevitt adatok a meghatározott, pl. függőségi feltételeknek megfelelnek-e (integrity constraints). Sajnos, amint az R sémát felbontottuk, már nem feltétlenül tudjuk az eredeti függőségeket alkalmazni a rész-relációkban annak elkerülésére, hogy "hamis" adatok kerüljenek be az adatbázisba, legfeljebb a függőségeknek a rész-sémákra való "vetületeit".

Definíció: Adott az R séma attribútumain értelmezett függőségek F halmaza. A *függőségeknek egy* $Z \subset R$ *attribútumhalmazra való vetítése* az a $\Pi_Z(F)$ függőséghalmaz, amelyre $\Pi_Z(F)=\{X \rightarrow Y \mid X \rightarrow Y \in F^+ \text{ és } XY \subseteq Z\}$

Ezek a vetített függőségek bizonyos esetekben elegendőek lehetnek az adatbázis integritásának megőrzéséhez. Ehhez az kell, hogy az R_i részsémákra vetített függőségek ugyanazt az információt hordozzák, mint az eredeti F függéshalmaz.

Definíció: Adott egy R relációs séma és egy, a sémán értelmezett F függéshalmaz. A séma $\rho=\{R_1, R_2, ..., R_k\}$ felbontása *függőségőrző*, ha $\{\Pi_{Ri}(F)\} \models F$.

Egy relációs séma veszteségmentes, de nem függőségőrző felbontása eredményezheti tehát, hogy a rész-sémákban nem tudjuk többé az eredeti sémában érvényes függőségeket alkalmazni. Ennek következtében a rész-relációinkba nem megengedett adatok is bekerülhetnek. Ez ellen csak úgy védekezhetnénk, ha minden egyes új sor bevitele előtt előállítanánk az eredeti relációt és ellenőriznénk azon a függőségi viszonyok fennállását. Ez érezhetően nagyon költségessé tenné sok sort tartalmazó relációk esetén az új sorok bevitelét. Ezt elkerülendő célszerű tehát olyan sémafelbontásokat készíteni, amelyek függőségőrzőek.

A definíciókon alapuló kézenfekvő, de nehézkesen alkalmazható lehetőséget nyújt egy sémafelbontás függőségőrző tulajdonságának tesztelésére az alábbi algoritmus:

- elkészítjük F^+-t
- vetítjük F^+ valamennyi függőségét valamennyi rész-sémára
- meghatározzuk a vetített függőségek lezárását
- ha ez azonos F^+-tal, akkor a felbontás függőségőrző, ellenkező esetben biztosan nem az.

Megj.: Egy felbontás lehet veszteségmentes és függőségőrző,
veszteségmentes és nem függőségőrző,
veszteséges és függőségőrző,
veszteséges és nem függőségőrző.

8.2.6 Sémadekompozíció adott normálformába

Az előzőek alapján belátható, hogy gyakorlati szempontból az olyan sémafelbontások bírnak jelentőséggel, amelyeknél biztosítható, hogy az eljárás eredménye meghatározott tulajdonságokat mutató sémák halmaza lesz. A legfontosabb tulajdonságok, amelyeket figyelembe kell venni (nem feltétlenül fontossági sorrendben):

- adott normál forma elérése,
- veszteségmentesség,
- függőségőrző tulajdonság.

Adott tulajdonságú felbontások létezését állítják a következő tételek:

Tétel: Minden R relációs séma és a sémán értelmezett F függéshalmaz esetén $\exists\rho$ sémafelbontás, amely veszteségmentes és függőségőrző, továbbá $\forall R_i \in \rho$-ra R_i 3NF tulajdonságú.

Bizonyítás: konstruktív. Képezzük az adott függéshalmaz egy minimális fedését, legyen ez G. Ha $G=\{X_1 \rightarrow A_1, X_2 \rightarrow A_2,...,X_n \rightarrow A_n\}$ alakú, akkor a felbontás legyen $\rho = \{X_1A_1, X_2A_2,...,X_nA_n\} \cup \{K\}$, ahol K az R séma egy kulcsa.

A ρ felbontás természetesen függőségőrző, mert a részsémákra vetített függőségek közül egy pontosan megegyezik az egyik G-beli függőséggel, így a vetített függőségek lezártja nyilván megegyezik G lezártjával.

A 3NF tulajdonságot indirekt módon bizonyítjuk. T. f. h. $\exists Y \rightarrow B \in G^+$, $YB \subseteq X_iA_i$, valamely i-re, amely az i-edik séma 3NF tulajdonságát sérti: azaz $B \notin Y$, Y nem szuperkulcsa R_i-nek és B nem elsődleges attribútum.

- Ha $B=A_i$, akkor $Y \subseteq X_i$ és mivel Y nem szuperkulcsa X_iA_i-nak, így $Y \subset X_i$. De ekkor a feltételezett $Y \rightarrow B$ miatt $Y \rightarrow A_i$ ugyanazt fejezi ki, mint $X_i \rightarrow A_i$, így G-ben $X_i \rightarrow A_i$ helyett $Y \rightarrow A_i$,-nek kellett volna szerepelnie.
- Most t. f. h. $B \neq A_i$. Mivel X_i szuperkulcs X_iA_i-ben, ezért $\exists Z \subseteq X_i$, amely kulcs. $B \neq A_i$ miatt $B \in X_i$, de $B \notin Z$, mert feltételeztük, hogy B nem elsődleges attribútum. Így X_i feltétlenül bővebb Z-nél legalább B-vel. Ekkor viszont az $X_i \rightarrow A_i$ függőség helyettesíthető G-ben $Z \rightarrow A_i$-vel.

Meg kell még vizsgálnunk, hogy a K kulcsként (esetleg) megjelenő n+1-edik séma is 3NF-e. Ha $Y \rightarrow B$ nemtriviális függés és $YB \subseteq K$, akkor K nem lehet minimális, hiszen belőle B elhagyható.

Láthatóan mindhárom esetben ellentmondásra jutottunk a feltétellel, az első kettőben azzal, hogy G minimális függéshalmaz, a harmadikban a K kulcs minimális tulajdonságával. Tehát valamennyi részséma 3NF.

Lássuk be most ρ veszteségmentességét. Ehhez a táblázatos tesztet fogjuk használni. Megmutatjuk, hogy a táblázat végállapotában a K sora csupa 'a'-t tartalmaz.

Képezzük először K^+-t a tanult algoritmussal. Ennek során rendre a $B_1, B_2,...,B_k$ attribútumokkal bővítjük $K^{(0)}=K$-t, ebben a sorrendben. Nyilván igaz, hogy $K \cup \{B_1, B_2,...,B_k\}=R$. Megmutatható, hogy a táblázat módosítása során B_i-k sorrendjében lehetőségünk van K sorába 'a'-kat írni. Ezt i-re vonatkozó teljes indukcióval láthatjuk be.

i=0 megfelel annak, hogy a K sorában a K attribútumainak oszlopaiban 'a'-k vannak.

T. f. h. i-1-re még működik, majd adjuk hozzá $K^{(i)}$-hez B_i-t valamely $Y \rightarrow B_i \in G$ függőség miatt. K^+ konstrukciója miatt biztos, hogy $Y \subseteq K \cup \{B_1, B_2,...,B_{i-1}\}$. Ekkor a táblázatban a K és az YB_i sémák sorainak attribútumértékei az Y oszlopaiban mind megegyeznek, mégpedig mind 'a', mert

- K sorában K attribútumainak oszlopai a táblázat konstrukciója miatt mindenkor 'a'-k, $B_1, B_2,...,B_{i-1}$ oszlopai pedig már 'a'-k,

- YB_i sorában Y attribútumainak oszlopaiban a táblázat konstrukciója miatt mindenkor 'a'-k állnak.

Mivel YB_i sorában Bi oszlopában 'a' áll, ezért jogosan írhatunk K sorába B_i oszlopába is 'a'-t.

Megjegyzések:

1. Vegyük észre, hogy a a K kulcs csak a veszteségmentes felbontáshoz kell. Ha a sémafelbontásból kihagyjuk, csak a függőségőrző és 3NF tulajdonság garantálható.
2. ρ konstrukciója során kiderülhet, hogy valamely R_i részséma tartalmazza R valamely kulcsát. Ekkor felesleges egy i+1-edik sémát külön definiálni egy kulcs számára, hiszen a veszteségmentességet biztosító valamennyi mechanizmus ekkor is működik.

Példa: Adott az R(C, T, H, R, S, G) séma és egy minimális függéshalmaz: G={C→T, HR→C, HT→R, CS→G, HS→R)

A ρ=(CT, HRC, HTR, CSG, HSR) sémafelbontás az 1. megjegyzés miatt 3NF, függőségőrző felbontás. Ha azt szeretnénk, hogy a felbontás veszteségmentes is legyen, akkor keressük meg R (egyik) kulcsát! Azt gondoljuk, hogy az SH attribútumhalmaz kulcs. Határozzuk meg ezért a lezárását:

$SH^{(0)}$=SH, $SH^{(1)}$=SHR, $SH^{(2)}$=SHRC, $SH^{(3)}$=SHRCG, $SH^{(4)}$=SHRCGT, tehát SH^{+}=SHRCGT, az attribútumok teljes halmaza, így SH legalábbis szuperkulcs. Ugyanakkor minimális is, mert F-ben sem S, sem H egyedül semmilyen más attribútumot nem határoz meg, S^{+}=S, H^{+}=H, vagyis az SH valóban kulcs.

Ha hozzávesszük a ρ felbontáshoz az SH sémát a 2. megjegyzés szerint, akkor az eredmény ρ-val azonos lesz, hiszen ρ utolsó sémája tartalmazza az SH attribútumokat. Tehát valójában ρ veszteségmentes is volt.

Tétel: Minden, legalább 1NF R sémának létezik veszteségmentes felbontása BCNF sémákba.

Bizonyítás: Iteratívan készítjük el a felbontást. Az iteráció minden fázisában igaz lesz, hogy a pillanatnyi felbontás veszteségmentes.

1. Ha R az adott F függőségek mellett BCNF, akkor nincs tennivaló, készen vagyunk.
2. Ha R nem BCNF, akkor $\exists X \rightarrow A \in F^{+}$, ami megsérti a BCNF tulajdonságokat, azaz $A \notin X$ és X nem szuperkulcsa R-nek. Legyen ekkor a felbontás $\rho_1(R_1, R_2)$, R1=XA, R2=R\A, melyről belátható, hogy
   - veszteségmentes felbontás és
   - R_1 és R_2 is kevesebb attribútumot tartalmaz, mint R. R_2 nyilván kisebb R-nél, hiszen az A attribútumokat nem tartalmazza. R_1 pedig azért kisebb, mert különben X→A nem sérthetné a BCNF tulajdonságot.
3. Vizsgáljuk meg, hogy R_1, ill. R_2 BCNF-e. Ehhez meg kell határoznunk a részsémákra vetített függőségeket. Ha mindkettő BCNF, akkor készen vagyunk.
4. Ha R_1, R_2 között van, amelyik nem BCNF, akkor arra a sémára ismételjük meg az eljárást a 2. ponttól. Mivel egy veszteségmentes felbontás veszteségmentes felbontása is veszteségmentes, így az iteráció tetszőleges mélységben folytatható. Másrészről, mivel a legfeljebb két attribútumot

tartalmazó sémák mind BCNF-ek, így előbb-utóbb a felbontás részei BCNF sémák lesznek.

Példa: Legyen adott a NYELV(DIÁKKÓD, DIÁKNÉV, OFŐNÖK, OFTEL, NYELV, FÉLÉVKÓD, OSZTÁLYZAT) relációs séma és a funkcionális függőségek alábbi (nem minimális) halmaza:

DIÁKKÓD→DIÁKNÉV
DIÁKKÓD→OFŐNÖK
OFŐNÖK→OFTEL
OFTEL→OFŐNÖK
DIÁKKÓD→OFTEL
DIÁKKÓD, NYELV,FÉLÉVKÓD→OSZTÁLYZAT

Bontsuk fel BCNF alakú sémákba veszteségmentes dekompozícióval!
A séma egyetlen kulcsa a {DIÁKKÓD, NYELV, FÉLÉVKÓD} attribútumhalmaz. Válasszuk ki az OFŐNÖK→OFTEL függőséget, és bontsuk fel a NYELV relációt két részre az előbbiek szerint:

R_1(OFŐNÖK, OFTEL)
R_2(DIÁKKÓD, DIÁKNÉV, OFŐNÖK, NYELV, FÉLÉVKÓD, OSZTÁLYZAT)

Az R_1 BCNF alakú, így csak az R_2-t vizsgáljuk tovább.
Ennek függőségei:

DIÁKKÓD→DIÁKNÉV
DIÁKKÓD→OFŐNÖK
DIÁKKÓD, NYELV, FÉLÉVKÓD→OSZTÁLYZAT,

tehát R_2 kulcsa továbbra is {DIÁKKÓD, NYELV, FÉLÉVKÓD}.
R_2 még mindig nem BCNF alakú, mert a DIÁKKÓD determináns, de nem szuperkulcs.
Az első két függőséget az alábbi formában is felírhatjuk:
DIÁKKÓD→DIÁKNÉV,OFŐNÖK
Konstruáljuk meg a következő felbontást ezen függőség alapján:

R_3(DIÁKKÓD, DIÁKNÉV, OFŐNÖK,)
R_4(DIÁKKÓD, NYELV, FÉLÉVKÓD, OSZTÁLYZAT)

Itt már R_3 és R_4 is BCNF alakú, tehát készen vagyunk. A keresett felbontás az R_1, R_3 és az R_4 sémák együttese.

Megjegyzések:

1. Ez a tétel az egyik oka annak, amiért nem érdemes minden esetben BCNF alakokra törekedni egy relációs adatbázis tervezése során. (A másik az, hogy sok kis relációból általában költségesebb, tehát adott gépen lassúbb egy lekérdezés eredményének összeállítása. Esetleg éppen a lekérdezési válaszidők csökkentése érdekében alkalmanként szándékosan redundanciát építenek bele a relációkba.)
2. Vegyük észre, hogy más függőségeket (ill. más sorrendben) választva a felbontás alapjául az eredményül kapott sémák is más-más attribútumokat tartalmazhatnak.
3. Egy másik, kézenfekvő lehetőség BCNF sémafelbontások előállítására, hogy a 3NF, függőségőrző és veszteségmentes sémák előállítására alkalmas, fentebb leírt algoritmussal előállított felbontásból indulunk ki. Minden rész-sémára megvizsgáljuk, hogy az BCNF-e. Ha egy séma nem BCNF, akkor két részre

bontjuk veszteségmentes dekompozícióval pontosan az imént leírt módszerrel. A módszer előnye, hogy hamarabb eredményre vezethet, mivel garantáltan 3NF sémákból indul ki.

4. A BCNF tulajdonság ellenőrzése igen költséges is lehet. Szerencsére, számos egyszerűsítésre nyílik lehetőség, amelynek a részleteire azonban nem térünk ki.

Könnyű példát mutatni arra, amikor egy séma nem bontható fel veszteségmentesen és függőségőrzően BCNF sémákba.

Vegyük ismét a már ismert R(VÁROS, ÚT, IR_SZÁM) = (V, U, I) sémát, amelyen most is az F = {VU→I, I→V} függőségeket értelmeztük. Így R kulcsa VU, az I→V függőség miatt R nem BCNF. Készítsük el a séma egy valódi, veszteségmentes felbontását! Könnyen ellenőrizhető, hogy $\rho = (UI, VI) = (R_1, R_2)$ az egyetlen lehetőség. A részsémák nyilván BCNF-ek. A vetített függőségek: $\Pi_{R_1}(F)$ csak triviális függőségeket tartalmaz, $\Pi_{R_2}(F)$ = {I→V, triviális függőségek}. Mivel a VU →I függőség a sémafelbontás során elveszett, ezért ρ nem függőségőrző.

Az előbbiek alapján a normalizálás elméletét egy másik módon is felépíthetjük: redundanciamentes relációkat akarunk létrehozni függőségőrző és veszteségmentes felbontással. A redundanciamentességéhez az szükséges, hogy minden sémán függőség csak szuperkulcstól lehessen (BCNF). De ekkor függőségőrző és veszteségmentes felbontásokat nem tudunk készíteni. Ezért célszerű egy enyhébb normál forma bevezetése is. Ez lesz a 3NF, amely "éppen annyi" redundanciát tartalmaz, hogy mellette függőségőrző és veszteségmentes felbontást lehessen garantálni. A 2NF jelentősége ebben a gondolatkörben marginális.

8.2.7 Többértékű függőségek

Induljunk ki az R(TANTÁRGY, TANÁR, JEGYZET) sémából. Az ehhez tartozó relációnak legyenek elemei mindazon tanárok, akik egy adott tantárgyat egy adott jegyzet felhasználásával tanítanak. Például az alábbi reláció adódott:

r(R)	TANTÁRGY	TANÁR	JEGYZET
	matematika	Fenyő	Vektoranalízis
	matematika	Fenyő	Num. analízis
	matematika	Lovász	Vektoranalízis
	matematika	Lovász	Num. analízis
	fizika	Öveges	Vektoranalízis
	fizika	Lovász	Vektoranalízis
	kémia	Fenyő	Szerves kémia

Az R sémán nem értelmeztünk funkcionális függőségeket. Ennek következtében R kulcsa kizárólag az attribútumainak teljes halmaza, tehát R BCNF alakú. Ennek ellenére úgy érezzük, hogy redundanciát tartalmaz. Ha pl. Lovász helyett Juhász jön matematikát tanítani, akkor ezt több helyen is ki kell javítani az adatbázisban (update anomália). Próbáljuk meg R-et felbontani és a redundanciát csökkenteni!

r_1	TANTÁRGY	TANÁR
	matematika	Fenyő
	matematika	Lovász
	fizika	Öveges
	kémia	Fenyő
	fizika	Lovász

r_2	TANTÁRGY	JEGYZET
	matematika	Vektoranalízis
	matematika	Num. analízis
	fizika	Vektoranalízis
	kémia	Szerves kémia

Ez a felbontás már mentes az említett anomáliától, ráadásul a két relációból az eredeti veszteségmentesen helyreállítható, tehát "jobb"! Ugyanakkor bizonyos, hogy a felbontás nem a funkcionális függőségek figyelembe vételén alapult, mint eddig minden esetben.

Az ok az úgynevezett *többértékű függőség*ekben rejlik, amely a funkcionális függőségek általánosításának tekinthető. A funkcionális függőségek esetén egy attribútum(halmaz) értéke egy másik attribútum(halmaz) értékét meghatározta. A többértékű függőségek esetén pedig egy attribútum(halmaz) értéke egy másik attribútum(halmaz) értékeinek egy halmazát határozza meg: jelen esetben egy tantárgyhoz egy tanár-halmaz (ill. egy jegyzet-halmaz is) tartozik.

Ez azonban nem elég ahhoz, hogy többértékű függőségről beszélhessünk a tantárgyak és a tanárok között, ehhez egy további szabályszerűség is tartozik, amely során a sémában található többi attribútumot is figyelembe kell venni:

Ha (tantárgy, tanár1, jegyzet1) valamint (tantárgy, tanár2, jegyzet2) is egy-egy eleme a relációnak, akkor (tantárgy, tanár1, jegyzet2) valamint (tantárgy, tanár2, jegyzet1) is a relációhoz kell, hogy tartozzon. (Ha töröljük az r(R) reláció első 4 sora közül bármelyiket, utána R_1 és R_2 egyesítése többé nem adja vissza R-et!)

Mindezt fogalmazzuk meg pontosabban:

Definíció: *többértékű függőség (TÉF)*:

Adott egy R séma és attribútumainak egy X és Y halmaza. Ha $\forall t_1,t_2 \in r(R)$-hez, amelyre $t_1[X] = t_2[X]$ (de más attribútumokon ez nem áll fenn!) $\exists t_3$, $t_4 \in r(R)$, hogy

- $t_3[X] = t_4[X] = t_1[X]$,
- $t_3[Y] = t_1[Y]$ és $t_3[\Omega\text{-}XY] = t_2[\Omega\text{-}XY]$,
- $t_4[Y] = t_2[Y]$ és $t_4[\Omega\text{-}XY] = t_1[\Omega\text{-}XY]$, akkor

Y többértékűen függ X-től (X multideterminálja Y-t). Jelölése: $X \twoheadrightarrow Y$.

Megj.: Vegyük észre, hogy a definíció szimmetriája miatt egyidejűleg fennáll $X \twoheadrightarrow \Omega$ -XY is.

$X \twoheadrightarrow Y$ tehát "ekvivalens" $X \twoheadrightarrow \Omega$-XY-nal, és mindkettõ azt jeleni, hogy X minden egyes értéke meghatározza Y és Ω-XY lehetséges értékeinek egy halmazát, és ezen értékek minden egyes kombinációja (tehát minden megengedett Y érték mindegyik megengedett Ω-XY értékkel) elõ is kell, hogy forduljon ilyen X mellett.

Tétel: Ha $X \rightarrow Y$ fennáll, akkor $X \twoheadrightarrow Y$ is igaz.

Biz.: A többértékű függőség definíciójából következik, ilyenkor az X-hez tartozó Y halmaznak csak egyetlen eleme van.

A többértékű függőségeknek szintén léteznek axiómái, melyek az Armstrong axiómákhoz hasonlóak. Számos tétel is általánosítható a többértékű függőségekre.

Részletes ismertetésük túlmutat a tárgy keretein, az alábbiak elsősorban illusztrációnak tekintendők.

Tétel: Legyen R egy séma, $\rho=(R_1, R_2)$ egy felbontása, D pedig az R sémán értelmezett funkcionális és többértékű függőségek halmaza. ρ akkor és csak akkor veszteségmentes, ha $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$ (vagy $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$) következik a D függőségekből.

Speciálisan, ha R(A, B, C) és ρ=(AB, AC) alakú, akkor ρ veszteségmentességének szükséges és elégséges feltétele, hogy $A \twoheadrightarrow B$ (vagy $A \twoheadrightarrow C$) fennálljon. Ennek elégséges feltétele, hogy $A \rightarrow B$ vagy $A \rightarrow C$ igaz legyen, összhangban az 8.2.4. szakasz 4. tételével. (A tétel azért is hasznos, mert egyúttal módszert is ad többértékű függőségek tesztelésére.)

Létezik egy általánosítása a BCNF-nek többértékű függőségek esetére, amit *negyedik normál formának (4NF)* neveznek.

Definíció: Egy relációs séma 4NF alakú, ha $X \twoheadrightarrow Y$ esetén X szuperkulcs, miközben
1. Y nem részhalmaza X-nek, és
2. XY-on kívül a sémának van más attribútuma is.

Megj. 1: A kulcs, ill. szuperkulcs definíciójában továbbra is csak funkcionális függéseket értelmezünk.
Megj. 2: Ha egy sémán csak funkcionális függőségeket értelmezünk, akkor 4NF= BCNF.
Megj. 3: Minden relációs séma, amelyen funkcionális és többértékű függőségeket is értelmezünk, felbontható veszteségmentes dekompozícióval 4NF alakú sémákba.

9 Tranzakciók adatbáziskezelő rendszerekben

Mindeddig arról volt szó - hallgatólagosan -, hogy valamely adatbáziskezelő rendszer egyidőben egyetlen felhasználó egyetlen programját szolgálja ki, egyéb igények csak a korábbiak kielégítése után következhetnek. A továbbiakban annak a problémakörét vizsgáljuk meg, ha több felhasználó/több program egyidejűleg kerül kiszolgálásra. Maga az alapprobléma és számos kapcsolódó algoritmus már ismert az Operációs rendszerek c. tárgyból, ezért itt azokra a sajátosságokra térünk ki elsősorban, amelyek az adatbáziskezelő rendszerek környezetére jellemzőek.

9.1 Bevezető

Definíció: Tranzakció: egy program egyszeri futása, amelynek vagy valamennyi hatása hatásos, vagy belőle semmi sem (→ atomicitás).

Definíció: (ütemezés, schedule) tranzakciók elemi műveleteinek összessége, melyben a műveletek időbeli sorrendje is egyértelműen meghatározott.

Problémák a tranzakciókkal:

- idő előtti befejeződés — nullával osztás/illegális művelet végzése
  nem fér hozzá adathoz
  kívülről lövik ki
- időosztásos rendszerben a tranzakciók összefésülődhetnek, mert a tranzakcióhoz rendelt időszelet hamarabb véget érhet.

A tranzakciók maguk is elemi lépésekből állnak: írás/olvasás + kiegészítő lépések: ezek szintén oszthatatlannak tekintettek.
A tranzakciók egyidejű végrehajtása, egymásba fésülődése csak akkor probléma, ha közös erőforráshoz akarnak hozzáférni (concurrency). A közös erőforrások jelen vizsgálatainkban csupán az adatok lesznek.
Az adathozzáférés

- egységeinek megválasztása ("granularity") kritikus rendszertechnikai kérdés,
- módjának szabályozása pl. zárakkal (lock) történhet.

Definíció: A zár egy hozzáférési pivilégium egy adategységen, amely adható és visszavonható.

Hogyan kezelik a zárak a konkurrenciát?
Példa: két tranzakciót tartalmazó ütemezés, zárak nélkül:

T_1: {read A A=A+1 write A}

T_2: { read A A=A+1 write A}

Végeredményben A értéke csak 1-gyel nőtt, bár 2-t várnánk.
Egy megoldás zárakkal a problémára:

T_1: {lock A read A A=A+1 write A unlock A}

T_2: {<itt T_2 nem fér hozzá A-hoz, ezért pl. vár> lock A read A A=A+1 write A unlock A}

Ekkor már helyesen, 2-vel fog nőni A értéke T_1 és T_2 lefutása után.

A {LOCK A.....UNLOCK A} műveletek között más tranzakció csak korlátozottan (vagy sehogyan sem) fér hozzá az A adategységhez. Ezért a zárak szinkronizációs primitívként is szolgálnak: ha egy tranzakció lockolni akar egy adategységet, amin egy másik tranzakció tart fenn zárat, akkor addig nem mehet tovább, amíg a zár -bármely okból kifolyólag- fel nem szabadul.

Definíció: legális az az ütemezés, amelyben

- a lock-olt adategységeket fel is szabadítják (unlock-kal), továbbá
- ha egy adategység már foglalt - mert egy másik tranzakció tart fenn zárat rajta (ami nem megosztható) -, akkor a tranzakció a zár felszabadulásáig várakozik.

9.2 Problémák a zárakkal

Ha egy T_m tranzakció azért nem tud továbblépni, mert egy olyan A adategység felszabadítására vár, amin egy olyan $T_n \neq T_m$ tranzakció tart fenn zárat, ami viszont azért nem tud továbblépni és a zárat felszabadítani, mert ehhez olyan adategységhez kellene hozzáférnie, amin már T_m, tart fenn zárat, akkor *patt*ról (deadlock) beszélünk.
Patt elképzelhető természetesen kettőnél több tranzakció részvételével is.
Megoldási lehetőségek:

1. A tranzakciók lock-oljanak mindent egyszerre, amire a futásukhoz szükségük lehet. Ha valamely zárat nem kaphatják meg, akkor el se induljanak.
2. Ha egy tranzakció "túl sokáig" várakozik, akkor valószínűsíthető, hogy patthelyzetbe került, ezért abortálandó.
3. Valamilyen egyértelmű sorrendet rendeljünk az adategységekhez és zárat csak ennek a - pl. növekvő - sorrendjében lehessen kérni.
4. Folyamatosan monitorozzuk a zárak elhelyezését, és ha pattot érzékelünk, akkor valamely tranzakciót, amely a pattot okozza, kilőjük.

Ez utóbbihoz szükségünk van egy eljárásra, amivel érzékeljük a patthelyzetet. Egy lehetőség: *várakozási gráf* rajzolása.

Definíció: (várakozási gráf) olyan irányított gráf, ahol a gráf csomópontjai a tranzakciók, egy élt pedig akkor rajzolunk a T_i csomópontból a T_j csomópont felé, ha a T_i tranzakció bármely okból várakoztatja a T_j tranzakciót úgy, hogy az nem tud továbbmenni.

Tétel: adott időpillanatban nincs patt $\Leftrightarrow$ a várakozási gráfban nincs kör (azaz a gráf irányított körmentes gráf, Directed Acyclic Graph, DAG)

Bizonyítás: (indirekt) t. f. h. van kör. Az élek rajzolásának szabálya miatt ez azt jelenti, hogy a körben résztvevő tranzakciók egymást várakoztatják, egyik sem tud továbblépni, ami éppen egy patthelyzetet jelent, ellentmondásban azzal, hogy nincs patt. Tehát ha nincs patt, akkor nem lehet kör a várakozási gráfban.
Másik irányba: ha a gráf DAG, akkor létezik topologikus rendezése, ekkor pedig ez a tranzakcióknak egy olyan sorbarendezése, amelyben a tranzakciók sorban egymás után elindulhatnak anélkül, hogy várakoztatnák egymást. Tehát nincs patt.

Nem a patt az egyetlen lehetőség arra, hogy legális ütemezésben szereplő tranzakció ne tudjon lefutni. Ha egy tranzakció egy adategység lock-olására vár, de közben más tranzakciók mindig lock-olják előtte a kérdéses adategységet, akkor *éhezésről* (starving, livelock) beszélünk. Egy lehetőség az éhezés elkerülésére, ha feljegyezzük a

sikertelen zárkéréseket, és ha egy adategység felszabadul, akkor zárat csak a zárkérések sorrendjében ítélünk oda (FIFO stratégia).

9.3 Ütemezések

Ha a tranzakciók egy rendszerben szigorúan egymás után futnak le úgy, hogy egyidejűleg mindig csak egyetlen tranzakció fut, akkor ez egy *soros ütemezés*. Egy soros ütemezés mindig megvalósítható, problémamentes, amennyiben a tranzakciók külön-külön megvalósíthatók.
Minden egyéb ütemezés neve: *nem soros ütemezés*. Ilyen ütemezés megvalósulhat egyetlen CPU-n (időosztással) vagy több CPU-n is. A nem soros ütemezések lehetnek *sorosíthatóak* vagy *nem sorosíthatóak*.

Definíció: sorosítható ütemezésről beszélünk, ha valamennyi hatása ekvivalens valamely soros ütemezéssel. Egy ütemezés nem sorosítható, ha ilyen ekvivalens soros ütemezés nincs.

Egy ütemezés sorosíthatóságának eldöntése a tranzakciókezelés központi kérdése. Ennek oka a következő:
Feltételezzük, hogy egy tranzakció elvárt, korrekt eredménye az, amit akkor kapunk, ha a tranzakció futása közben más tranzakció nem fut (izolációs elv).
Ez a helyzet a soros ütemezéseknél. Ha nem lehet egy nem soros ütemezésnél biztosítani, hogy u. azt az eredményt produkálja, mintha a tranzakcióit valamely sorrendben sorosan egymás után futtatnánk le, akkor az ütemezést nem tekintjük helyesnek, korrektnek.

Definíció: egy ütemezés pontosan akkor korrekt, ha sorosítható.

Példa soros, sorosítható és nem sorosítható ütemezésekre:

T_1	T_2
read A	
A:=A-10	
write A	
read B	
B:=B+10	
write B	
	read B
	B:=B-20
	write B
	read C
	C:=C+20
	write C

a.) soros ütemezés

T_1	T_2
read A	
	read B
A:=A-10	
	B:=B-20
write A	
	write B
read B	
	read C
B:=B+10	
	C:=C+20
write B	
	write C

b.)sorosítható ütemezés

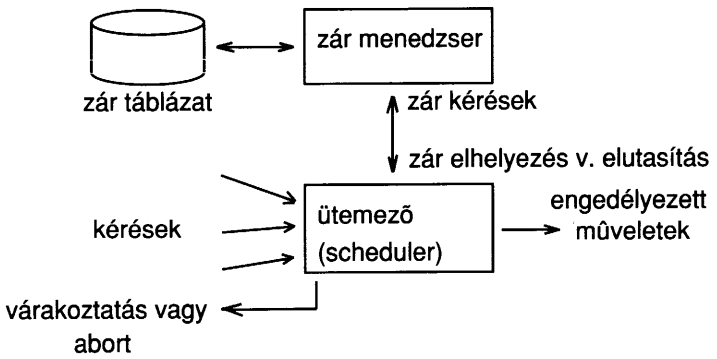
T_1	T_2
read A	
A:=A-10	
	read B
write A	
	B:=B-20
read B	
	write B
B:=B+10	
	read C
write B	
	C:=C+20
	write C

c.) nem sorosítható

Megjegyzések:
1. Általában nehéz eldönteni, hogy egy ütemezés sorosítható-e, hiszen elvileg minden tranzakció minden adategységét meg kell vizsgálni, márpedig k számú tranzakció esetén k! számú soros ütemezés képzelhető el.
2. Kisebb hibát vétünk akkor, ha egy sorosítható ütemezést nem sorosíthatónak minősítünk, mint fordítva.
3. Gyakorlati megoldás:

- olyan szabályok (*protokollok*) megalkotása, amelyeknek a megtartása garantálja a sorosíthatóságot
- az ütemezés analízise alapján a sorosíthatóság eldöntése (pl. sorosíthatósági gráf rajzolása, ld. később).

A konkurrens működés elemei:



9.3.a. ábra: A konkurrens működés elemei

Ütemező: A DBMS azon része, amely az adatelérési igények megítélése felett dönt a sorosíthatóság biztosítása és a pattok feloldása érdekében. Ennek során
- várakoztathatja, ill.
- abortálhatja, újraindíthatja a tranzakciókat.

Az ütemező bonyolultsága jelentősen függ az alkalmazott protokolltól (ld. később). Zárkezelésen alapuló tranzakció menedzsment esetén szorosan együttműködik a zár menedzserrel. A zár menedzser kezeli a zár táblázatot, melynek felépítése:
<adategység > <zártípus> <tranzakció>

9.4 Tranzakció modellek

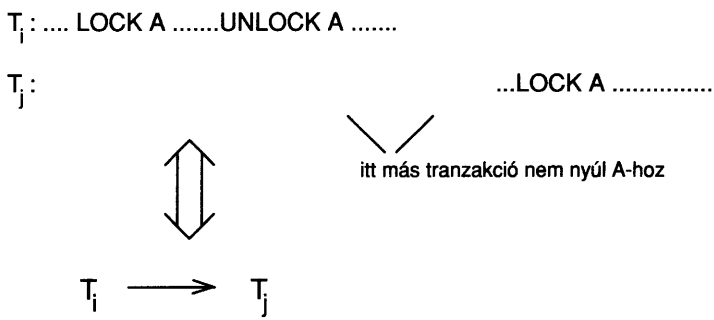
A szabályos tranzakciókat jellemző tulajdonságok gyűjteménye.

Definíció: egyszerű tranzakció modell

- csak egyfajta zár létezik
- egy adatelem zárolása (LOCK) után a tranzakció vége előtt a zár elengedése (UNLOCK) következik
- egy adatelemen egyidőben csak egyetlen zár lehet.

Adott adatelemen zár elhelyezésének következménye, hogy a LOCK ..... UNLOCK műveletek között más tranzakció az adott adathoz semmilyen módon nem férhet hozzá. Ugyanakkor feltételezzük, hogy a zárat elhelyező tranzakció az adatelemet írta is és olvasta is.
Egy adott ütemezés sorosíthatósága eldönthető a sorosítási (precedencia) gráf segítségével.

Definíció: A *sorosítási gráf* olyan irányított gráf, amelynek a csomópontjai a tranzakciók, egy élt pedig akkor rajzolunk a T_i csomópontból a T_j csomópont felé, ha van olyan A adategység, amelyen egy adott S ütemezésben a T_i tranzakció zárat helyezett el, majd a zár felszabadítása után először a T_j tranzakció helyez el zárat A-n.



9.4.a. ábra: Precedenciagráf rajzolása egyszerű tranzakció modellben

A definíció értelme az, hogy, amikor élt rajzolunk a T_i csomópontból a T_j csomópont felé, akkor a T_j tranzakció az A adategységnek olyan értékét olvassa, amit T_i hozott létre. Tehát az ütemezés minden soros ekvivalensében T_i meg kell, hogy előzze T_j-t.

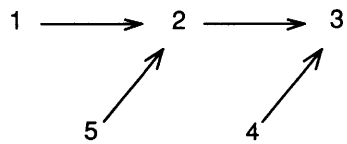
Tétel: Egy S ütemezés sorosítható $\Leftrightarrow$ a sorosítási gráf DAG.

Bizonyítás:
Előre (indirekt): t. f. h. S sorosítható, de tartalmaz kört a sorosítási gráfja. Ekkor a kört alkotó tranzakciók közül egyik sem előzi meg a másikat, tehát egy soros ekvivalensben egyik sincs legelöl, azaz az ütemezés nem sorosítható, ellentmondásban a feltétellel.
Visszafelé: ha a gráf DAG, akkor létezik topologikus rendezése. Belátjuk, hogy ebből a rendezésből legalább egy soros ekvivalens előállítható. U. is a legelöl álló tranzakció (vagy tranzakciók) nem olvashat(nak) olyan adategységet, amin egy másik tranzakció korábban már zárat helyezett el, tehát először lefuthat(nak). Ha a gráfból eltávolítunk egy ilyen tulajdonságú tranzakciót ($T_{első}$) jelölő csomópontot, akkor a maradék gráf továbbra is DAG, tehát továbbra is létezik topologikus rendezése. A korábbi megfontolás továbbra is érvényes, így megadható(k) az(ok) a tranzakció(k), amelyek $T_{első}$ után lefuthatnak, mert vagy egyáltalán nem olvas(nak) olyan adategységet, amin egy másik tranzakció korábban már zárat helyezett el, vagy csak $T_{első}$ által már korábbal lock-olt adategységen helyeznek el zárat. Mindez addig folytatható, amíg a gráf valamennyi csomópontját eltávolítottuk a gráfból: az eltávolítás sorrendje az előbbiek alapján az ütemezésnek egy soros ekvivalense lesz.

Példa: Adott az alábbi ütemezés:

T_1	T_2	T_3	T_4	T_5
				LOCK A
LOCK B				
				UNLOCK A
			LOCK C	
UNLOCK B				
	LOCK A			
	LOCK B			
	UNLOCK A			
		LOCK A		
		UNLOCK A		
			UNLOCK C	
	UNLOCK B			
		LOCK C		
		UNLOCK C		

Sorosítási gráfja:



Soros ekvivalens ütemezések lehetnek:

$T_1T_5T_2T_4T_3$ vagy
$T_5T_1T_2T_4T_3$ vagy
$T_1T_5T_4T_2T_3$ vagy
$T_5T_1T_4T_2T_3$ vagy
$T_4T_1T_5T_2T_3$ vagy
$T_1T_4T_5T_2T_3$ vagy
$T_5T_4T_1T_2T_3$ vagy
$T_4T_5T_1T_2T_3$.

Tehát az ütemezés nyilvánvalóan sorosítható.

A sorosítási gráf segítségével tehát a sorosíthatóság eldöntése visszavezethető kör keresésére egy irányított gráfban. A kör keresés művelete minden zárkérés előtt elvégzendő, így az egyidejűleg futó tranzakciók és az általuk közösen használt adatelemek számának függvényében az ütemező működése jelentősen lassulhat. A gyakorlatban ezért elterjedtebbek azok a megoldások, amikor egy ütemezésben található valamennyi tranzakció adott protokollt követ, amely protokoll mellett a sorosíthatóság vagy automatikusan teljesül, vagy egyszerű módszerekkel biztosítható.

9.4.1 Kétfázisú zárolás (Two-phase locking, 2PL)

Definíció: egy tranzakció a kétfázisú zárolás protokollt követi, ha az első zárfelszabadítást megelőzi valamennyi zárkérés.

Tehát a tranzakció az első fázisban zárakat kér, a második fázisban pedig felszabadítja azokat.

Példa: az előző példában T_3 kivételével valamennyi tranzakció kétfázisú.

Tétel: ha egy legális ütemezés valamennyi tranzakciója a 2PL protokollt követi, akkor az ütemezés sorosítható.

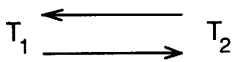
Bizonyítás:
Mivel a sorosíthatóságnak szükséges és elégséges feltétele, hogy a sorosítási gráfban ne legyen kör, elegendő ezt belátni.
Ehhez (indirekt) t. f. h. a csak kétfázisú tranzakciókból álló ütemezés sorosítási gráfjában mégis van kör.
Tekintsük az (egyik) kört, melyet az alábbi tranzakciók alkotnak:
$T_{i1} \rightarrow T_{i2} \rightarrow T_{i3} \rightarrow \ldots\ldots \rightarrow T_{ik} \rightarrow T_{i1}$
A $T_{i1} \rightarrow T_{i2}$ él megléte azt jelenti, hogy az ütemezésben van egy
T_{i1}: LOCK A_{i1}.....UNLOCK A_{i1}..... T_{i2}: LOCK A_{i1} szekvencia.
Hasonlóan, a $T_{i2} \rightarrow T_{i3}$ él megléte azt jelenti, hogy az ütemezésben van egy
T_{i2}: LOCK A_{i2}.....UNLOCK A_{i2}..... T_{i3}: LOCK A_{i2} szekvencia.
.
.
Végül a $T_{ik} \rightarrow T_{i1}$ él megléte azt jelenti, hogy az ütemezésben van egy
T_{ik}: LOCK A_{ik}.....UNLOCK A_{ik}..... T_{i1}: LOCK A_{ik} szekvencia.
Láthatóan így a T_{i1} tranzakció megsérti a kétfázisúság szabályát, hiszen UNLOCK után LOCK következik. Tehát az ellentmondásból arra jutottunk, hogy nem lehet kör a gráfban, az ütemezés sorosítható.

A későbbiek érdekében érdemes a bizonyításnak egy másik módját is megvizsgálni. Ehhez definiáljuk a zárpont fogalmát.

Definíció: zárpont az az időpont, amikor egy kétfázisú protokoll szerinti tranzakció az utolsó zárját is megkapja.

A tétel bizonyításához rendezzük a tranzakciókat a növekvő zárpontjuk szerinti sorrendbe. Beláthatjuk, hogy ez egy soros ekvivalens ütemezés lesz.
T. f. h. az ütemezésben a T_i: LOCK A után következik a T_j: LOCK A művelet (azaz minden soros ekvivalensben T_i meg kell, hogy előzze T_j-t). Ehhez nyilván az kell, hogy T_i felszabadítsa a zárat A-n (T_i: UNLOCK A), mielőtt T_j: LOCK A következne. Viszont T_i is kétfázisú, így meg kell hogy kapja valamennyi zárját T_j: LOCK A előtt. Emiatt T_i biztosan megelőzi T_j-t a zárpontok növekvő sorrendjében, valamennyi soros ekvivalensnek megfelelően. Így a növekvő zárpontok szerinti sorrend nem mond ellent a soros ekvivalense(ke)t meghatározó feltételeknek, azaz egyike a lehetséges soros ekvivalenseknek, az ütemezés sorosítható.

Megjegyzések:
1. ha valamely ütemezés tartalmaz egyetlen nem kétfázisú tranzakciót (T_1) is, akkor létezik olyan tranzakció (T_2), amellyel együtt az ütemezés már nem sorosítható.
   Példa:
   T_1: LOCK A, UNLOCK A.....LOCK B, UNLOCK B
   T_2: LOCK A,B...UNLOCK A,B
   Ennek a (legális!) ütemezésnek a sorosítási gráfja:



mivel kört tartalmaz, az ütemezés nem sorosítható.

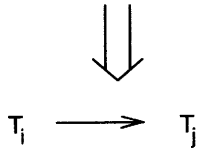
2. A 2PL protokollt követő tranzakciók esetén a kapcsolódó ütemező olyan egyszerű lehet, hogy emiatt a gyakorlatban igen gyakran alkalmazzák a 2PL protokollt.

Definíció: az RLOCK-WLOCK modellben

- kétfajta zár létezik:
  RLOCK (megosztható, puha, shareable, soft) és
  WLOCK (nem megosztható, kemény).
  Ha T: RLOCK A érvényes, akkor más tranzakció is olvashatja A-t, de senki nem írhatja.
  Ha T: WLOCK A érvényes, akkor semmilyen más tranzakció nem fér hozzá A-hoz, sem írásra, sem olvasásra.
- egy adatelem zárolása (RLOCK vagy WLOCK) után a tranzakció vége előtt a zár elengedése (UNLOCK) következik.

A kétféle zár bevezetésétől azt várjuk, hogy kevesebb legyen a várakozás és a tranzakció abort, hiszen több tranzakció is képessé válik egyidőben ugyanazon adategységet olvasni.
Hasonlóan az egyszerű tranzakció modellhez, itt is lehetőség van a sorosítási (precedencia) gráf segítségével eldönteni egy ütemezés sorosíthatóságát.

T_i: ...RLOCK A .....UNLOCK A.....
T_j: ....WLOCK A.....
T_i: ...WLOCK A .....UNLOCK A.....
T_j: ....WLOCK A.....
T_i: ...WLOCK A .....UNLOCK A.....
T_j: ....RLOCK A.....



A fenti ábra mutatja azt a három szekvenciát, amelyek esetén a precedenciagráfban $T_i \rightarrow T_j$ ág rajzolandó.
Az első esetben T_j feltétlenül T_i után kell, hogy következzen, mert T_i-nek T_j-től független A értéket kell olvasnia.
A második esetben T_j-nek azért kell T_i után következnie, mert a szekvencia végén T_j-től függő A értéknek kell maradnia A-ban.
A harmadik esetben T_j-nek olyan A értéket kell olvasnia, amit T_i állított elő, így T_j-nek T_i után kell következnie.
Vegyük észre: valójában a hatékonyságnövekedést az okozza, hogy a

T_i: ...RLOCK A .....UNLOCK A.....

T_j: ....RLOCK A.....

szekvenciák esetén nem kell ágat húzni T_i és T_j között (ha az egyszerű tranzakció modellben gondolkodunk, akkor itt is lenne ág, tehát nagyobb lenne a kör keletkezésének valószínűsége).

Tétel: egy RLOCK-WLOCK modellbeli S ütemezés sorosítható ⇔ a fenti szabályok szerint rajzolt precedenciagráf DAG.

Bizonyítás: analóg az egyszerű tranzakció modell hasonló tételével.

Definíció: egy RLOCK-WLOCK modell szerinti tranzakció kétfázisú, ha minden RLOCK és WLOCK megelőzi az első UNLOCK-ot.

Tétel: ha egy ütemezésben csak kétfázisú, RLOCK-WLOCK modell szerinti tranzakciók vannak, akkor az ütemezés sorosítható.

Bizonyítás: analóg az egyszerű tranzakció modell hasonló tételével.

Az RLOCK-WLOCK bevezetésével elérhető előnyök gondolata alapján további zár-módok is definiálhatók. Ettől nyilván akkor várhatunk további hatékonyságnövekedést, ha az értelmezett zár-módok között minél több olyan van, amely egy adategységen egyidejűleg tartható fenn, azaz összeférhetők, kompatibilisek. A zár-módok közötti összeférhetőséget a *zár kompatibilitási mátrix*ban szokás ábrázolni.
Példa: Az RLOCK-WLOCK modell kompatibilitási mátrixa:

		meglévő zár az adategységen RLOCK	WLOCK
zár kérés	RLOCK	Igen	Nem
	WLOCK	Nem	Nem

Egy mátrix elem "igen", ha egy, az adatelemen lévő adott típusú zár mellett az új zárkérés is elfogadható, és "nem" akkor, ha az új zárkérés nem teljesíthető.
További példa: vezessük be az inkrementális zár fogalmát! INCR A akkor helyezendő el A-n, ha úgy növeljük eggyel A értékét, hogy közben nem olvassuk ki azt. Két ilyen zárkérés felcserélhető, tehát kompatibilis egymással. Kompatibilitási mátrixa:

	RLOCK	WLOCK	INCR
RLOCK	I	N	N
WLOCK	N	N	N
INCR	N	N	I

Ha ismerjük az alkalmazott zárak kompatibilitási mátrixát, akkor segítségével könnyen megrajzolhatjuk a sorosítási gráfot is: $T_1 \rightarrow T_2$ élt húzunk akkor, ha T_1 valamely A adategységet i módban zárolt, majd elengedett és utána először T_2 zárolja j módban, és a zár kompatibilitási mátrixban a_{ji}=N.

9.5 Zárak hierarchikus adategységeken

Mindeddig az említett zárkezelési mechanizmusok tökéletesen függetlenek voltak attól, hogy az adategységek milyen struktúrába szervezettek, ill. szervezettek-e egyáltalán. Látni fogjuk, hogy ennek a járulékos információnak a felhasználása a zárkezelés hatékonyságát tovább növelheti.
Kitüntetett a jelentősége annak az esetnek, ha az adategységek valamely hierarchiába szervezettek. Pl.:
1. hierarchikus adatbázis rekordjai
2. B-fa elemei
3. egymásba ágyazott adatelemek: ezek közül a legfontosabb a relációs adatbázis-reláció-blokk-tuple hierarchia.

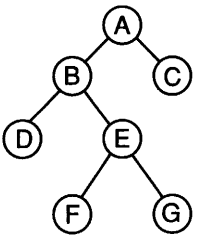
Kihasználva a hierarchikus szerkezetet, lehetőségünk van arra, hogy egy adategység zárolása esetén minden, a hierarchiában alacsonyabban lévő adategységet is egyidejűleg zároljunk. Jól kihasználható ez egymásba ágyazott adatelemek esetén. Pl. a relációs adatbázisoknál, ha egy tábla valamennyi sorát zárolni kell, akkor ez soronként igen költséges lehet, de - az előbbiek szerint - megoldható a táblára helyezett egyetlen zárral is.
Más hierarchiák esetén, pl. B-fáknál nem feltétlenül célravezető az előbbi stratégia. Egy jó példa erre a *fa protokoll*, míg az előbbi megközelítésre a *figyelmeztető protokoll*.

9.5.1 A fa protokoll

Az egyszerű tranzakció modellt követjük és egy csomópont zárolása nem jelenti a gyerekek zárolását is.
Szabályai:
1. egy tranzakció az első LOCK-ot akárhová teheti
2. további LOCK-ok csak akkor helyezhetők el, ha az adategység szülőjére u. az a tranzakció már rakott zárat
3. egyazon tranzakció kétszer ugyanazt az adategységet nem zárolhatja.

Vegyük észre, hogy az UNLOCK-ra nincs előírás, aminek következménye, hogy a fa protokoll nem feltétlenül kétfázisú, mint pl. a következő példában:

LOCK A
LOCK B
UNLOCK A
LOCK E



Ez a szekvencia a valóságban is gyakran előfordul, amikor egy tranzakció insert műveletet hajt végre egy B-fán. Ha B egy olyan csomópont, amiben van hely egy újabb indexrekord számára, akkor a fa átstruktúrálódása az insert következtében a B csomópont szüleit már nem érintheti, így a rajtuk lévő zár felszabadítható, lehetővé téve, hogy más tranzakció zárolja a C csomóponthoz tartozó részfát.

Tétel: a fa protokollnak eleget tevő **legális** ütemezések sorosíthatók.

Bizonyítás: (vázlat)

1. Rendeljük hozzá minden tranzakcióhoz egy irányított gráf egy csúcsát.
2. Ebben a gráfban adott szabályok szerint rajzoljunk éleket.
3. Bebizonyítjuk, hogy az így rajzolt gráf DAG.
4. Bebizonyítjuk, hogy a gráf minden topologikus rendezése az ütemezésnek egy soros ekvivalense.

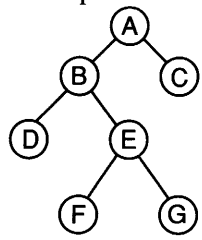
Fentiekből részletesebben csak a 2. pontot tárgyaljuk:
Legyen E(T) az a csúcs (adategység), amit a T tranzakció elsőnek zárol.
Ha $E(T_i)$ és $E(T_j)$ közül egyik sem őse a másiknak, akkor nem rajzolunk élet T_i és T_j között, hiszen a protokoll garantálja, hogy ekkor T_i és T_j sohasem fog közös csúcsot (adategységet) zárolni.
T. f. h. tehát, hogy $E(T_i)$ őse $E(T_j)$-nek.

- Ha T_i zárolja először $E(T_j)$-t, mielőtt T_j zárolná, akkor egy $T_i \rightarrow T_j$ élet rajzolunk a gráfba (ugyanis a közös, $E(T_j)$ alatti adatokat ekkor T_i fogja tudni először zárolni).
- Ha T_j zárolja először $E(T_j)$-t, mielőtt T_i zárolná, akkor pedig egy $T_j \rightarrow T_i$ élet rajzolunk a gráfba.

9.5.2 A figyelmeztető protokoll

Az egyszerű tranzakció modellt követjük és egy csomópont zárolása a gyerek csomópontok zárolását is jelenti (*implicit lock*).
Ez utóbbi feltételezés azonban azt is eredményezheti, hogy két tranzakció tart fenn egyidejűleg zárat ugyanazon adategységen. Tekintsük az alábbi példát:

T1: LOCK A
T2: LOCK E



Ezután F-en és G-n T_1 és T_2 is (implicit) zárat tart fenn.
A jelenség neve *zárkonfliktus*, amit alkalmas szabályok megtartásával el kell kerülni.
A figyelmeztető protokoll zárműveletei:

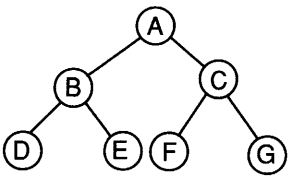
LOCK A zárolja A-t és valamennyi gyerek csomópontot is. Két különböző tranzakció nem tarthat fenn egyidejűleg zárat ugyanazon adategységen.
WARN A A-ra figyelmeztetést (warning) rak. Ekkor A-t más tranzakció nem zárolhatja.
UNLOCK A eltávolít zárat, figyelmeztetést A-ról.

Szabályai: mint az egyszerű tranzakció modellben, továbbá

1. egy tranzakció első művelete kötelezően LOCK gyökér vagy WARN gyökér
2. LOCK A vagy WARN A akkor helyezhető el, ha A szülőjén már van WARN
3. UNLOCK A akkor lehetséges, ha A gyerekein már nincs LOCK vagy WARN
4. kétfázisú: az első UNLOCK után nem következhet LOCK vagy WARN.

Példa:

T_1	T_2	T_3
WARN A		
	WARN A	
		WARN A
WARN B		
	LOCK C	
LOCK D		
	UNLOCK C	
UNLOCK D		
	UNLOCK A	
UNLOCK B		
		LOCK B
		WARN C
		LOCK F
UNLOCK A		
		UNLOCK B
		UNLOCK F
		UNLOCK C
		UNLOCK A



Az ütemezéshez tartozó hierarchia

Tétel: a figyelmeztető protokollt követő legális ütemezések konfliktusmentesek (1) és sorosíthatók (2).

Bizonyítás:

(1) A protokoll 1-3 szabályai biztosítják, hogy bármely tranzakció csak akkor tehessen zárat egy adategységre, ha figyelmeztetés van annak valamennyi ősén. Emiatt egyidejűleg más tranzakció nem tehet zárat az adategységnek egyetlen ősére sem, tehát nem alakulhat ki zárkonfliktus.

(2) Megmutatjuk, hogy az adott R ütemezés átalakítható egy olyan ekvivalens S ütemezésbe, amely az egyszerű tranzakció modellnek felel meg és minden adategységet explicit módon zárolunk.

S-t tehát úgy állítjuk elő, hogy

1. eltávolítjuk az összes figyelmeztetést és UNLOCK párjait R-ből,
2. LOCK X esetén explicit zárat helyezzünk X valamennyi gyerekére is,
3. UNLOCK X esetén eltávolítjuk a zárat X valamennyi gyerekéről is.

Ezekután S legális, mert R is legális volt és az átalakítással semmi olyat nem tettünk, ami miatt illegálissá válhatna, továbbá kétfázisú, mert R is kétfázisú volt és ez az átalakítás során a kétfázisú tulajdonság megmaradt. Ezek elégséges feltételek S sorosíthatóságához.

Megjegyzések:

1. A WARN-LOCK kompatibilitási mátrix

	LOCK	WARN
LOCK	N	N
WARN	N	I

formálisan azonos a RLOCK-WLOCK kompatibilitási mátrixszal. Ez azonban nem jelenti azt, hogy a WARN azonos lenne egy RLOCK-kal, hiszen más a szemantikájuk: a WARN hierarchikus adategységeken adott szabályok szerint helyezhető csak el.

2. Az RLOCK-WLOCK modellnek megfelelően értelmezhetünk hierarchikus adategységek esetén RWARN-t és WWARN-t is, ami a tranzakciós teljesítmény további növekedését eredményezheti.

9.6 Tranzakcióhibák kezelése

Mindeddig nem foglalkoztunk azokkal a problémákkal, amelyek akkor lépnek fel, ha egy tranzakció nem fut le teljesen, valamely ok miatt idő előtt befejeződik. Ahhoz, hogy hatékonyan kezelni tudjuk ezeket a hibákat, először gyűjtsük össze a lehetséges okokat.

1. tranzakció félbeszakad (felhasználó megszakítja, 0-val osztás, nem fér hozzá valamely adategységhez)
2. az ütemező patt miatt kilövi
3. az ütemező amiatt lövi ki, mert a sorosíthatóság így biztosítható
4. valamilyen *rendszerhiba* lép fel, emiatt az adatbáziskezelő hibásan kezd működni (szoftver és/vagy hardver eredetű hibák egyaránt tartozhatnak ide)
5. a háttértár tartalma (is) megsérül (*média hiba*).

Az 1-2-3 hibákban közös az, hogy a memória- és a háttértár-struktúrák érintetlenek, ellenben a 4. ponthoz tartozó hibákkal. Rendszerhibáról beszélünk tehát akkor, ha a felejtő tár tartalma, a memória sérül. A legrosszabb, ha a háttértár tartalma is károsodik, ez persze együtt is járhat valamely memóriastruktúra sérülésével is. Ebben a sorrendben egyre bonyolultabb a reguláris működés biztosítása, az adatbázis konzisztenciájának fenntartása, az adatvesztés elkerülése. A legsúlyosabb esetben már csak az segít, ha biztonsági másolatunk van az adatbázisról és valahogy rekonstruálni lehet, hogy mi történt az adatbázissal a legutóbbi mentés óta.
Első lépésben csak azokkal a kérdésekkel foglalkozunk, hogy mi a teendő az 1-2-3 hibaokok esetén (ú.n. *tranzakcióhibák*).

Definíció: *konzisztens állapot* az adatbázisnak olyan állapota, amely csak teljesen lefutott tranzakciók hatását tükrözi.

Ez a definíció akkor jogos, ha feltételezhetjük, hogy az egyes tranzakciók az adatbázist egyik konzisztens állapotból egy másik konzisztens állapotba viszik át. Ehhez természetesen egyéb módszerek kellenek, amelyek az egyes tranzakcióknak ezt a tulajdonságát biztosítják.

Definíció: *commit pont* (*kész pont*) az az időpillanat, amikor egy tranzakció futása során már minden befejeződött, ami az 1-2-3 okok miatti abortját eredményezheti.

Gyakorlatilag ez azt jelenti, hogy a tranzakció már minden számítást elvégzett, és minden zárat megkapott. Ekkor egy COMMIT utasítás szokott következni, de ez az időpont még nem feltétlenül azonos azzal, amikor minden eredmény már véglegesítve is lett. Ehhez további műveletek is szükségesek lehetnek (ld. később). Ha viszont a tranzakció nem jutott el eddig a pontig, és ekkor *abort* következik be, akkor a tranzakciónak minden esetleges hatását törölni kell az adatbázisból.

Definíció: a "piszkos" adat olyan adat, amit az előtt írt valamely tranzakció az adatbázisba, mielőtt commit-tált volna.

A piszkos adat az adatbázisból mihamarabb eltávolítandó. Azonban előfordulhat, hogy egy másik tranzakció olvassa a piszkos adatot, és ez a tranzakció már sikeres.

Ennek ellenére, eredménye nem feltétlenül tekinthető helyesnek, így az adatbázisból ez is eltávolítandó. A jelenség iteráltan is előfordulhat, ekkor a neve *lavina*.
Példa:

T_1	T_2
LOCK A	
READ A	
A=A-1	
WRITE A	
LOCK B	
UNLOCK A	
	LOCK A
	LOCK C
	READ A
	C=A*2
READ B	
	WRITE C
	COMMIT
	UNLOCK A
B=B/A	
ABORT	
	UNLOCK C

T_1 abortja után az alábbi tevékenységek végzendők:

1. T_1 által B-n fenntartott zárat az adatbáziskezelő rendszernek kell eltávolítani.
2. A eredeti értéke helyreállítandó, ehhez szükség van A régi értékére is.
3. T_2 rossz A-t olvasott, ezért T_2 teljes hatása törlendő az adatbázisból (C régi értéke is helyreállítandó).
4. Ezután T_2 újra lefuttatandó és minden más tranzakció is, amely esetleg még olvasta időközben A értékét (*redo*).

Láthatóan egy abortált tranzakció így sok járulékos költséget eredményezhet.
Kezelése:

- nem commit-tált tranzakciótól nem olvasunk
- ha megengedjük, akkor minden piszkos értéket olvasott tranzakció hatása törlendő az adatbázisból (*undo*)
- lehetetlenné tesszük, hogy a tranzakciók piszkos adatot olvassanak (ld. a szigorú protokollokat alább)

9.6.1 Szigorú kétfázisú protokoll (strict 2PL)

Igen gyakori. Egy tranzakció ezt a protokollt követi, ha kétfázisú, továbbá

1. nem ír az adatbázisba, amíg a kész pontját el nem érte,
2. a zárait csak az adatbázisba írás után engedi el.

Azaz a COMMIT, adatbázisba írás, zárak elengedése pontosan ebben a sorrendben következik.
Vegyük észre, hogy az 1. szabály ahhoz kell, hogy ne kelljen az adatbázist helyreállítani, ha egy tranzakció abortál (csak redo kell). A 2. szabály biztosítja valójában azt, hogy piszkos adatot ne lehessen olvasni - hiszen commit után nem lehet abort -, tehát a lavinákat elkerüljük.
Mindez természetesen csak akkor igaz, ha nem kell rendszerhibákkal is számolni, amelyek az írási folyamat megzavarásával eredményezhetik, hogy piszkos adat kerül

az adatbázisba. Ez ellen azonban más módszerekkel kell védekezni (naplózás, ld. később).
Ezután az alábbi tétel állítása csaknem triviális:

Tétel: A szigorú kétfázisú protokollt követő tranzakciókból álló ütemezések sorosíthatók és lavinamentesek.

Bizonyítás: az ütemezés sorosítható, mert kétfázisú; továbbá lavinamentes, mert nincs lehetőség piszkos adat olvasására.

9.6.2 Agresszív és konzervatív protokollok

A szigorú kétfázisú protokollok is valójában egy családot alkotnak, mivel számos alternatíva létezik még, amely az ütemezésekre hatással lehet (pl. egy tranzakció valamennyi zárját előre kéri, és csak akkor indul, ha már valamennyit megkapta).
Ezen alternatívák elsősorban annak alapján értékelhetők, hogy mennyire segítik elő az adatbáziskezelő rendszer "hatékony" konkurrens működését.
"Hatékonyság" mércéje lehet pl.:

- egy adott tranzakció minél hamarabb fusson le
- adott idő alatt minél több tranzakció fusson le sikeresen (=*tranzakciós teljesítmény*).

Ezen kritériumok ellentmondásosak, adott esetben mindkettő lehet fontosabb a másiknál, mégis, gyakran a tranzakciós teljesítményt kívánjuk maximalizálni az ütemező és a protokoll alkalmas megválasztásával.
Hogyan befolyásolják ezek a tranzakciós teljesítményt?

a) számos műveletet igényel a zár táblák kezelése, a zárakra várakozó sorok karbantartása, pattok érzékelése és feloldása, annak eldöntése, hogy a zár odaítélhető-e vagy sem
b) a később abortált tranzakciók által végzett műveletek mind kárbavesznek csakúgy, mint az általuk igényelt és fel nem oldott zárak felkutatása és felszabadítása
c) nem szigorú 2PL esetén az adatbázis helyreállítása (undo), a lavinaeffektus felszámolása szintén csökkenti a tranzakciós teljesítményt.

Definíció: egy protokoll *agresszív*, ha megpróbál olyan gyorsan lefutni, amennyire csak lehetséges, nem törődve azzal, hogy ez esetleg abort-hoz is vezethet (keveset foglalkozik a)-val).

Definíció: egy protokoll *konzervatív*, ha megkísérli elkerülni az olyan tranzakciók futtatását, amelyek nem biztos, hogy eredményesek lesznek (sokat fordít a)-ra).

Példa: egy nagyon konzervatív protokollt követő tranzakció

1. szigorú 2PL
2. az összes zárat előre kéri és csak akkor indul, ha már az összeset megkapta
3. csak akkor kapja meg a zárjait, ha már előtte senkinek nem kellenek a sorban.

Vegyük észre: 1. miatt sorosítható és lavinamentes, 2. miatt pattmentes, 3. miatt éhezésmentes. Ennek ára, hogy
2. és 3. miatt nagy lehet a késleltetés, amíg egyáltalán a tranzakció elindulhat,
2. és 1. miatt más tranzakciókat is szükségtelenül késleltet,
2. miatt előre ismerni kell(ene) valamennyi zárat.
Másik véglet: nagyon agresszív egy protokoll, ha a zárakat közvetlenül írás vagy olvasás előtt kéri. Amennyiben a zárakat csak azután engedi el, hogy valamennyit megkapta, akkor egyszerűen sorosítható, ellenben a sorosíthatóság biztosítása is külön

probléma. Patt és éhezés is lehetséges, viszont a tranzakciók igen gyorsan lefuthatnak és a többi tranzakciót is kevéssé fogják vissza.

Választási szempont: ha kevés a tranzakciók által közösen használt adat, akkor kicsi a nem sorosítható ütemezés, patt és éhezés veszélye, ilyenkor az agresszív protokollok hatékonyabbak.

9.7 Helyreállítás rendszerhibák és médiahibák után

Eddig csak arról volt szó, hogy mi a teendő, ha egyes tranzakciók aborttal fejezik be futásukat. A továbbiakban az előző szakaszban 4-5. pont alatt szerepelt rendszerhibák és médiahibák kezeléséről lesz szó.
A rendszerhibák elleni védekezés általános módszere a *naplózás* (journal, log). A naplózásnak számos módja lehet, itt csak a leggyakoribbakra térünk ki.
A napló a mi értelmezésünk szerint az adatbázison végrehajtott változások története.
Általában az alábbi szerkezetű rekordokat tartalmazza:
(<Tranzakció azonosító>, begin)
(<Tranzakció azonosító>, commit)
(<Tranzakció azonosító>, abort)
(<Tranzakció azonosító>, <adategység>, <új érték>, <régi érték>)
Ha tudható, hogy undo műveleteket nem kell a napló alapján végezni, akkor elég az adategység új értékének naplózása. Néha még ez is túl nagy méretű, ekkor esetleg az kerülhet a naplóba, hogy hogyan kell az adategység új értékét előállítani.
Alapszabály, hogy a naplót azelőtt írjuk, mielőtt a naplózott műveletre sor kerülne (kivétel: abort).

9.7.1 Hatékonysági kérdések

A helyreállítás képességének igénye miatt a napló stabil tárban (háttértáron, diszken) tárolandó. Ennek költsége - mint ismert - a háttértárra írt napló-blokkok számával arányos. A gyakorlatban a napló-blokkokat valamilyen lapozási stratégiával kezelik. Számos napló oldal lehet egyidejűleg a memóriában és egy lapmenedzser gondoskodik a háttértárra kiírásról, meghatározott stratégia és feltételek szerint. Tipikus, hogy ugyanez igaz az adatbázis blokkokra is.
Ha változtatunk az adatbázisunkon, a változás elveszhet egy rendszerösszeomlás esetén, ha a naplót vagy a megváltozott adatbázis oldalakat nem írjuk ki a stabil tárba. Hatékonyabb, ha a naplót írjuk, mert a napló tömörebben tartalmazza a változásokat, mint maguk az adatbázis oldalak. Ekkor tehát kevesebb blokkműveletet kell végeznünk. Ugyanakkor szabály, hogy egy tranzakció vége után a napló akkor is kiírandó a háttértárra, ha a lapozási stratégia ezt még nem követelné meg.
Az adatbázis oldalakat lehet a naplótól függetlenül is, egy másik lapozási stratégia szerint cserélni.

9.7.2 A redo protokoll

Nevét onnan kapta, hogy olyan tranzakciókezelést valósít meg, amely rendszerhiba (és tranzakcióhiba) után szükségtelenné teszi az undo műveletet, csak redo kell.
A protokoll két része a *redo naplózás* és a *redo helyreállítás*.
A redo naplózás a szigorú 2PL finomítása.

Lépései:
1. (T, begin) naplóba
2. (T, A, <A új értéke>) naplóba, ha T megváltoztatja valamely A adategység értékét
3. (T, commit) naplóba, ha T elérte a commit pontját
4. a napló mindazon oldalainak stabil tárba írása, amikkel ez még nem történt meg
5. az "A" értékeknek a tényleges írása az adatbázisba
6. zárak elengedése

Megjegyzések:
Az a tranzakció lett véglegesítve, amelyik eljutott a 4. pont végéig, hiszen a napló alapján az adatbázison végzett műveletei már bármikor megismételhetők. A 4. pont végéig el nem jutott tranzakciók hatása pedig részben vagy egészben elveszhet.
Az 5. pontban az adatbázisoldalak írásához az érintett blokkokat be kell hozni a memóriába, az írást elvégezni, de a megváltozott oldalak azonnali visszaírása a háttértárra már opcionális.
A 6. pont után férnek hozzá csak más tranzakciók a T által megváltoztatott adatokhoz, így nincs lavinaeffektus sem.

A redo helyreállítás
Az adatbázist egy konzisztens állapotba viszi át a helyreállítás végére.
Lépései:
1. valamennyi zár felszabadítása
2. napló vizsgálata visszafelé: feljegyezzük azon tranzakciókat, amelyekre találunk (T, commit) bejegyzést a naplóban
3. addig megyünk visszafelé a naplóban, ameddig nem találunk egy konzisztens állapotot (ld. később)
4. a 2.pontnak megfelelő tranzakciókra vonatkozó bejegyzések segítségével az adatbázisban található értékeket felülírjuk az újakkal

A redo helyreállítás eredményeként a 2. pontnak megfelelő tranzakciók hatása megmarad, a többié elvész és az adatbázis egy újabb konzisztens állapotba kerül.
Ha a redo helyreállítás elszáll, akkor egyszerűen megismételendő, hiszen a 4. pontban végzett műveletek hatása az adatbázisra idempotens.

Példa egy redo protokoll szerinti tranzakcióra:

tranzakció	napló	megjegyzés
	(T, begin)	
LOCK A		
LOCK B		Ha itt jön a rendszerhiba, akkor
	(T, A, v)	a helyreállítás során A és B értéke
	(T, B,w)	változatlan marad.
	(T, commit)	Ezután a napló stabil tárba írandó.
WRITE A		Ha ezután jön a rendszerhiba,
WRITE B		akkor később már bármikor minden
UNLOCK A		helyreállítható.
UNLOCK B		

9.7.3 Ellenőrzési pontok (checkpointing)

A redo helyreállításnál könnyen előfordulhat, hogy igen régi időpontra kell a naplóban visszafelé menni ahhoz, hogy alkalmas időpontot találjunk a helyreállítás megkezdésére. Ezen a problémán segítenek az ellenőrzési pontok, amikor kikényszerítik az adatbázisnak egy konzisztens állapotát:

1. Ideiglenesen megtiltjuk új tranzakció indítását és megvárjuk, amíg minden tranzakció befejeződik vagy abortál
2. Megkeressük azokat a memóriablokkokat, amelyek módosultak, de még nem kerültek a háttértárba kiírásra
3. Ezeket a blokkokat visszaírjuk a háttértárra
4. Ellenőrzési pont tényét naplózzuk
5. Naplót kiírjuk a háttértárra.

Előnyök:

1. redo helyreállításnál csak a legutóbbi ellenőrzési pontig kell a naplóban visszamenni,
2. emiatt a napló korábbi részei eldobhatók (amennyiben más érv ez ellen nem szól),
3. csökkenti a lavinák számát,

Hátrány:

csökkenti a tranzakciós teljesítményt (többlet írások a háttértárra, tranzakció indítások késleltetése)

Ütemezése:

- adott idő eltelte után
- adott számú tranzakció lefutása után
- kombinálva az előző kettőt.

Mivel az ellenőrzési pontokra elsősorban a helyreállításkor van szükség, ez pedig ritka eseménynek számít, ezért ellenőrzési pontokat is viszonylag ritkán iktatnak be. Következmény: a helyreállítás tovább tart.

9.7.4 Médiahibák elleni védekezés

- Rendszeres mentések (backup). Modern adatbáziskezelő rendszerek ezt úgy hajtják végre, hogy egy pillanatnyi üzemszünetet sem okoz.
- Az adatbázis file-ok duplikálása lehetőleg egy másik fizikai eszközön, néha más földrajzi helyen (→elosztott adatbázisok)

Érdemes a napló file-okat is védeni: jó gyakorlat, ha az adatbázis és a napló más-más fizikai eszközön található. Szóba jön itt is a duplikálás.

9.8 Időbélyeges tranzakciókezelés

A tranzakciók sorosíthatósága biztosításának egy másik módja. Akkor praktikus elsősorban, ha a tranzakciók között kevés a potenciális sorosítási konfliktus.
Az időbélyeges tranzakciókezelés során az adategységekhez egy (vagy néhány) járulékos adatot rendelünk hozzá (*időbélyeg*). Segítségével eldönthető, hogy egy tranzakció adott adategységre vonatkozó kérése sérti-e a sorosíthatóságot. Ha igen, akkor a tranzakciót abortálni kell. Alapvetően agresszív protokoll.

Definíció: az időbélyeg olyan érték, amelyet minden tranzakcióhoz szigorú egyediséget biztosítva rendelünk hozzá és amely arányos (legegyszerűbb esetben azonos) a tranzakció kezdőidejével. Jele: t(Tranzakció).

Figyelembe véve, hogy
1. az időbélyegek a tranzakcióknak egy egyértelmű sorrendjét határozzák meg, továbbá
2. képzelhetjük úgy, mintha a tranzakciók a kezdőidejükben (csaknem) zérus idő alatt futnának le

a kezdőidők növekvő sorrendjébe állítva a tranzakciókat ez *lehet* egy soros ekvivalens ütemezés.

Ha az időbélyeges ütemező tehát gondoskodik arról, hogy csak olyan műveleteket engedélyezzen, amely a tranzakciók növekvő időbélyegével ekvivalens soros ütemezés hatásaival egyezik meg, akkor a tranzakciók indulási sorrendje valóban egy soros ekvivalens ütemezés lesz.

(Vö: soros ekvivalens ütemezés kétfázisú zárkezeléssel: a zárpontok szerinti növekvő sorrend a soros ekvivalens (legalábbis az egyik)

soros ekvivalens ütemezés időbélyegesen: a $t(T_{i1}) < t(T_{i2}) <....< t(T_{in})$-nek megfelelő $T_{i1}, T_{i2},....,T_{in}$)

Az időbélyeges ütemező működése:
1. megvizsgálja minden írás-olvasás előtt a hivatkozott adategység időbélyegét
2. ha ez a sorosítási szabályokkal összhangban van (ld. később), akkor az adategység időbélyegét felülírja a műveletet végrehajtó tranzakció időbélyegével, ha nincs összhangban, akkor pedig abortálja a tranzakciót.

Megjegyzés: elképzelhető, hogy egy ütemezés sorosítható
- időbélyegesen, de zárakkal nem (Példa 1)
- időbélyegesen is és zárakkal is (pl. minden olyan ütemezés, amelyben a tranzakciók nem használnak közös adatokat)
- időbélyegesen nem, de zárakkal igen (Példa 2) valamint
- időbélyegesen sem és zárakkal sem.

Példa 1:

T1	T2	T3
READ A		
	READ A	
		READ D
		WRITE D
		WRITE A
	READ C	
WRITE B		
	WRITE B	

Soros ekvivalens sorrend: T_1, T_2, T_3
Időbélyegesen: T_1, T_2, T_3
(kétfázisú) zárakkal: $ZP(T_1)<ZP(T_2)$ a B adategység miatt (ZP: zárpont), továbbá $ZP(T_3)<ZP(T_1)$, így a T_3, T_1, T_2 sorrendben érik el a zárpontjaikat.

Példa 2:

T1	T2
	READ B
READ A	
WRITE C	
	WRITE C

Soros ekvivalens sorrend: T_1, T_2

Időbélyegesen:T_2, T_1
(kétfázisú) zárakkal: $ZP(T_1)<ZP(T_2)$, így a T_1, T_2 sorrendben érik el a zárpontjaikat.

Tanulság: sem a zárakkal, sem az időbélyegekkel való sorosítás nem jobb egyértelműen a másiknál.
Időbélyegek kiosztása során az egyértelműség kritikus.
Egyprocesszoros környezetben a tranzakció indulásakor a rendszeróra által mutatott érték jó alap az időbélyeg meghatározásához.
Többprocesszoros (akár elosztott) környezetben ehhez még a processzor (node) azonosítója is hozzáveendő (ld. elosztott időbélyeges tranzakciókezelés).
A sorosítási feltételek megsértése többféle modell alapján is detektálható. Itt is létezik:

- egyszerű modell
- read/write modell
- és további műveletek is megkülönböztethetők.

9.8.1 Időbélyeges tranzakciókezelés R/W modellben

Definíciók:
R(A): az adategység olvasási ideje, annak a tranzakciónak az időbélyege, amely utoljára olvasta az A adategységet.
W(A): az adategység írási ideje, annak a tranzakciónak az időbélyege, amely utoljára írta az A adategységet.

Az alábbi táblázat alapján eldönthető, hogy egy tranzakció műveleti igénye jogos volt-e vagy sem, azaz az időbélyeges soros ekvivalensnek megfelel-e.

	T olvasni akar	T írni akar
t(T)<R(A) t(T)<W(A)	abort T (1)	abort T (2)
t(T)<R(A) t(T)>W(A)	T olvassa A-t, de R(A) nem változik (3)	abort T (4)
t(T)>R(A) t(T)<W(A)	abort T (5)	abort T (6)
t(T)>R(A) t(T)>W(A)	T olvassa A-t és R(A):=t(T)	T írja A-t és W(A):=t(T)

Magyarázat:
(1): egy később indult tranzakció már írta A-t, tehát nem olvashatjuk
(2): egy később indult tranzakció már olvasta A-t, tehát nem írhatjuk
(3): egy később indult tranzakció már olvasta A-t, ettől még T is kiolvashatja, de az időbélyeget a későbbi értéken kell hagyni
(4): u. az, mint (2)
(5): u. az, mint (1)
(6): egy később indult tranzakció már írta A-t, így T nem írhatja felül, azaz T-nek abortálnia kell.
Megjegyzések:
Az időbélyeg tesztelése és maga a művelet oszthatatlan kell, hogy legyen.
A fenti táblázatban a t(T)=W(A) vagy t(T)=R(A) teljesülése nyilván akkor következhet be, ha maga T írta/olvasta legutóbb A-t.
Összefoglalva:

1. abort T, ha T olvasni akar és t(T)<W(A) vagy T írni akar és t(T)<R(A) vagy t(T)< W(A)
2. a READ művelet elvégzendő, ha t(T)≥W(A)

3. WRITE elvégzendő, ha t(T)≥R(A) és t(T)≥W(A).

(2.-3. esetén egyidejűleg az időbélyegek is beállítandók, kivéve, ha T olvas és t(T)< R(A))

Példa:

T_1	T_2	időbélyegek:
$t(T_1)=50$	$t(T_2)=60$	R(A)=0 W(A)=0
READ A		$t(T_1)>R(A)$, $t(T_1)>W(A)$ tehát olvashat és R(A):=50
	READ A	$t(T_2)>R(A)$, $t(T_2)>W(A)$ tehát olvashat és R(A):=60
A:=A+1		
	A:=A+1	
	WRITE A	$t(T_2)=R(A)$, $t(T_2)>W(A)$ tehát írhat és W(A):=60
WRITE A		$t(T_1)<R(A)$, $t(T_1)<W(A)$ tehát abort T_1.
abort T_1		

9.8.2 Időbélyegek kezelése

Zár alapú tranzakció kezelésnél praktikus megközelítés, ha
-külön tároljuk a zárinformációt az adategységektől (zártábla), és
-feltételezzük az adategységek default "nem zárolt" állapotát. Tehát a zártáblában csak azokról az adategységekről tárolunk információt, amelyeken van zár.
Analóg módon: időbélyeges esetben az adategységek default időbélyege lehet -∞, és csak az ettől eltérő értékeket kell explicit módon tárolni. Ezek az értékek megjelenhetnek közösen egy időbélyeg-táblában, ami lehetőség szerint memóriában tárolandó. A tábla nem nő végtelenségig, mert a táblából kitakaríthatók azok az időbélyegek, amelyek kisebbek, mint a futó tranzakciók legkisebb időbélyegének az értéke.

9.8.3 Tranzakcióhibák és az időbélyegek

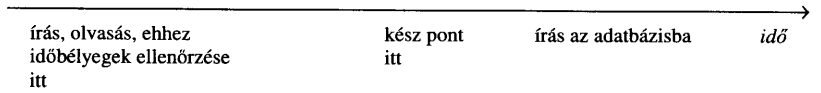
Elképzelhető, hogy T_2 olvas T_1 által előállított értéket, majd később T_1 abortál bármely (korábban 1-2-3 pontok alatt hivatkozott) ok miatt. Ez a piszkos adat olvasásának esete, amikor tehát lavinaveszéllyel kell számolni.
A megoldás:

- elfogadjuk a lavinákat, hiszen az időbélyeges tranzakciókezelést tipikusan olyan környezetben használjuk, ahol kevés az abort, tehát a lavina még kevesebb, vagy
- megakadályozzuk a piszkos adat olvasását pl. azzal, hogy nem írunk az adatbázisba, amíg a tranzakció el nem érte a commit pontját (→*időbélyeges szigorú protokoll*)

Lépései **lennének**:

1. módosítások csak munkaterületen elvégezve
2. tranzakció eléri a kész pontját

3. írások véglegesítése az adatbázison
Az írásokkal azonban baj van:



Az időbélyeg ellenőrzése a kész pont előtt kell, hogy történjen, hiszen utána már a tranzakció akkor sem abortálhat, ha az időbélyegek vizsgálatából ez következne. Így azután egy írandó adatelem időbélyegének ellenőrzése és tényleges írása között jelentős idő telhet el, ami problémát okozhat:
T.f.h. t(T)=100 és T írni akarja A-t. Legyen R(A)=W(A)=60 az írás előtt, tehát T írhat (a munkaterületen). Ha jönne egy T_1 tranzakció a kész pont előtt, ahol $t(T_1)=80$ és olvasni akar, akkor

- olvashat (helytelenül), amennyiben nem állítjuk be A időbélyegét az írással egyidőben,
- nem olvashat (helyesen), amennyiben beállítjuk A időbélyegét az írással egyidőben W(A)=100-ra.

Tehát elvégeztünk egy írást A-n, ennek megfelelően beállítottuk az írási időbélyegét, de A megváltozott értékét más tranzakciók mégsem látják, mert csak munkaterületen történt az írás.
Megoldás: a tényleges írásig A-ra zárat kell tenni... Ha T közben abortál, akkor a zárat el kell engedni és W(A) értékét helyreállítani.
További alternatívák:
a. akkor ellenőrizzük az időbélyegeket, amikor az írás/olvasási igény megjelenik
b. közvetlenül a kész pont előtt ellenőrizzük az időbélyegeket.
Az a. esetben (pesszimista stratégia) a zárak hosszabbak, de az abort valószínűsége kisebb, míg a b. esetben (optimista stratégia) éppen fordítva van.

9.8.4 Verziókezelés időbélyegek mellett

Feltételezés: minden adatelem írásakor a régi értéket is megőrizzük.
Kézenfekvő megoldás, ha idősor jellegű adatokat kívánunk tárolni (betegek adatai, tőzsdei árfolyamváltozások, fejlesztési verziók és azok verziói, stb.)
Fizikai megoldás: pl. egyszer írható optikai diszkek
Segít az időbélyeges tranzakciókezelés mellett az abortok számát is csökkenteni, ha a verzió keletkezési idejét (értsd: időbélyegét) is tároljuk.
T.f.h. T tranzakció olvasni akarja A-t. Abort T kellene, ha $t(T)<W(A)$.
Amennyiben rendelkezésre áll A-nak az az A_i verziója is, amelyre $t(T)>W(A_i)$ és az ilyen tulajdonságú A_i-k közül ez a legkésőbbi, akkor T olvashatja A_i-t és mehet tovább.
Probléma: ha a T tranzakció írni akarja A-t, akkor abort T kell $t(T)<R(A)$ esetén. Verziók esetén csak akkor kell abort, ha van egy A_i változat, amelyre $W(A_i)<t(T)<R(A_i)$ (hiszen annak a tranzakciónak, amely A_i-t olvasta, valójában azt az értéket kellett volna olvasnia, amit T hozna létre).

Példa:

T_1	T_2	A_0	A_1	B_0	B_1
$t(T_1)=10$	$t(T_2)=20$	$R(A_0)=0$		$R(B_0)=0$	
		$W(A_0)=0$		$W(B_0)=0$	
READ A		$R(A_0)=10$			
	READ A (3)	$R(A_0)=20$			
	WRITE B				$W(B_1)=20$
READ B (1)				$R(B_0)=10$	
WRITE A (2)			$W(A_1)=10$		

(1): A verziók tárolása nélkül itt T_1-nek abortálnia kellene, így viszont B_1 olvasása helyett B_0 olvasásával T_1 továbbmehet.

(2): mivel T_1 korábban indult, T_2-nek feltétlenül T_1 által írott értékeket szabad csak olvasnia. Ezzel szemben (3) alatt T_2 ezt megsérti (van egy változat, i=0, amelyre $W(A_i)<t(T_1)<R(A_i)$), ezért T_1 mégis abortra kényszerül.

Összefoglalva:

- alapvetően konzervatív módszer
- sok extra háttértárat igényelhet
- bonyolult adatvisszakeresési algoritmus
- a DBMS-nek kell felderítenie, ha egy verzió a futó tranzakciók számára már közömbös, így törölhető (akkor persze, ha egyéb okból nincs rá szükség).

9.8.5 Időbélyeges módszerek áttekintése

	zárak	abort	helyreállítás	hátrány
általános	nincs	lehet	lavina lehet	helyreállítás
"pesszimista"	hosszabb időre	lehet	redo	zárak
"optimista"	rövid időre	gyakoribb	redo	abortok
verziók	nincs	ritkább	redo	tárigény, felesleges értékek eltávolítása

10 Elosztott adatbázisok

Definíció: *elosztott adatbázis*: csomópontok (node, site) összessége, amelyekben egy-egy számítógép üzemel saját CPU-val, háttértárral és adatbáziskezelővel. A csomópontok egymással össze vannak kötve adatcserére alkalmas módon úgy, hogy az adatbáziskezelők felhasználói csupán egyetlen, logikailag egységes adatbázist érzékelnek.

Sebezhetőbb az elosztott rendszer, mert a node-okon kívül a linkek változatos hibái is befolyásolhatják a működést.
A megbízható működés fenntartása fontos szempont. Módszerek:
1. adatok multiplikált tárolása, különböző fizikai helyeken (ez az adatelérés idejét is csökkentheti)
2. linkek hibái ellen többszörös elérési utak biztosítása a hálózatban
3. nagy megbízhatóságú komponensek alkalmazása.

1. esetén problémaként merül fel a másolati példányok írásának kérdése, amely atomi kell, hogy legyen valamennyi másolatot beleértve. Külön megfontolást igényel az az eset, amikor egy csomópont elérhetetlen, ilyenkor ugyanis az adatbázis működése nem állhat meg.

Definíció: *globális adat* az, amely egyetlen logikai egységnek számít az egész elosztott adatbázis szempontjából, valójában részekből állhat, tipikusan fizikailag különböző helyeken. Ezek a részek a *lokális (fizikai) adatok.*

A fizikai adatok lehetnek másolatok (megbízhatóság, gyorsabb elérés) és/vagy egy nagyobb adategység különböző részei is (ha pl. egy közigazgatási adatokat tartalmazó reláció n-esei különböző megyei szervekhez tartozó csomópontokon tárolódnak).
Fontos elv: a műveleteket logikai egységeken fogalmazzuk meg, hiszen kifelé nem látszanak a lokális adatok, de a valóságban lokális adatokon végzett műveletekre kell ezeket visszavezetni.
Példák:
Egy (globális) adatelem zárolása egyetlen logikai művelet (aminek ráadásul oszthatatlannak kellene lennie), amely a lokális adategységekre visszavezetett fizikai zárakon keresztül valósul meg. Eközben jelentős idő telhet el! Problémák: ld. elosztott zárak. Jelölés: ha A a logikai adatelem, akkor A_i-k a fizikai adatelemek.

Egy globális (logikai) tranzakció a valóságban különböző csomópontokban futó fizikai tranzakciókon keresztül valósul meg. A fizikai tranzakciók kezelik a lokális adatokat. A globális tranzakciókra megfogalmazott (pl. atomiságra, sorosíthatóságra vonatkozó) követelmények is a lokális tranzakciók révén teljesülhetnek: ld. elosztott tranzakciók. Jelölés: ha T a globális tranzakció, akkor T_i-k a lokális tranzakciók.

10.1 Elosztott zárak

A lokális példányokon a zárakat úgy kell elhelyezni, hogy támogassák a globális példányok zárolását.

A R/W modellben ha egy T_i tranzakció WLOCK A_k műveletet eredményez, akkor egyetlen más lokális tranzakció sem helyezhet semmilyen zárat A egyetlen lokális példányára sem.
Másrészről, ha csak egyetlen másolat létezik valamely adategységről, akkor a globális zárkezelés megegyezik a lokális zárkezeléssel, azaz a globális zárkezelés pontosan akkor korrekt, amikor a lokális zárkezelés korrekt.
Lefolyása: az N_i csomópontbeli T_j tranzakció zárolni akarja az A adategységet, amelynek egyetlen A_1 példánya az N_k csomópontban van. Ezért az N_i-ből egy üzenet megy N_k-ba, ahol az ottani zármenedzser eldönti a zárkérés jogosságát és az eredményről visszaüzen az N_i csomópontba.
Több lokális példány esetén számos lehetőség van, ahogyan a globális zárkérések lokális zárkérésekké lefordíthatók.
Ezek a protokollok különbözhetnek a

- bonyolultságukban és
- az üzenetek számában.

Az üzenetek lehetnek

- kontroll üzenetek, melyek rövidebbek, olcsóbbak vagy
- adat üzenetek, melyek hosszabbak, így drágábbak.

10.1.1 A WALL (write locks all) protokoll

A zárak és szemantikájuk a lokális példányokon pontosan megegyezik a R/W modellnél megismerttel. Globális adategységeken ezután zárakat az alábbiak szerint értelmezünk:

1. RLOCK A érvényes, ha RLOCK A_i érvényes A-nak legalább egy példányán
2. WLOCK A érvényes, ha WLOCK A_i érvényes A-nak az összes példányán.

Hatására a globális zár kompatibilitási mátrix az alábbi lesz:

	RLOCK	WLOCK
RLOCK	I	N
WLOCK	N	N

azaz formálisan megegyezik a lokális adategységekre értelmezett zár kompatibilitási mátrixszal. Ugyanis:
A fenti protokoll következtében két különböző tranzakció nem tarthat WLOCK-ot ugyanazon logikai adategységen. Ha T_i már zárolta A-t, akkor az összes A_i-n T_i: WLOCK A_i érvényes kell, hogy legyen, így T_j nem tud egyetlen A_i-re sem (sem író, sem olvasó) zárat rakni. Emiatt T_i: WLOCK A nem kompatibilis sem T_j: WLOCK A-val, sem T_j: RLOCK A-val. Másrészt ha T_i: RLOCK A érvényes, akkor T_i: RLOCK A_i áll fenn valamely A_i-re, amely kompatibilis T_j: RLOCK A_i-vel, tehát T_i: RLOCK A kompatibilis T_j: RLOCK A-val.
A WALL hatékonysága a szükséges üzenetváltások számával mérhető le.
Feltételezések, melyeket későbbi módszereknél is alkalmazni fogunk:
n db csomópontban van példány az A adategységből, és ismert az, hogy hol találhatók (melyik csomópontban) a lokális példányok.
Hozzávetőleges analízis:
A globális WLOCK-hoz tehát n csomópontba kell kérés (kontroll) üzenetet küldeni, innen n válasz érkezik, majd - ha mind pozitív volt - n helyre kell A új értékét

szétküldeni. Szükséges lehet még n db UNLOCK üzenet, de ez megtakarítható az elosztott kész pont képzése során (ld. később).
A globális RLOCK-hoz elég egyetlen csomópontba kontroll üzenetet küldeni (tudjuk, hol vannak a példányok, ezekből egy tetszés szerint kiválasztható), innen pozitív válasz esetén egyetlen adatüzenet küldendő.

10.1.2 Többségi zárolás

Feltételezések a lokális zárakról: mint a WALL protokollnál. Globális adategységeken ezután zárakat az alábbiak szerint értelmezünk:

1. RLOCK A érvényes, ha RLOCK A_i érvényes A lokális példányainak többségén
2. WLOCK A érvényes, ha WLOCK A_i érvényes A lokális példányainak többségén

Tétel: A globális zár kompatibilitási mátrix azonos a WALL protokoll mátrixával.

Bizonyítás:
T_i: WLOCK A_i (lokálisan) nem kompatibilis T_j: WLOCK A_i-val, így egyidőben nem lehet A lokális példányainak többségét zárolni. Emiatt (a globális) T_i: WLOCK A nem kompatibilis T_j: WLOCK A-val.
T_i: WLOCK A nem kompatibilis T_j: RLOCK A-val, mert T_i: WLOCK A-hoz A példányainak többségén WLOCK van, ami nem kompatibilis RLOCK-kal, így a többségen már nem lehet RLOCK. Emiatt T_i: WLOCK A nem kompatibilis T_j: RLOCK A-val és T_i: RLOCK A nem kompatibilis T_j: WLOCK A-val.
T_i: RLOCK A_i kompatibilis T_j: RLOCK A_i-vel, így egyidőben több tranzakció is zárolhatja az A_i-k többségét.
A többségi zárolás hatékonysága:
WLOCK A létesítéséhez legalább az n példány többségének, $\lceil (n+1)/2 \rceil$ példánynak kell kontroll üzenetet küldeni, legalább $\lceil (n+1)/2 \rceil$ válasz (kontroll) üzenet jön vissza, majd n adatüzenettel valamennyi lokális példányt írni kell.
RLOCK A-hoz ugyanannyi kontrollüzenet kell, de olvasni utána elég egyetlen lokális példányt.
Összehasonlító táblázat a WALL-lal:

	adat üzenet	kontroll üzenet
WALL írás	n	2n
többségi írás	n	n+1
WALL olvasás	1	1
többségi olvasás	1	n

Konklúzió: ha sok az olvasás, akkor a WALL, ha az írás több, akkor a többségi zárolás hatékonyabb az üzenetek száma alapján.
Más szempont: ha két tranzakció azonos adategységet akar közel egyidőben zárolni, akkor WALL esetén valószínűleg patt lesz az eredmény (aminek a feloldása költséges), többségi zárolásnál pedig az egyik sikeres lesz, a másik pedig nem (abort vagy várakozás).

10.1.3 k az n-ből protokoll

Ez a módszer az előző kettő általánosítása.
n=csomópontok száma, továbbá $\lceil (n+1)/2 \rceil \leq k \leq n$

Szabályai:
1. WLOCK A érvényes, ha WLOCK A_i érvényes k db lokális példányon,
2. RLOCK A érvényes, ha RLOCK A_i érvényes (n+1-k) db lokális példányon.

k=n esetén megfelel a WALL protokollnak, k=$\lceil (n+1)/2 \rceil$ esetén pedig a többségi protokollnak.
k alkalmas megválasztásával a protokoll "hangolható".

10.1.4 Elsődleges példányok módszere

Az eddig megismert protokolloknál a lokális adategységek felett az egyes csomópontok zármenedzserei rendelkeztek. Egy globális zár elhelyezéséhez ismerni kellett, hogy hol találhatók az adategységek példányai és ezek csomópontjai közül "számosnak" kellett üzenetet küldeni. Javíthat a zárkezelés hatékonyságán, hogyha egy A adategység valamennyi példányának elérhetőségét egyetlen csomópont, az X_A zármenedzsere felügyeli. A különböző adategységekre ez a csomópont általában különböző, tipikusan olyan csomópont, amelyen van a kiválasztott adategységnek másolata (ez lesz az A adategységnek az *elsődleges példánya*). Ugyanakkor az is elképzelhető, hogy valamennyi adategység felett ugyanaz a csomópont rendelkezik (*centrális csúcs módszere*).
Hatékonysága:
1. WLOCK A-hoz kell egy kérés (kontroll) üzenet X_A-ba, erre jön egy válasz, majd (jó esetben) írandó A-nak valamennyi másolata.
2. RLOCK A-hoz kell egy kontroll üzenet X_A-ba, erre jön egy válasz, majd kiolvasható A-nak valamely másolata.

Tehát sokkal hatékonyabb, mint az előbbiekben megismertek, hiszen a kontrollüzenetek száma konstans, nem n-nel arányos. Viszont sebezhetőbb, mert egy adategység egyáltalán nem elérhető, akárhány másolata is van, ha az X_A csomópont kiesik.

10.1.5 Elsődleges példányok tokennel

Ez a protokoll az elsődleges példányok módszerének továbbfejlesztése olyan módon, hogy egy adategység elérését kontrolláló csúcs kijelölése dinamikusan változhat.
Minden A adategységhez értelmezünk egy írási WT(A) és egy olvasási tokent RT(A), amelyek az adategység elérését szabályozzák. Ha létezik WT(A), akkor nem létezhet RT(A), ha nincs WT(A), akkor viszont akárhány RT(A) létezhet.
A tokenek szemantikája:
Ha az X csomópontban van a WT(A) token, akkor az X csomópont zármenedzsere jogosult az RLOCK A-t vagy WLOCK A-t megítélni az X csomópontban futó (globális) tranzakciók számára.
Ha az X csomópontban van az RT(A) token, akkor az X csomópont zármenedzsere jogosult az RLOCK A-t megítélni az X csomópontban futó tranzakciók számára.
Amennyiben pl. egy tranzakció az Y csomópontban a B adategységet írni akarja, akkor ehhez WT(B)-t az Y csomópontba kell juttatnia. Ha nem lenne ott, akkor ezt üzenetváltásokkal érheti el:
Y-ból WT(B)-t kérő üzeneteket küld valamennyi (m db.) csomópontba, amely az elosztott adatbázishoz tartozik → m db. kontroll üzenet
A csúcsok válaszolnak a.) vagy b.) üzenettel (→ m db. kontroll üzenet), mégpedig:

a.)-t válaszolnak, ha nincs náluk sem RT(B) sem WT(B), vagy náluk van ugyan, de lemondanak róla
b.)-t válaszol egy csúcs, ha nála van RT(B) vagy WT(B), és kell is neki (tehát nem engedi át a tokent, mert pl. még használja vagy már más csomópontnak ígérte). Ekkor a csúcs megjegyzi a kérést.

Ezután az Y csomópont összegyűjti a válaszokat:

- ha mindenki a.)-t üzen, akkor Y tudja, hogy a tokent megszerezheti. Ezért üzeneteket küld minden csomópontba, hogy semmisítsék meg a B-re vonatkozó tokenjüket.
- ha valaki b.)-t üzen, akkor Y visszavonja a kérését azoktól a csomópontoktól, akik a.)-t válaszoltak.

Ez legfeljebb újabb m db. kontrollüzenetet jelent.

Az RT(B) megszerzése hasonló:

Y-ból RT(B)-t kérő üzeneteket küld valamennyi csomópontba.

- A csomópontok nem válaszolnak, ha RT(B) van náluk, vagy
- a csomópont a.) vagy b.) üzenettel válaszol, mégpedig:

  a.)-t válaszol, ha nincs nála WT(B), vagy nála van ugyan, de lemond róla

  b.)-t válaszol, ha nála van WT(B), és kell is neki. Ekkor a csúcs megjegyzi a kérést.

  Ezután az Y csomópont összegyűjti a válaszokat:
  - ha csak a.) jött, akkor Y tudja, hogy szerezhet RT(B)-t. Ezért üzeneteket küld azokba a csomópontokba, ahonnan a.) jött, hogy semmisítsék meg a WT(B) tokent.
  - ha valaki b.)-t üzen, akkor Y nem tud RT(B)-t szerezni.

Értékelés: a token mozgatása nagyon költséges, de ha a token már a megfelelő csomópontban van, akkor az újabb tranzakcióknál a tokenmozgatás járulékos költsége már zérus. A módszer tehát adaptív.

10.1.6 Összefoglaló

A zárkezelési protokolloknál a kontroll üzenetek szükséges száma (az adat üzeneteké azonos):

	kontroll üzenet íráskor	kontroll üzenet olvasáskor	megjegyzés
WALL	2n	1	jó, ha sok az olvasás
többségi zárolás	≥n+1	≥n	jó, ha sok az írás
elsődleges példányok	2	1	hatékony, de sebezhető
elsődleges példányok tokennel	0-3m	0-3m	adaptív
centrális csúcs	3	2	nagyon sebezhető, centralizált hálózati forgalmat okoz

10.2 Elosztott tranzakciók problémái

A cél a tranzakciók atomiságát és sorosíthatóságát elosztott környezetben is biztosítani. Ehhez rendelkezésre állnak a (globális) zárak és változatos protokollok,

amelyek nem egyszerű kiterjesztései a nem elosztott környezetben működő protokolloknak.

Definíció: elosztott sorosíthatóság: tranzakciók egy ütemezése egy elosztott adatbázison sorosítható, ha hatása a logikai adategységeken ugyanaz, mintha a tranzakciók valamely soros ütemezésben futottak volna le, azaz mindegyik logikai tranzakcióhoz tartozó fizikai tranzakció is befejeződik, mielőtt a soros ekvivalensben rákövetkező logikai tranzakció elkezdődik.

Nem elosztott környezetben a kétfázisú zárolás elégséges feltétele volt a sorosíthatóságnak. Elosztott környezetben ez nincs így.
Példa:
két globális tranzakció egyenként két-két lokális tranzakcióból áll:
$T_1=T_{11}+T_{12}$, $T_2=T_{21}+T_{22}$
amelyek közül T_{11} és T_{21} valamint T_{12} és T_{22} azonos csomópontban fut.

csomó pont 1		csomó pont 2	
T_{11}	T_{21}	T_{12}	T_{22}
WLOCK A			WLOCK B
UNLOCK A			UNLOCK B
	WLOCK A	WLOCK B	
	UNLOCK A	UNLOCK B	

A lokális sorosítási gráfok:

$T_1 \rightarrow T_2$ $\qquad$ $T_1 \leftarrow T_2$

A globális sorosítási gráf:

$$T_1 \rightleftarrows T_2$$

Azaz a precedenciagráfban hurok van, a globális ütemezés nem sorosítható, mégha a lokális sorosíthatóság fenn is áll a lokális kétfázisúság következtében.

10.2.1 Elosztott kétfázisú zárolás

A globális sorosíthatósághoz nyilván elégséges feltétel, hogyha a globális tranzakciók *globálisan kétfázisú*ak, azaz egy T_i tranzakció egyetlen T_{ij} résztranzakciója sem engedhet el egyetlen zárat sem, ameddig bármelyik T_{ij} még kérhet új zárat.

A megvalósítás úgy képzelhető el, ha a résztranzakciók üzenetváltásokkal informálják egymást arról, hogy elérték a zárpontjukat, több zárra nincs szükségük. Azaz kell egy *közös zárpont*.

10.2.2 Szigorú kétfázisú zárolás

A lavinák problémája az elosztott környezetben is megmaradt. Elkerülése itt is lehetséges, ha biztosítani tudjuk, hogy egyetlen résztranzakció se írjon az adatbázisba mindaddig, amíg minden résztranzakció el nem érte a kész pontját. A lavinák ellen tehát *közös kész pont*ot kell létrehozni, amihez az kell, hogy minden résztranzakció valamennyi számítását befejezze, valamennyi zárját megkapja. Ha ez minden résztranzakcióra teljesül, akkor a tranzakció folytatható, egyébként pedig valamennyi résztranzakciónak abortálnia kell. Tehát a közös kész pontot megelőzi a közös zárpont elérése, vagy akár egybeeshet vele. Ha valamely protokoll tehát biztosítja a közös kész pont képzését, akkor ez közös zárpontnak is megfelelő. A közös kész pont megvalósítása különböző hibalehetőségeket is figyelembe véve egy *elosztott*

megegyezési feladatra vezet, amely költséges művelet. A következő szakasz részletesebben ismerteti.

10.2.3 Elosztott kész pont képzése - a kétfázisú kész protokoll (2PC)

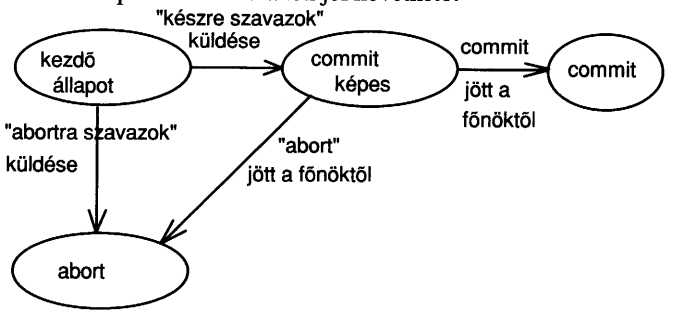
Először egy viszonylag egyszerű protokollt nézzünk meg: az elosztott kész pont képzését akkor, ha nem kell hálózati hibákkal számolni.
Feltételezések:
Adott egy T logikai tranzakció, számos T_i lokális tranzakcióval az N_i csomópontokban. Az a csomópont, ahol a tranzakciót kezdeményezték legyen a *főnök*, a többit *résztvevő*nek nevezzük.
Minden résztvevő szavaz egy-egy, a főnöknek küldött üzenet formájában:

- készre szavaz akkor, ha a résztranzakció elérte a kész pontját
- abortra szavaz akkor, ha bármely ok miatt a résztranzakció abortálni kényszerült.

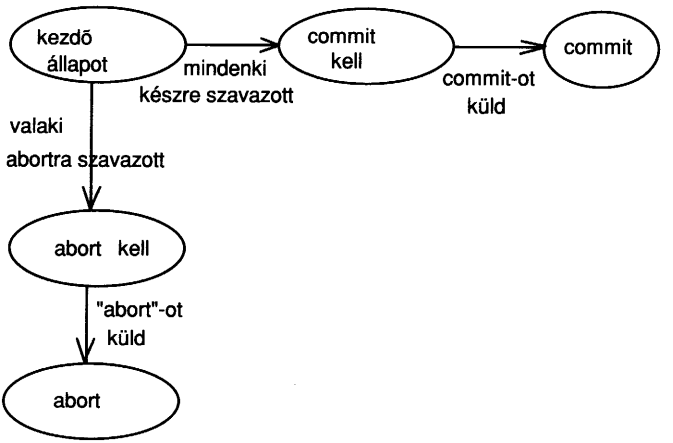
A főnök ezután két dogot tehet:

- ha minden résztvevőtől "készre szavazok" üzenetet kapott, akkor "commit" üzenetet küld valamennyi résztvevőnek. Ebből minden résztvevő megtudja, hogy minden résztvevő készre szavazott, tehát a tranzakció elérte a kész pontját.
- ha bárhonnan is "abortra szavazok" üzenetet kap, akkor a tranzakciónak is abortálnia kell, ezért "abort" üzenetet küld valamennyi résztvevőnek, akik ennek hatására abortálják a résztranzakciókat.

Mindez az alábbi állapotátmenet ábrákon jól követhető.



10.2.3.a. ábra: Egy résztvevő állapotai

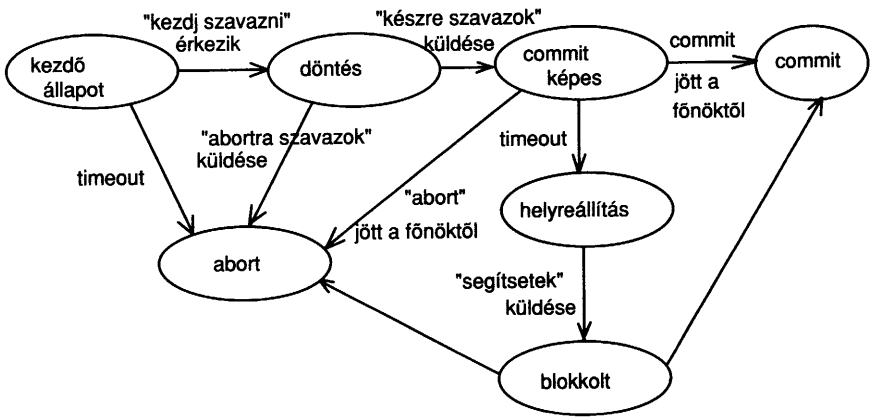


10.2.3.b. ábra: A főnök állapotai

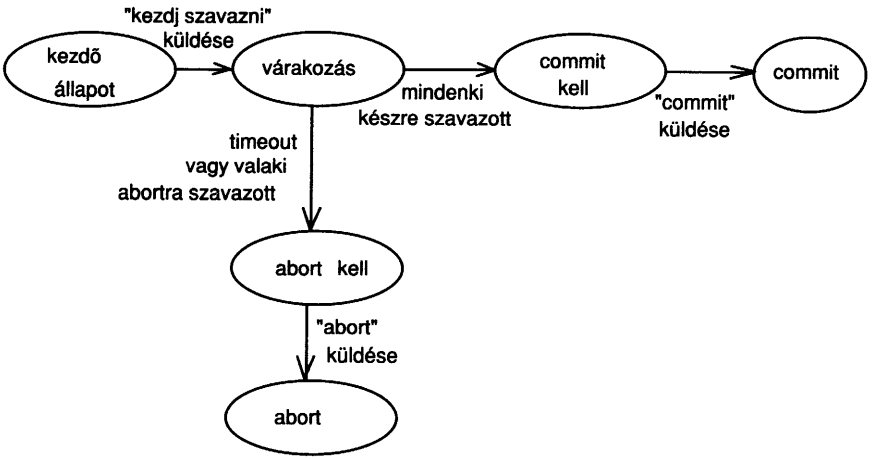
Megjegyzések:
1. a főnök is résztvevő egyben, ő is küld magának üzeneteket, csak éppen ezek nem költséges üzenetek
2. ha nincs főnök, csak résztvevők, akkor mindenki mindenkinek kell, hogy küldjön üzenetet. Ilyenkor az üzenetek száma n^2-tel lesz arányos
3. hálózati hibák esetén a protokoll nem használható: ha u. is T_i zárat tart fenn valamely adategységen és commit képes állapotban van, akkor itt is marad, ha nem jön üzenet a főnöktől (pl. hálózati hiba miatt), mert
   - commit állapotba nem léphet, mert jöhet "abort" üzenet,
   - abort állapotba sem léphet, mert jöhet "commit" üzenet és
   - a zárait sem engedheti el, mert sérülhet a globális kétfázisúság.

A jelenség neve *blokkolás*, kiküszöbölése a hálózati hibák esetén is működőképes protokollok alapproblémája.

Második lépésben vizsgáljuk meg az elosztott kész pont kétfázisú képzését (two-phase commit, 2PC protokoll).
Az előző protokoll továbbfejlesztése, amikoris a résztvevők a commit-abort eldöntése után két fázisban szavaznak a logikai tranzakció commitjáról, ill. abortjáról. Ennek eredményeképpen bizonyos hibák esetén elkerülhető a blokkolás, de a lehetősége továbbra is fennmarad. A leggyakoribb algoritmus az elosztott megegyezés megvalósítására.
Jelentősebb változások:
1. A résztranzakciók mérik azt az időt, hogy mióta várják a választ a szavazatukra. Ha túl sok idő telik el, akkor a hálózati hiba valószínű, így egy olyan állapotba lépnek be, aminek a deklarált célja a probléma feloldása.
2. A főnök üzenetküldéssel szólítja fel a résztvevőket a szavazás megkezdésére.



10.2.3.c. ábra: Egy résztvevő állapotai a 2PC protokollban



10.2.3.d. ábra: A főnök állapotai a 2PC protokollban

A helyreállítás folyamata:
ha a résztvevő belép a "helyreállítás" állapotba, akkor "segítsetek" üzenetet küld valamennyi többi résztvevőnek (ehhez ismernie kell őket). Erre válaszul jöhet

- commit üzenet, mert ha a résztvevő már commit állapotban van, akkor ezt az üzenetet kell küldenie, vagy
- abort üzenet, abort állapotban lévő résztvevőtől vagy olyantól, aki még kezdő állapotban van.

Ha a résztvevő commit képes állapotban van, akkor nem tud hasznos üzenetet küldeni, így nem is küld semmit.
A "blokkolt" állapotban lévő tranzakció ezután annak megfelelően jár el, amilyen üzeneteket kapott. Ez a döntése helyes, mert

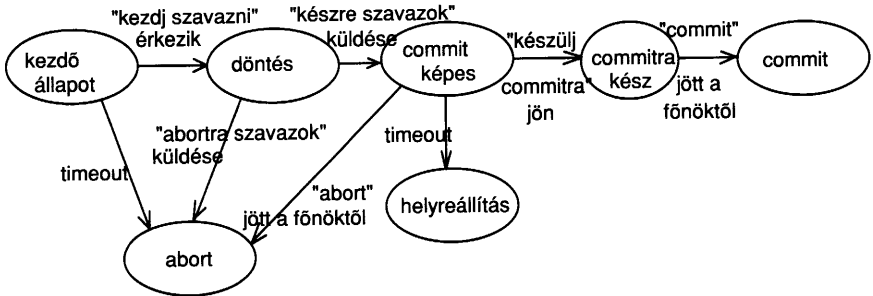
- nem kaphat commit és abort választ is a "segítsetek" üzenetére,
- nem kaphat abortot, ha a főnök commit-ot küldött, és
- nem kaphat commitot, ha a főnök abortot küldött.

Amennyiben nem jön semmilyen üzenet a segítségkérésre - mert pl. a többi résztvevőtől el van vágva, vagy a többiek commit képes állapotban vannak -, akkor a blokkolás a 2PC-nél is bekövetkezhet.
A főnök a várakozás állapotból timeout-tal az abort állapotba kerülhet, ha valamely résztvevő hosszú idő után sem válaszol. Ekkor u. is joggal feltételezhető, hogy a résztvevő nem elérhető vagy meghibásodott. Előbbi esetben a résztvevő könnyen blokkolt állapotban maradhat.
A résztvevő, ha hosszú idő után sem kap "kezdj szavazni" üzenetet, akkor abort állapotba megy át, mert azt tételezi fel, hogy a főnök elérhetetlenné vált vagy meghibásodott. Ebben az állapotában már elengedheti a zárjait, ha pedig mégis megjön a "kezdj szavazni" üzenet, akkor egyszerűen abortra szavaz.
Megjegyzés: egy résztranzakció kész vagy abortált állapotában is jöhetnek kérések, ha más résztranzakció segítséget kér. Ilyenkor egy napló alapján van lehetőség korrekt választ adni.

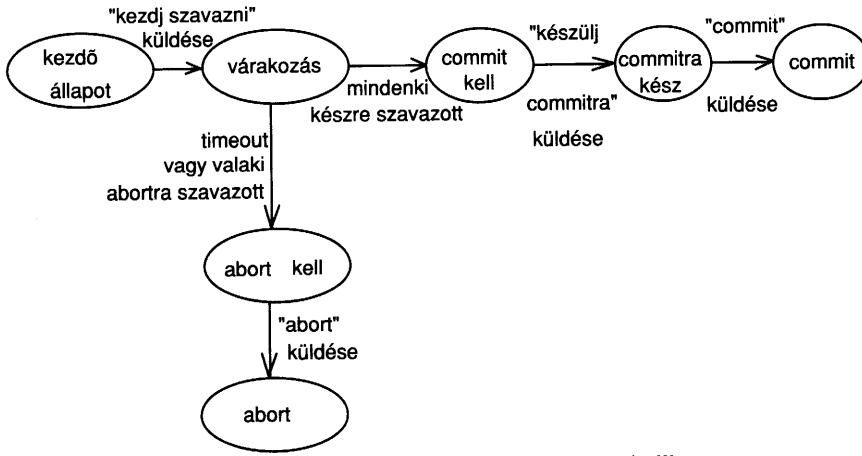
10.2.4 Egy "blokkolásmentes" kész protokoll - 3 fázisú kész protokoll (3PC)

Valójában ez a megoldás is csak csökkenti a blokkolás valószínűségét (hálózatszakadás esetén egyetlen protokoll sem tudja garantálni!). Ugyanis 2PC-nél a blokkolást az okozza, ha valamely résztvevő nem tudja, hogy mindenki készre szavazott. Ha tudná, akkor nem lenne baj abból, ha ezután a kapcsolat a többi csomóponttal megszakad, hiszen tudja, hogy commit-ot kell végrehajtania. Ezért a 3PC-nél commit állapotba csak akkor kerülhet egy résztvevő, ha már tudja, hogy mindenki tudja, hogy mindenki commitra szavazott. Mindez egy újabb üzenetcserével realizálható. Eredmény: olyan protokoll, amely a következő esetekben blokkolásmentes:

- csak csomóponthibák vannak, a hálózat nem esik szét,
- a csomópont meghibásodása esetén hallgat, nem küld hamis üzenetet,
- a hibás csomópont nem áll vissza anélkül, hogy a kiesését megfelelően le ne kezelné,
- működő csomópont a timeout időnél gyorsabban válaszol,
- a hálózat üzenetet nem veszít és megtartja az üzenetek sorrendjét.



10.2.4.a. ábra: Egy résztvevő fontosabb állapotai a 3PC protokollban



10.2.4.b. ábra: A főnök állapotai a 3PC protokollban

A három fázis értelmezése:
amikor egy résztvevő megkapja a "készülj commitra" üzenetet, akkor tudhatja, hogy mindenki készre szavazott. A résztvevők ezt nyugtázzák, ezután küldi a főnök a "commit jön" üzenetet. Ha egy résztvevő ezt is veszi, akkor ebből már tudja, hogy minden résztvevő már megtudta, hogy minden résztvevő commitra szavazott. Ezután committálhat.

10.3 Elosztott időbélyeges tranzakciókezelés

Az egyprocesszoros elvek jó része átvihető elosztott környezetre is.
Ha egy tranzakció egy N csúcsban írja és/vagy olvassa az A adategységnek valamely A_i példányát, akkor rajta hagyja az alkalmazott modellnek megfelelő időbélyegét. (Írás esetén természetesen valamennyi példányt írni kell és a végleges adatbázisba írás előtt egy elosztott megegyezésnek kell lezajlania.) A tranzakciónak az időbélyeget természetesen az a csomópont adja, ahol a tranzakciót kezdeményezték. Az egyértelműség biztosítása itt is alapvető követelmény, ami sérülhet, ha a csomópontok saját óráik alapján állítják elő az időbélyegeket. Egy megoldás: a helyi időből képzett időbélyeghez annak LSB részeként hozzáfűzzük a *csomópont azonosítóját.*
Ezután annak ellenőrzése, hogy a kezdeményezett adathozzáférési műveletek összhangban vannak-e az időbélyegek növekvő sorrendje szerinti soros ekvivalenssel hasonló módon történhet, mint egyprocesszoros esetben. Majdnem minden elosztott zár alapú tranzakciókezelési módszernek megvan az *elosztott időbélyeges megfelelője.* A WALL-lal analóg protokoll szerint pl. read A esetén egyetlen $R(A_i)$, $W(A_i)$ vizsgálatából, míg write A esetén valamennyi $R(A_i)$, $W(A_i)$ vizsgálatából döntendő el, hogy a tervezett adatelérés legális-e.
Egy másik problémát jelenthet a különböző csomópontok óráinak eltérése. Az órák egzakt szinkronizmusa fizikai okok miatt nem tételezhető fel, de szerencsére nincs is rá szükség.
T. f. h. az N csomópont órája jelentős (több óra, vagy akár több nap) késést mutat a többi órához képest. Minden itt kezdeményezett tranzakció ezt a nagyon régi időbélyeget kapja meg. Ha valamely adategységhez hozzá akar férni, amelyet más csomópontokban kezdeményezett tranzakciók is használnak, akkor nagy a valószínűsége, hogy az adategységnek az időbélyegei fiatalabbak. Ez gyakorlatilag azt jelenti, hogy a tranzakció abortra kényszerül.
Ellenkezőleg, ha valamely csomópont órája jóval előbbre jár, mint a többié, akkor az itt kezdeményezett tranzakciók gyakorlatilag sohasem fognak sorosítási feltétel megsértése miatt abortálni.
Kis óraeltérések esetének vizsgálatához arra gondoljunk, hogy az időbélyeges tranzakciókezelés azt utánozza, mintha a tranzakciók az időbélyegük által meghatározott időpillanatban zérus idő alatt futnának le. Tehát a kis óraeltérések nem kritikusak, azonban késő óra esetén csökken, sietős óra esetén pedig nő a tranzakció sikeres lefutásának valószínűsége.
A nagy óraeltérések könnyen korrigálhatók, ha a csomópontok egymáshoz küldött üzeneteihez hozzáillesztjük a küldő csomópont órájának pillanatnyi értékét is. Ha egy csomópont azt tapasztalja, hogy a jövőből kap üzenetet, akkor a saját óráját ehhez igazítja. Ha tehát egy csúcs - pl. meghibásodás miatt - leáll az órájával együtt, akkor újraindulás után szinte biztos, hogy az általa kezdeményezett tranzakció abortálni fog. A közben folyó üzenetváltások során azonban korrigálhatja az óráját, aminek hatására

az újból elindított tranzakció a második kísérletre már átlagos valószínűséggel sikeres lesz.

10.4 Csúcsok helyreállítása rendszerhibák után

Ha egy elosztott adatbázisban valamely csomópont meghibásodik, ez nem okozhatja a teljes adatbázis kiesését. Egy csomópont hibája viszont könnyen eredményezheti azt, hogy bizonyos adategységek elérhetetlenné válnak. A korábbiakban megismertek szerint egy tranzakció csak akkor hajthat végre az adatbázison sikeresen módosításokat, ha a módosítandó adategységnek valamennyi lokális példánya elérhető. Emiatt mindazon tranzakciók sikertelenek lesznek, amelyek olyan adategységekre hivatkoznak, amelyeknek a kiesett csomópontban is van példánya. Ez - szerencsétlen esetben - akár az adatbázis teljes elérhetetlenségét is okozhatja. Elkerülendő ezt a helyzetet biztosítani kell, hogy az adatbázis - akár csökkent funkcionalitással, de - tovább működjön.
A naplózás ebben az esetben minden csomópontnál szükséges. Ha a hálózati hibák ellen is akarunk vele védekezni, akkor a csomópont által küldött és vett üzeneteket is naplózni kell.
Amikor egy csomópont "feléled", akkor gondoskodnia kell arról, hogy a lokális adatait konzisztens állapotba hozza a többi csomópontéval.
Ehhez minden csomópont, amely észleli, hogy egy csomópont (pl. N) kiesett, naplózza ezt a tényt saját magánál, majd folyatja a működését, amennyiben ez lehetséges. Amint az N csomópont feléledt, minden csomópontnak üzenetet küld. Ezt az üzenetet véve a többiek a naplójuk alapján - a korábban megismertekhez hasonló módon - kiderítik, hogy mely adategységek változtak meg azok közül, amelyeknek N-ben is van példányuk. Ezután ezeket az adatokat elküldik N-nek, aki így a lokális adatait fel tudja frissíteni (eközben természetesen az érintett adategységekre zárakat kell helyezni).

10.5 Elosztott pattok keletkezése és kezelése

Patthelyzetek természetesen elosztott környezetben is előfordulhatnak. A keletkezésük analóg az elosztott tranzakciók sorosíthatóságának alakulásával (ld. 10.2. szakasz ütemezését):

1. előfordulhat, hogy valamely csomópontban alakul ki patthelyzet, a lokális tranzakciók között, ill.
2. előfordulhat, hogy bár lokálisan sehol sincs patt, mégis a tranzakciók egymást várakoztatják.

Az első esetben a lokális várakozási gráf nem DAG, a második esetben pedig a globális várakozási gráf nem DAG.
Ebből viszont az is következik, hogy a globális várakozási gráf vizsgálata nem képzelhető el anélkül, hogy a csomópontok ne küldenének üzeneteket egymásnak arról, hogy a lokális várakoztatási viszonyok hogyan alakulnak. Bár a módszer működőképes - különösen, ha egy centrális csomópont foglalkozik a pattdetekcióval -, mégis hatékonyabb lehet számos esetben, ha a pattok létrejöttét akadályozzuk meg.
Láttuk, hogy a patt elkerülhető egyprocesszoros környezetben, ha minden tranzakció zárakat csak az adategységek növekvő sorrendjében kér.

Ha zárkérésre a centrális csúcs módszert alkalmazzuk, és a másolatokat is egyedi adategységeknek tekintjük, akkor az változtatás nélkül működőképes és elkerülhetők a pattok.

Ha viszont az "*n* a *k*-ból" módszert használjuk, akkor be kell tartani az alábbi szabályokat:

1. ha A <B, akkor LOCK A_i minden i-re meg kell, hogy előzze bármely LOCK B_i-t,
2. az adategységek másolatait is sorrendbe kell állítani és a zárkérések csak a másolatok növekvő sorrendjében teljesíthetők.

Nyilvánvaló akadálya az alkalmazásának, hogy előre kellene ismerni az adategységeket, amelyeken a tranzakció zárat akar elhelyezni. Ellenkező esetben szükségtelenül sok zárkérést kell menedzselni.

11 Gyakorló feladatok

ER diagramok

1. Hogyan alakítható át egy ternális kapcsolatot tartalmazó ER diagram ekvivalensen csak bináris kapcsolatot tartalmazó ER diagrammá?

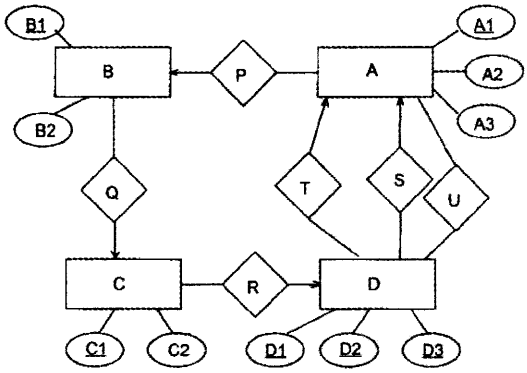
2. Egy menza havi menüit szeretnénk tárolni egy adatbázisban. A menü minden nap egy levest, egy főételt és egy édességet tartalmaz. Egy étel többször is előfordulhat az adott hónapban, de tudjuk, hogy egy adott leves-főétel kombinációhoz csak egy édesség illik. Minden ételnek tárolni szeretnénk a nevét, az energiatartalmát, és hogy melyik hozzávalóból mennyi kell az elkészítéséhez. Készítsen ER diagramot az adatbázishoz!

3. Adott a következő laikus leírás:
   Egy paciensnek számos betegsége is lehet, vannak betegségek, amiben pillanatnyilag senki sem szenved. Minden pacienst egyetlen mentőállomáson kezelnek, akár több orvos is. Az orvosoknak több paciensük is lehet, akik különböző mentőállomásokon is fekhetnek. Egy mentőállomás lehet akár üres is, és mindig pontosan egy kórházhoz tartozik. Egy kórháznak esetleg több mentőállomása és több orvosa is van. Egy orvost legfeljebb 3 kórház alkalmaz. A kórházat mindig egy olyan igazgató vezeti, aki a kórház orvosa is, közgazdászdiplomával is rendelkezik és más kórházzal nincs munkaviszonya.
   Készítsen a fentiekről egyed-kapcsolati (ER) diagramot! A tanult szintaktikával tüntesse fel pontosan a kapcsolatok funkcionalitását is! Azonosítsa az egyedeket célszerűen megválasztott attribútumokkal, határozza meg a kulcsokat!

4. Adott a következő laikus leírás:
   Egy kórház számos osztályból áll, mindegyiknek van egy osztályvezető főorvosa és akárhány főorvosa. Ha nincs osztályvezető főorvos, akkor van megbízott osztályvezető. Ők valamennyien a kórház - orvosdiplomával is rendelkező - alkalmazottai, másik kórházban nincs állásuk. Egy kórháznak ezen kívül még számos más dolgozója is van: orvosok, nővérek, segédszemélyzet. Az orvosok és a nővérek mindig egy meghatározott osztályon dolgoznak, míg a segédszemélyzet közvetlenül a kórházhoz is tartozhat. Minden alkalmazottnak van kódszáma, de a orvosoknak nyilvántartják a kamarai tagsági számukat is. A kórházat mindig egy olyan igazgató vezeti, aki a kórház orvosa is, közgazdászdiplomával is rendelkezik és más kórházzal nincs munkaviszonya. Egy beteg - ha bekerül a kórházba - számos osztályt is megjárhat, amíg meggyógyul, és eközben számos betegséggel kezelhetik.
   Készítsen a fentiekről egyed-kapcsolati (ER) diagramot! A tanult szintaktikával tüntesse fel pontosan a kapcsolatok funkcionalitását is! Azonosítsa az egyedeket célszerűen megválasztott attribútumokkal, aláhúzással jelölje meg a kulcsokat!

Relációs sémaábrázolás, relációalgebra

5. Alakítsa át az alábbi ER diagramot relációs sémákba! Törekedjen minél kevesebb séma kialakítására!



6. a) Alakítsa át a 3., ill. a 4. feladatban kapott ER diagramot relációs sémákká!
   b) Végezze úgy az átalakítást, hogy a kapcsolatok megvalósításához a lehető legkevesebb relációs sémát definiálja!

7. Adott két relációs séma R(A,B) és S(B,C) valamint két reláció r(R) és s(S). Tudjuk, hogy r és s egyesítésével kapott relációnak (b,c) és (d,e) is elemei. Tudjuk továbbá, hogy a természetes illesztésükkel kapott reláció pedig:

A	B	C
a	b	a
f	c	g

   Határozza meg a két relációt!

8. Adott egy r és egy s reláció, melyek rendre az R(A,B) illetve az S(B,C) sémára illeszkednek. r-nek van n_r csupa különböző sora, s-nek pedig n_s. Legfeljebb és legalább hány sora lehet (n_r és n_s függvényében) az r és s természetes illesztésének, ha
   - A kulcs R-ben
   - B kulcs R-ben
   - B kulcs R-ben és S-ben is?
   - A kulcs R-ben B kulcs S-ben

9. Ha egy A attribútum kardinalitása kisebb, mint az A doménje elemeinek száma, akkor A nem lehet (egyszerű) kulcs. Igazolja vagy cáfolja az állítást!

10. Legyen R, S két azonos attribútumokkal rendelkező reláció, X pedig ezen közös attribútumhalmaz egy részhalmaza. Melyek igazak az alábbi állítások közül?

$$\pi_x(R \cup S) = \pi_x(R) \cup \pi_x(S)$$

$$\pi_x(R \setminus S) = \pi_x(R) \setminus \pi_x(S)$$

11. Adott a következő sémaleírás, adjon relációalgebrai kifejezéseket a kérdésekre!
TERMÉK(GYÁRTÓ, MODELL, TÍPUS)
PC(MODELL, SEBESSÉG, MEMÓRIA, MEREVLEMEZ, CD, ÁR)
LAPTOP(MODELL, SEBESSÉG, MEMÓRIA, MEREVLEMEZ, KÉPERNYO, ÁR)
NYOMTATÓ(MODELL, SZÍNES, TÍPUS, ÁR)
Kérdések:
    - Mely nevű áruk azok, amelyekkel van azonos egységárú másik áru?
    - Melyek azok a PC modellek, amelynek sebessége legalább 150?
    - Mely gyártók készítenek legalább egy gigás merevlemezSu laptopot?
    - Adjuk meg a B gyártó által gyártott összes termék modellszámát és árát típustól függetlenül!
    - Melyek azok a gyártók, akik laptopot gyártanak, de PC-t nem?
    - Melyek azok a gyártók, amelyek gyártanak legalább két, egymástól különböző legalább 133 MHz-en működő PC-t vagy Laptopot? (Nincs két azonos modellszám!)
    - Az összes 133MHz-nél gyorsabb PC és LAPTOP gyártója
    - Azon gyártók, melyek olyan LAPTOP-okat hoznak forgalomba, melyekkel megegyező tulajdonságú PC-ket is árulnak.
    - Azon gyártók, melyek olyan PC-ket hoznak forgalomba, melyekkel megegyező tulajdonságú LAPTOP-okat is árulnak.

12. Tekintsük az alábbi csillagflotta adatbázissémát:
CSILLAGHAJÓ(HAJÓNÉV,ÉV,FAJ),
DOLGOZÓ(DOLGOZÓNÉV,AZONOSÍTÓ,SZÜLETÉS),
BEOSZTÁS(AZONOSÍTÓ,HAJÓNÉV,RANG)
A relációk jelentése a következő:
Csillaghajó: a hajó neve, gyártási éve, és az hogy melyik faj tervei alapján készült
Dolgozó: neve, Csillagflotta-azonosítója, mikor született
Beosztás: melyik dolgozó melyik hajón, milyen beosztásban dolgozik
Adjunk relációalgebrai kifejezést, mely megadja azon dolgozók nevét, akik Catherine Janeway kapitány hajóján dolgoznak.

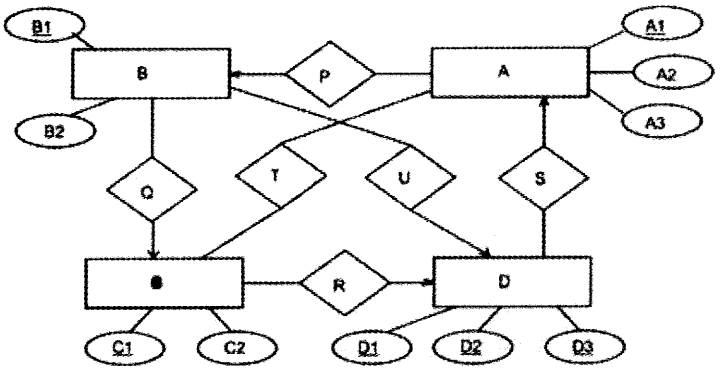
13. Tekintsük a következő alaprelációkat (a kézenfekvő értelmezéssel): Kedvel(személy, sör), Kapható(söröző, sör), Látogat(személy, söröző). Fejezze ki relációs algebra segítségével
    - azon sörök összességét, melyeket minden látogató kedvel azokban a sörözőkben ahol kaphatók.
    - azon személyek összességét, akik minden sört kedvelnek azokban a sörözőkben, amelyeket látogatnak.

Hálós sémaábrázolás

14. Adott egy ternáris kapcsolat az attribútumaival. Milyen mezőt tartalmaznak az ekvivalens hálós modellben a member típusú rekordok?

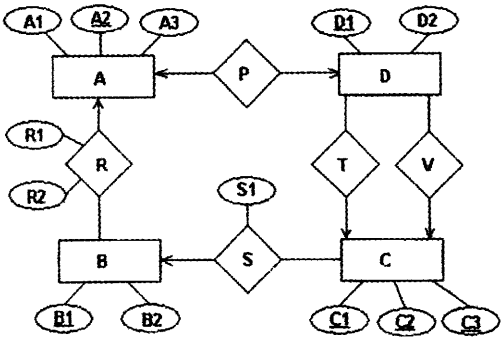
15. Bizonyítsa be, hogy a hálós modell esetén egy set-típuson belül az owner példányokhoz kapcsolódó memberek halmazai diszjunktak.

16. Alakítsa át az alábbi ER diagrammot hálós sémába úgy, hogy minimális számú set-típust definiáljon! Definiálja a rekord típusokat is, a definíciókat rendezze ABC sorrendbe!



17. Alakítson ki hálós sémát ami személyeket, munkahelyeiket és munkáikat leíró (egyszerűsített) adatbázis alapjául szolgálhat. Az alábbiakat szeretnénk ábrázolni:
    - Személy: név, személyi szám, lakcím, munkahely(ek), beosztás(ok), projektek, amiken dolgozik.
    - Eszköz: megnevezés, azonosító, tulajdonos cég, a porjekt(ek) amiben használják.
    - Cég: név, cím, vezető, dolgozók, projektek.
    - Projekt: elnevezés, vezető, érintett cége(ek), határidő, résztvevők.

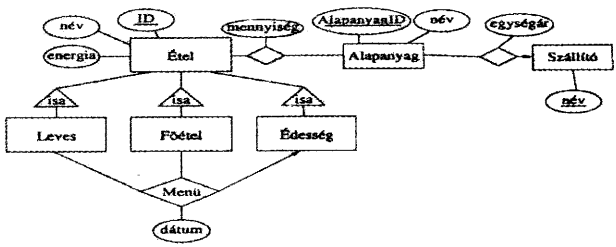
18. Adott az alábbi ER diagram, amely egy videokölcsönző üzlet működéséhez szükséges elemeket tartalmazza: a kazettákat, ügyfeleket, kölcsönzéseket, előjegyzéseket.
    - Alakítsa át a diagramot hálós sémákba úgy, hogy a lehető legkevesebb set-típust definiálja! A hálós séma elemeit a megfelelő E-R diagrambeli elemmel azonosan nevezze el. Rajzolja le a hálós sémát a tanult jelölésrendszert használva. Definiálja a rekordtípusok szerkezetét is.
    - Alakítsa át a diagramot relációs sémákba úgy, hogy a lehető legkevesebb relációs sémát definiálja! A relációs séma elemeit a megfelelő ER diagrambeli elemmel azonosan nevezze el.



19. Mindenki tudja, hogy a Nyuszi létezik, de azt már kevesen, hogy sok-sok húsvéti nyuszi van! Minden bizonnyal a Húsvétinyuszi Rt. is rendelkezik adatbázissal az alkalmazottairól. A cégnek vannak kirendeltségei a kontinenseken. Egy kontinensen, akár több is. Minden kirendeltség élén egy fő nyuszi áll, de egy nyuszi állhat akár több kirendeltség élén is. Azt tudjuk, hogy a vezérigazgató, Tapsi Hapsi, a Húsvét szigeteki kirendeltség vezetője. Minden nyuszi be van sorolva egy kirendeltséghez, ahol dolgozik és készíti az színes tojásokat. A Nyusziadatbázis tartalmazza minden gyerekről, hogy melyik nyuszi felelős az ő tojásainak kézbesítéséért. Ez csak olyan nyuszi lehet, aki abban a körzetben dolgozik, ahol a kisgyerek lakik. A Húsvétinyuszi Rt-re is hat az információs társadalom, így szabványba foglalták, hogy a gyerekek jósága 1..10-es skálán mérhető és minden jóság mértékhez megadták a standard ajándéktojások mennyiségét is. Minden nyuszi licenccel rendelkezik adott jóságú gyerekeknek tojást vinni, de persze egy nyuszinak lehet több licence is. A leírás alapján tervezzen a nyusziadatbázishoz
    - ER diagramot
    - Relációs sémát
    - Hálós sémát
    - Adjon relációalgebrai kifejezést a következő meghatározásához: Tapsi Hapsi saját maga szeretné megajándékozni a 10-es faktorú gyerekeket, és kíváncsi, hogy melyik körzetben laknak.

Relációs lekérdező nyelvek

20. A következő ER modellhez illeszkedő relációs sémát definiáljon SQL-ben. (Hozzon létre megfelelő táblákat SQL utasításokkal.) Ne feledkezzen meg a különféle megszorításokról sem!



21. Adott a következő séma: Jegyek(Neptun, DiákNév, Tárgykód, Jegy). (Melyik diák melyik tárgyból milyen jegyet kapott, a séma kulcsa a Neptun és a Tárgykód együtt.)
    - Fejezze ki sorkalkulussal azokat a tárgykódokat, amely tárgyakból csak olyan diákok szereztek jegyet, akik legalább egy tárgyból szereztek legalább elégségest.
    - Adjon SQL lekérdezést, ami kilistázza azokat a tárgykódokat, amely tárgyakból csak olyan diákok szereztek jegyet, akiknek az (összes szerzett jegyük) átlaga nagyobb, mint 4.

22. Legyenek $R(A,B,C,D)$ és $S(C,D,E)$ alaprelációk és legyenek $\pi_{AC}(\sigma_{E=2}(R \triangleright\!\triangleleft S))$, illetve $\pi_{CD}(R) \cap \pi_{CD}(S)$ belőlük képzett leszármaztatott relációk. Fejezze ki ez utóbbiakat sorkalkulussal, oszlopkalkulussal!

23. Vizsgálja meg, hogy biztonságos-e a sorkalkulus. Minden változó doménje a természetes számok halmaza, $alma^1 = \{2,3\}$.
$\{x^{(1)} \mid (\exists t^{(1)})x^{(1)}[1] = t^{(1)}[1] \wedge t^{(1)}[1] > 2 \wedge alma^{(1)}(x^{(1)})\}$.

24. Az R séma attribútumai (A,B,C,D,E), az S séma attribútumai pedig (A,B,F,G). Fejezze ki oszlopkalkulus segítségével $R \triangleright\!\triangleleft S$ -et!

25. Az R reláció attribútumai (A,B,C,D), az S-é pedig (C,D). Ekkor R÷S, R és S hányadosa, azon (A,B) attribútumú t sorokból áll, melyekre igaz hogy bármely S-beli s sor esetén a ts sor R-ben van. Fejezze ki R÷S-t sorkalkulus segítségével! Feltehetjük, hogy S nem üres.

26. Adott az alábbi relációs adatbázis: GYÁRT(CÉG, TÍPUS, ÁR), SZERETI(VEVŐ, TÍPUS), DOLGOZIK(VEVŐ, CÉG, BEOSZTÁS).
    A relációk jelentése rendre: azon autóTÍPUSok, amiket egy CÉG gyárt az azok eladási ÁRa; egy VEVŐ melyik autóTÍPUS-t szereti; a VEVŐ melyik CÉGnél dolgozik, milyen BEOSZTÁSban.
    - Adjon meg egy sorkalkulus kifejezést, amely azokat a vevőket tartalmazó relációt állítja elő, akik olyan cégnél dolgoznak, amelyek gyártanak olyan autótípust, amilyet a vevő szeret!
    - Vizsgálja meg, hogy a fent leírt kifejezés biztonságos-e!

Objektum orientált adatmodell

27. Az objektum orientált adatmodellnél megismert típuskonstruktorok segítségével készítse el az egyszerűsített „újság" típust! Tárolnunk kell az újság címét, a főszerkesztőt, a helyettes főszerkesztőt (sorrendhelyesen), a kiadót, a rovatokat a rovatvezetőkkel és rovatújságírókkal.

Fizikai szervezés

28. Milyen módszerekkel támogatható a több kulcs szerinti keresés?

29. Egy 25.000 rekordból álló állományt szeretnénk ritka index (ISAM) szervezéssel tárolni. A rekordhossz 850 byte, egy blokk kapacitása (a fejrészt nem számítva) 4000 byte. A kulcs 50 byte-os, egy mutatóhoz 18 byte kell.
    - Legalább hány blokkra van szükség a teljes struktúra tárolásához?
    - Mennyi ideig tart legfeljebb egy rekord tartalmának kiolvasása, ha az operatív tárban rendelkezésünkre álló szabad hely 6000 byte? (egy blokkművelet ideje 5msec)
    - Segít-e a rekordhozzáférési idő csökkentésében, ha 10-szer (100-szor) ennyi szabad memóriával gazdálkodunk? Hogyan célszerű a többletmemóriát felhasználni?

30. Egy 15525 rekordból álló állományt szeretnénk ritka index (ISAM) szervezéssel tárolni. A rekordhossz 850 byte, egy blokk kapacitása (a fejrészt nem számítva) 4000 byte. A kulcs 50 byte-os, egy mutatóhoz 18 byte kell.
    - Legalább hány blokkra van szükség?
    - Mennyi ideig tart legfeljebb egy rekord tartalmának kiolvasása, ha az operatív tárban rendelkezésünkre álló szabad hely 5000 byte? Segít-e a legnagyobb rekordhozzáférési ido csökkentésében, ha 10-szer (100-szor) ennyi szabad memóriával gazdálkodhatunk?

31. Egy állományt sűrű index, majd erre épített egyszintes ritka index segítségével szeretnénk tárolni. Adjon értelmes alsó becslést a szükséges blokkok számára az alábbi feltételek mellett:
    - az állomány $3 \cdot 10^6$ rekordból áll,
    - egy rekord hossza 300 byte
    - egy blokk mérete 1000 byte
    - a kulcshossz 45 byte
    - egy mutató hossza 5 byte

32. Egy 270 000 rekordból álló állományt akarunk tárolni. Két lehetőség közül választhatunk: vagy sűrű indexre épített 1 szintes ritka indexet használunk, vagy 3 szintes ritka indexet. Melyik megoldást lehet kevesebb lap felhasználásával megvalóstani, ha még azt is el szeretnénk érni, hogy sem az indexállományban, sem a főállományban ne legyenek 80 %-nál telítettebb lapok? Tudjuk hogy egy lap mérete 1900 byte, egy rekord hossza 300 byte, a kulcs hossza 35 byte, a mutató hossza pedig 15 byte.

33. Egy adatbázisban egymilliárd rekordot akarunk tárolni. Egy rekord mérete 100 byte, a blokkméret 4000 byte. Egy blokkművelet 5 msec hosszú. Ket kulcs van, mindkettő 10 byte-os. A mutatók 32 bitesek. Az egyszerűség kedvéért feltételezzük, hogy egyszerre csak egy blokk fér el a memóriában.
    - Javasoljon tárolási módszert, ha mindkét kulcs szerint akarunk majd keresni úgy, hogy a keresés maximum 40 ms-t vegyen igénybe. A módszernek támogatnia kell az intervallumkeresést is. Készítsen magyarázó ábrát!
    - Egy konkret keresés a rekordok várhatóan 8%-at adja eredményül. Adjon minél hatékonyabb módszert a keresésre!

34. Vödrös hash alkalmazása esetén mit szükséges módosítani az adattároló struktúrán úgy, hogy az adatelérési idő megfeleződjön?

35. Egy adatstruktúrában hash alapú tárolást építünk ki egy CD lemezeket tároló adatbázisban. Minden CD-ről eltároljuk, hogy képeket, zenéket, videót vagy adatot tárol. Mindezt egy karakter típusú mezőben: K, Z, V, A. Milyen hash fv-t célszerű választani, ha ezen mezőre szeretnénk alapozni a hash tárolást? Mi a mező kardinalitása, mi lesz a doménje?

36. Egy adatbázisban szeretnénk 1.000.000 rekordot tárolni vödrös hash szervezéssel. 1 rekord mérete 110 byte, 1 blokk 3000 byte 1 kulcs 25 byte, 1 mutató pedig 64 bit méretű. A rekordelérési idő max. 20 msec, a blokkelérés 5 msec. A vödörkatalógus befér a memóriába, a hash függvény egyenletesen szór.
    - Mennyi az átlagos rekordelérési idő?
    - A vödörkatalógus hány byte-ot foglal el a memóriában?
    - Mennyi többletmemóriára lenne szükség, hogy a rekordelérési idő a felére csökkenjen?

Funkcionális függések

37. Bizonyítsa be, hogy az alábbi három szabályból következnek az Armstrong axiómák. (Azaz csak ezen három szabályt használva axiómaként levezethetők az Armstrong axiómák mint „tételek".)
    Ha X,Y,Z,C egy relációséma attribútumhalmazai, akkor:
    - $X \to X$
    - $X \to YZ$ és $Z \to C$ akkor $X \to YZC$
    - $X \to YZ$ akkor $X \to Y$.

38. Mutassa meg, hogy igaz a tranzitivitási axióma!

39. Bizonyítsa be a bővítési axiómát!

40. Igaz-e, hogy a következő axiómarendszer teljes, azaz levezethető-e felhasználásukkal minden logikai következmény?
    - Ha $X \subseteq R$ akkor $X \to X$.
    - Ha $X,Y \subseteq R$ és $X \to Y$, akkor $XW \to YW$ igaz tetszőleges $W \subseteq R$-re.
    - Ha $X,Y,Z \subseteq R$, $X \to Y$ és $Y \to Z$, akkor $X \to Z$.

41. Adjon egy $R(A,B,C)$ sémára illeszkedő r relációt, melynek 4 sora van és nem teljesül rá semmilyen nemtriviális funkcionális függés!

42. Adott egy (R,F) séma, ahol $R = ABCGWXYZ$ és
    $F = \{XZ \to BGYZ; AY \to CG; C \to W; B \to G\}$.
    Adja megy F -nek egy minimális fedését! Igaz-e, hogy $AXZ \to BY \in F^+$?

43. Igazak-e az alábbi szabályok? (A,B,C,D tetszőleges attribútumhalmazok egy R sémán.)
    - $A \to B,C \to D \Rightarrow A \cup (C-B) \to BD$,
    - $A \to B,C \to D \Rightarrow C \cup (D-A) \to BD$.

44. Adott egy $R(A,B,C)$ sémára illeszkedő r reláció, melynek 3 sora van. Bizonyítsa be, hogy meg lehet adni olyan nemtriviális funkcionális függést, amit r kielégít!

Normál formák

45. Mutassa meg, hogy egy 2NF sémára illeszkedő reláció is lehet redundáns. Magyarázza el, hogyan lehet megszüntetni. Adjon példát legalább 3 elemű 2NF sémára illeszkedő relációra, mely nem redundáns.

46. Bizonyítsa be, hogy F és G függéshalmaz pontosan akkor ekvivalens, ha $F \subseteq G^+$ és $G \subseteq F^+$!

47. Hányadik legmagasabb normál formában van az $R(A,B,C,D)$ relációs séma, ha $F = \{C \to B; B \to D; AB \to AC; CD \to B\}$?

48. Vizsgálja meg, hogy hányadik legmagasabb normál formában van az R(I,S,T,Q) relációs séma az $F = \{I \to Q; ST \to Q; IS \to T; QS \to I\}$ függéshalmaz esetén!

49. Bizonyítsa be, hogy ha az R relációs séma nem BCNF, akkor $\exists A,B$, hogy $A,B \in R$ és $R - AB \to A$!

50. Állapítsa meg, hogy tartalmazhat-e redundanciát (funkcionális függések következtében) az (R,F) sémára illeszkedő reláció, és ha igen, akkor milyen jellegűt? R(X,Y,Z,W), $F = \{XY \to Z, YZ \to W, X \to W, WY \to X\}$.

Relációs sémafelbontás

51. Adott egy (R,F) séma, ahol $R = (A,B,C,D,E)$ és $F = \{AB \to C; D \to A; AE \to B; CD \to E; BE \to D\}$.
    - BCNF-ben van-e ez a séma?
    - Ha igen bizonyítsa be, ha nem, akkor adja meg a séma egy veszteségmentes felbontását BCNF sémákra!

52. Adott az $R(L,M,N,O,P)$ relációs séma és a séma attribútumain értelmezett $F = \{MOP \to L, LN \to ON, NO \to M, OP \to N, PN \to LP\}$ funkcionális függéshalmaz. Az R séma egy veszteségmentes felbontását szeretnénk elkészíteni tetszőleges formában, de nemtriviális módon úgy, hogy az egyik részséma $R(L,M,O)$ legyen.

53. Adott az $R(A,B,C,D,E,F)$ relációs séma és az $F = \{A \to B, AC \to DB, C \to AD, AF \to ECB\}$, csak funkcionális függőségeket

tartalmazó függéshalmaz. Adja meg a séma egy veszteségmentes, függőségőrző felbontását 2NF sémákba, törekedve minél kevesebb relációs séma definiálására!

54. Adott az $R(G,H,J,K,L)$ séma. Adjon egy veszteségmentes, függőségőrző felbontást 3NF részsémákra, ha az ismert funkcionális függések halmaza $F=\{HJ \to J, GH \to IJ, HI \to JG, G \to J\}$!

55. Vizsgálja megy, hogy lehet-e az $S(L,M)$ relációs séma része az $R(L,M,N,O)$ relációs séma valamely veszteségmentes felbontásának az $F=\{MN \to O; NO \to L; N \to M; M \to N\}$ függéshalmaz esetén!

56. Adott a következő relációs séma: $R(A,B,C,D,E)$, $F=\{AB \to C, A \to E, B \to D\}$. Adja meg R egy veszteségmentes felbontását BCNF részsémákra!

57. Vizsgálja meg, hogy az $R(A,B,C,D,E,F,G)$ relációs séma σ=(ACEFG, BCDE) felbontása az $F=\{AB \to C, AC \to D, C \to F, D \to B, E \to G\}$ függéshalmaz mellett veszteségmentes-e!

58. Tekintsük a következő (R,F) sémát, akol $R=(A,B,C,D,E)$ és $F=\{B \to E; E \to A; A \to D; D \to E\}$.
Igaz-e, hogy a $\rho=(AB,BCD,ADE)$ felbontás veszteségmentes?

59. Bizonyítsa be, hogy egy reláció tetszőleges vertikális felbontása után a részrelációknak a természetes illesztésével sorok nem tűnhetnek el, csak újak jelenhetnek meg.

60. Van egy relációnk, és annak egy nem függőségőrző felbontása. Ha a felbontásban az egyik relációhoz hozzáadunk egy új elemet, melyen nem érvényesül(nek) az „elveszett" függőség(ek), akkor a relációba egy helytelen elem kerülhet. Ezután ha vesszük a felbontásban szereplő relációk természetes illesztését, akkor az eredeti relációnál bővebb relációt kapunk, tehát a nem függőségőrző felbontás nem veszteségmentes. Hol a hiba a gondolatmenetben?

61. Vizsgálja meg, hogy igaz-e a következő állítás: minden r(R,F)-re $\pi_{R_1}(r) \triangleright\triangleleft \pi_{R_2}(r) \triangleright\triangleleft \pi_{R_3}(r) = r$, ahol $R(A,B,C,D,E,H,G)$ relációs séma, $F=\{AB \to C, AC \to D, C \to H, D \to B, E \to G\}$ függéshalmaz és $\sigma=\{R_1(ACE), R_2(EHG), R_3(BCDE)\}$ egy sémafelbontás.

Tranzakciókezelés

62. Tekintsük a T1, T2, T3, T4 tranzakciók alábbi ütemezését:
T2: RLOCK A; T3: RLOCK A; T2: WLOCK B; T2: UNLOCK A; T3: WLOCK A; T2: UNLOCK B; T1: RLOCK B; T3: UNLOCK A; T4: RLOCK B; T1: RLOCK A; T4: UNLOCK B; T1: WLOCK C; T1: UNLOCK A; T4: WLOCK A; T4: UNLOCK A; T1: UNLOCK B; T1: UNLOCK C.

Rajzolja meg az ütemezés precedencia gráfját, és döntse el, hogy az ütemezés sorosítható-e!

63. Rajzolja fel a várakozási és a precedencia gráfot. Használjon zárakat. Hogyan változnak a gráfok, ha 2 fázisú a rendszer?

$T1$	$T2$
	WriteA
WriteB	
	WriteB
WriteA	

64. Mi történik a redo protokoll szerint, ha a tranzakció a bejelölt (1-6) pontokban abortál?

	naplóba(T,BEGIN)
1	
	LOCK(A);
	LOCK(B);
2	
	naplóba(T,<A régi értéke>,<A új értéke>);
	naplóba(T,<B régi értéke>,<B új értéke>);
3	
	naplóba(T,COMMIT);
4	
	write(A);
	write(B);
5	
	UNLOCK(A);
	UNLOCK(B);
6	

65. A következő tranzakció szigorú 2PL? Ha nem, módosítsa úgy, hogy az legyen! Mit biztosít ez a protokoll?

```
Lock A
Read A
A :=A*2
Write A
Commit
Unlock A
```

66. Miért lehet előnyös zárakat is használni időbélyeges tranzakció kezelésnél?

Készült a 2005. őszi előadástematika nyomán.

Források:

- Példafeladatok - http://www.inf.bme.hu/~pts/pelda.pdf
- Info Site - http://info.sch.bme.hu

Függelék: Szemistrukturált adatok[9]

A félig strukturált, vagy más szóval szemistrukturált adatok az élet minden területén jelen vannak. A Weben oly gyakori HTML és XML oldalak túlnyomó többsége is ebbe a kategóriába tartozik.

Mitől szemistrukturált egy adat?

A félig strukturált, avagy szemistrukturált adatok fogalmának megértéséhez először tisztázni kell, mit is értünk strukturált, ill. strukturálatlan adatok alatt. Az itt közölt definíciók természetesen nem általános érvényűek, de a továbbiak megértéséhez elégségesek.

Strukturált adatnak olyan adatokat tekintünk, melyeknél a *szintaxis*, azaz az adatok ábrázolása megfelel az adatok alkalmazása során felhasznált *szemantikájának*, azaz a jelentésüknek. Mivel az adatok szemantikáját a feldolgozás előtt előre ismernünk kell, ilyen esetben lehetőség van stabil, az adatok szerkezetét jól tükröző sémák előzetes definíciójára, amely az összes rendelkezésre álló, és jövőben megjelenő adat szintaktikáját leírja. Ilyen típusú adatok hatékonyan jellemezhetők a jelenleg széles körben használt relációs és objektum-orientált adatmodellekkel, ill. hatékonyan kezelhetők az ezekre a modellekre épülő adatbáziskezelő-rendszerekben. A relációs és objektum-orientált modelleket ezentúl mint „hagyományos adatmodelleket", a rájuk épülő rendszereket mint „hagyományos adatbázis-kezelőket" fogjuk emlegetni.

Strukturálatlan adatoknak olyan adatokat tekintünk, melyeknek az aktuális alkalmazás szempontjából semmilyen használható szemantikája és emiatt felismerhető szerkezete nincs. Ezek alapján a *szemistrukturált adatok* olyan adatok, melyek az aktuális alkalmazás szempontjából hordoznak ugyan értékes szemantikus információt, de a reprezentációjuk, struktúrájuk eltér a hasznos szemantikus jelentés által meghatározottól.

Lássunk néhány példát arra - a teljesség igénye nélkül -, hogy mely tulajdonságok árulkodnak a szemantikus jelentés és a struktúra eltéréséről, és így jellemzőek a szemistrukturált adatokra:

- *Az adatok struktúrája szabálytalan:* Az általános struktúrától igen sok adatelem eltér, különböző formában. Külön említést érdemel az az eset, amikor többlet elem jelenik meg, hiszen ez egy hagyományos adatbázisban olyan sémaelemet indukálna, amely a legtöbb adatelem esetében csak üres értéket tartalmazna. Amennyiben ilyen eltérő adatelemből viszonylag sok van, az eredményül kapott adatbázis nagy részben csak üres elemeket tartalmazna. Másik eltérés lehet, ha az adatok típusa változik, pl. egy lakcím egyszer egyszerű karakterlánc, máskor pedig egy struktúra (utca, házszám, irányítószám). Ebben az esetben is új elemek felvételére kényszerülnénk hagyományos adatmodellek esetén (hiszen egy adatmező csak egy típussal rendelkezhet), ami a sémát súlyosan összezavarná.

[9] Megjelent: Gajdos-Nagypál: Hálószemantika címmel az InfoByte Magazin 1. számában (2001. december)

- *Implicit struktúra:* A struktúra definíciója nem, vagy nem teljes egészében található meg az adatforrásban, azt részben vagy teljes egészében nekünk kell kinyerni az adatokból. Pl. egy HTML-oldal tartalmaz ún. tag-eket, amelyek biztosítanak valamilyen struktúrát a dokumentumnak, ez a struktúra mégis sokszor csak részlegesen fedi a dokumentum logikai felépítését.
- *Részleges struktúra:* Szemistrukturált dokumentumok sokszor tartalmaznak olyan részeket, amelyek egy adott nézőpontból tekintve nem strukturálhatók (pl. képek egy HTML oldalon, ha a szöveges információkat szeretnénk feldolgozni). Olyan részei is lehetnek az adathalmaznak, melyeket szándékosan nem is akarunk tovább strukturálni (pl. egy szöveges termékleírás egy katalógusban).
- *Csak a-posteriori sémainformáció áll a rendelkezésünkre:* Míg a hagyományos adatbáziskezelő rendszereknél az adatok struktúrája, típusa az adatbázissémában előre rögzített, és gondoskodunk róla, hogy az új adatok ennek az előre rögzített sémának pontosan megfeleljenek, addig szemistrukturált adatok esetében sok esetben csak az adatok adatbázisba való betöltése után lehet valamilyen sémainformációt kinyerni.
- *A szemistrukturált séma nem azonos a hagyományos adatbáziskezelő rendszerekben használatos sémával,* több olyan tulajdonsága is lehet, amely a hagyományos adatbáziskezelő rendszereket alkalmatlanná teszi ilyen típusú adatok kezelésére. Néhány példa ezekre:
  - Az adatok nagy változékonysága miatt a séma mérete igen nagy is lehet. Így nem tételezhető fel, hogy a felhasználó a lekérdezés megfogalmazásánál ismeri a sémát. Sőt, *a séma lekérdezésére is eszközöket kell biztosítani.*
  - Amennyiben a séma nem előre definiált, hanem csak a mindenkori adatokból következtetünk rá, maga a *séma is igen változékony* lesz.
  - Amennyiben a szemistrukturált séma előre ismert, akkor is csak *laza kényszereket határoz meg az adatokra nézve*, azaz opcionális és alternatív adatelemek is előfordulhatnak. Az adattípusok kezelése sem olyan szigorú, mint a hagyományos esetekben.
- *Gyakori, hogy az adatokat a felhasználók a sémainformációtól függetlenül csak böngészni szeretnék,* ellentétben a hagyományos esettel, amikor az adatokat csak a séma ismeretében, lekérdezések útján kaphatjuk meg a rendszertől.

Fontos, hogy nem kell az összes felsorolt tulajdonságnak teljesülnie ahhoz, hogy egy adat kiérdemelje a szemistrukturált jelzőt, noha az összes tulajdonság egyidejű megléte sem kizárt. Például egy BibTeX adatbázis (mely bibliográfiai adatokat tartalmaz) vagy egy egyszerű DTD-vel leírható XML állomány, melyeket szemistrukturált dokumentumoknak szoktak tekinteni, csak a definíció néhány elemét elégítik ki, hiszen egy, a lehetséges adatokra csak laza megkötéseket alkalmazó séma leírja a dokumentumokat, így a séma nem implicit és nem csak a-posteriori ismert. Olyan szemistrukturált adatra, amelyet az összes felsorolt tulajdonság jellemez, példa lehet egy HTML dokumentum, amely a mindenkori heti tévéműsort tartalmazza. Ez esetben a nyilvánvalóan jelenlevő, noha nem teljesen szabályos struktúra csak implicit, automatikus eszközökkel nehezen felfedezhető.

Fontos megjegyezni azt is, hogy az, hogy mi tekinthető strukturált, szemistrukturált vagy akár teljesen strukturálatlan adatnak, nézőpont kérdése. Így az olyan adat, amely bizonyos szempontból szemistrukturált, vagy éppen strukturálatlan, más szempontból

elképzelhető, hogy strukturáltnak bizonyul, mert az aktuális alkalmazási terület számára hasznos szemantika által meghatározott struktúra éppen megegyezik az adatok reprezentációja által meghatározott szerkezettel.

Hol találhatók szemistrukturált adatok?

Vajon a valóságban is léteznek nagy mennyiségben szemistrukturált adatok? Ha igen, akkor hogyan keletkeznek?

Általánosságban elmondható, hogy a szemistrukturáltnak tekinthető adatok a keletkezésük szempontjából lényegében két csoportba sorolhatók. Az első, ún. *dokumentumközpontú* kategóriába olyan, elsősorban emberi fogyasztásra készült dokumentumok tartoznak, amelyek valamilyen szinten strukturált információt hordoznak. Jó példa egy táblázatot tartalmazó HTML oldal, esetleg egy Word dokumentum. Itt a legfontosabb tulajdonság a struktúra implicit volta, amelyet legtöbbször csak utólag, az adatok feldolgozása után lehet kinyerni az adathalmazból.

A másik, ún. *adatközpontú* kategóriába olyan adatok tartoznak, amelyek független adatforrások egyesítésekor, ill. független adatforrások közti adatcsere során keletkeznek. Adatforrások integrációjakor igen kényelmes olyan adatmodellben gondolkodni, amely nem igényli egy előre meghatározott, és – ami talán még fontosabb – részletesen kidolgozott séma meglétét. Ez különösen igaz arra az esetre, ha nem előre ismert számú és tulajdonságú adatforrásról van szó, vagy az adatforrások ugyan ismertek, de a nagy számuk miatt egy, az összes adatforrás lényeges részét egyszerre leíró séma megalkotása túl nagy erőfeszítésbe kerülne. Adatforrások integrációjakor a szemistrukturált adatok tulajdonságai közül gyakran a legfontosabb az, hogy a szemistrukturált séma csak lazán, vázlatosan írja le az adatokat, ami a legtöbb esetben elegendő arra, hogy a felhasználók dolgozni tudjanak az integrált adathalmazzal.

Amennyiben szemistrukturált adatokat információs rendszerek közötti adatcserére használunk, tipikus a részletes, jól definiált séma, és ebben az esetben a szemistrukturált adatformátum használata annyiban indokolható, hogy egy közös platformot, egy „lingua franca"-t teremt, amely általánossága miatt minden környezetben használható (ld. pl. alkalmazásintegráció üzenetorientált middleware-ekkel).

A szemistrukturált adatok hőskora

A szemistrukturált adatokkal kapcsolatos kutatások a kilencvenes évek elején kezdődtek meg. A kutatást azt tette szükségessé, hogy a szemistrukturált adatok néhány tulajdonsága olyan követelményeket támasztott az adatbázis-kezelőkkel szemben, amelyeknek a hagyományos adatbáziskezelő-rendszerek nem (vagy nem hatékonyan) tudtak megfelelni. Néhány példa:

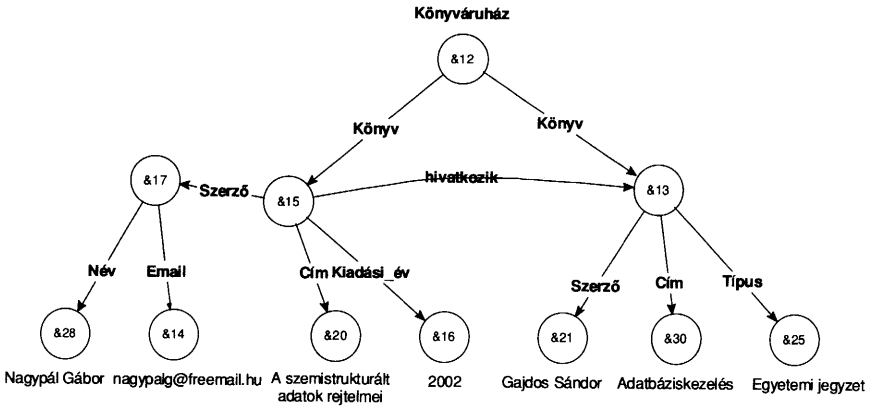
- a séma változékonyságának, ill. lazaságának tolerálása
- a séma lekérdezésének biztosítása
- az adatok szabad böngészésének biztosítása

Új adatmodelleket, új adatmanipulációs nyelveket és ezek alapján működő adatbáziskezelőket kellett alkotni, melyek képesek voltak a fenti követelményeknek megfelelni. A kutatások eredményeképpen megszületett néhány adatmodell, melyek lényegében azonos ötleten alapultak: a modell egy gráf, melynek bizonyos elemeit címkékkel látjuk el. Az adatmodellek közül a legismertebb és legsikeresebb az *OEM* (Objext Exchange Model – objektum adatcsere modell) [1], a hozzá kifejlesztett *LOREL* adatmanipulációs nyelv [2] illetve *LORE* adatbáziskezelő [3] lett.

Az „adatmodell" kifejezés használatával kapcsolatban egy fontos megjegyzést kell tennünk. Mint az jól ismert, egy adatmodell hagyományosan két részből áll: egy formalizált jelölésrendszerből adatok, adatkapcsolatok leírására, valamint az adatokon végrehajtható műveletekből. A szemistrukturált adatok esetében a fő hangsúly mindig a formális reprezentáció megalkotásán van, és az adatokon végezhető műveletek rögzítése sok esetben még nem történt meg, ill. egy adott formális reprezentációhoz több, különböző műveletha lmaz is tartozhat, különböző lekérdező erejű adatmanipulációs nyelvek formájában. Egy, a gráf éleinek és csúcsainak törlését ill. beszúrását megvalósító műveletha lmaz természetesen minden esetben elképzelhető. Cikkünkben az egyes modellek formális reprezentációjának bemutatására koncentrálunk, és a műveleteket elhanyagoljuk, elsősorban azért, mert sok esetben (pl. a bemutatásra kerülő XML és RDF esetén is) az elérhető műveletek köre még jelenleg is változik, ahogy új és új adatmanipulációs nyelvek születnek. Ettől függetlenül jogosnak érezzük az adatmodell szó használatát, hiszen az ismertetett formális reprezentációk mindig csak egy konkrét műveleti halmazzal, egy konkrét adatmanipulációs nyelvvel együtt használhatók, így a valós életbeli alkalmazás során mindig egy teljes értékű adatmodellel találkozunk.

Az OEM modell lényege, hogy az adatokat objektumokként fogjuk fel: egy objektum vagy egy konstans érték, vagy további objektumok halmaza, ahol az objektumok halmazbeli szerepét (ez a hagyományos modellek esetén az attribútumoknak megfelelő fogalom) egy (beszédes) címkével adjuk meg. A konstansoknak típusleíró információt kell adnunk, de ez ténylegesen csak leíró információ, nem az ellenőrzést szolgáló eszköz. Például új típust bármikor létrehozhatunk azáltal, hogy egy „attribútum" típusának olyat adunk meg, amit eddig még nem használtunk. Minden objektumhoz egyedi azonosító tartozik, az objektum-orientált szemléletnek megfelelően.

Szemléletesen egy OEM adatbázis egy irányított gráfnak tekinthető, ahol az objektumok a gráf csúcsai, míg a címkék/attribútumok a gráf élei, és a konstans értéket tartalmazó csúcsokból már nem indul ki él. Így egyrészt egy nagyon flexibilis, másrészt egy önmagát leíró (self-describing) modellt kapunk, melynél a séma információ az adatokkal együtt tárolódik, így az is lekérdezhető. Esetünkben sémainformációnak az élek címkéinek összességét, valamint az egyes csomópontoknál megadott típusleírásokat tekinthetjük, adatnak pedig a csomópontokban levő értékeket. Azonban látható, hogy a sémainformáció és az adatok nem válnak el egymástól élesen, ami a szemistrukturált modellek ill. adatok egyik legjellegzetesebb tulajdonsága. Az 1. ábrán egy képzeletbeli könyváruház OEM modelljének részlete látható.



1. Ábra: Könyváruház OEM modellje

A LOREL nyelv szintaxisa hasonlít a jól ismert SQL nyelvéhez, és az OEM által meghatározott absztrakt gráfban történő navigálást teszi lehetővé.

Az OEM adatmodellhez egyáltalán nem kapcsolódik sémaleíró nyelv, noha lehetőség van ún. adattérképek (data guide) generálására, amelyek segíthetnek a felhasználóknak eligazodni egy OEM alapú adatbázis struktúrájában. Fontos megemlíteni azonban, hogy ez az adattérkép mindig dinamikusan jön létre az adatbázis mindenkori tartalma alapján, azaz csak dokumentációs célokat szolgál, semmiképpen sem az újonnan érkező adatok formátumának meghatározását, mint azt a hagyományos sémáknál megszoktuk! Egy adattérkép maga is egy OEM gráf, amely pontosan és tömören leírja az adatbázis gráfban aktuálisan előforduló utakat, ezzel segítve a felhasználó navigálását. A *pontosság* itt azt jelenti, hogy minden, az adatbázisban bejárható út megkapható az adattérkép gráfjának bejárásával, és fordítva, minden, az adattérképben szereplő út megtalálható az adatbázisban is. A *tömörség* pedig azt jelenti, hogy minden lehetséges út csak egyféleképpen járható be az adattérképen.

A legelterjedtebb szemistrukturált formátum: az XML

Napjainkban a szemistrukturált adatok egyik legnépszerűbb megjelenési formája az XML dokumentum. Adatmodellezési szempontból egy XML állomány az OEM modellhez hasonlóan egy irányított gráfot ír le. A gráf csomópontjai az XML elemek (element), melyek határát az XML dokumentumban a nyitó és záró címkék (tag-ek) jelölik ki. Ezekre az elemekre ezentúl csak objektumokként fogunk hivatkozni. A gráf élei egyrészt az objektumok közti tartalmazási relációt (azaz mely objektum melyik másiknak a belsejében található az XML dokumentumban), másrészt pedig az egyes elemek közti hivatkozásokat reprezentálják.

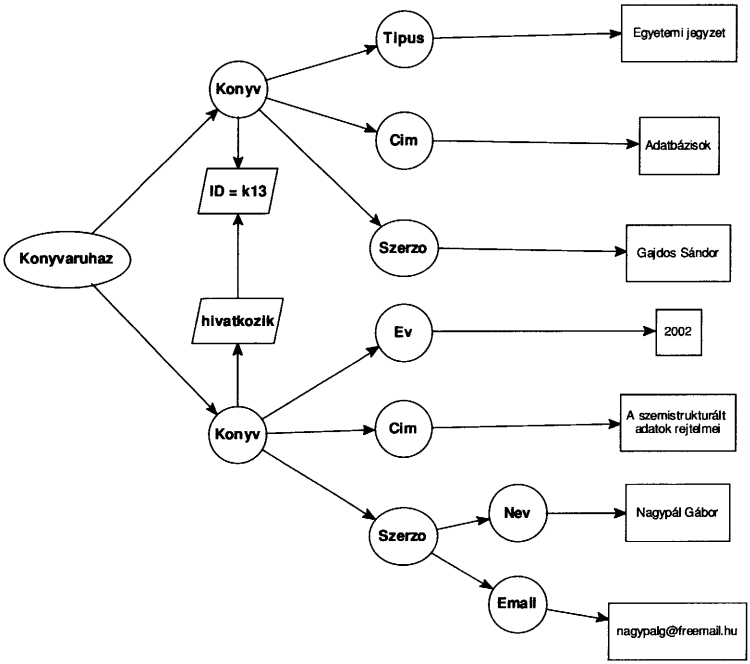
Jóllehet az XML formátum által meghatározott adatreprezentáció hasonló az OEM-nél látottakhoz, néhány fontos különbség is van. Az egyik közülük az, hogy XML esetén a csomópontok vannak névvel ellátva az élek helyett. Ennek egyik következménye, hogy egy adott entitás mindig csak egy néven szerepelhet a különböző kapcsolatokban. Míg az OEM modell esetén „Kovács János" lehet egyszer „apa", másszor pedig „vevő", XML esetében mindig döntenünk kell a szerepről, vagy

ugyanazt az adatot többször meg kell ismételnünk más és más név alatt. További különbség, hogy a csomópontok sorrendje fontos, a gráf élei mindig rendezettek az elemek dokumentumbeli előfordulási sorrendje szerint. Szemistrukturált szempontból ez a tulajdonság inkább zavaró, mint hasznos, ráadásul igény esetén könnyen szimulálható „1" „2" stb. címkéjű élek felvételével. Nem véletlen, hogy az XML Schema szabványban újra bevezették az SGML szabványban [18] már meglévő *„all"* konstrukciót (az XML az SGML egy részhalmaza), amellyel jelezni lehet az elemek sorrendjének érdektelenségét. Végezetül XML esetén maguk a csomópontok nem homogének, hiszen egy csomópont lehet egy „elem" (element), egy „attribútum" (attribute) vagy egy szövegmező (text). Az OEM modellnél ismertetett példa az 1. listán látható szöveges formában ill. a 2. ábrán grafikus alakban.

```
<Konyvaruhaz>
    <Konyv hivatkozik="k13">
        <Szerzo>
            <Nev>Nagypál Gábor</Nev>
            <Email>nagypalg@freemail.hu</Email>
        </Szerzo>
        <Cim>A szemistrukturált adatok rejtelmei</Cim>
        <Ev>2002</Ev>
    </Konyv>
    <Konyv ID="k13">
        <Szerzo>Gajdos Sándor</Szerzo>
        <Cim>Adatbázisok</Cim>
        <Tipus>Egyetemi jegyzet</Tipus>
    </Konyv>
</Konyvaruhaz>
```

1. Lista: A könyváruház példa XML forrása

XML adatok esetében – az OEM modellel ellentétben – lehetőségünk van sémával korlátozni a szóba jöhető adatok körét. A sémát számtalan módon megadhatjuk, a leggyakoribb az XML szabványban is szereplő *document type definition (DTD)* [4], és a nemrégiben W3C szabvánnyá előlépett *XML Schema* [5] használata. Ez utóbbival – ha kedvünk tartja – akár relációs adatmodelleket megszégyenítő alapossággal is leírhatjuk adatainkat, de ez távolról sem kötelező, a különböző XML séma nyelvek (a már említett XML DTD-n és az XML Schema-n kívül pl. az *XDR* [19] és a *Schematron* [20]) kiválóan alkalmasak szemistrukturált adatok leírására is. Ennek illusztrálására a 2. listán bemutatjuk a láthatóan szemistrukturált könyváruházas példa sémáját leíró DTD-t.



2. Ábra: A könyváruház XML modellje grafikusan

```
<!ELEMENT Konyvaruhaz (Konyv*)>
<!ELEMENT Konyv (Szerzo, Cim, Ev?, Tipus?)>
<!ATTLIST Konyv
    ID ID #IMPLIED
    hivatkozik IDREFS #IMPLIED
>
<!ELEMENT Szerzo (#PCDATA | Nev | Email)*>
<!ELEMENT Cim (#PCDATA)>
<!ELEMENT Ev (#PCDATA)>
<!ELEMENT Tipus (#PCDATA)>
<!ELEMENT Nev (#PCDATA)>
<!ELEMENT Email (#PCDATA)>
```

2. Lista: A könyváruház példa DTD sémája

Fontos megjegyezni, hogy bár XML-ben is lehetőség van tetszőleges gráf leírására, ez nehézkesebb, mint azt az OEM esetében bemutattuk. Egyik oka, hogy az egyes objektumok nem kapnak automatikusan egyedi azonosítót, ezek létrehozásáról magának a felhasználónak kell gondoskodnia, ha egy adott objektumra hivatkozni szeretne. Másrészt, hogy egy adott objektum éppen referenciát tartalmaz-e vagy csak egy egyszerű szövegfüzért, csak sémainformáció ismeretében lehet megállapítani, azaz gráfstruktúra esetén elveszítjük a séma nélküli feldolgozhatóságot. Ráadásként a legelső - és mindmáig népszerű - sémaleíró nyelv, a magában az XML szabványban található DTD igen korlátozott lehetőségeket nyújt hivatkozások definiálására. Csak speciális attribútumok (típusuk *IDREF* vagy *IDREFS*) hivatkozhatnak speciális *ID* típusú attribútumokra, tehát szó sincs arról, hogy bármely csomópont bármely másikra hivatkozhatna. Az újabb XML Schema szabványban ezt a hiányosságot már orvosolták, itt tetszőleges csomópont hivatkozhat tetszőleges másikra. A gráf struktúrákkal kapcsolatos nehézkesség fő oka abban keresendő, hogy az XML

formátumot eredetileg szöveges dokumentumok címkézésére (markup) találták ki, és szemistrukturált adatformátumként való használata csak melléktermeknek tekinthető.

Esetleg zavarhatja a megértést, hogy az XML Schema nyelvvel kifejezetten alapos, részletes sémák megadására is mód van. Jogos lehet az a kérdés, hogy miért szemistrukturált az ilyen részletes sémával meghatározott adat. A válasz az, hogy XML formátumban megadott adat lehet strukturált is, szemistrukturált is. Amennyiben az adatok jól meghatározott sémáját előre ismerjük, akkor a szóban forgó XML dokumentum nyilván nem tekinthető szemistrukturáltnak, hiszen a szemistrukturált adatok egyetlen egy tulajdonságát sem teljesíti. Természetesen minden szemistrukturált adatmodell alkalmas strukturált adatok tárolására is, más kérdés, hogy ilyen típusú adatokat általában érdemesebb a hagyományos módszerek szerint kezelni, hiszen azok erre a célra sokkal hatékonyabbak.

A szemistrukturált adatok kétféle nagy típusáról elmondottak természetesen XML dokumentumokra is igazak, itt is megkülönböztetünk adatközpontú és dokumentumközpontú XML állományokat. Adatközpontú XML dokumentumokat napjainkban elsősorban főleg e-business, e-commerce területén adatcserére használnak, gyakori azonban az XML formátum alkalmazása strukturált információforrások (és szemistrukturált információforrások, pl. egy Webes hírforrás) integrációjánál is.

Dokumentumközpontú XML dokumentumra jó példa egy DocBook [7] formátumban íródott felhasználói kézikönyv. Noha a dokumentum nagy része strukturálatlannak tekinthető (folyó szöveg), a megfelelő helyen elhelyezett információk a dokumentum struktúrájáról (bekezdés, fejezetcím) lehetővé teszik az automatikus formázást, valamint a kevés strukturált metainformáció megjelölése (pl. szerző, cím) lehetővé tesz néhány extra szolgáltatást, pl. pontosabb és gyorsabb keresést.

A jövő szemistrukturált formátuma, az RDF

A *Resource Description Framework (RDF)* [11] azaz „forrásleíró rendszer" névre hallgató modell, mint az eljövendő „szemantikus web" alapja, szintén egyfajta szemistrukturált adatmodellnek tekinthető. Az RDF alapvető célja az, hogy bármilyen *forrásról (resource)*, azaz olyan dologról, ami egyedi azonosítóval rendelkezik, egyszerű *állításokat (statement)* legyünk képesek tenni. A Weben az egyedi azonosító minden esetben valamilyen *URI*-t [8] jelent, és mivel az RDF elsősorban a Weben fellelhető dolgok leírására készült, itt sincs ez másként. A továbbiakban a könnyebb érthetőség kedvéért a forrásokat is csak objektumként fogjuk emlegetni.

Ahhoz, hogy az RDF modell értelmét és tulajdonságait megértsük, néhány szót kell szólni a „Szemantikus Web" (Semantic Web) [9] kezdeményezésről. „A Szemantikus Web egy új formája a Weben fellelhető tartalomnak, amely számítógépek számára is értelmes, és forradalmian új távlatokat nyit" jellemzi a fogalmat Tim Berners-Lee, a jelenlegi Web megalkotója a Scientific American hasábjain [10]. Más szavakkal a kezdeményezés célja egy olyan globális információs hálózat létrehozása, melynek tartalmát gépi intelligencia is képes megérteni, feldolgozni, átalakítani, és belőle következtetéseket levonni, azaz új tényeket alkotni. Mindezt a funkcionalitást a már jelenleg is rendelkezésre álló digitális aláírásokkal megtámogatva egy olyan rendszert kapnánk, amely automatikusan és megbízhatóan tudná levenni a vállunkról a legtöbb, jelenleg manuálisan végzett feladat terhét. Egy ilyen rendszerben lehetővé válna pl.

az, hogy az optimális nyaralási programot egy automatikus ügynök állítsa össze számunkra, vagy hogy a problémánk megoldására a legalkalmasabb szakértőt a személyes ügynökünk keresse meg, a Weben fellelhető adatokat automatikusan feldolgozva. Jelenleg ezek a feladatok csak hosszadalmas manuális keresgéléssel oldhatók meg, ha megoldhatók egyáltalán.

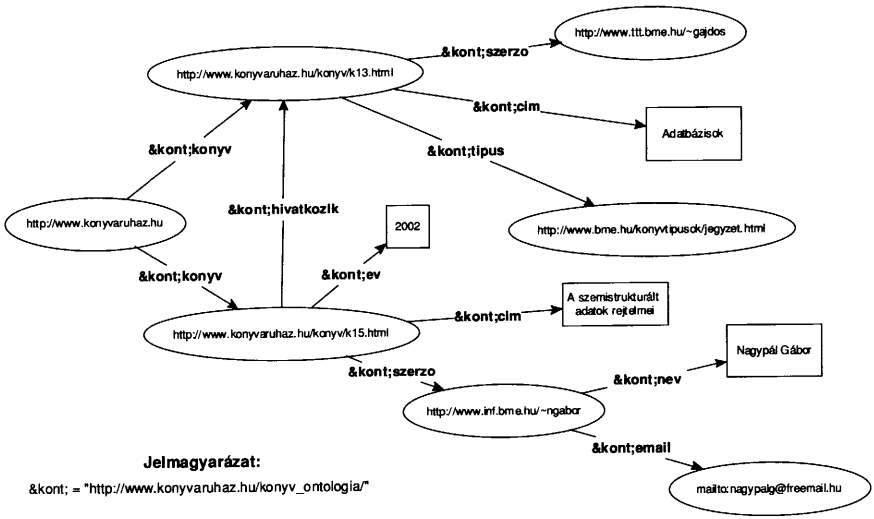
Egy ilyen, egyelőre csak álmainkban létező rendszer megvalósításához két út vezethet: vagy a gépi intelligencia színvonalát kell jelentősen növelni, vagy a rendelkezésre álló adatokat kell ellátni a megfelelő szemantikus információval. A Szemantikus Web létrehozói az utóbbi utat választották, és a jelenlegi munka ezen az úton halad. Az első állomás az RDF modell létrehozása volt. Jelenleg a sémaleíró nyelvek vannak soron, utána pedig sorban jönnek majd a különféle logikai funkcionalitást nyújtó rétegek. A lényeg az, hogy az RDF modellt elsősorban abból a célból hozták létre, hogy azonos szemantikájú adatokat lehetőleg csak egyféleképpen lehessen vele leírni, ezzel biztosítva megfelelő alapot a Szemantikus Web további funkcionalitása számára. Magáról a Szemantikus Webről további információk a [9], [10] Web oldalakon találhatók.

Logikai szempontból az RDF modell egyszerű, (állítás, alany, tárgy) hármassal leírható állítások halmaza, kifejező ereje még az egyszerű ítéletlogika szintjét sem éri el, hiszen nincs benne pl. negáció, csak pozitív állításokat tehetünk. Például a (név, http://people.com/kistvan.html, „Kovács István") hármas azt mondja ki, hogy a http://people.com/kistvan.html azonosítójú objektum neve „Kovács István".

Ha az RDF-t mint adatmodellt tekintjük, lényegében az OEM modellt kapjuk vissza, apró változtatásokkal. Az adatmodell itt is egy irányított gráfot ír le, ahol az egyedi azonosítóval rendelkező objektumokat az objektumok egyes *tulajdonságait* (*properties*) jelképező nyilak kötik össze. Egyszerű karakterfüzérek, azaz *literálok* (*literals*) is lehetnek a gráf csomópontjai, azonban ezekből már nem indulhat ki él. Fontos megemlíteni, hogy itt az egyes objektumok globálisan egyedi azonosítóval rendelkeznek, azaz ha két objektum URI-ja megegyezik, abból következik, hogy a két objektum azonos. További lényeges különbség az eddigi modellekkel szemben, hogy az RDF esetében maguk a tulajdonságok is objektumok, azaz egy egyszerű szöveges név helyett globális azonosítóval rendelkeznek. Ez két dolgot von maga után. Először is, maguk a tulajdonságok így globálissá válnak, azaz ha a http://www.konyvaruhaz.hu/konyvek/szerzo/nev tulajdonságot két helyen is használjuk ugyanabban az adatbázisban (ill. bármelyik adatbázisban), akkor biztosak lehetünk abban, hogy a két tulajdonság szemantikailag ugyanazt jelenti. Másodszor pedig magukról a tulajdonságokról (azaz a gráf éleiről) is tehetünk állításokat, ami OEM és XML esetén lehetetlen. Az eddigi példánk az RDF modellben leírva a 3. ábrán látható.

Az RDF adatmodellhez tartozó sémaleíró nyelvek még jelenleg is fejlesztés alatt vannak, legismertebb képviselőik az RDFS [12] ill. az erre épülő DAML+OIL [13]. Általánosságban elmondható, hogy az XML-től (és a hagyományos adatbázisoktól) eltérően a séma – amelyet a Szemantikus Web terminológiájában *ontológiának* neveznek – itt főként a szóba jöhető objektumok szemantikájának leírására szolgál, és kevésbé a szintaxis megszorítására. Azaz főképpen objektum és tulajdonság hierarchiákat alkotunk, és azt próbáljuk formálisan megragadni, hogy mi az a fogalom, hogy „apa", ahelyett, hogy azt próbálnánk meghatározni, hogy az „apa" objektum milyen formátumban fog megjelenni. Másként megfogalmazva: a szintaktika helyett a szemantikát írjuk le. Jellemző, hogy az adattípusok fogalma

először a DAML+OIL szintjén jelenik meg, ahol egyébként az XML Schema szabványban megjelenő adattípusokat [6] használták fel újra az alkotók.



3. Ábra: A könyváruház példa RDF modellje

Érdekesség, hogy az RDF világában legfontosabb elemmé a tulajdonságok, azaz a gráf élei lépnek elő, hiszen itt az a fontos, hogy már meglévő objektumokról akár az objektum létrehozójától teljesen függetlenül is tudjunk különböző állításokat tenni. Tipikus esetben az RDF leírás létrehozója csak a tulajdonságokat használhatja kreatívan, a leírandó objektumok, források már tőle függetlenül léteznek. Másik érdekesség, hogy míg a hagyományos és az XML sémáknál az elv az, hogy csak olyan adat megengedett, amit a séma maradéktalanul leír, addig az RDF esetében a filozófia az, hogy ami a sémában specifikálva van, annak úgy kell lennie a dokumentumban is, ami nincs, az pedig lehet bárhogy. Például attól, hogy meghatároztuk hogy a http://www.konyvaruhaz.hu/konyvek/szerzo/nev tulajdonság „ember" csomópontokból mutat karakterfüzér típusú csomópont felé, még nyugodtan használhat bárki egy http://www.konyvaruhaz.hu/konyvek/konyv/nev tulajdonságot is, noha erről említést sem tettunk a sémában. Ez a filozófia lehetővé teszi a már meglévő szabványos ontológiáknak a kiegészítését az eredeti szerzőtől függetlenül. Az elv hasonlít a HTML oldalak összekapcsolásánál alkalmazottra: egy adott oldalra bárki mutathat anélkül, hogy arról az oldal szerzőjének bármiféle tudomása lenne. Mint az a hagyományos Web világában már bebizonyosodott, ennek az egyszerű és bizonyíthatóan nem kevés hátulütőkkel rendelkező (pl. semmibe mutató linkek) megoldásnak köszönhető a Web viharos elterjedése. Az eddigi tapasztalatok alapján globális méretekben csak olyan rendszer lehet sikeres, amelyet a felhasználók egymástól teljesen függetlenül tudnak bővíteni.

Szemistrukturált adatok tárolása

Mint láttuk, a szemistrukturált adatoknak két nagy kategóriája létezik. Az első csoportba a csak megjelenésükben szemistrukturált, de valójában alapvetően strukturált információt hordozó adatok tartoznak. Esetükben – megfelelő szintaktikai

átalakítás után – semmi akadálya annak, hogy hagyományos adatbázis-kezelőkben tároljuk őket. Ennek megfelelően például a legtöbb modern relációs adatbáziskezelő már nyújt valamilyen szintű XML export/import funkcionalitást, és maga a téma is, hogy miképpen lehet egyre hatékonyabban és egyszerűbben XML formátumú adatokat tárolni és lekérdezni hagyományos (főként relációs) adatbázis-kezelőket használva, népszerű kutatási terület. Természetesen az XML-re kitalált technikák átültetése más szemistrukturált formátumban érkező adatra minimális erőfeszítéssel megtehető, a szemistrukturált modellek hasonlósága következtében. A hasonlóságot kiemelendő, példaképpen megemlítjük, hogy az RDF modellt adatcserénél leggyakrabban éppen XML formátumban szokás szerializálni. Az irodalomjegyzékben található [14] [15] források XML és RDF adatok relációs adatbázisokban való tárolási módjait, lehetőségeit elemzik.

A második kategóriába a ténylegesen szemistrukturált adatok tartoznak, és mint láttuk, ezek általában olyan dokumentumok, amelyek eredetileg emberi fogyasztásra készültek. Esetükben igaz a megállapítás, miszerint a hagyományos módszerekkel való kezelésük alacsony hatékonyságú. Itt egy gyakran használt naiv megoldás a szemistrukturált adatmodell szerkezeteinek megfelelő tárolási forma alkalmazása, azaz a gráf csomópontjainak és éleinek tárolása. Másik lehetséges megoldás natív szemistrukturált adatbázis-kezelők használata, melyek különböző, már meglévő hagyományos tárolási technikák (fájlrendszer, különböző indexek, relációs és objektum-orientált adatbázisok) alkotó ötvözésén alapulnak. Jó példa ilyen rendszerekre az OEM modellen alapuló LORE [3] és a Tamino [16] nevű XML adatbázis-kezelők. A [17] oldalon pedig további részletes információk találhatók különböző natív XML adatbázis-kezelőkről. Általánosságban elmondható, hogy a natív szemistrukturált adatbázis-kezelők valódi szemistrukturált adatok esetében jobb teljesítményt nyújtanak hagyományos társaiknál, de az alapvetően strukturált adatok kezelésére továbbra is a hagyományos adatbáziskezelőket érdemes alkalmazni, amennyiben ez a struktúra könnyen felderíthető és stabil.

Konklúzió

Cikkünkben a szemistrukturált adatok néhány jellemző vonásának felvillantása után bemutattuk, hogy adatkezelési szempontból miben különböznek a népszerűbb szemistrukturált adatmodellek a hagyományos modellektől, valamint egymástól.

A szemistrukturált adatok az élet minden területén jelen vannak, és szerepük várhatóan egyre nőni fog. A Web következő generációja a jelenlegihez hasonlóan a szemistrukturált adatok végtelen tárháza lesz. Ugyanakkor a jelenleg is dinamikusan fejlődő e-business megoldásokhoz is elengedhetetlen a szemistrukturált adatformátumok alkalmazása, az egyre fontosabb együttműködési követelmények miatt.

Források a függelékhez

[1] *Y. Papakonstantinou, H. Garcia-Molina, J. Widom:* **Object Exchange Across Heterogeneous Information Sources.** In Proceedings of the Eleventh International Conference on Data Engineering p.251-60, 1995, URL: http://dbpubs.stanford.edu/pub/1995-6

[2] *S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. L. Wiener:* **The Lorel Query Language for Semistructured Data.** In Journal of DigitalLibraries volume 1:1, 1997, URL: http://dbpubs.stanford.edu/pub/1996-35

[3] **LORE adatbáziskezelő projekt honlapja**, URL: http://www-db.stanford.edu/lore/

[4] **Extensible Markup Language (XML) 1.0 specifikáció**, URL: http://www.w3.org/TR/2000/REC-xml-20001006

[5] **XML Schema specifikáció, Part 1: Structures**, URL: http://www.w3.org/TR/xmlschema-1/

[6] **XML Schema specifikáció, Part 2: Datatypes**, URL: http://www.w3.org/TR/xmlschema-2/

[7] **DocBook honlap**, URL: http://www.docbook.org

[8] **IETF URI specifikáció, RFC 2396**, URL: http://www.ietf.org/rfc/rfc2396.txt

[9] **W3C Szemantikus Web kezdeményezés honlapja**, URL: http://www.w3.org/2001/sw

[10] *Tim Berners-Lee, James Hendler and Ora Lassila,* **The Semantic Web**, Scientific American, 2001 május, URL: http://www.sciam.com/2001/0501issue/0501berners-lee.html

[11] **W3C RDF specifikáció: Resource Description Framework (RDF) Model and Syntax Specification**, URL: http://www.w3.org/TR/REC-rdf-syntax

[12] **W3C RDFS specifikáció: Resource Description Framework (RDF) Schema Specification 1.0**, URL: http://www.w3.org/TR/rdf-schema

[13] **DAML+OIL specifikáció**, URL: http://www.daml.org/2001/03/daml+oil-index.html

[14] *Ronald Bourret,* **XML and Databases**, 2001, URL: http://www.rpbourret.com/xml/XMLAndDatabases.htm

[15] *Sergey Melnik,* **Storing RDF in a relational database**, URL: http://www-db.stanford.edu/~melnik/rdf/db.html

[16] **Tamino XML adatbáziskezelő honlapja**, URL: http://www.softwareag.com/tamino/

[17] *Ronald Bourret,* **XML Database Products**, URL: http://www.rpbourret.com/xml/XMLDatabaseProds.htm

[18] **ISO 8879:1986, Standard Generalized Markup Language (SGML)**, URL: http://www.iso.ch

[19] **XML-Data reduced (XDR) specifikáció**, URL: http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm

[20] **Schematron honlap**, URL: http://www.ascc.net/xml/resource/schematron/schematron.html

Irodalomjegyzék

[1] Ullman, J. D.: Principles of Database and Knowledge-Base Systems, Vol. I-II, Comp. Science Press, 1988-89.
[2] Lockemann, P., G. Krüger and H. Krumm: Telekommunikation und Datenhaltung, Carl Hanser Verlag, 1993.
[3] Lang, S. M. and P. C. Lockemann: Datenbankeinsatz, Springer 1995.
[4] Date, C. J.: An Introduction to Database Systems, Addison-Wesley, 1990.
[5] Demetrovics J., J. Denev, R. Pavlov: A számítástudomány matematikai alapjai, Tankönyvkiadó, Budapest, 1985.
[6] Tanenbaum: Számítógép hálózatok, Novotrade Kiadó Kft., Budapest, 1992.
[7] Kiss István: Az SQL nyelv, Oktatási segédanyag, BME-MMT, 1993.
[8] Gajdos-Németh-Kiss: Informatika II., Jegyzet, Műegyetemi Kiadó 1996.
[9] Maier, D.: The Theory of Relational Databases, Comp. Science Press, 1983.
[10] Silberschatz, Korth and Sudarshan: Database System Concepts, Third edition, McGraw-Hill, 1998.