Adatbázisok

Gajdos Sándor

2019

Szerző:
Gajdos Sándor

egyetemi oktatási célra

A 2015. évi kiadás negyedik javított utánnyomása
ISBN 978-963-313-195-4
Terjedelem: 20 (B5) ív
Nyomdai munkák:
A-SzínVonal 2000 Nyomdaipari Kft., Szeged

Előszó

Ma, amikor az információ és a tudás a legfontosabb erőforrássá kezd válni, különösen felértékelődik az a képesség, amely (nagy) adatbázisok tervezésére, megvalósítására és/vagy hatékony lekérdezésére tesz valakit alkalmassá. Ilyen céllal már számos jegyzet, könyv íródott, a jelenlegi azonban kifejezetten a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar Műszaki informatika szakos hallgatói számára készült, ugyanis a felépítése az ő előképzettségükhöz és igényeikhez igazodik. Ennek megfelelően célja nem a naprakész gyakorlati ismeretek átadása, hanem olyan elméleti alapok nyújtása, amelyek egy-egy konkrét megvalósítástól függetlenek, és olyan alapokat jelentenek, amelyek széles körben megkönnyítik az adatbázis-kezelők működésének, felépítésének megértését, használatuknak a hatékony elsajátítását. Ezt szolgálja a példatár is, amely megmutatja, hogy a leírt elmélet elemeit milyen módon lehet gyakorlati problémák megoldására felhasználni.

Az adatbázisok gazdag és folyamatosan bővülő témaköreinek a leírtak természetesen csak egy (szűk) részét képezik, hiszen egyetlen félév anyagába kell mindennek belefférnie. A bővítés – és a technológiai fejlődés – legfontosabb irányait (szemistrukturált adatkezelés, NoSQL adatbázis-kezelők) a függelékekbe szorult fejezetek mutatják.

Jelen mű nem előzmények nélküli, közvetlen elődjének egy Microsoft Wordben készült jegyzet tekinthető. Lelkes és tehetséges hallgatók azonban jó pár hónapja rávettek arra, hogy ez a könyv már inkább LaTeX-ben készüljön, és ennek érdekében teljes mértékben magukra vállalták a konvertálás feladatait. Noha sejthető volt, hogy a szükséges ráfordítás irreálisan magas lesz a várható előnyökhöz képest, nem tudtam lebeszélni őket. Ezért a könyv esztétikai szépsége elsősorban nekik köszönhető: az ő kitartásukat dicséri, hogy szebbek lettek a betűk, határozottan elválnak egymástól a definíciók, példák, magyarázatok, tételek, bizonyítások; elkészült egy magyar-angol tárgymutató, továbbá számos helyen pontosítottunk a szövegezésen. Mindezek a tanulás segítését célozzák az olvashatóság javításán, ill. a nem egyértelmű megfogalmazások megszüntetésén/csökkentésén keresztül.

Ezért ezúton is hálás köszönetemet fejezem ki korábbi hallgatóimnak, akik áldozatos munkája nélkül ez a mű ebben a formában nem jelenhetett volna meg: Szárnyas Gábor, Stein Dániel, Szepes Nóra, Marton József, Darvas Dániel, Kocsány Dóra, Sebők Márton, Horváth Benedek, Frigó Erzsébet és Búr Márton vettek részt az elkészítésében.

A konverzió sajnos rengeteg manuális beavatkozást is igényelt, ezért óhatatlanul megjelenhetnek új sajtóhibák is. Előre is köszönöm, ha az Olvasó megosztja ezeket vagy egyéb észrevételeit velünk a `https://www.db.bme.hu/sajtohibak` címen keresztül annak érdekében, hogy a következő kiadásban ezek már ne maradjanak benne, ill. a javaslatokat figyelembe vehessük.

Budapest, 2015. augusztus

A szerző

Tartalomjegyzék

Bevezetés

Adatok gyors, gépesített tárolásának és visszakeresésének igénye már akkor felmerült, amikor még csak (elektro-)mechanikus számológépek léteztek. A népességnyilvántartásban kezdetben „lyukkártyás tabulátorokat" alkalmaztak („Hollerith kártyák"), amelyek ősi adatbázis-kezelőknek 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 adatbázis-kezelők, alapvetően a kártyás működést utánozták. Ma, 2019-ben a soros helyett közvetlen hozzáférésű háttértárak (mágneslemezek, optikai tárak, sőt egyre inkább a félvezető alapú nem felejtő tárak) dominálnak, és az egykori kártyás adattárolóra a mai számítógépek már egyáltalán nem hasonlítanak, azonban az adatbázis-kezelők dominánsan rekordorientált szemlélete máig megmaradt (de: ld. a NoSQL adatbázis-kezelőkről szóló C. függeléket). 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 és fizikai struktúrája tetszőlegesen bonyolulttá tehető, ami változatosabb és bonyolultabb lekérdezések hatékony végrehajtását is lehetővé teszi.

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 a 90-es években is még ezen a nyelven készült, megelőzve a C, C++ nyelveket is, amiket inkább rendszerfejlesztésre használnak.) Nem sokkal ezután megjelentek az első adatbázis-kezelő rendszerek is, melyek a fájlkezelőkből 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 adatbázis-kezelők 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 hódítottak az új elvek, mint az objektumorientáltság vagy a logikai programozás, a hálózatok elterjedésével az elosztott adatbázis-kezelők jelentősége is növekszik. Emellett egyre nagyobb szerepet kapnak a strukturált adatkezeléstől eltérő felépítésű és funkciójú, szövegszerű kezelést megvalósító (szemistrukturált, ill. strukturálatlan adatokon dolgozó, ld. B. függelék) információs rendszerek is, mint ahogyan az adatok kezelése helyett egyre fontosabbá válik az „értelmezett adat" (információ), ill. a tudás (kontextusba helyzett információ) kezelése (ld. tudásbázisok). A XXI. század elején pedig petabájtos adatmennyiségeket perzisztensen tárolni tudó, de csak korlátozott lekérdezhetőséget támogató rendszerek hódítanak (ld. pl. Yahoo!, Twitter, Facebook). Természetesen ezek a korlátok csak átmenetiek, mára – állítólag – elkészült az első, akár yottabájt (10^{24}) mennyiségű[1], a legkülönbözőbb forrásból származó és típusú adatot (szöveg, szám, hang, kép, video) egyaránt feldolgozni képes adatbázis az USA Nemzetbiztonsági Hivatala (NSA) számára.[2] Az adatok sok terabájtnyi hányada már folyamatosan a memóriában található, ami ezekhez az adatokhoz való hozzáférést több nagyságrenddel gyorsítja meg ahhoz képest, mintha azokat mágnes- vagy optikai lemezen tárolnánk (ld. *memóriarezidens adatbázis*, IMDB).

Itt tartunk most…

[1] Ennek a nagyságát segít elképzelni, ha tudjuk, hogy a teljes világháló forgalma 2010-ben „csak" mintegy 250 exabájt (250×10^{18} bájt) volt.

[2] A borítón az NSA főhadiszállása látható Fort Meade-ben, Maryland államban. Forrás: Wikipedia.

1. fejezet

Alapfogalmak

Adatbázisnak a valós világ egy részhalmazának leírásához használt adatok összefüggő, rendszerezett 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, *adatbázis-kezelő rendszernek* (database management system, DBMS) nevezzük. Az adatbázis-kezelőt alapvetően jellemző tulajdonságok:

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

Az adatbázis-kezelő rendszerek adatait (még) ma (2019) is túlnyomórészt merevlemezes mágneses háttértárakon (hard disk drive, HDD, „winchester") tárolják, kisebb részben mágnesszalagon, optikai lemezen, de rohamosan növekvő hányadban félvezető alapú memóriában is, beleértve a szilárdtest-meghajtókat (solid-state drive, SSD)[1] is. Így jelen jegyzetben a mágneslemezes háttértárak alkalmazására, sajátosságaira koncentrálunk. Az adatbázis-kezelő gazdag struktúrája azt jelenti, hogy a tárolt rekordok között változatos logikai kapcsolatok hozhatók létre, amelyek meghatározott adatbázis-műveleteket megkönnyíthetnek (értsd: meggyorsíthatnak). Egyes adatbázis-kezelők bizonyos logikai kapcsolatokat megengedhetnek az adatok között, másokat nem, így különböző adatmodelleken (ld. 4.1. szakasz) alapulhatnak. 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 adatbázis-kezelő úgy működik (működhet), hogy a felhasználó anélkül tudja előírni a teendőket, hogy az

[1] Olyan adattároló eszköz, amely blokkos elérést (ld. 3. fejezet) biztosít az adatokhoz, és a tápellátás megszűnése esetén sem veszíti el azokat.

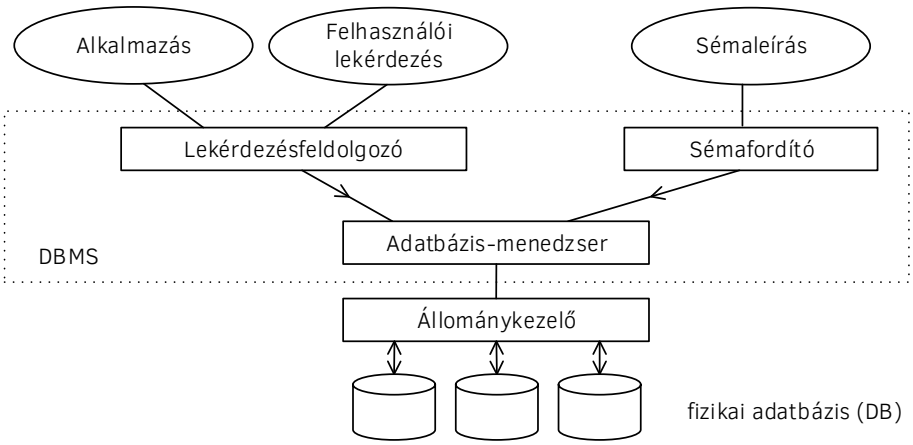
adatbázis-kezelő algoritmusairól vagy az adatok (fizikai) tárolási elvéről ismeretei lennének.

A továbbiakban megvizsgáljuk, hogy melyek a különböző adatbázis-kezelők fontosabb közös vonásai.

1.1. A programozó és a felhasználó kapcsolata az adatbázis-kezelő rendszerrel

Az 1.1. ábra egy DBMS általános felépítését és környezetét mutatja be.

A „klasszikus” adatbázis-kezelő 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 vagy struktúrájának megteremtését jelenti (1. fázis). Ennek az adatai (mint ún. technikai *metaadatok*) az adatbázisnak egy különösen védett részében kerülnek tárolásra, hiszen az elvesztésük/sérülésük a teljes adatbázis tartalmát hozzáférhetetlenné teszi. 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 adatbázis-kezelő válaszol rá. Másrészt, a lekérdezések önálló programként vagy egy alkalmazás (applikáció) részeként is megfogalmazódhatnak. Mindkét esetben egy interpreter, egy lekérdezésfeldolgozó alakítja azokat az *adatbázis-menedzser* által értelmezhető formába. Eközben általában a lekérdezés optimalizálása is megtörténik, hiszen egy-egy végeredményhez gyakran több „úton” (műveletek különböző sorozatán keresztül) is eljuthatunk, amelyek számításigénye azonban akár nagyságrendileg is különbözhet.



1.1. ábra. A DBMS és környezete

Az 1. fázist egy speciális nyelv, az ún. *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 az adatbázis-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 az adatbázis-kezelés hőskorában határozottan szétvált egymástól, de az utóbbi időben gyakran jelennek meg egy egységes nyelvként, mint pl. a szabványosított SQL nyelvben (ld. 5.4. szakasz).

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

Az *állománykezelő* (file manager) biztosítja a hozzáférést a fizikai adatbázishoz, és 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. fejezet). Az adatbázisokban az adatvesztés veszélye miatt az adatokat végül mindig valamilyen perzisztens tárolást biztosító eszközre is ki kell írni (ld. a 10.1. szakasz, ACID tulajdonságok).

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

1.2. Járulékos feladatok

Az adatbázis-kezelőtől 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, vagy egyáltalán nem (ld. pl. a személyes adatokat, mint tipikus, különösen érzékeny adatokat). 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. (ld. 10. fejezet).

1.2.3. Integritás (integrity)

Fontos, hogy legyen olyan beépített szolgáltatás, amely segítségével az adatbázis adatainak „helyessége", ellentmondás-mentessége – 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, gyakran azonban az adatbázis alkalmazásokra hárul ennek a feladatnak egy része.[2] 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. A 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 *adatbáziskényszer* (data constraint) (ld. még 9.2.2. alszakasz) is, amelyek miatt pl. 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 (synchronization, synchronism)

A ma használatos adatbázis-kezelő rendszerek általában többfelhasználósak (ld. 10. fejezet), és egyre gyakrabban térben elosztottan (ld. 11. fejezet), egy számítógép-hálózatnak megfelelően üzemelnek.

Fontos, hogy az azonos adatokon közel egyidőben műveleteket végző felhasználók beavatkozásainak ne legyenek nemkívánatos mellékhatásai egymás tevékenységére,

[2] Vannak olyan, széles körben elterjedt (pl. ERP, CRM családjába tartozó) alkalmazások/rendszerek, amelyeknek sokféle adatbázis-kezelővel kell együttműködniük. Az adatbázis-kezelők különbözőségei miatt szinte semmit nem használnak ki azok beépített (gazdag) lehetőségeiből, szinte csak rekordmenedzsernek használják őket.

illetve az adatbázis tartalmára. Ezt a *tranzakciókezelés* (transaction processing) fogalmába tartozó módszerek képesek biztosítani jól bevált eszközök – pl. *zárak* (lockok) – rendszerével.

1.3. Az adatbázissal kapcsolatos tevékenységek szintjei

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

***Képzetlen felhasználó* (naive 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. webes áruház katalógus nézegetője).

***Alkalmazásprogramozó* (application programmer):** 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 (legalább) logikai szintű (ld. 2. fejezet) elérését.

Ez olyan feladat, amely programozót igényel, de megoldásához nem feltétlenül szükséges, hogy az illető belelásson az adatbázis belső szerkezetébe is, és egyáltalán nem szükséges, hogy a szerkezetet (a tárolt adatok megőrzése mellett) módosítani tudja.

***Adatbázis adminisztrátor* (database administrator):** 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.
- *Jogosultságok 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 pl. a régi adatok mellé újabbakat is felveszünk a tárolandók közé.
- *Mentés-visszaállítás.* Célszerű lehet adatbiztonsági okokból időnként vagy bizonyos időközönként 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* (DBMS designer/programmer):** Tudja, hogyan kell DBMS-t készíteni, ami különösen specializált tudást igényel.

1.4. A fejezet új fogalmai

adat, információ, tudás, (fizikai) adatbázis, adatbázis-kezelő rendszer, adatbázis alkalmazás, metaadat, adatbázis séma, sémafordító, DML, DDL, állománykezelő, SQL, lekérdezés feldolgozás, adatbiztonság, szinkronizálás, integritás, adatstruktúra, strukturált adat, szemistrukturált adat, nemstrukturált adat, adatszótár, adatbázis adminisztrátor

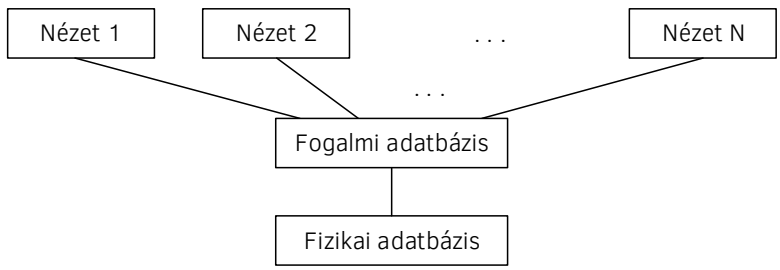
2. fejezet

Az adatbázis-kezelők felépítése

A mai adatbázis-kezelők bonyolult hardver-szoftver rendszerek. Komplexitásuk az operációs rendszerekével összemérhető, sőt, gyakran nagyobb annál. Egy ilyen rendszer megtervezése, implementálása és karbantartása nem egyszerű feladat, amelyre kifinomult módszerek léteznek. Ismertetésük túlmutat e jegyzet keretein, itt csak a legfontosabb modellezési, tervezést segítő elvek bemutatására van lehetőség.

Mint a mérnöki gyakorlatban olyan sok más helyen, itt is eredményes a rétegezési koncepció, vagyis egy *rétegmodell* (layered model) 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 számítógép-hálózatok ISO-OSI modellje [11]. Hasonló modell, sőt modellek léteznek az adatbázis-kezelők számára is: a legegyszerűbb 3 rétegűtől kezdve a 7 rétegű modellig. Jelen jegyzetben részletesebben egy 3 rétegűvel ismerkedünk meg (2.1. ábra).

A legalsó réteg a *fizikai adatbázis* (physical database). Itt valósul meg az adatbázis adatainak a fizikai tárolókon való elhelyezése. Ide értjük azokat az adatstruktúrákat is, amelyekben a (fizikai) adattárolás megvalósul (ld. 3. fejezet). Ehhez a réteghez tartozó fogalmak: *kötet, állomány, blokk, track, szektor, vödrös hashing* (ld. 3.2. szakasz) stb.



2.1. ábra. Adatbázis-kezelők 3 rétegű architektúrája

Középen helyezkedik el a *fogalmi (logikai) adatbázis* (conceptual (logical) database). Ez nem más, mint a való világ egy darabjának leképezése, egy sajátos modell, ahogyan az adatbázis tükrözi a valóság egy részét. A fogalmi adatbázis van szorosabb kapcsolatban azzal, ahogyan az adatokat értelmezni kell. Pl. egy könyvtári adatbázisban ide tartoznak a következők: a kölcsönző személyek neve, kölcsönzőjegyének száma, egy kötet lelőhelye, ETO-száma, példányszáma, címe, szerzője, kiadója, értéke stb. A fogalmi adatbázishoz tartozó sémát *fogalmi (logikai) sémának* (conceptual (logical) schema) nevezik.

Nézet (view) 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 sémának* (external schema) 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 változatlanok maradnak. Az adatbázis-kezelés világában ezt az *adatfüggetlenség* (data independence) elvének nevezik.

Kétféle adatfüggetlenségről lehet beszélni a háromrétegű modellben: a fizikai és a fogalmi adatbázis között értelmezhető *fizikai adatfüggetlenségről*, ill. a fogalmi adatbázis és a nézetek között értelmezhető *logikai adatfüggetlenségről*.

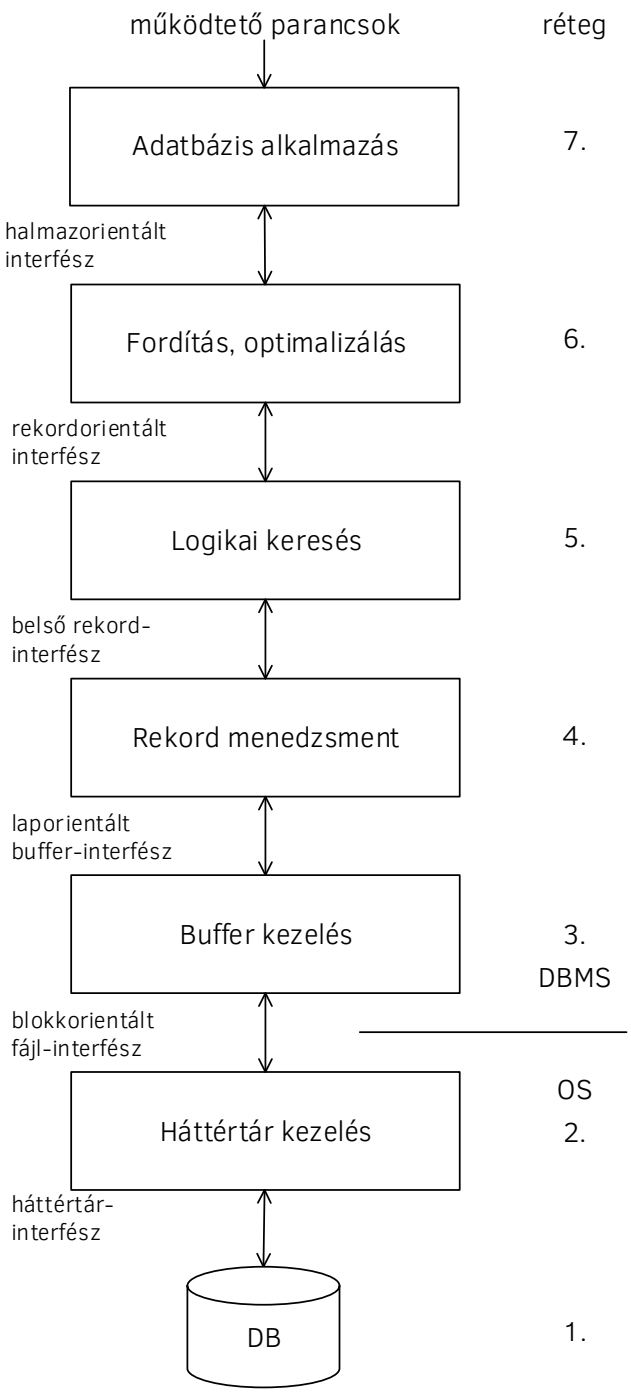
A *fizikai adatfüggetlenségen* (physical data independence) (kb. eszközfüggetlenségen) azt értjük, hogy a fizikai szinten, a fizikai működés sémáiban véghezvitt változások nem érintik a fogalmi (logikai) adatbázist. Ha ez teljesül (gyakorlatilag mindig), akkor a fizikai adathordozó egy teljesen eltérő fizikai paraméterekkel rendelkezőre is kicserélhető (pl. meghibásodás, technikai fejlődés stb. miatt), vagy az állományszervezés módja megváltoztatható anélkül, hogy az adatbázisban bármilyen logikai változás érzékelhető lenne (a rendszer teljesítőképessége, válaszidejei azonban jelentősen változhatnak).

Logikai adatfüggetlenségről (logical data independence) 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 adatbázis-kezelőnek és környezetének egy tipikus, hétrétegű modelljét.

2.1. A fejezet új fogalmai

adatbázis nézet (view), modell, rétegmodell, fizikai adatbázis, logikai (fogalmi) adatbázis, külső séma, logikai adatfüggetlenség, fizikai adatfüggetlenség



2.2. ábra. Adatbázis-kezelő (és környezete) statikus 7 rétegű modellje

3. fejezet

A fizikai adatbázis

A 2.1. ábrának megfelelően most az adatbázis-kezelő rendszerünk legalsó szintjét vizsgáljuk meg: hogyan lehet az adatrekordjainkat célszerűen tárolni a háttértárolón annak érdekében, hogy egy keresett rekordot vagy rekordok egy halmazát minél gyorsabban el lehessen érni. Ennek megvalósításához az adott tárolóeszköz ismerete is szükséges. Jelen fejezetben a mágneslemezes háttértáron való hatékony tárolás lehetőségeit vizsgáljuk meg (*diszkrezidens adatbázis*, DRDB). Mindez természetesen nem jelenti azt, hogy a diszkek szerepe kizárólagos. Különösen érdekes – és jelentősen eltérő – az olyan adatbázis-kezelők felépítése és működése, ahol a teljes adatbázist memóriában tárolják (*memóriarezidens adatbázis*, IMDB). Ilyenek kb. 2005 óta már kereskedelmi forgalomban is kaphatók.

A mi, diszkekre vonatkozó megállapításaink is csupán számos egyszerűsítő feltétel fennállása esetén igazak. Ezeket foglaljuk össze először.

Négy műveletet tervezünk támogatni, melyek a

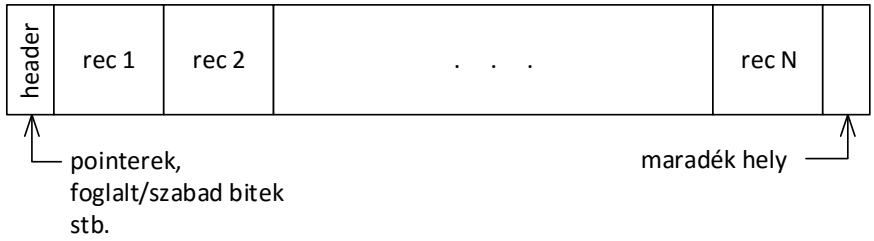
- keresés,
- beszúrás,
- törlés és a
- módosítás.

A számítógépen futó operációs rendszer az adatbázis adatait is állományokban (fájlokban) tárolja. Az állományok azonos méretű blokkokból épülnek fel, a blokkméret általában 0,5 . . . 32 kilobájt. Az operációs rendszer tartja nyilván, hogy egy állományhoz mely blokkok tartoznak. Minden blokknak abszolút címe (is) van, egyetlen fejmozgatással és *be-/kimeneti* (input/output, I/O) művelettel a blokk elérhető, adatai az *operatív tárba* (memory) juttathatók. A szükséges adatokat minden esetben a mágneslemezről kell beolvasni, mert nincs lehetőségünk arra, hogy egyidőben több blokkot a számítógép memóriájában tároljuk (ez a gyakorlatban természetesen általában nem igaz, azonban a következőkben alkalmazzuk ezt az egyszerűsítő feltételezést, hiszen DRDB esetén az adatok kis hányadát tételezhetjük fel az operatív tárban elérhetőnek).

Cél: a fent felsorolt négy művelet minél gyorsabb elvégzése.

Mivel a lemezműveletek időigénye ($\sim$ ms) több nagyságrenddel nagyobb ahhoz képest, mintha az adatokat a memóriában kellene megkeresni ugyanilyen szervezés mellett ($\sim$ μs), ezért a *fenti műveletek időigényét alapvetően az fogja meghatározni, hogy egy blokk tartalmáért hányszor kell a háttértárhoz fordulni.* Ezért a fizikai adatszervezés célját DRDB esetén 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 3.1. ábrán látható.



3.1. ábra. Egy blokk szerkezete

Lényeges, hogy a rekordok *blokkhatárt nem lépnek át*, így a blokkok általában nincsenek teljesen kihasználva. (A továbbiakban ezért feltételezzük, hogy a blokkméret mindig legalább akkora, mint egyetlen rekord mérete.)

A rekordok általános szerkezete a 3.2. ábrán látható.

3.2. ábra. Egy adatrekord szerkezete

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

A rekordokat számos módon megcímezhetjük. A 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 ofszetet, amely a blokkon belüli kezdőcímet adja meg. Ezen kívül *logikailag is megcímezhető* egy

rekord, pl. ha megadjuk egy kulcsának az értékét, hiszen az is alkalmas lehet a rekord egyértelmű azonosítására.

Másrészről, a rekordok lehetnek rögzített vagy változó formátumúak / hosszúságúak is. Változó lehet a formátumuk, ha pl. változó hosszúságú mezőt vagy ismétlődő csoportot (tipikus a hálós adatbázisoknál, ld. 7. fejezet) tartalmaznak. A változó hosszúságú rekordok problémájával csak a 3.4. szakaszban foglalkozunk.

A továbbiakban tehát feltételezzük, hogy az adatállományok minden rekordjában a mezők ugyanabban a sorrendben fordulnak elő, és a hosszuk is minden rekordban azonos.

Tárolnunk kell valahol[1] a különböző típusú rekordokkal kapcsolatban azt az információt is, hogy milyen a felépítésük, az egyes mezőket hogyan kell értelmezni (egész szám, lebegőpontos, karakterlánc, stb.). Így ha a *blokk fejléc* (*fejrész*) után a rekordok közvetlenül egymás után következnek, egyértelműen tudjuk dekódolni a mezőket. Ekkor már csak arról kell gondoskodni, hogyan különböztessük meg az „élő" rekordokat a még soha nem használt üres helyektől. Egy lehetőség, ha a blokk headerben egy számláló jelzi a blokkon belüli élő rekordok mindenkori számát, de más megoldások is elképzelhetők.

A továbbiakban a hasznos – a fejrész méretét nem tartalmazó – blokkméretet b-vel, az r állomány rekordméretét s_r-rel, rekordjainak számát n_r-rel jelöljük. Az r állományban egy blokkban elhelyezhető rekordok számát *blocking factornak* nevezzük és f_r-rel jelöljük:

$$f_r = \left\lfloor \frac{b}{s_r} \right\rfloor .$$

Az állomány által elfoglalt blokkok számát b_r-rel jelöljük:[2]

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil .$$

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

3.1. Heap szervezés (heap file organization)

Ebben az esetben közelítjük meg legegyszerűbben a tárolás problémáját (heap: halom, kupac). Az adatokat (legalább) annyi blokkban tároljuk, amennyit a rekordok száma és mérete megkövetel/igényel, de nem rendelünk hozzá semmilyen kiegészítő/segéd-struktúrát.

[1] Erre a célra az adatbázisnak egy kitüntetett területét szokás használni, amit a különböző kereskedelmi adatbázis-kezelő rendszerek különböző nevekkel illetnek (ld. data dictionary, data repository, …).

[2] A valóságban ennél több blokkra is szükség lehet, ha töredezett az állomány.

Keresés

Mivel a már megismert struktúrákon kívül (blokk, rekord, mező) újat nem hozunk létre, így a tárolásban nincs egyéb szabályosság. Ha valamely érték (= keresési kulcs) alapján egyetlen rekordot tudunk azonosítani, akkor egymás után beolvassuk a háttértárról a memóriába a kérdéses rekordokat tartalmazó állomány blokkjait, majd végigolvassuk a blokkokat mindaddig, amíg rá nem találunk a keresettre (lineáris keresés). Szerencsés esetben csak egyetlen blokkot, szerencsétlen esetben az állomány minden blokkját végig kell olvasnunk. Így egy egyedi kulcsérték által meghatározott egyetlen rekord megtalálásához átlagosan $(\textit{blokkok száma} + 1)/2$ számú blokkműveletet kell elvégezni.

Törlés

A törlendő – t. f. h. egyetlen – rekordot megkeressük, ennek időigénye már ismert. A rekord fejlécben jelezzük, hogy a terület (= *subblock*) felszabadult, tehát ha a rekord nem kötött, akkor a terület felülírható, ezután a megváltozott blokkot még vissza kell írni a diszkre a megtalálása után. Időnként szükség lehet a gyakori törlések következtében szétszóródott lemezterületek összegyűjtésére és egyesítésére. Az ún. *szemétgyűjtő* (garbage collector) programmodul a törölt jelzés alapján tudja, hogy ezt a területet felül lehet írni.

Beszúrás

Beszúrásnál ügyelni kell arra, hogy a rekordok egyediséget biztosító mezőinek értékei valóban egyediek maradjanak a beszúrás után is. 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, ill. az adatbázis adminisztrátort kell megkérni, hogy bővítse az állományt. Ennek végrehajtása komplex feladat lehet, ugyanis célszerű egy fizikailag egybefüggő (kevés fejmozgatással elérhető) diszkterülettel bővíteni az adatbázist.

Módosítás

A módosítás egyszerű művelet, ha az egyediséget biztosító mező(k) értéke nem változik. Ekkor a módosítás egy rekord megkeresését, felülírását, majd a tartalmazó blokk háttértárra való visszaírását jelenti. Minden egyéb esetben gondoskodni kell arról, hogy két megkülönböztethetetlen rekord ne kerüljön az adatbázisba.

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 az ún. 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 blokkművelettel elérhetjük a rekordot az általában hosszadalmas, blokkok közötti *keresés*[3] helyett.

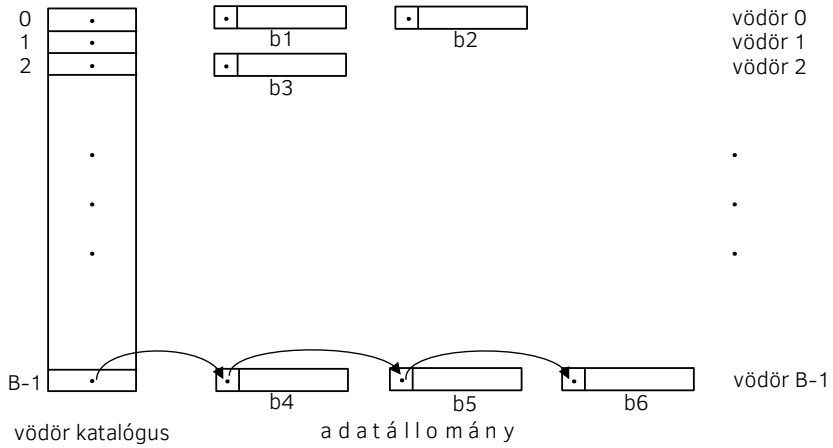
Ez a megközelítés – bár igen gyors rekordhozzáférést tesz lehetővé – ebben a formájában gyakorlatban alig használható, mert a háttértárat igen rosszul használná ki.

Egy gyakorlati megoldást az ún. *vödrös hashelés* (bucket hashing) jelent, melynek alapgondolata és fogalmai a 3.3. ábrán láthatók.

Osszuk fel az adatállományt B (*vödör*, bucket) részre úgy, hogy minden rész legalább egy blokkból álljon! Hozzunk létre egy B számú mutatóból álló ún. *vödörkatalógust* (hash table, bucket directory), amelyben minden mutató az állomány egy-egy blokkcsoportjának (vödörnek) a címét tartalmazza. Definiáljunk továbbá egy h *hash-függvényt* (hash function), amely a kulcsok szóbajöhető értékkészletét leképezi a $[0, B-1]$ tartományra mégpedig lehetőleg egyenletesen, ha a kulcs befutja a szóba jöhető értékeinek tartományát.

A vödrös hash-szervezés lényege 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 $h(K) = (c \cdot K) \bmod B$, ahol c egy alkalmasan megválasztott állandó.



3.3. ábra. Vödrös hash-szervezés

[3] Ld. heap (3.1. szakasz) ill. indexelt szervezés (3.3. szakasz).

Keresés

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

Végigolvassuk a vödör első blokkjának mindegyik nem törölt és nem üres rekordhelyét. Ha valamely kiolvasott rekordnak K a kulcsa, akkor megtaláltuk a rekordot. Ha nem, akkor a vödör következő, ún. túlcsordulási blokkjait vizsgáljuk végig hasonló módon. (Ezt a blokkot a blokk headerben lévő mutató címzi.) Ha a vödör egyik blokkjában sem találtuk meg a K kulcsú rekordot, akkor az biztosan nincs az adatbázisban.

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

Beszúrás

Helyezzük el az állományban a K kulcsú rekordot! Beszúrásnál természetesen itt is ügyelni kell arra, hogy a rekordok egyediséget biztosító mezőinek értékei valóban egyediek maradjanak a beszúrás után is. 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, hogy a kulcs egyediségének megsértése elkerülhető legyen. 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 hozzáfűzni, és az új rekordot itt kell elhelyezni.

Törlés

Megkeressük a kívánt rekordot a már jól ismert módon. A törölt bitet bebillentjük, a módosított blokkot visszaírjuk a diszkre.

Módosítás

Ha a módosítás nem érint kulcsmezőt, akkor egyszerűen végrehajtható: megkeressük a módosítandó rekordot tartalmazó blokkot, és a művelet elvégzése után a blokkot visszaírjuk a háttértárra. Ha nem találtuk a rekordot, 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-alapú műveletek igen gyorsak lehetnek, ha a vödrök hossza kicsi. Szélsőséges esetben, ha minden vödör csak egyetlen blokkból áll, akkor az összes rekord egyetlen blokkhozzáféréssel elérhető, feltéve, hogy a vödör katalógus elég kicsi ahhoz, hogy a memóriában lehessen tárolni (ezt általában feltételezzük). 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. Tekintettel arra, hogy manapság a diszkterület az egyik legolcsóbb erőforrás, nem a diszkterülettel való takarékosságra érdemes törekedni, ha a keresést gyorsítani kell.

A hash-állományszervezés másik jellegzetessége, hogy az ún. „tól–ig" kereséseket (*intervallumkeresés*) 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

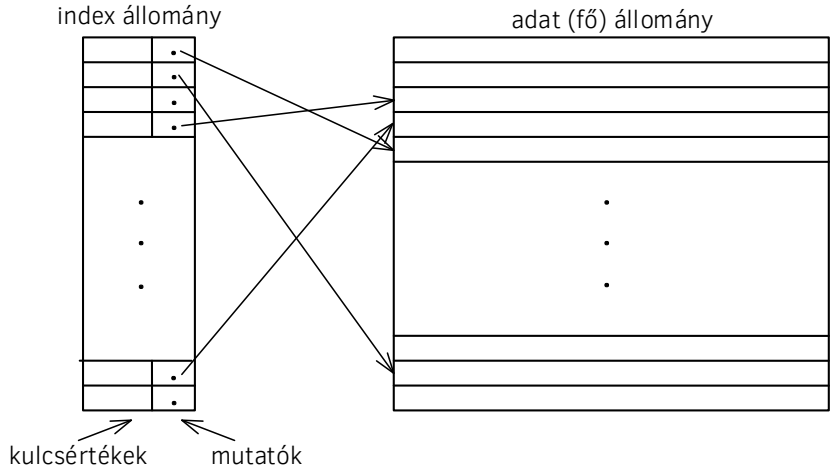
Ha egy könyvtárban egy könyvet keresünk, nem nézzük végig a könyvtár raktárát és olvassuk végig valamennyi könyvcímet és/vagy szerzőt. Helyette a könyvtári katalógust (= egy segédstruktúrát) lapozgatjuk, amelynek mérete töredéke a teljes könyvállományénak, így könnyebben kezelhető (katalóguscédulák, mikrofilm), ráadásul ábécébe rendezett, szemben a könyvraktárral. Továbbá, a különböző katalógusok többféle szempont szerint is lehetnek 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 ún. indexállományban (kb. katalógus) megismételjük, és a kulcshoz egy mutatót rendelünk, amely a tárolt adatrekord helyére mutat. Az indexelt állományszervezés alapstruktúráját és fontosabb fogalmait a 3.4. ábra mutatja.

A kulcsot és a mutatót is rögzített hosszúsággal ábrázoljuk (hosszukat rendre k és p jelöli, több kulcs esetén: $k_1, k_2, \dots$). Az indexállomány blocking factora $f_i = \left\lfloor \frac{b}{k+p} \right\rfloor$, ahol i az indexállomány neve.

Az indexállományt mindig rendezve tartjuk. Ha a kulcs numerikus, akkor a rendezés triviális. Ha szöveges, akkor lexikografikus rendezést alkalmazhatunk. *Összetett kulcs* (composite key) esetén, vagyis amikor a kulcs több mezőből áll, definiálnunk kell a rendezés módját, azt, hogy a kulcsmezők mely sorrendje alapján történjen a rendezés. Ennek a sorrendnek az alkalmas megválasztása jelentősen befolyásolhatja az index használatának a hatékonyságát. Általában nincs akadálya, hogy több indexállományt is létrehozzunk ugyanazon adatállományhoz, különböző (vagy különbözően rendezett) kulcsmezőkkel, bár vannak nehézségek (ld. 3.3.4. alszakasz). Még az sem biztos, hogy ez a mező (ill. ezek a mezők) egy

rekordot egyértelműen azonosítanak. Ebben a kontextusban tehát *(keresési) kulcs lesz minden, ami szerint egy indexet felépítünk, ill. a keresést végezzük.* Vegyük észre, hogy az indexállomány (azonos hosszúságú) rekordjai szabadok, így könnyen mozgathatók, jól karbantarthatók. Ugyanakkor az adatállomány rekordjai valamilyen értelemben kötöttekké válnak.



3.4. ábra. Indexelt szervezés

Mindeddig nem volt szó arról, hogy az indexrekordokat hogyan feleltetjük meg az indexelt állomány rekordjainak. Két alapvetően különböző megoldás lehetséges:

1. indexrekordot rendelünk minden egyes adatrekordhoz, vagy
2. indexrekordot rendelünk adatrekordok egy csoportjához, tipikusan az egy blokkban levőkhöz.

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

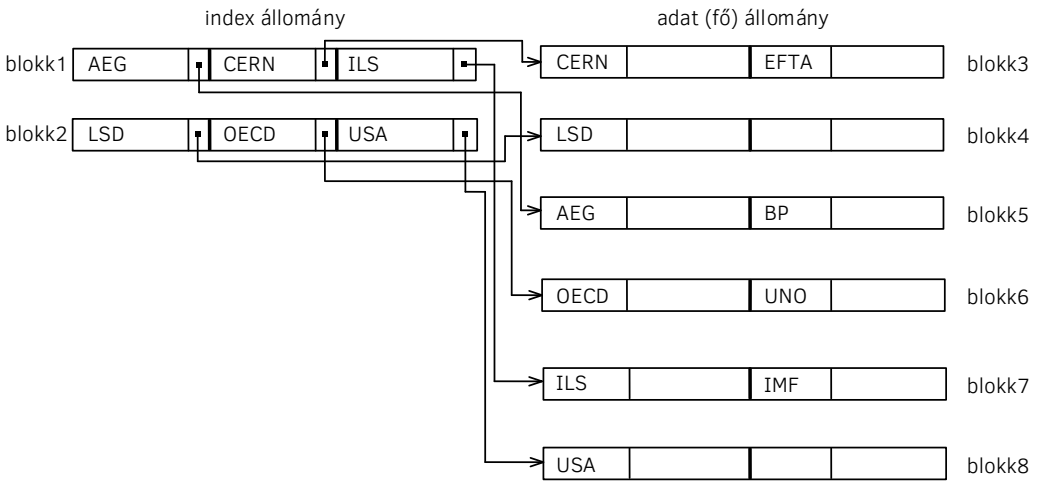
3.3.1. Ritka indexek (sparse indices)

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 adatbázis-kezelésben 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 minden olyan adatrekord, amelyeknek a kulcsa egy meghatározott intervallumba esik. Az adott blokkra mutató indexrekord a blokk címét, valamint a legkisebb (vagy a legnagyobb) értékű kulcsot fogja tartalmazni.

Az ri ritka indexállomány blokkjainak száma $b_{ri} = \left\lceil \frac{n_{ri}}{f_{ri}} \right\rceil$, ahol az előzőekkel összhangban $n_{ri} = b_r$, ha az r állományhoz épített ritka indexről beszélünk.

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 (vagy éppen egyenlő vele). 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.



3.5. ábra. Ritka index szervezés

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 rekordjainak számát (beleértve a k_1 kulcsút is) megfelezzük B_i és B_n között (természetesen a rendezettséget megőrizve). 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.

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.

Módosítás

A módosítás egyszerű, ha nem érint kulcsot. Ekkor meg kell keresni a szóban forgó rekordot, elvégezni a módosítást, majd az érintett adatblokkot visszaírni a háttértárra.

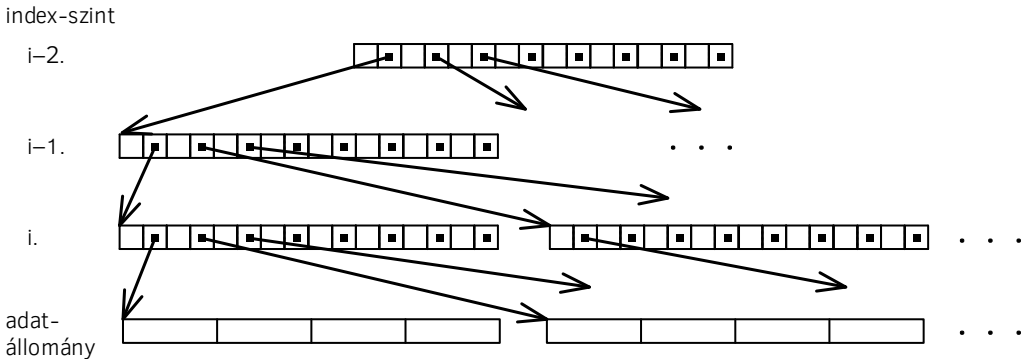
Ha a módosítás kulcsmezőt is érint, akkor egy törlést követő beszúrás valósíthatja meg egy rekord módosítását.

3.3.2. B*-fák, mint többszintes ritka indexek

Az indexelt szervezésnél $\log_2 b_i$-vel (b_i az i indexállomány blokkjainak száma) arányos átlagos keresési idő érhető el, amely lényegesen kisebb, mint a heap szervezésé (b_r-rel arányos), de elmarad a hash-szervezésé (melynek keresési blokkműveletigénye 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 $\log_k b_i$-vel arányos keresési időt érjünk el. Igen nagy méretű adatállományok esetén, ill., ha k elég nagy, akkor 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 többletadminisztráció 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 indexállomány egy-egy részfáját azonosítják (ld. 3.6. ábra).



3.6. 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"). Szokásos még, hogy az egy csomópontban ábrázolt l mutatóhoz csak $l-1$ 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. B*-fa esetén a blocking factor, azaz a fa elágazási tényezője:

$$f_i = \left\lfloor \frac{b+k}{k+p} \right\rfloor ,$$

ahol k a kulcs hosszát jelöli. Az r állományhoz tartozó B*-fa szintjeinek számát HT_i-vel (Height of Tree) jelöljük (ld. még a 6.2.2. alszakaszt):

$$HT_i = \left\lceil \log_{f_i} b_r \right\rceil$$

A gyakorlati megvalósításokban a leveleket gyakran egyik vagy mindkét irányban összeláncolják, amivel az intervallumkereséseket lehet gyorsítani.

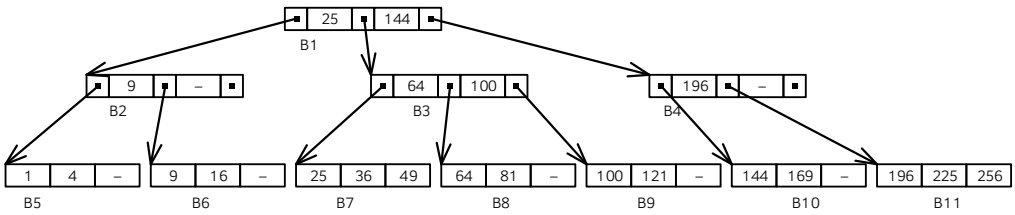
Keresés

Az eljárás hasonló ahhoz, mint amit az egyszintű ritka indexek esetén kellett végrehajtani, csupán az indexállományban több lépésben végezzük a keresést.

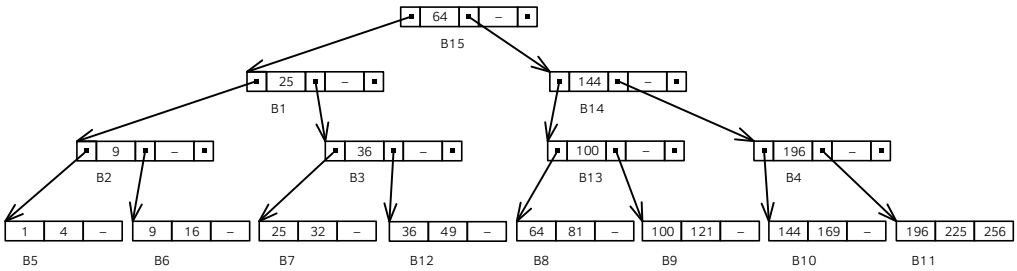
Tételezzük fel, hogy a v_1 kulcsú rekordra van szükségünk. Az indexállomány csúcsán álló blokkban megkeressük azt a rekordot, amelyiknek v_2 kulcsa a legnagyobb azok közül, amelyek még kisebbek v_1-nél (vagy egyenlő vele). Ennek a rekordnak a mutatója az eggyel alacsonyabb szintű indexben rámutat arra 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 (vagy egyenlő vele). Az eljárás mindaddig folytatandó, ameddig az utolsó mutató már az adatállomány egy olyan blokkját azonosítja, amelyben a v_1 kulcsú rekordnak lennie kell.

Beszúrás

Az eljárás nagymértékben hasonló a 3.3.1. szakaszban a beszúrásnál 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 minden szintjén igényelheti néhány blokk megváltoztatását. A követendő eljárást a 3.7. és a 3.8. ábrák szemléltetik. Az ábrákon azt az egyszerűsítő feltételezést tettük, hogy az adatrekordoknak csupán egyetlen mezőjük van, ami egyben nyilván kulcsmezőül is szolgál.



3.7. ábra. Egy adatstruktúra B*-fa szervezés esetén



3.8. ábra. A B*-fa a 32 kulcsú rekord beszúrása után

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 indexállomány érintett részfájából. Ehhez adott esetben a fa minden szintjén szükség lehet néhány blokk módosítására.

Módosítás

Elvben azonos a 3.3.1. pontban leírtakkal, a bonyolultabb indexstruktúrából adódó követelményeket értelemszerűen alkalmazva.

3.3.3. Sűrű indexek (dense indices)

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

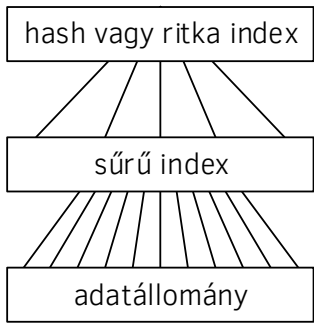
a memóriába beolvasnunk. A blokkon belüli keresés már igen gyors, így elegendő az indexállományban a blokkcímeket tárolni a bennük található legkisebb kulcsértékkel együtt. Ennek az ára viszont az, hogy az adatállományt is rendezetten kell tárolni, hacsak nem megengedhető az „egy adatblokk = egy adatrekord” 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.

Megjegyzés. 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. alszakasz).

Az si sűrű indexállomány blokkjainak száma $b_{si} = \left\lceil \frac{n_{si}}{f_{si}} \right\rceil$, ahol az előzőekkel összhangban $n_{si} = n_r$, ha az r állományhoz épített sűrű indexről beszélünk.

A sűrű indexek tipikus alkalmazása a 3.9. ábrán látható.



3.9. ábra. Sűrű index alkalmazása

A bemutatott megoldásnak a hátrányain kívül számos előnye is van.

Hátrányok:

- a sűrű indexnek plusz helyigénye van,
- eggyel több indirekció kell egy rekord kiolvasásához,
- plusz adminisztrációval jár a sűrű index karbantartása.

Viszont:

- az adatállományt nem kell rendezetten tárolni, így 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 (csaknem) szabadokká tehetők, ha minden további rekordhivatkozás a sűrű indexen keresztül történik (egyetlen mutatót kell megváltoztatni).

Keresés

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

Törlés

Megkeressük a kívánt rekordot. A hozzátartozó törölt bitet szabadra állítjuk. A kulcsot kivesszük az indexállományból, és az indexállományt időnként – műveletigényessége miatt nem minden törlés után – tömörítjük.

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 indexállományba.

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. 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 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 ún. *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 indexállományt is létrehozunk, minden keresési kulcshoz egyet.

Definíció – invertált állomány *(inverted file)*. Azt az indexállományt, amely nem kulcsmezőre[a] tartalmaz indexeket, *invertált állománynak* nevezzük.

[a] azaz egyediséget nem biztosító mezőre

Az invertált állomány mutatói

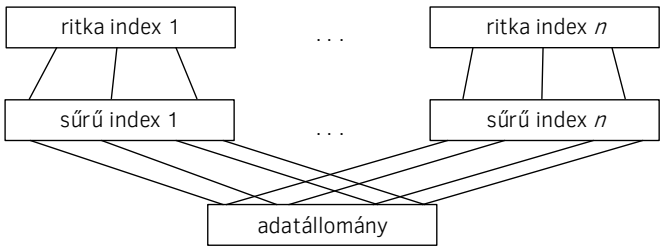
1. lehetnek fizikai mutatók, amelyek pl. mutathatnak
   a) közvetlenül az adatállomány megfelelő blokkjára (esetleg közvetlenül a rekordra), vagy
   b) az adatállomány elsődleges kulcsa szerinti (sűrű) indexállomány megfelelő rekordjára, ill.
2. lehetnek logikai mutatók, amelyek az adatállomány valamely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

Az 1.*a)* esetben az adatállomány rekordjai kötöttek és ráadásul csak egyetlen invertált állomány esetén használható.

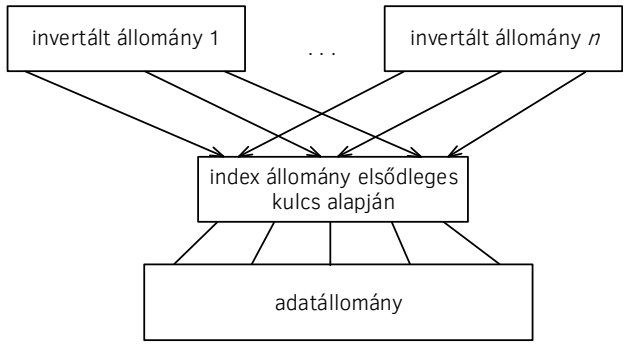
Az 1.*b)* esetben eggyel több indirekción keresztül érjük el a keresett rekordot, de az adatrekordok változtatásakor csak az érintett mezőt (mezőket) tartalmazó invertált állományokat és az indexállományt kell módosítani.

Ha a 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 3.10. ábra azt mutatja be, hogyan lehet egy állományhoz az 1.*b)* esetben sűrű index segítségével tetszőleges számú ritka indexet rendelni, a 3.11. ábra pedig ugyanennek a problémának a megoldását mutatja, ha az invertált állományokban logikai mutatót (az elsődleges kulcs értékeit) alkalmazunk a rekordok azonosítására – ez a fenti 2. eset.



3.10. ábra. Adatállomány elérése több indexen keresztül, sűrű index



3.11. 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. 7. fejezet).

Általánosan elterjedt megoldás, hogy egy rekord változó hosszúságú részeit a rekord mezőlistájának a végén helyezzük el. Így a rekord eleje fix hosszúságú marad.

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: az *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 építünk indexeket. Minden specifikált mező-érték alapján előállítjuk a találati rekord- (vagy legalább mutató-) halmazt, majd ezeknek a metszetét képezzük. Ez nem igazán praktikus.

Másik lehetőség: *particionált* (feldarabolt) *hash-függvények* 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 $h(m_1, m_2, \ldots, m_k) = h_1(m_1) * h_2(m_2) * \ldots * h_k(m_k)$ alakú, ahol az 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 $x_1 + x_2 + \ldots + x_k$ 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.

3.6. A fejezet új fogalmai

DRDB, IMDB, operatív tár, háttértár, elsődleges/másodlagos háttértár, soros hozzáférés, direkt/közvetlen hozzáférés, mágnesszalag, mágnesdob, mágneslemez, track, cilinder, szektor, fájl, blokk, blokk header, rekord, rekord header, mező, mutató (pointer), fizikai/logikai címzés, (keresési) kulcs, invertált állomány, kötött/szabad rekord, blocking factor, rekordelérési idő (min., max., átlagos), heap szervezésű állomány, fizikai segédstruktúra, hash szervezésű állomány, bucket hashing (vödrös hash), indexelt állományszervezés, ritka index, sűrű index, B*-fa, kiegyensúlyozott fa (balanced tree), fa magassága, elágazási tényező, elsődleges/másodlagos index, intervallumkeresés, több kulcs szerinti keresés, részleges információ alapján történő keresés, particionált hash

4. fejezet

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 *adat*bázist létrehozunk, a cél az, hogy benne a való – vagy ritkábban egy kitalált – világ *adat*ait 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- vagy jelenség-) körrel kapcsolatos összes adat tárolására, így adatoknak csak egy 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 adja vissza.

Amikor ezt a képet elkezdjük kialakítani, követhetünk bizonyos konvenciókat, ami számos előnnyel járhat. A konvenciók egy része arra vonatkozik, hogy milyen formában, milyen kapcsolatok kialakítását támogassuk az adataink között és hogy milyen műveleteket engedjünk meg az adatainkon. Így ún. adatmodelleket hozunk létre. Természetesen, a konvenciókhoz való alkalmazkodás járhat hátrányokkal is, ez esetben megfontolandó egy teljesen egyedi adatmodell megalkotása.

Egy *adatmodell* (data model) tehát hagyományosan két részből áll:

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

Az adatmodell tulajdonságai alapvetően meghatározzák az azt használó 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 egy ezzel

analóg, de esetleg másképpen nevezett konstrukció). Mivel egy adatbázis struktúráját jelentős részben a rekordtípusok közötti kapcsolatok határozzák meg, 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. 7. fejezet) a rekordtípusok között (pl. mutatók segítségével) tetszőleges függvényszerű kapcsolatokat szervezhetünk. A relációs adatmodellnél a kapcsolatok kialakítására nincs külön strukturális elem, magukat a kapcsolatokat is relációkkal ábrázoljuk (ld. 5. fejezet). Az objektumorientált adatmodell (ld. 8. fejezet) 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 adatbázis-kezelő rendszer legalapvetőbb tulajdonságait rögzíti. Egy adatbázis-kezelő rendszer ezért csaknem mindig egyetlen adatmodellnek megfelelően működik.

4.2. Egy majdnem-adatmodell: az egyed-kapcsolat modell

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

4.2.1. Az ER-modell elemei

Az ER-modell elemei:

- *egyedtípusok*;
- *attribútumtípusok*;
- *kapcsolattípusok*.

Természetesen, a *típusokhoz* mindenütt tartoznak konkrét *példányok* (*eset*, *előfordulás*) is, de maga a modellezés a típusok szintjén történik. Összhangban azzal, amit általában típusnak nevezünk, a típus itt is a konkrétan létező – de hasonló – 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.

4.2.1.1. Entitások

Definíció – *egyed,* entitás *(entity).* A valós világban létező, logikai vagy fizikai szempontból saját léttel rendelkező dolog, amelyről adatokat tárolunk.

Megjegyzés. Ennek megfelelően az egyedeknek megkülönböztethetőknek kell lenniük, mint ahogyan a matematikai értelemben definiált halmazok elemei is azok. 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.

Definíció – tulajdonság *(property).* Az entitásokat jellemzi, amelyen vagy amelyeken keresztül az entitások megkülönböztethetők.

Megjegyzés. 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* egy mesterségesen hozzájuk rendelt, egyediséget biztosító *attribútummal* (attribute) – a gyakorlatban pl. az elkülönített tárolásuk segítségével –, így a hangyák is entitásokká válhatnak.

Definíció – egyedhalmaz *(entity set).* Az azonos attribútumtípusokkal jellemzett egyedek összessége.

Az entitások közös attribútumtípusait zárójelben szokás az entitáshalmaz neve után felsorolni.

Példa.

- `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

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 kapcsolattípusokat definiálunk. Természetesen ezeknek a meghatározása szintén modellezési kérdés: az adott feladat dönti el, hogy egy konkrét adatbázisban milyen kapcsolattípusok definiálása szükséges.

Definíció – kapcsolat *(relationship)*. Entitások névvel ellátott viszonya.

Formálisan egy kapcsolattípus nem más, mint entitás típusok névvel ellátott sorozata.

Példa.

- *DOLGOZIK*: EMBER, CÉG
  Ez a bináris kapcsolattípus azt fejezheti ki, hogy valaki egy cégnél dolgozik.
- *ALÁÍR*: EMBER, CÉG, SZERZŐDÉS
  Ez a ternáris (hármas) kapcsolattípus azt fejezheti ki, hogy egy személy egy cég nevében egy szerződést aláírt.
- *TESTVÉRE*: EMBER, EMBER
  Ez a bináris kapcsolattípus 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 entitáshalmaz 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 példány, ha igen, akkor mennyi a kapcsolódó egyedek minimális, maximális száma stb. A kapcsolatok teljes mélységű jellemzése gyakran szükségtelen, mi is csak olyan mélységben fogjuk megtenni, amit a tervezett felhasználás indokol.

Megjegyzés. A mindennapi gyakorlatban (sajnos) rendszerint elfeledkezünk a típusok (halmazok) és a konkrét példányok megkülönböztetéséről. Igen gyakran emlegetünk entitást, kapcsolatot akkor is, amikor valójában entitás- vagy kapcsolattípusról/halmazról 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 *(one-to-one relationship)*. Olyan (bináris) kapcsolat, amelyben a résztvevő entitáshalmazok példányaival egy másik entitáshalmaznak legfeljebb egy példánya lehet kapcsolatban.

Példa.

- *HÁZASSÁG*: EMBER, EMBER
- *FŐNÖK*: OSZTÁLY, EMBER

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

Megjegyzés *(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 pontosan egy főnöke van. 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 *(many-to-one relationship)*. Egy *K*: E1, E2 kapcsolat több-egy, ha E1 példányaihoz legfeljebb egy E2-beli példány tartozhat, viszont E2 példányai tetszőleges számú E1-beli példányhoz tartoznak.

Példa. *TANUL*: DIÁK, OSZTÁLY

Itt tételezzük azt fel, hogy egy véletlenszerűen kiválasztott diák általában egy osztályban tanul (egyidejűleg), de egy meghatározott osztályba számos diák jár (egyidejűleg).

Definíció – több-több kapcsolat *(many-to-many relationship)*. Egy *K*: E1, E2 kapcsolat több-több, ha E1 példányaihoz is tetszőleges számú E2-beli példány tartozhat, és E2 példányaihoz is tetszőleges számú E1-beli példány tartozhat.

Példa. *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 adatbázis-kezelésben a több-egy (egy-több) kapcsolatok kitüntetett jelentőségűek, mert viszonylag egyszerűen ábrázolhatók, ugyanakkor elegendően általánosak, kifejezőek is.

4.2.2. Kulcs

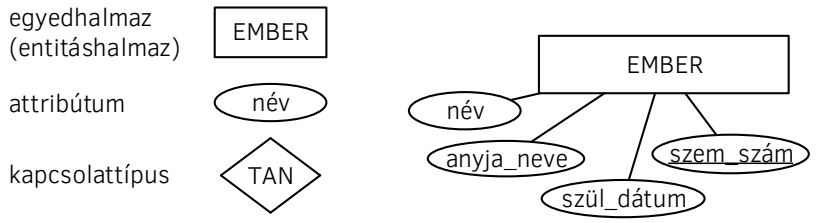
Definíció – kulcs *(key)*. Az ER-modellezésnél az attribútumoknak azt a halmazát, amely az entitás példányait egyértelműen azonosítja, *kulcsnak* nevezzük.

Példa. 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.

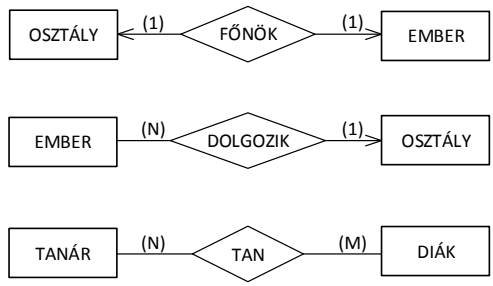
Minden egyedhalmaznak legalább egy kulcsa mindig van, hiszen az egyedeknek megkülönböztethetőknek kell lenniük. Ehhez pedig az attribútumok teljes halmaza elegendő, tehát az attribútumok teljes halmaza mindig kulcs. A kulcs attribútumait hagyományosan aláhúzással jelöljük.

4.2.3. Az ER-modell grafikus ábrázolása: ER-diagram

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



4.1. ábra. Az ER-diagram elemei



4.2. ábra. Kapcsolatok funkcionalitásának egy ábrázolása az ER-diagramoknál

Példa. A Nekeresdi Általános Biztosítónak számos kirendeltsége működik szerte Nekeresdország városaiban, néhányban több is. Minden kirendeltségnek külön kódszáma is van, amely egyértelműen azonosítja őket. 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 január 1-jei dátummal –, és a Nekeresdi általános Biztosítón belül másik kirendeltséghez mennek dolgozni.

A leírás alapján pl. az alábbi ER-modellt alkothatjuk.

Entitáshalmazok:

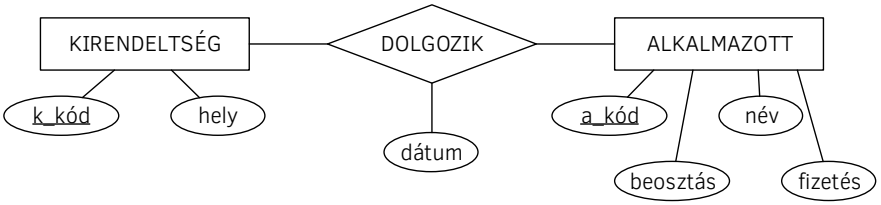
- `KIRENDELTSÉG(k_kód, hely)`
- `ALKALMAZOTT(a_kód, név, beosztás, fizetés)`

Kapcsolattípus:

- *`DOLGOZIK`*`: KIRENDELTSÉG, ALKALMAZOTT; dátum`

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 ER-modell azonban eddig nem engedte meg ilyen attribútumok definiálását. Megtehetjük, hogy ilyen esetekben a dátumot egy olyan entitáshalmaznak képzeljük el, amelynek egyetlen attribútuma a dátum. Ezt egyszerűsítve úgy is ábrázolhatjuk, ahogyan az a 4.3. ábrán látható.

Megjegyzés. Mivel a diagramot a fenti ER-modell – és nem az eredeti leírás – alapján alkottuk, ezért nem látszik rajta, hogy a *`DOLGOZIK`* kapcsolathalmaz 1:N funkcionalitású. Természetesen ezt egy jobb, pontosabb diagramon akár ábrázolhatnánk is.

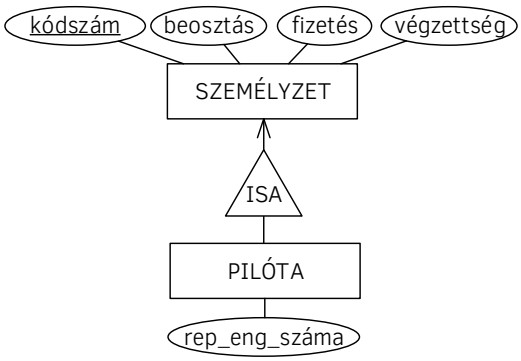


4.3. ábra. ER-diagram a fenti ER-modell alapján

Az ER-diagramok eszköztárát évtizedek alatt sokan, sokféle módon terjesztették ki, leginkább annak érdekében, hogy a gyakorlatban felmerülő számos különböző jelentést hordozó kapcsolatokat meg lehessen különböztetni. Így jöttek létre a különböző EER-diagramok (Extended ER). Az alábbiakban ennek néhány elemét mutatjuk be.

Gyakori az a modellezési szituáció, amikor egy entitáshalmaz minden eleme rendelkezik egy másik (általánosabb) entitáshalmaz attribútumaival, de azokon kívül még továbbiakkal is (specializáció). Ez a viszony a kapcsolatok egy sajátos típusával, az ún. „isa”[1] kapcsolattal írható le.

Példa.



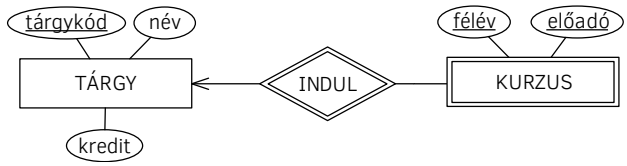
4.4. ábra. Specializáció ábrázolása ER-diagramon

Az isa kapcsolatnak az objektumorientált modelleknél kitüntetett szerepe van.

Szintén gyakori, hogy a modellezés során egy entitáshalmaznak nem tudunk kulcsot meghatározni, hanem az egyedek azonosításához valamely kapcsolódó egyed(ek)re is szükség van. Ebben az esetben *gyenge egyedhalmazról* (weak entity set) beszélünk. A gyenge egyedhalmaz identitását egy (vagy ritkán több) ún. *tulajdonos egyedhalmaz* (owner entity set) biztosítja, amely a gyenge egyedhalmazzal több-egy kapcsolatban áll. A kapcsolat neve *determináló kapcsolat* (identifying relationship).

[1] is a: angol.

A gyenge egyedhalmaz és determináló kapcsolatának szokásos jelölése a 4.5. ábra diagramján látható, ahol a `KURZUS` egyedhalmaznak nincs kulcsa, mert pl. a 2012/2013/1. félévben Gipsz Jakab több kurzust is vezethet. A kurzusokhoz a megfelelő tárgykódokat hozzárendelve lesznek az egyes kurzusok mint entitások egyértelműen megkülönböztethetők. A `KURZUS` tehát gyenge egyedhalmaz, az *`INDUL`* a determináló kapcsolata, ezért a `KURZUS` példányok egyedisége csak a `TÁRGY` példányaival együtt biztosítható.



4.5. ábra. Gyenge egyedhalmaz és a determináló kapcsolat ábrázolása ER-diagramon

A gyenge egyedhalmaz példányait (a hozzájuk tartozó tulajdonos egyedhalmaz kulcs attribútumaival együtt) egyértelműen megkülönböztető attribútumokat a kulcs attribútumokhoz hasonlóan aláhúzással jelöljük.

4.3. A fejezet új fogalmai

modell, modell az adatokról, adatmodell, formális jelölésrendszer, egyed-kapcsolat (ER) modell, ER-diagram, entitás, kapcsolat, tulajdonság (attribútum), egyedtípus, kapcsolattípus, tulajdonságtípus, kapcsolat fokszáma, kapcsolat funkcionalitása (kardinalitása), isa kapcsolat, gyenge egyed, determináló kapcsolat, kulcs

5. fejezet

A relációs adatmodell

A relációs adatmodellen alapuló adatbázis-kezelők ma a legelterjedtebbek, emiatt tanulmányozásuk kitüntetett figyelmet érdemel. Ezért a modell alapvető tulajdonságait ismertető jelen szakasz után a relációs adatbázisok logikai tervezésével a 9. fejezet külön foglalkozik.

5.1. Az adatok strukturálása

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

Definíció – reláció *(relation)*. Halmazok Descartes-szorzatának részhalmaza.

Adott n (valódi, azaz azonos elemeket nem tartalmazó) halmaz. A halmazokban található értékek egy-egy ún. *tartományból* (domain) kerülnek ki. Legyenek ezek rendre $D_1, D_2, \ldots, D_n$. A tartományok $D_1 \times D_2 \times \ldots \times D_n$ Descartes-szorzatában megtalálhatók mindazok a $(v_1, v_2, \ldots, v_n)$ n-esek (*tuple, n-tuple, elem, rekord, ennes*), amelyekre igaz, hogy $v_i \in D_i$, $\forall i = 1, 2, \ldots, n$-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.: $r_1 = \{(1,y), (1,z), (3,z)\}$, $r_2 = \{(2,y), (1,z)\}$

A modellben elvileg sem a domainek (domének) sorrendjének, sem az egyes relációkban található elemek sorrendjének nincs érdemi jelentősége. Vegyük észre, hogy a relációs adatmodellben tárolt adatok esetén a hasznos információt lényegében az hordozza, hogy az egyes relációkban, ill. egy reláció *elemeiben* mely értékeket

tároljuk *együtt* (azon a trivialitáson túl természetesen, hogy milyen adatok vannak a relációban).

Áttekinthetőbben ábrázolhatjuk relációnkat táblázatos formában. Az oszlopokat (ezek az *attribútumok,* amelyeknek szintén nevet adunk) hozzárendeljük a tartományokhoz, amelyekből az egyes oszlopokban található értékek kikerülnek. Az egyes attribútumok különböző értékeinek száma az attribútum *kardinalitása.* A táblázat sorai pedig a reláció *elemei*, az n-esek konkrét előfordulásai. A fejlécben az attribútumok megnevezése található.

Példa.

r_1:

D_1	D_2
1	y
1	z
3	z

r_2:

NÉV	SZÁM	HELYSZÍN
Chao Phraya	37	Bangkok
Kis József	45	Budapest
Nagy Imre	72	Bréma

Láthatóan az adatelemekhez (számokhoz, karakterláncokhoz) rendelhető információt – többek között – az hordozza (vagy inkább csak sugallja!), hogy milyen neveket adtunk az attribútumoknak. Az elsőnek a *NÉV* nevet adtuk, amiből leginkább csak arra következtethetünk, hogy pl. a Chao Phraya egy tulajdonnév. A SZÁM – mint attribútumnév – még kevésbé informatív, a *HELYSZÍN*-ből pedig pl. annyit tudhatunk meg, hogy a „Bangkok" karakterlánc egy helyszínt (is) jelöl. Az egyes attribútumértékekhez rendelhető információ – az adat értelmezése – jelen esetben tehát bizonytalan.

További információra akkor tehetünk szert az adatbázisból, ha azt is tudjuk, hogy a reláció egyazon elemének attribútumértékei összetartoznak. Az összetartozás szemantikája egyáltalán nem nyilvánvaló, és megfelelő dokumentáltság hiányában megfogalmazhatnánk pl. azt, hogy „Chao Phraya egy 37 éves bangkoki lakos". De akár azt is, hogy „a Chao Phraya utcában 37 ház van Bangkokban", azaz az ilyen módon dokumentált adatok segítségével kinyerhető információ értéke szinte zérus. A gyakorlatban súlyos tévedések forrása lehet, ha az attribútumok pontos jelentését csupán azok rövid neve sugallja a felhasználóknak, vagy a relációkhoz kapcsolódó szemantika nem/sem (kellően) definiált (ld. üzleti információs metaadattár, szemantikus adatkezelés, információ-, ill. tudásmenedzsment).

Gyakran magára a relációra nincs szükségünk, csak arra az információra, hogy melyik relációban milyen attribútumok találhatók. Ez a *relációs séma* (relational schema)[1]. Szokásos jelölése: $R(A_1, A_2, \ldots, A_k)$, ahol R a relációs séma neve, A_i-k az attribútumok nevei, ahol $1 \leq i \leq k$.

Példa. $SZEMÉLY(NÉV, KOR, FOGLALKOZÁS)$

Ha egy r reláció sémája R, akkor azt így jelöljük: $r(R)$.

[1] Figyelem: egyes kereskedelmi adatbázis-kezelők a séma fogalmát más értelemben használják.

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) száma a *reláció foka* (arity, aritás).
- A relációban lévő sorok száma (a konkrét előfordulások száma) a *reláció számossága*.

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 számítana akkor, ha csak a sorszámukkal hivatkoznánk rájuk.

Megjegyzés. Ebben a tárgyban az oszlopok (attribútumok) sorrendjének nem lesz jelentősége, a relációs sémát lényegében egy *attribútumhalmaznak* tekintjük. Létezik a relációs adatmodellnek olyan matematikai felépítése is, amikor a domaineknek, ill. az attribútumoknak a sorrendje jelentőséggel bír, ilyenkor a relációs séma valójában egy *attribútumlista*, természetesen csak ilyenkor van értelme a listaelemekre esetenként sorszámmal hivatkozni. A jegyzetben a precizitás rovására – különböző helyeken – mindkét módon hivatkozunk az attribútumokra, ha azt a könnyebb érthetőség indokolja.

5.2. Műveletek relációkon

A relációs adatmodell a relációkon megengedett műveletek meghatározásával válik teljessé. Ezen műveletekből épül fel az ún. *relációs algebra* (relational algebra). Tekintve, hogy a relációk is halmazok, mégpedig valódi halmazok (ismétlődés nem engedhető meg bennük) néhány halmazalgebrai műveletet relációs algebrai műveletként is fel kívánunk használni.

5.2.1. Egyesítés, unió (set union)

Az egyesítés feltétele, hogy az egyesítendő relációk sémáinak ugyanannyi attribútumból kell állniuk.[2] 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.

Példa.

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

5.2.2. Különbségképzés (set difference)

Ugyanazok a megkötések érvényesek a különbségképzés műveletre, mint az egyesítésnél.

Példa.

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

$r_1 \setminus r_2$:

1	2	3
a	b	c
c	b	a

Bevezethetnénk a *metszetképzés* (set intersection) műveletét is, azonban ez szükségtelen, mert a különbségképzés segítségével a metszet kifejezhető:

$$A \cap B = A \setminus (A \setminus B).$$

5.2.3. Descartes-szorzat (Cartesian product, cross product)

A reláció definíciójánál leírtaknak megfelelően az $r_1 \times r_2$ Descartes-szorzat eredménye az összes olyan $(n_1 + n_2)$-esekből áll, amelyeknek első n_1 attribútuma az első operandusból, második n_2 attribútuma a második operandusból származik, ebben a rögzített sorrendben. Az operandusok szerkezetére ebben az esetben semmilyen megkötést nem kell tennünk.

[2] Ez a korlátozás csak a relációs adatmodell implementálását könnyíti meg, hiszen a halmazelméleti unióképzésnél nincs ilyen megszorítás.

Példa.

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

5.2.4. Vetítés, projekció (projection)

A vetítés egyoperandusos művelete azt jelenti, hogy a reláció összes rekordjának egyes attribútumait megtartjuk, a többit pedig töröljük. Ennek során ki kell jelölnünk, hogy mely attribútumokat kívánjuk felhasználni.[3]

Példa.
Adott egy *GÉPKOCSI*(*ÁR*, *RENDSZÁM*, *ÉVJÁRAT*, *ELSŐ_TULAJDONOS*, *VIZSGA_ÉRVÉNYESSÉGE*, *TÍPUS*, *FOGYASZTÁS*) sémára illeszkedő reláció. Ekkor a $\pi_{TÍPUS,ÉVJÁRAT,FOGYASZTÁS}(gépkocsi)$ vetítés a *gépkocsi* reláció n-eseiből csak a *TÍPUS*, *ÉVJÁRAT*, *FOGYASZTÁS* azonosítójú attribútumoknak megfelelő hármasokat tartja meg.

A művelet implementációjakor, amennyiben az eredményben ismétlődések fordulnának elő, azokat meg kell szüntetni.

5.2.5. Kiválasztás, szelekció (selection)

A kiválasztás egyoperandusos művelete egy részhalmaz képzése az r reláción, amelynek vezérlésére egy logikai formula szolgál. Az r reláció minden elemére kiértékeljük a formulát, és azokat az elemeket vesszük be az új relációba, amelyekre a formula igaz értéket vesz fel.

Jelölése: $\sigma_F(r)$, 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 ($\wedge \vee \neg$).

Példa. A $\sigma_{KOR<23 \wedge NÉV='\text{Kovács}'}(névsor)$ kifejezés a *névsor* reláció azon elemeinek halmazát jelenti, amelyeknek *KOR* azonosítójú attribútuma kisebb huszonháromnál, *NÉV* azonosítójú attribútuma pedig Kovács.[4]

[3] Ha az előző lábjegyzetnek megfelelően attribútumhalmazok helyett attribútumlistákkal dolgozunk, akkor az egyes attribútumok sorszámukkal is azonosíthatók. Pl.: $R(A_1, A_2, \ldots, A_k)$ 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 $S(A_1, A_3, A_7, A_2)$.

[4] Attribútumlistás reprezentáció esetén az egyértelműség érdekében a numerikus konstansokat is aposztrófok közé írjuk, hogy meg lehessen különböztetni az attribútumok sorszámától.

Az 5.2.1–5.2.5. szakaszokban definiált műveletek a relációs algebra *alapműveletei*. Az alapműveletek láthatóan relációkat állítottak elő relációkból, a műveletek tehát *zártak a relációk halmazára*. Segítségükkel ugyanakkor változatos manipulációkat végezhetünk a relációinkon, néha akár többféle módon is. Ennek ellenére célszerű további, ún. (*le)származtatott műveletek*et is definiálni, mint pl. a különböző illesztések vagy a hányados műveletét, amelyekkel gyakori, viszonylag bonyolult műveleteket lehet nagyon tömör formában leírni.

5.2.6. Természetes illesztés (natural join)

Ez a művelet különösen nagy jelentőséggel bír majd a relációs adatbázisok logikai tervezéséhez kapcsolódóan (ld. 9. fejezet).

Adott két reláció, amelyeknek tipikusan van legalább egy – de akár több – megegyező nevű attribútuma. Vegyük sorra a két reláció minden rekordját (elemét), és válasszuk ki azokat, amelyeknek a megegyező nevű attribútumai érték szerint is megegyeznek. Fűzzük össze ezeket olyan rekordokká, amelyben a mindkét relációban szereplő, azonos nevű és értékű attribútumokat csak egyszer vesszük figyelembe. Ezen rekordokból képzett reláció lesz a természetes illesztés eredménye.

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

Legyen R és S a két adott reláció sémája. Legyenek $X_1, X_2, \ldots, X_n$ az azonos attribútumnevek. Ekkor

$$r \bowtie s = \pi_{R \cup S} \sigma_{(R.X_1 = S.X_1) \wedge \ldots \wedge (R.Xn = S.Xn)} (r \times s)$$

Az $R \cup S$ kifejezésben az unió művelet garantálja, hogy az azonos nevű attribútumok csak egyszer fordulnak elő az eredményhalmaz sémájában.

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

Példa *(1).* Kövessük végig mindezt egy egyszerű példán:

r:

A	B
a	b
a	c
b	b

s:

A	C
a	c
b	c

$r \times s$:

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

Példa *(2).* Adott az *osztály* nevű reláció, amelynek az elemei azt tartalmazzák, 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 reláció megfelelő elemeit (a közös *NÉV* attribútumon keresztül[5], pl. a természetes illesztés műveletével) összekapcsoljuk. Az így kapott *osztály* $\bowtie$ *személy* reláció tartalmazza minden olyan személy nevét, osztályának nevét, lakcímét és születési dátumát, akik neve mindkét relációban szerepelt. Így tehát már egyetlen relációban megtalálható az összes szükséges adat. Ebből kell kiválasztanunk azokat a sorokat, amelyekben az *OSZT_NÉV* pl. a Pénzügyi Osztálynak felel meg (PO), majd vetítenünk kell az eredményt a kérdéses attribútumokra: *NÉV*, *LAKCÍM*.

Formálisan: $\pi_{NÉV,LAKCÍM}\sigma_{OSZT\_NÉV='\mathrm{PO}'}(osztály \bowtie személy)$

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 ún. *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).

5.2.7. Θ-illesztés (Θ-join, theta join)

Legyen r és s két reláció, Θ pedig egy kvantormentes feltétel, melyet az r és s relációk egy-egy attribútuma között definiálunk. A táblák Descartes-szorzatából a rekordpáron értelmezett Θ feltétel szerint választunk ki sorokat: $t = \sigma_\Theta(r \times s)$. Az így definiált t reláció állítja elő az r és s relációkon végzett Θ-illesztés műveletet. Jelölése: $r \underset{\Theta}{\bowtie} s$

Példa. Nézzük két egyszerű példát.

[5] Itt hallgatólagosan azt is feltételezzük, hogy az azonos nevű attribútomokhoz azonos szemantika tartozik, hiszen valójában számunkra ez a fontos akkor, amikor a helyes lakcímet meg akarjuk kapni. Ha csak formálisan végzünk el valamilyen illesztést azonos nevű attribútomok mentén, akkor az eredményreláció elemeihez nem feltétlenül fogunk tudni használható információt hozzárendelni.

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

$r \underset{\mathrm{B=D}}{\bowtie} s$:

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

$r \underset{\mathrm{B<D}}{\bowtie} s$:

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

5.2.8. Hányados (division)

Jelölje $r \div s$ azt a relációt, amelyre igaz az, hogy az s-sel alkotott Descartes-szorzata a lehető legbővebb részhalmaza r-nek: $(r \div s) \times s \subseteq r$. Ezzel a relációval definiáljuk a relációs algebra r és s relációk között értelmezett hányados műveletét.

Példa. Nézzük egy egyszerű példát.

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

$r \div s$:

C	D
a	c
c	d

5.2.9. Példák a relációs algebra alkalmazására

Egy boltban az alábbi adatokat gyűjtik:

- záróösszeg (a pénztár tartalma a nap végén) ($\ddot{O}SSZEG$)
- 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 $\ddot{O}SSZEG - 4000$ kerül ($BEFIZ$).

A 9.1. szakaszban meg fogjuk konstruálni ehhez a feladathoz az alábbi relációs sémákat, melyeket most megelőlegezünk:

- $ÁRU(ÁRUKÓD, ÁRUNÉV, EGYSÁR)$
- $MENNYISÉG(DÁTUM, ÁRUKÓD, DB)$
- $BEVÉTEL(DÁTUM, \ddot{O}SSZEG)$

– $BEFIZ(\ddot{O}SSZEG, BEFIZ)$[6]

Ezen sémákra illeszkedő relációk segítségével fejezzük ki az alábbi relációkat:

– Az 1997. jan. 1. és utána következő napok bevételei a dátummal együtt:

$$\sigma_{DÁTUM \geq \text{'19970101'}}(bevétel)$$

– Ha meg akarjuk cserélni az attribútumok sorrendjét:

$$\pi_{\ddot{O}SSZEG, DÁTUM} \sigma_{DÁTUM \geq \text{'19970101'}}(bevétel)$$

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

$$\pi_{\ddot{O}SSZEG, BEFIZ} \sigma_{DÁTUM = \text{'19970115'}}(bevétel \bowtie befiz)$$

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:

$$\pi_{\ddot{O}SSZEG, BEFIZ}((\sigma_{DÁTUM = \text{'19970115'}}(bevétel)) \bowtie befiz)$$

Természetesen, a természetes illesztés helyett az alapműveleteket is használhatjuk:

$$\pi_{\ddot{O}SSZEG, BEFIZ} \ \sigma_{DÁTUM = \text{'19970115'} \wedge \text{BEFIZ}.\ddot{O}SSZEG = \text{BEVÉTEL}.\ddot{O}SSZEG}(bevétel \times befiz)$$

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

$$\pi_{DB, ÁRUNÉV, EGYSÉGÁR} \ \sigma_{ÁRUKÓD = \text{'A1'} \wedge DÁTUM = \text{'19970115'}}(mennyiség \bowtie áru)$$

– Melyek azok a napok, amikor mindenféle áruból adtak el?

$$(\pi_{DÁTUM, ÁRUKÓD} mennyiség) \div \pi_{ÁRUKÓD} áru$$

5.3. Relációs lekérdező nyelvek (relational query languages)

A legtöbb relációs adatbázis-kezelő rendszer lekérdező nyelve alapvetően

– relációs algebra (pl. ISBL) vagy
– relációs sorkalkulus (pl. QUEL) vagy

[6] Megjegyzendő, hogy nem szerencsés az adatbázis sémába felvenni *BEFIZ* attribútumot, amely származtatott értékeket tartalmaz, de egyéb szempontok miatt most tekintsünk el ettől.

– 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 5.2. 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 eredményhalmaz elemeinek tulajdonságaira 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 adatbázis-kezelők sikerének.[7]

5.3.1. Relációs sorkalkulus (tuple relational calculus)

A relációs sorkalkulus (a továbbiakban: sorkalkulus) egy elsőrendű nyelv,[8] amely tehát *kvantorokat* (quantifier) is tartalmazhat és a kvantorok sorvektor változókat kvantifikálhatnak.

Felépítése: a nyelv *szimbólumaiból atomokat* (alapformulákat, prímformulákat) hozhatunk létre, amelyek *formulákká* építhetők össze, a formulák pedig egy *kifejezésbe* építve alkalmasak arra, hogy segítségükkel relációkat írjunk le.

Szimbólumai (symbol):

– zárójelek: (,)
– aritmetikai relációjelek: $<$, $>$, $=$, $\neq$, $\leq$, $\geq$
– 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[9]
– konstansok: c
– kvantorok: $\exists$, $\forall$

Az *atomok* (atomic formulae) 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
– $s^{(n)}[i]\Theta\, c$
– $R^{(n)}(c_1, c_2, \ldots, c_n)$

[7] A lekérdezések deklaratív megfogalmazásának lehetőségén kívül.
[8] Ld. matematikai logika, formális nyelvek.
[9] A kalkulus kifejezéseknél – a korábban használt konvencióval ellentétben – a relációkat nagybetűvel írjuk, hogy megkülönböztethetők legyenek a sorváltozóktól.

A *formulák* (formulae) 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* (free tuple variable), akkor $(\exists s^{(n)})\Psi$ és $(\forall s^{(n)})\Psi$ is formulák[10], amelyekben $s^{(n)}$ már *kötött sorváltozó* (bound tuple variable).

A *kifejezések* (expressions) felépítése:

$$\left\{ s^{(m)} \,\middle|\, \Psi(s^{(m)}) \right\},$$

ahol $s^{(m)}$ a $\Psi(s^{(m)})$ formula egyetlen szabad sorváltozója.

Példa.

- 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 (értelmezését) rögzítve, hogy a változók és a konstansok milyen értékeket vehetnek fel. Ehhez meg kell adnunk ezeknek az értékeknek a halmazát, ez lesz a formulához tartozó *interpretációs halmaz* (domain of discourse). Az egyszerűség kedvéért most minden változóhoz és konstanshoz egyetlen közös halmazt rendelünk hozzá.

Legyen pl. A tetszőleges, véges, számítógépben ábrázolható számhalmaz. Tételezzük fel, hogy $c \in A, s^{(n)} \in A^n, R^{(m)} \subseteq A^m$.

Ezen interpretációs halmaz elemeit felhasználva a sorváltozók minden komponense értéket kaphat, ami után minden formulához egy igazságérték rendelhető.

Az *igazságértékek* (truth values) 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 ($s^{(n)} \in A^n, u^{(k)} \in A^k, c \in A$).
- $R^{(n)}(c_1, c_2, \ldots, c_n)$ pontosan akkor igaz, ha $(c_1, c_2, \ldots, c_n) \in R^{(n)}$ ($c_i \in A$).
- Ψ_1 szabad sorváltozói az $s_t^{(n_t)} \in A^{n_t}$, $t = 1, 2, \ldots, r_1$ értékeket, míg Ψ_2 szabad sorváltozói a $v_j^{(k_j)} \in A^{k_j}$, $j = 1, 2, \ldots, r_2$ 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.

[10] A kvantor hatóköre a sor végéig, ill. zárójelezett formula – pl. $((\exists s^{(n)})\ldots)$ – esetén a bezáró zárójelig tart.

– Ha Ψ_1 szabad sorváltozója $l^{(k)}$ és sorváltozói az $s_t^{(n_t)} \in A^{n_t}$ $(t = 1, 2, \ldots, r)$ értékeket vették fel, akkor
  - $(\exists l^{(k)})\Psi(l^{(k)}, s_1^{(n_1)}, s_2^{(n_2)}, \ldots, s_r^{(nr)})$ pontosan akkor igaz, ha van olyan $u^{(k)} \in A^k$, amelyre $\Psi(u^{(k)}, s_1^{(n_1)}, s_2^{(n_2)}, \ldots, s_r^{(nr)})$ igaz, továbbá
  - $(\forall l^{(k)})\Psi(l^{(k)}, s_1^{(n_1)}, s_2^{(n_2)}, \ldots, s_r^{(nr)})$ pontosan akkor igaz, ha minden $u^{(k)} \in A^k$ esetén $\Psi(u^{(k)}, s_1^{(n_1)}, s_2^{(n_2)}, \ldots, s_r^{(nr)})$ igaz.

A kifejezések interpretációja: $\left\{ s^{(m)} \middle| \Psi(s^{(m)}) \right\}$ pontosan azoknak az $s^{(m)} \in A^m$-eknek a halmaza, amelyekre a Ψ formula igaz. A kifejezések 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.

Példa *(Az 5.2.9. alszakasz relációit fejezzük ki ismét).*

– Az 1997. jan. 1. és az utána következő napok bevételei a dátummal együtt:

$$\left\{ t^{(2)} \middle| \text{BEVÉTEL}(t) \wedge t[1] \geq \text{'19970101'} \right\}$$

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

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

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

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

Természetes módon adódik a kérdés, hogy van-e olyan „jó” a sorkalkulus, mint a relációs algebra, azaz minden relációs algebrai kifejezéshez meg tudjuk-e konstruálni annak sorkalkulus megfelelőjét? A választ pontosabban az alábbi tétel adja meg.

Tétel. Minden, az $R_k^{(f_k)} \subseteq A^{f_k}$, $(k = 1, 2, \ldots, r)$ relációkból felépített E relációs algebrai kifejezéshez van olyan $\Psi(s^{(m)})$ sorkalkulus formula, hogy Ψ csak az $R_k^{(f_k)}$-k közül tartalmaz relációkat és az E kifejezés megegyezik az $\left\{ s^{(m)} \middle| \Psi(s^{(m)}) \right\}$ kifejezéssel.

Bizonyítás. Az E kifejezésben található műveletek száma n szerinti teljes indukcióval.

$n = 0$, azaz nincs művelet E-ben. E szükségszerűen egyetlen relációt tartalmaz, legyen ez a k-ik. Ekkor $\Psi(s^{(f_k)}) = R_k^{(f_k)}(s^{(f_k)})$ miatt teljesül az állítás.

T. f. h. a legfeljebb n műveletet tartalmazó E relációs algebrai kifejezésekre az állítás még igaz. Igaz-e $n+1$ műveletre is? Vizsgáljuk meg az öt relációs algebrai alapművelet szerinti esetszétválasztással az $n+1$ műveletet tartalmazó relációs algebrai kifejezéseket.

Igaz volt tehát, hogy E_1, E_2 rendre $n_1, n_2 \leq n$ műveletet tartalmazó relációs algebrai kifejezéshez létezik $\Psi_1(t_1^{(x_1)}), \Psi_2(t_2^{(x_2)})$ sorkalkulus formula, hogy $E_1 = \left\{ t_1^{(x_1)} \middle| \Psi_1(t_1^{(x_1)}) \right\}$ és $E_2 = \left\{ t_2^{(x_2)} \middle| \Psi_2(t_2^{(x_2)}) \right\}$.

1. $E = E_1 \cup E_2$. Itt $x_1 = x_2$ és $n_1 + n_2 = n$. Pontosan ekkor lesz $E_1 \cup E_2$ elvégezhető, $n+1$ műveletet tartalmazó relációs algebrai kifejezés. Ekkor az $E = \left\{ t^{(x_1)} \middle| \Psi_1(t^{(x_1)}) \vee \Psi_2(t^{(x_2)}) \right\}$ az $E_1 \cup E_2$ megfelelője.
2. $E = E_1 \setminus E_2$. Itt $x_1 = x_2$ és $n_1 + n_2 = n$. Pontosan ekkor lesz $E_1 \setminus E_2$ elvégezhető, $n+1$ műveletet tartalmazó relációs algebrai kifejezés. Ekkor az $E = \left\{ t^{(x_1)} \middle| \Psi_1(t^{(x_1)}) \wedge \neg\Psi_2(t^{(x_2)}) \right\}$ az $E_1 \setminus E_2$ megfelelője.
3. $E = E_1 \times E_2$, ami pontosan az $n_1 + n_2 = n$ esetben $n+1$ műveletet tartalmazó, elvégezhető relációs algebrai kifejezés. Ekkor az $E_1 \times E_2$ kifejezés sorkalkulus megfelelője.

$$E = \Big\{ t^{(x_1+x_2)} \Big| (\exists t_1^{(x_1)})(\exists t_2^{(x_2)}) \Psi_1(t_1^{(x_1)}) \wedge \Psi_2(t_2^{(x_2)}) \wedge$$
$$t^{(x_1+x_2)}[1] = t_1^{(x_1)}[1] \wedge \ldots \wedge t^{(x_1+x_2)}[x_1] = t_1^{(x_1)}[x_1] \wedge$$
$$t^{(x_1+x_2)}[x_1+1] = t_2^{(x_2)}[1] \wedge \ldots \wedge t^{(x_1+x_2)}[x_1+x_2] = t_2^{(x_2)}[x_2] \Big\}$$

4. $E = \pi_{i_1, i_2, \ldots, i_y}(E_1)$. Itt $n_1 = n$ és $1 \leq i_1, i_2, \ldots, i_y \leq x_1$. A kifejezés sorkalkulus megfelelője:

$$E = \Big\{ t^{(y)} \Big| (\exists t_1^{(x_1)}) \Psi_1(t_1^{(x_1)}) \wedge t^{(y)}[1] = t_1^{(x_1)}[i_1] \wedge$$
$$t^{(y)}[2] = t_1^{(x_1)}[i_2] \wedge \ldots \wedge t^{(y)}[y] = t_1^{(x_1)}[i_y] \Big\}$$

5. $E = \sigma_F(E_1)$. Itt $n_1 = n$. A kifejezés sorkalkulus megfelelője: $E := \left\{ t^{(x_1)} \middle| \Psi_1(t^{(x_1)}) \wedge F' \right\}$, ahol F' úgy kapható az F logikai formulából, hogy az E_1 eredményreláció sémájának i-ik ($1 \leq i \leq x_1$) attribútumára történő hivatkozást $t^{(x_1)}[i]$-vel helyettesítjük. Ekkor belátható, hogy F' sorkalkulus formula lesz, melynek egyetlen szabad változója $t^{(x_1)}$ és csak olyan relációkat tartalmaz, amelyeket F is tartalmazott.

Az esetszétválasztás az összes relációs algebrai alapművelet, mint a kifejezés $n+1$-ik művelete esetén megmutatta, hogy létezik a keresett $\Psi(s^{(m)})$ sorkalkulus formula, ha teljesül az indukciós feltétel.

Összegezve tehát megállapítható, hogy a sorkalkulus ***kifejezőereje* (expressive power) legalább akkora**, mint a relációs algebráé.

A tétel megfordítása semmiképpen nem igaz, hiszen a

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

relációt nem tudjuk a relációs algebra alapműveleteivel kifejezni. Tehát a sorkalkulus kifejezőereje nagyobb, mint a relációs algebráé.

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 (emlékezzünk, hogy A akár tetszőlegesen nagy, de *véges*, számítógépben ábrázolható halmaz) mellett már kis n-ek esetén is ábrázolhatatlanná teheti az eredményhalmazt.[11] A probléma úgy is jelentkezhet, hogy csak a részformulák eredményeiként keletkező közbenső relációk nőnek kezelhetetlen méretűvé, maga a végeredmény már nem ilyen.[12] Mivel ez implementációs problémákat okoz, ezért célravezetőnek tűnik a sorkalkulus szűkítése.

5.3.1.1. Biztonságos sorkalkulus (safe tuple relational calculus)

A probléma megoldására vezették be az ún. *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 halmazát, amelyek a sorkalkulus kifejezés formuláját igazzá tehetik egy olyan halmazra, amely magából a bemeneti relációkból és esetleges egyéb konstansokból áll. Ez a halmaz a *formula doménje*, DOM(Ψ) lesz.

[11] Ilyenkor kvázi végtelen (eredmény)halmazról beszélhetünk.

[12] Vegyük észre, hogy a relációs algebra a definiált 5 alapműveletével zárt abban az értelemben, hogy véges relációkból véges relációkat állít elő.

Definíció – formula doménje *(domain)*. $\mathrm{DOM}(\Psi) \equiv \{\Psi\text{-beli alaprelációk összes attribútumának értékei}\} \cup \{\Psi\text{-ben előforduló konstansok}\}$

$\mathrm{DOM}(\Psi)$ tehát egy egydimenziós halmaz.

Példa. $\Psi(x^{(2)}) := (x^{(2)}[1] = 28) \wedge R^{(2)}(x^{(2)})$ esetén

$$\mathrm{DOM}(\Psi) = \{28\} \cup \pi_1(r(R)) \cup \pi_2(r(R)).$$

Definíció – biztonságos kifejezés *(safe expression)*. $\{t \,|\Psi(t)\}$ biztonságos, ha

a) minden $\Psi(t)$-t kielégítő t minden komponense $\mathrm{DOM}(\Psi)$-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 $\mathrm{DOM}(\omega)$-beli (a *részformula biztonságos*).

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

Módszerek biztonságosság eldöntésére:

- $(\exists u)R(u) \wedge \ldots$ alakú formulák mindig biztonságosak u-ban.
- $(\forall u)\omega(u)$ alakú részformulák ekvivalensek $\neg(\exists u)\neg\omega(u)$-val.

Megjegyzés. Egyes szerzők egy harmadik feltételt is definiálnak, hogy az ellenőrzéshez ne kelljen átalakítani az univerzális kvantort tartalmazó részformulákat:

c) Ψ-nek minden $(\forall u)\omega(u)$ alakú részformulájára teljesül, hogy ha u valamely komponense nincs $\mathrm{DOM}(\omega)$-ban, akkor u kielégíti ω-t az ω-beli szabad változók minden értéke mellett.

Ez a feltétel levezethető a *b)* feltételből az alábbiak szerint. Tudjuk, hogy $(\forall u)\omega(u)$ formula ekvivalens a $\neg(\exists u)\neg\omega(u)$ formulával. A *b)* feltétel miatt $\neg(\exists u)\neg\omega(u)$ akkor biztonságos, ha a $(\exists u)\neg\omega(u)$ részformulára teljesül, hogy ha u kielégíti ω-t az ω-beli szabad változók valamely értéke mellett, akkor u minden komponense $\mathrm{DOM}(\neg\omega)$-beli. Jelölje $t \in^* \mathrm{DOM}(\alpha)$ azt, hogy t minden komponense $\mathrm{DOM}(\alpha)$-beli, ekkor formálisan: $(\exists u)\neg\omega(u)$ akkor és csak akkor biztonságos, ha $\forall u_0$-ra $\neg\omega(u_0) \rightarrow u_0 \in^* \mathrm{DOM}(\neg\omega)$. Az implikációt[a] átalakíthatjuk az $A \rightarrow B \equiv \neg A \vee B$ azonosság alapján, így a feltétel: $\omega(u_0) \vee u_0 \in \mathrm{DOM}(\neg\omega)$. A domén definíciója miatt $\mathrm{DOM}(\neg\omega) = \mathrm{DOM}(\omega)$, hiszen $\neg\omega$ ugyanazokat az alaprelációkat és konstansokat tartalmazza, mint ω, ezért a feltétel $\omega(u_0) \vee u_0 \in \mathrm{DOM}(\omega)$. Átrendezve $\neg(u_0 \notin^* \mathrm{DOM}(\omega)) \vee \omega(u_0)$, amit átírhatunk implikációra: $\forall u_0$-ra $u_0 \notin^* \mathrm{DOM}(\omega) \rightarrow \omega(u_0)$. Vagyis minden doménen kívüli u_0 elem kielégíti ω-t az ω-beli szabad változók minden lehetséges értéke mellett.

[a] Az implikációt ld. részletesebben az A. függelékben.

Példa *(Biztonságos kifejezések).*

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

Összetett kifejezés biztonságosságának vizsgálata. Vizsgáljuk meg a

$$\{t\,|R_1(t) \wedge t[1] > 2 \wedge ((\exists u)\neg\mu(t,u) \wedge R_2(u)) \wedge ((\forall v)\neg R_3(v) \vee \lambda(t,v))\}$$

kifejezés biztonságosságát, ha tudjuk, hogy $\mu(x,y)$ és $\lambda(x,y)$ biztonságos formulák!

- Az első feltétel teljesül, mivel a kifejezés formulája $(R_1(t) \wedge \ldots)$ alakú, ezért az összes, formulát kielégítő t csak R_1-beli értéket vehet fel.
- A második feltételhez mindkét kvantort tartalmazó részformulát megvizsgáljuk:
  - a $(\exists u)\neg\mu(t,u) \wedge R_2(u)$ részformula biztonságos u-ban, mivel az összes, $(\neg\mu(t,u) \wedge R_2(u))$ formulát igazzá tevő u kizárólag az R_2 reláció elemei közül kerülhet ki.
  - a $(\forall v)\neg R_3(v) \vee \lambda(t,v)$ univerzális kvantort tartalmazó részformulát először alakítsuk át a következőképpen: $\neg(\exists v)R_3(v) \wedge \neg\lambda(t,v)$.

Mivel a kifejezés kielégíti a biztonságosság mindkét feltételét, így a kifejezés biztonságos.

Megjegyzés *(1).* Érdemes megvizsgálni a $(\forall v)\neg R_3(v) \vee \lambda(t,v)$ részformulát a harmadik, kiegészítő feltétel tekintetében is. Mivel a részformula magja $\neg R_3(v) \vee \ldots$ alakú, ezért kijelenthető, hogy a v kötött változó minden doménen kívüli értéke kielégíti a részformulát a t szabad változó minden lehetséges értéke mellett, így az előzővel összhangban a részformula biztonságos.

Megjegyzés *(2)*. Fontos megfigyelés, hogy egy biztonságos kifejezés formuláját negálva nem biztonságos kifejezést kapunk. Az állítás visszafelé nem igaz, nem biztonságos kifejezés formuláját negálva nem feltétlenül kapunk biztonságos kifejezést. Pl. a $\Psi(t^{(1)}) = (t^{(1)}[1] < 2)$ formula esetén a $\left\{ t^{(1)} \middle| \Psi(t^{(1)}) \right\}$ kifejezés eredményhalmaza az A halmaz 2-nél kisebb, a $\left\{ t^{(1)} \middle| \neg\Psi(t^{(1)}) \right\}$ kifejezés a 2-nél nagyobb vagy egyenlő számait tartalmazza,[a] vagyis egyik kifejezés sem korlátozott a DOM(Ψ)-re, ezért egyik sem biztonságos.

[a] Az A halmaz ismeretének hiányában ennél többet nem állíthatunk az eredményhalmazról, az tartalmazhat például negatív vagy valós számokat is.

Megjegyzés *(3)*. A formulák biztonságosságának meghatározása és a formula kiértékelése két különböző dolog. Pl. egy $(\neg S_1(t) \wedge S_2(t))$ formula *biztonságos*, hiszen csak S_2-beli t helyettesítések tehetik igazzá. Tehát ki *lehet* véges idő, ill. véges (közbülső) eredményhalmazok mellett is értékelni. Ugyanakkor ha egy egyszerű automata megpróbálná kiértékelni (balról jobbra), akkor $\neg S_1(t)$ kiértékelése nagy valószínűséggel sikertelen lenne, emiatt a teljes formuláé is az lehet.

Megjegyzés *(4)*. Egy Ψ *biztonságos* sorkalkulus kifejezés eredményhalmazát előállító egyszerű algoritmus az alábbi elven működhet. Ha a Ψ kifejezés biztonságos, akkor az eredményhalmazban csak DOM(Ψ)-beli elemekből álló n-esek szerepelhetnek. Az első lépés tehát a DOM(Ψ) meghatározása a Ψ kifejezésben szereplő alaprelációk és konstansok alapján. Ezután végigiterálhatunk a DOM(Ψ) elemeiből mint komponensekből képzett összes (a kifejezésnek megfelelő dimenziójú) $v^{(n)}$ n-esen, és megvizsgáljuk, hogy kielégíti-e a kifejezés formuláját: ha igen, megy az eredményhalmazba, különben folytatjuk a következő n-essel.

A kvantoros részformulák igazságértékét a következőképpen határozzuk meg. Meghatározzuk a doménjét, majd az ebből képzett $w^{(m)}$ m-eseken végigiterálunk, és ellenőrizzük, hogy a külső iteráció $v^{(n)}$ n-esére és a belső ciklus $w^{(m)}$ m-esére teljesül-e a belső formula minden ($\forall$) vagy legalább egy ($\exists$) iterációban. Többszintű kvantálás esetén ezt rekurzívan ismételjük.

Tétel. A relációs algebra és a biztonságos sorkalkulus kifejezőereje ekvivalens.

Bizonyítás.

Relációs algebra $\Rightarrow$ biztonságos sorkalkulus

Teljes indukcióval, az előző tétel alapján belátható. Csupán azt kell még feltételezni, hogy az n-edik lépésben $E_1 = \left\{ t_1^{(n)} \middle| \Psi_1(t_1^{(n)}) \right\}$ és $E_2 = \left\{ t_2^{(m)} \middle| \Psi_2(t_2^{(m)}) \right\}$ 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:

- Egyrészt $\Psi_1(t) \vee \Psi_2(t)$ 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 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 relációs algebrai alapműveletre a biztonságosság hasonlóképpen belátható.

Biztonságos sorkalkulus $\Rightarrow$ relációs algebra

Nem bizonyítjuk.

5.3.2. Relációs oszlopkalkulus (domain relational calculus)

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:

– ...

A kifejezések felépítése:

$$\{x_1, x_2, \ldots x_m |\, \Psi(x_1, x_2, \ldots x_m)\},$$

ahol Ψ olyan formula, amelynek szabad változói csak $x_1, x_2, \ldots x_m$.

Példa. (Azonosak a sorkalkulus szakasz példáival.)

– Az 1997. jan. 1. és az utána következő napok bevételei a dátummal együtt:

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

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

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

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

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

Példa *(2)*. Fogalmazzuk meg oszlopkalkulussal az alábbi kifejezést: azokat a boltokat keressük, ahol a boltban kapható minden termék finom. A boltok listája a $bolt^{(1)}(bolt)$, a finom termékek listája a $finom^{(1)}(termék)$ relációban, az egyes boltok kínálata a $kapható^{(2)}(bolt,\ termék)$ relációban található.

bolt

bolt
Alfa Cukrászda
Lambda Cukrászda
Szigma Cukrászda

finom

termék
dobostorta
almás pite

kapható

bolt	*termék*
Lambda Cukrászda	dobostorta
Lambda Cukrászda	almás pite
Szigma Cukrászda	fűrészporos pogácsa

A feladat szerint olyan boltokat keresünk, amelyekre igaz, hogy minden termékük szerepel a *finom* relációban. Egy tipikus hiba, hogy a lekérdezést az alábbi módon fogalmazzuk meg:

$$\left\{ b \,\middle|\, bolt^{(1)}(b) \wedge (\forall t) kapható^{(2)}(b, t) \wedge finom^{(1)}(t) \right\}$$

Ezzel a lekérdezéssel ugyanis olyan b boltokat keresünk, amelyekre igaz, hogy minden t termékkel a (b,t) pár eleme a *kapható* relációnak és a t termék eleme a *finom* relációnak. A minden kvantor miatt azonban a t oszlopváltozó értéke tetszőleges értéket felvehet. Például $t =$ „rémes krémes” esetén a finom$^{(1)}(t)$ – a fenti relációk esetén – hamis, ezért a kvantor utáni formula értéke is hamis, így a kifejezés eredményhalmaza üres lesz.

Fontos, hogy a „rémes krémes” terméknek nem kell sem a *kapható*, sem a *finom* relációban előfordulnia. Ennek oka, hogy a minden kvantor definíciója szerint $(\forall l^{(k)})\Psi(l^{(k)}, s_1^{(n_1)}, s_2^{(n_2)}, \dots s_r^{(nr)})$ pontosan akkor igaz, ha minden $u^{(k)} \in A^{(k)}$ esetén$\Psi(u^{(k)}, s_1^{(n_1)}, s_2^{(n_2)}, \dots s_r^{(nr)})$ igaz, ahol A egy tetszőleges, véges, de nem korlátos, számítógépen ábrázolható számhalmaz. Fontos, hogy az A halmaz véges, de nem meghatározott méretű, ezért tetszőlegesen nagy lehet. Így az A halmazban biztosan szerepel a „dobostorta”, „almás pite” stb. mellett a „rémes krémes” is.

A kifejezést helyesen az alábbi módon fogalmazhatjuk meg:[13]

$$\left\{ b \,\middle|\, \mathit{bolt}^{(1)}(b) \wedge (\forall u)\mathit{kapható}^{(2)}(b,u) \rightarrow \mathit{finom}^{(1)}(u) \right\}$$

Azaz azokat a b boltokat keressük, amelyekre igaz, hogy ha egy u termék a b boltban kapható, akkor u finom is. A kalkulus kifejezések formális nyelvtanában nem szerepel az implikáció, ezért átírjuk úgy, hogy csak $\neg$, $\wedge$, $\vee$ logikai műveleteket használjon:

$$\left\{ b \,\middle|\, \mathit{bolt}^{(1)}(b) \wedge (\forall u)\neg\mathit{kapható}^{(2)}(b,u) \vee \mathit{finom}^{(1)}(u) \right\}$$

A kifejezés szerint az eredményhalmazba olyan boltok kerülnek, ahol minden termékre igaz, hogy az adott boltban nem kapható vagy finom – azaz minden termékük finom.

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. $\left\{ s^{(m)} \,\middle|\, \Psi(s^{(m)}) \right\}$ ekvivalens $\{ x_1, x_2, \dots, x_m |\, \Psi' \}$-vel, hiszen $s^{(m)}$ helyére $x_1, x_2, \dots, x_m$-et írunk és Ψ-ből Ψ'-t úgy kapjuk, hogy $R^{(n)}(t)$ helyére $R^{(n)}(x_1, x_2, \dots, x_n)$-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 $\left\{ s^{(m)} \,\middle|\, \Psi(s^{(m)}) \right\}$ biztonságos, akkor

[13] Az implikációt ld. részletesebben az A. függelékben.

a neki megfelelő $\{ x_1, x_2, \ldots, x_m | \Psi' \}$ is biztonságos lesz. Ezzel beláttuk, hogy a relációs algebra, a biztonságos sorkalkulus és a biztonságos oszlopkalkulus kifejezőerejüket tekintve ekvivalensek egymással.

A kereskedelmi rendszerek konkrét lekérdező nyelvei ennél valamivel bővebbek, tipikusan aritmetikával, *aggregációval* (aggregation) is kiegészítettek (képesek az aritmetikai alapműveleteken kívül átlag, szórás, minimum stb. képzésére is), továbbá támogatják a rekordok megváltoztatását, új rekordok bevitelét is. Amennyiben mindaz lekérdezhető velük, mint ami a megismert három ekvivalens kifejező erejű absztrakt nyelv közül bármelyikkel, akkor *relációsan teljesnek* (relationally complete) nevezzük a lekérdező nyelvet.

5.4. Az SQL nyelv[14]

A relációs adatbázis-kezelőkkel történő kommunikáció manapság legelterjedtebb nyelve az ún. *strukturált lekérdezőnyelv* (Structured Query Language, SQL).

5.4.1. Az SQL története

- Codd 1970-es cikkében[15] vezeti be a relációs adatmodellt az adatbáziskezelés világába.
- 1974–75-ben kezdték – a relációs adatmodellre alapozva – a kifejlesztését az IBM-nél, az „eredeti" neve SEQUEL (Structured English QUEry Language).
- 1979-től több cég (pl. IBM, Relational Software Inc., a mai Oracle Corp.) kereskedelmi forgalomban kapható termékeiben.
- 1987-től ANSI szabvány.

5.4.2. A nyelv jelentősége

- Szabvány, amelyet jelenleg csaknem minden relációs adatbázis-kezelő alkalmaz (kisebb-nagyobb módosításokkal).
- Tömör, felhasználó közeli nyelv, alkalmas hálózatokon adatbázis-kezelő szerver és kliensek közötti kommunikációra.
- Nem-procedurális programozási nyelv (legalábbis a lekérdezéseknél).

5.4.3. A példákban szereplő táblák

A leírás az SQL nyelv Oracle „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.

[14] A [4] irodalom alapján.

[15] Edgar F. Codd: A Relational Model of Data for Large Shared Data Banks, Communications of the ACM, vol 13. No 7. (1970. június)

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

5.1. táblázat. DEPT tábla a cég részlegeinek adataival

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

5.2. táblázat. EMP tábla az alkalmazottak adatainak tárolására

5.4.4. A nyelv definíciója

A nyelvet a következő rész(nyelv)ekre oszthatjuk:

- *adatdefiníciós nyelv* (data definition language, DDL)
- *adatlekérdező és -manipulációs nyelv* (data manipulation language, DML)
- egyes források a lekérdező nyelvet a DML-től különválasztják, ekkor külön részнyelvként megjelenik az ún. *adatlekérdező nyelv* (data query language, DQL)
- *adatelérést vezérlő nyelv* (data control language, DCL)

A nyelvben – a szöveg literálok kivételével – a kis- és nagybetűket nem különböztetjük meg. A megadott példáknál a könnyebb érthetőség miatt a nyelv alapszavait csupa nagybetűvel, míg a programozó saját neveit kisbetűvel írjuk.

A parancsok több sorba is átnyúlhatnak, a sorokra tördelésnek, és a tabulálásnak nincs szemantikai jelentősége. Az SQL parancsokat pontosvessző zárja le.

5.4.4.1. Táblák, nézetek, indexek létrehozása, törlése

Tábla létrehozása Új táblát a

```
CREATE TABLE <táblanév>
       (<oszlopnév> <típus> [NOT NULL]
       [, <oszlopnév> <típus> [NOT NULL], ...]
);
```

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

- `CHAR(n)`, `VARCHAR2(n)` – max n karakter hosszú szöveg
- `LONG(n)` – mint a `CHAR`, de hosszára általában nincs (vagy nagyon nagy a) felső korlát
- `NUMBER(w)` – az előjel nélkül max. w karakter széles egész szám; az Oracle `INTEGER` típusa megegyezik a `NUMBER(38)`-cal
- `NUMBER(w,d)` – w a szám tízes számrendszerbeli helyi értékeinek a számát, míg d a pontosságot, azaz a tizedes vessző után álló helyi értékek számát jelenti
- `DATE` – dátum (és általában időpont).

Ha valamely oszlop definíciója tartalmazza a `NOT NULL` módosítót, a megfelelő mezőben mindig valamilyen megengedett, nem üres értéknek kell szerepelnie.

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)
);
```

5.4.4.1.1. 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-más nézeteken keresztül szemlélhetik az adatokat. A nézet általában csak olvasható, az adatmódosító műveletekben csak akkor szerepelhet, ha egyetlen táblából keletkezett és nem tartalmaz számított értékeket.

5.4.4.1.2. Index létrehozása Az indexek a táblákban történő keresést gyorsíthatják meg. Létrehozásuk:

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

Az indexeket az adatbázis-kezelő a táblák minden módosításánál frissíti. 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 éppen bizonyos lekérdezéseket gyorsítanak. Indexeket azokra az oszlopokra érdemes definiálni, amelyek gyakran szerepelnek keresésekben. Index nélkül minden kéréshez be kell olvasni az egész táblát. Szerencsés esetben viszont az adattáblában egyáltalán nem kell keresni, a kívánt rekord csupán az index alapján közvetlenül is kiválasztható. Ha a keresési feltételhez van megfelelő index, akkor az adatbázis-kezelő a diszkről csak a valóban szükséges adatokat olvassa be.

Pl. a

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

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

5.4.4.1.3. Törlések A fenti adatbázis objektumokat a `DROP` paranccsal lehet törölni.

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

5.4.4.2. 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 növelhetjük.

5.4.4.3. Adatok bevitele, törlése, módosítása

5.4.4.3.1. Adatok bevitele A `CREATE TABLE` utasítással létrehozott táblák kezdetben üresek.

Új sorokat egy meglévő táblába az

```
INSERT INTO <táblanév> [(<oszlopnév> [, <oszlopnév>, ...])]
VALUES (<kif1> [, <kif2>, ...]);
```

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 nem definiált, így a beillesztés sem feltétlenül a tábla „végére” történik.)

Amennyiben nem adjuk meg az oszlopok nevét, akkor a – tábla deklarálásánál megadott sorrendben – minden mezőnek értéket kell adni, ha viszont megadtuk az egyes oszlopok neveit, akkor csak azoknak adunk értéket, mégpedig a felsorolásuk sorrendjében, a többi mező `NULL` értékű lesz.

Az egyes mezőknek `NULL` értéket is adhatunk, ha a deklaráció alapján az adott mezőnek lehet `NULL` értéke.

Figyelem: Amennyiben olyan oszlopnak akarunk `NULL` értéket adni, amelynek nem lehet `NULL` értéke, úgy a parancsvégrehajtás hibával leáll!

Egy ilyen paranccsal egyszerre csak egy sort tudunk felvenni a táblába.

Két új termék bevitele pl. az alábbi SQL utasítással lehetséges:

```
INSERT INTO product (prodid, descrip)
VALUES (111111, 'Gozeke');

INSERT INTO product
VALUES (111112, 'Oracle 6.0', NULL, 'Relacios adatbazis-kezelo');
```

5.4.4.3.2. Adatok törlése Sorokat kitörölni a

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

paranccsal lehet.

Ha a `WHERE` hiányzik, akkor a tábla valamennyi sorát, egyébként csak a logikai kifejezés által kiválasztott sorokat törli.

Ha az előzőekben felvett adatok közül a 111112 azonosítójút (prodid) törölni akarjuk, akkor ezt a következő módon lehetséges:

```
DELETE
  FROM product
 WHERE prodid = 111112;
```

5.4.4.3.3. Adatok módosítása Sorokban az egyes mezők értékeit az

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

paranccsal lehet módosítani.

Ha a `WHERE` hiányzik, akkor a parancs a tábla valamennyi sorában módosít, egyébként csak a logikai kifejezés által kiválasztott sorokban.

Ha a PRODUCT táblában a „Gozeke”-hez megjegyzést akarunk fűzni, akkor ezt a következőképpen lehet megoldani:

```
UPDATE product
   SET comments = 'mezogazdasagi gep'
 WHERE descrip = 'Gozeke';
```

5.4.4.4. 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 egy oszlopa és csak (nulla vagy) egy sora lesz. Az eredmény tábla a lekérdezés után megjelenik vagy a tábla felhasználható egy másik parancsba beágyazva (pl. halmazműveletek).

- A `<jellemzők>` definiálják az eredmény tábla oszlopait,
- a `<táblák>` adják meg a lekérdezésben résztvevő táblák nevét,
- a `<logikai kifejezés>` segítségével szűrhetünk („válogathatunk") a bemenet sorai között,
- a `<csoportosítás>` a szűrt bemenet sorait rendezi csoporokba úgy, hogy az eredménytáblában csoportonként pontosan egy rekord szerepel. A képzett csoportok között opcionálisan egy logikai kifejezéssel szűrhetünk („válogathatunk") is,
- a `<rendezés>` az eredménysorok sorrendjét határozza meg.

Nézzük meg, hogy a lekérdezés műveletével hogyan lehet megvalósítani a relációs algebra alapműveleteit.

5.4.4.4.1. Vetítés (projekció, projection) A vetítés művelete egy táblából adott oszlopokat válogat ki. A `<jellemzők>` között kell felsorolni azokat az oszlopokat, amelyekre a `<táblanév>` által meghatározott táblát vetíteni akarjuk.

```
SELECT <jellemzők>
  FROM <táblanév>;
```

Például az alkalmazottak neve és fizetése:

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

Minden oszlop kiválasztása:

```
SELECT *
  FROM emp;
```

Ha a <jellemzők>-ben csak a *-ot adjuk meg, akkor az adott tábla minden oszlopát kiválasztja. (Az egész táblát megjeleníti.)

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, oszlopfüggvényeket (lásd később).

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 (valójában ez felel meg a relációs algebra *vetítés* (projection) műveletének):

Az összes különböző munka megnevezése:

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

5.4.4.4.2. Szelekció (selection) A kiválasztás műveleténél a tábla sorai közül válogatunk. A <logikai kifejezés> igaz értékeinek megfelelő sorok kerülnek be az eredmény táblába.

```
SELECT <jellemzők>
  FROM <táblanév>
 WHERE <logikai kifejezés>;
```

A 2000 dollárnál magasabb 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 (a dátumliterál formája: `DATE'éééé-hh-nn'`, ahol `éééé` az évszám 4 számjegyen, `hh` és `nn` pedig rendre a hónap és a nap két számjegyen, ha kell, vezető nullával)
- 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 (egymásba ágyazott lekérdezések, 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 jelzése
  - `_` 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 DATE'1982-01-01' AND DATE'1989-12-31';
```

- A 2000 dollárnál kevesebbet kereső és prémiumot nem kapó dolgozók:

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

5.4.4.4.3. Illesztés (join) A természetes illesztés műveleténél két vagy több tábla soraiból hozunk össze egy-egy új rekordot akkor, ha a két sor azonos nevű mezőinek értéke megegyezik. A `SELECT` kifejezésben a `<táblák>`-ban kell megadni az érintett táblák neveit vesszővel elválasztva, a `WHERE` mögötti logikai kifejezés definiálja azokat az oszlopokat és feltételeket, amely értékei szerint történik meg az illesztés. Vegyük észre, hogy ez a megoldás valójában a – természetes illesztésnél általánosabb – Θ-illesztést (5.2.7. alszakasz) valósítja meg.

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 illesztést 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 eredményt, láthatjuk, hogy a 40-es osztály nem szerepel a listában. Ennek az az oka, hogy az osztálynak nincs egyetlen dolgozója sem, tehát az illesztésnél 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 illesszünk, azaz az összekapcsolás során az `emp` táblához hozzáillesztünk egy üres sort is. Ez az illesztési típus a *külső illesztés* (outer join). 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 amelyikben ha nincs a másik táblához illeszkedő sor, akkor egy üres mezőket tartalmazó sort kell hozzávenni. Az illesztésnél 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 illesztés mellett egyidejűleg más logikai kifejezéseket is használhatunk. A fizetések fenti vizsgálatát felfoghatjuk úgy is, mint a természetes illesztés műveletének általánosított esetét (ld. Θ-illesztés, 5.2.7. alszakasz), ahol nem csak az egyenlőség művelete használható, valamint úgy is, hogy a már egyesített táblából zárjuk ki a fizetésekre szabott követelményeknek meg nem felelő sorokat.

Az SQL programozó számára átlátszó, hogyan (milyen algoritmusokkal) hajtódik végre a megírt lekérdezés.[16] Itt fedezhető fel a nyelv eredmény-orientált (deklaratív, a „mit számítsunk ki" kérdésre fókuszáló) jellege, szemben az algoritmus-orientált (imperatív) nyelvekkel, amelyek a „hogyan számítsuk ki" kérdésre fókuszálnak.

5.4.4.4.4. Oszlopfüggvények (aggregáció, aggregation) A lekérdezés eredményeként előálló táblák egyes oszlopaiban lévő értékeken végrehajthatunk – a procedurális nyelveken tipikusan – ciklussal/ciklusokkal 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;
```

[16] Ez gyakran előny – hiszen a fejlesztést hatékonyabbá tudja tenni, a programkód hamarabb elkészül. Azonban komplex, hosszan futó lekérdezéseknél – ahol egyáltalán nem biztos, hogy a lekérdezés optimalizáló (ld. 6. fejezet) valóban a legjobb végrehajtási tervet állítja elő – fontos lehet, hogy a fejlesztő is tisztában legyen azzal, hogy hogyan hajtódik végre a lekérdezése.Ekkor a programozónak – az SQL implementációtól függő módon, tipikusan – lehetősége van a végrehajtási tervbe beavatkozni, amit „hintelésnek" nevezünk.

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, kivétel ezalól a COUNT(*), ami minden rekordot számol.)

Mivel az oszlopfüggvény eredménye egyetlen értéket állít elő, az eredménytábla oszlopainak definiálásakor az oszlopfüggvény mellé vagy más oszlopfüggvényeket írhatunk, vagy olyan értéket írhatunk, amelyik az összes kiválasztott sorban azonos. Ha nem oszlopfüggvény eredményéből (és nem konstansból) származó érték is része az eredménytáblának, akkor csoportosítani kell ezen értéklista szerint. A csoportosítás szükséges akkor is, ha a programozó biztos benne, hogy az összes kiválasztott sorban azonosak az értékek. Például írhatnánk:

```
SELECT COUNT(*), AVG(sal)
  FROM emp;
```

De hibás az alábbi:

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

5.4.4.5. Egymásba ágyazott lekérdezések

A WHERE mögött álló logikai kifejezésben, illetve a FROM mögött álló táblamegadásban állhat egy teljes SELECT utasítás is zárójelek között. Például:

A nem New Yorkban 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 Yorkban vannak, majd azt vizsgáljuk, hogy az ezekből képzett halmazban található-e az adott dolgozó osztályának azonosítója. Mellesleg ugyanezt a listát megkaphatnánk az illesztés 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 minden elemére 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);
```

5.4.4.5.1. Csoportosítás (grouping) 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, konstansok és az ezekből képzett kifejezések szerepelhetnek.

A csoportosítás után az eredményből bizonyos csoportok kihagyhatók a `HAVING`

<logikai kifejezés> utasításrész használatával. A <logikai kifejezés> igaz értékeinek megfelelő csoportokhoz tartozó 1-1 rekord kerül be az eredmény táblába.

Foglalkozásonkénti átlagfizetés az 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 – szerepelhetnek. Természetesen a csoportosítás előtt a WHERE feltételek használhatók. Célszerű – gyorsabb – WHERE feltételeket használni mindenhol, ahol csak lehet, és a HAVING szerkezetet csak akkor alkalmazni, amikor a teljes csoporttól függő értékeket akarjuk vizsgálni.

5.4.4.5.2. Rendezés (sorting) Az eddig tárgyalt lekérdezések eredményében a sorok „véletlenszerű” – a programozó által nem megadható – sorrendben szerepeltek. A sorrendet az ORDER BY által megadott rendezéssel lehet szabályozni. A rendezés több oszlop 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 alapesetben emelkedő (ASC), de a DESC módosítóval csökkenő rendezés írható elő.

A 30-as osztály dolgozói munkakör neve szerint emelkedő, azon belül fizetés szerint csökkenő sorrendben:

```
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);
```

5.4.4.5.3. Egyéb, nem részletezett műveletek Az egyes lekérdezések által előállított táblák halmazként is felfoghatók, az SQL nyelv ezen táblák kombinálására tartalmazza a szokásos halmazműveleteket is.

`UNION`	egyesítés
`INTERSECT`	metszetképzés
`MINUS`	különbségképzés

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” a KING nevű elnöktől kezdődően.

A `CONNECT BY` klózban megadott feltételt kielégítő rekordpárok a hierarchiában szülő-gyerek viszonyban vannak. A `PRIOR` kulcsszóval jelölt kifejezés a szülő rekordon értelmezett.

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.

5.4.4.6. Adatelérések szabályozása

Az adatbázis-kezelő rendszereket tipikusan több felhasználó használja, ezzel kapcsolatban újabb feladatok merülnek fel.

5.4.4.6.1. Jogosultságok definiálása Az egyes felhasználók részint az adatbázis-kezelő rendszerrel, részint az egyes objektumaival különböző műveleteket végezhetnek. Ezeknek a megadására szolgálnak a `GRANT`, megvonására pedig a `REVOKE` utasítások, amelyek az SQL adatelérést vezérlő nyelv részei. A

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

REVOKE [DBA | CONNECT | RESOURCES]
  FROM <felhasználó>, ...;
```

parancsok az egyes felhasználóknak az adatbázishoz való hozzáférési jogát szabályozzák. 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 adatbázis-kezelőbe történő belépésre jogosít.

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

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

REVOKE <jogosultság>, ...
    ON <tábla vagy nézetnév>
  FROM <felhasználó>;
```

parancsok határozzák meg. A `<jogosultág>` az objektumon végezhető műveleteket adja meg, lehetséges értékei:

```
ALL
SELECT
INSERT
UPDATE
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.

5.4.4.6.2. 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 adatbázis-kezelő leáll. Ilyenkor a tárolt adatok inkonzisztens[17] állapotba kerülhetnének, hiszen egyes módosításokat már elvégeztünk, ehhez szorosan hozzátartozó másokat viszont még nem.

[17] Ld. ACID tulajdonságok, a 10.1. szakasz.

A *tranzakció* (transaction) 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 [ON | IMM | OFF];
```

azonban az `OFF` állapotban is commit-olnak az `ALTER`, `CREATE`, `DROP`, `GRANT` és `EXIT` utasítások[18]. Az `ON` és az `IMM`, mint *immediate* ugyanazt a videlkedést váltja ki.

A rendszer hardver hiba utáni újraindulásakor automatikusan `ROLLBACK`-et hajt végre.

5.4.4.6.3. Párhuzamos hozzáférések szabályozása Az adatbázist több felhasználó akár egyidejűleg is használhatja. Ennek speciális problémáiról ld. a 10. fejezetet. 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.

[18] Az Oracle DBMS motorjában igazából nincs is autocommit mód, a set autocommit on (vagy imm) a kliens viselkedését vezérli, ha az érti, és az fogja kiadni a commit-ot. Pl. az SQL*Plus nevű kliens érti.

5.4.5. Bővítések

5.4.5.1. Konzisztencia feltételek, kényszerek

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 azután 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, vagy adott maszkra illeszkedés);
- 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 vagy `UNIQUE` index alapjául szolgáló oszlopának valamelyik létező elemével;

Amennyiben a táblák módosításánál valamelyik feltételt megsértenénk, a rendszer egy kivételt (exception) generál, és lefuttatja a kivételhez tartozó kiszolgáló kódot, ha van ilyen.

5.5. A fejezet új fogalmai

halmaz, domén, reláció, attribútum, n-es (tuple, rekord), reláció fokszáma, reláció elemszáma, attribútum kardinalitása, relációs séma, relációs algebra, relációs algebrai alapműveletek (unió, különbség, keresztszorzat, vetítés, kiválasztás), származtatott műveletek (metszet, természetes illesztés, theta illesztés, külső (jobb- és baloldali) illesztés), relációs algebra zártsága, relációs lekérdező nyelv teljessége, sorkalkulus, oszlopkalkulus, formális nyelv, megengedett szimbólum, atom v. atomi formula, formula, kötött/szabad változó, sor/oszlopkalkulus kifejezés, formális nyelv interpretációja (értelmezése), interpretációs halmaz, igazságértékek hozzárendelése, formula doménje, biztonságos formula, lekérdező nyelv kifejezőereje

6. fejezet

Relációs lekérdezések optimalizálása

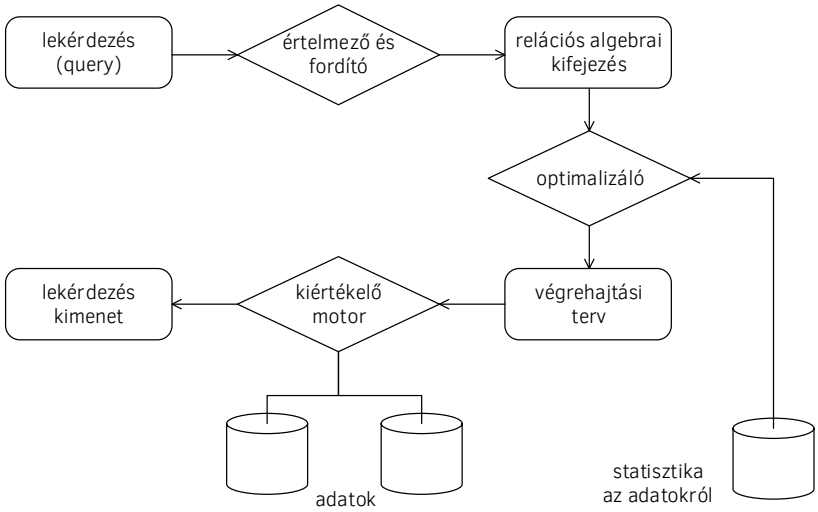
A fejezet anyaga a „relációs lekérdezések feldolgozása és optimalizálása” gazdag témakörének csak egy viszonylag kis, de alapvető részét fedi le. Nem foglalkozunk a lekérdezés fordításával és szintaktikai ellenőrzésével, a dinamikus programozás alkalmazásával, adaptív technikákkal és még számos más kapcsolódó problémával sem.

6.1. Áttekintés

A lekérdezés feldolgozás elsődleges célja az adatok adatbázisból való kinyerése. Az egyértelműen, de deklaratívan megfogalmazott célhoz vezető megoldás megtalálása azonban korántsem egyszerű. Számos bonyolult részfeladatot kell az adatbázis-kezelő rendszernek megoldania, amíg eljut a kívánt végeredményhez. Ezek a következők, melyeket az a 6.1. ábrán is szemléltetünk:

1. Elemzés (szintaktikus), fordítás
2. Költségoptimalizálás
3. Kiértékelés

Az első lépés a szintaktikai elemzés és a fordítás. Egy magas szintű nyelven (tipikusan az SQL valamilyen dialektusában) megfogalmazott kérést a számítógép számára használhatóbb formára kell hozni. Az SQL-t, mint kommunikációs interfészt az emberi igényekhez tervezték: minden SQL mondat megfeleltethető egy természetes nyelv (speciálisan az angol) egy mondatának. Sajnálatos módon azonban a számítógép ezt az idegen nyelvet csak tolmács segítségével beszéli. A legtöbb adatbázis-kezelő rendszer anyanyelve a relációs algebra valamilyen kiterjesztett változatára épül.



6.1. ábra. A lekérdezés feldolgozás tipikus lépései

Persze a fordítás csak akkor valósulhat meg, ha az SQL nyelvet a felhasználó helyesen beszéli, ezért a fordítást minden esetben megelőzi a szintaktikai elemzés. Az elemzés során megvizsgáljuk a lekérdezés „helyességét”: meggyőződünk arról, hogy a lekérdezésben szereplő relációnevek ténylegesen előfordulnak-e az adatbázisban, azok elérésére van-e jogosultsága a felhasználónak, stb. A lekérdezés elemeit ezután le kell fordítani, és valamilyen belső – általában relációs algebra alapú – reprezentációba kell átalakítani.

Miután sikerült egy matematikailag helyes kifejezést konstruálnunk az SQL mondatból, érdemes pár dolgot végiggondolni. Vajon egyértelmű-e egy lekérdezés, ill. a hozzárendelt belső reprezentáció a kiértékelés módját és a lépések sorrendjét tekintve? Lehetséges-e formális módszerekkel a lekérdezésünkkel ekvivalens másik lekérdezés(eke)t konstruálnunk? Mert ha lehetséges, akkor nyilvánvalóan több végrehajtási út létezik, amelyek sebességben, végrehajtási időben, lemezhasználatban stb.-ben esetleg eltérnek egymástól. Ebben az esetben érdemes lenne különböző végrehajtási terveket is kidolgoznunk.

Ha már kidolgoztunk több tervet, akkor ezeket valamilyen szempont szerint össze kellene hasonlítani. Nyilvánvalóan az optimális megoldást keressük a problémára, de mi az a paraméter, amely alapján az optimalitást értelmezzük? Esetleg nem egy, hanem több paramétert is számításba kell venni? Mivel bizonyosan lesznek eleminek tekintett műveleteink, vajon milyen algoritmusok léteznek a végrehajtásukra, és melyiket válasszuk? Összefoglalva: optimalizációs stratégiák alapján végrehajtási terveket kell készíteni, amelyeket előbb értékelni kell, majd közülük a legjobbat kiválasztva azt végrehajtani.

A hálós (ld. 7. fejezet) és a hierarchikus modellben, ill. a NoSQL rendszereknél a lekérdezések optimalizálása legtöbbször a programozó feladata. Ennek oka, hogy az adatmanipulációs nyelven (DML) megfogalmazott kifejezések általában

a programokba beágyazva szerepelnek, és ezeknek optimális hálós illetve hierarchikus lekérdezéssé transzformálása az egész program ismerete nélkül általában nem megoldható. Az optimalizációhoz tehát bonyolult algoritmus futtatására lenne szükség, amely túl nagy terhelést jelentene – különösen az akkori – rendszerek számára.

A relációs rendszerekben az optimalizálás megvalósítható a programozó közbeavatkozása nélkül is. Mivel a dominánsan deklaratív szemléletű mondatok elsősorban nem azt mondják meg, hogy hogyan hajtsunk végre valamit, hanem leginkább azt, hogy mit szeretnének eredményül kapni, ezért a belső megvalósítás rejtve maradhat. A deklaratív elvek realizálása algebrai eszközökkel megoldható. Köszönhetően a formális matematikai módszereknek viszonylag könnyű lesz egyazon kérdéshez több, eredményét tekintve ekvivalens alakot találni, és kiválasztani közülük a legkevésbé költségeset.

A továbbiakban megnézzük, hogy az algebrai formák hogyan támogatják a lekérdezések optimalizálását. Formális lépéseken keresztül megkeressük egy adott SQL mondat több különböző ekvivalens alakját. Az eltérő alakok eltérő kiértékelési sorrendet jelenthetnek. Megpróbáljuk megbecsülni, melyik végrehajtása mekkora terhelést jelentene a rendszer számára. Egy egyszerű példával illusztrálva a lépéseket nézzük az alábbi SQL mondatot:

```
SELECT DISTINCT egyenleg
  FROM számla
 WHERE egyenleg < 2500;
```

Egy bank nyilvántartásából szeretnénk megtudni, milyen (különböző) 2500 egységnél kisebb értékű egyenlegek léteznek. A lekérdezést átalakíthatjuk például a következő két, egymással ekvivalens relációs algebrai alakba:

$$\pi_{\text{egyenleg}}(\sigma_{\text{egyenleg}<2500}(\text{számla}))$$

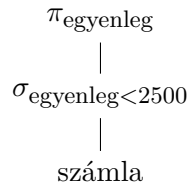
$$\sigma_{\text{egyenleg}<2500}(\pi_{\text{egyenleg}}(\text{számla}))$$

Láthatóan a két alak két különböző végrehajtási sorrendet kínál. Az első esetben a *számla* relációból kiválasztjuk azokat az elemeket, amelyben az egyenleg értéke kisebb 2500-nál, majd ezután végrehajtunk egy projekciót az így kapott relációra. A második forma pont a fordított sorrendet írja le, először egy projekció, majd egy szelekció műveletet kell elvégezni. Ezután el kell döntenünk, melyik végrehajtási sorrendet kövessük.

Az optimum mértéke lehet a lekérdezés teljes (fiktív) „költsége”, amelynek a kiszámításához szükséges az egyes, eleminek tekintett műveletek költségeinek ismerete. Sajnos a helyzet feloldása nem ennyire egyszerű, mert egy elemi művelet költsége eltérő környezetekben más és más lehet. Tekintsük például a szelekció műveletet, amelynek végrehajtási ideje, ha lineáris keresést alkalmazunk, arányos a reláció összes elemének számával, azonban ha valamilyen indexet használha-

tunk a szelekció feltételére, akkor a kapcsolódó költséget akár egy nagyságrenddel is csökkenthetjük.

Érezhetően nem mindegy, hogy milyen algoritmusokat tudunk alkalmazni az elemi operációk vagy *kiértékelési primitívek* végrehajtásánál. A primitívek összefoghatóak egy nagyobb munkafolyamati egységbe, ahol egy primitív feldolgozza a bemenetét, majd a kimenetét átadja a sorban utána álló műveletnek. Ha a primitív műveletek a relációs algebra műveletei, akkor a primitív műveletek szekvenciája a *relációs algebrai fa* (relational algebra tree). A 6.2 ábra a példánkra mutat be egy lehetséges relációs algebrai fát.



6.2. ábra. Egy lehetséges relációs algebrai fa

A különböző relációs algebrai fák megalkotása még önmagában nem elegendő, hiszen meg kell mondani, hogy az egyes elemi műveletek végrehajtásához pontosan milyen algoritmusokat alkalmazunk, milyen fizikai segédstuktúrákat használunk, és hogy az egymásra épülő műveletek párhuzamossága hogyan alakul (6.4. szakasz). A relációs algebrai fa az előbbi információkkal kiegészítve a *végrehajtási terv* (execution plan). A lehetséges tervek közül az optimális megtalálása nem egyszerű feladat.

6.2. Katalógus költségbecslés

A lekérdezés feldolgozáshoz a megfelelő végrehajtási terv kiválasztása valamilyen becsült költségmérték hozzárendelése alapján végezhető. Nem lehet eléggé hangsúlyozni, hogy becslésről van szó, nem szabad és nem is érdemes egzakt számokat várni. A becslés elvégzéséhez az adatbázis-kezelő rendszernek a relációkról különböző statisztikákat, mérőszámokat kell karbantartania, amelyek alapján kritikus, hogy a költségbecslés hatékonyan legyen elvégezhető.

A statisztikákat elvileg minden adatbázis módosító művelet után módosítani kellene, de ez óriási terhelést jelentene a rendszer számára. A valós alkalmazásokban a statisztika aktualizálása ezért csak akkor történik meg, amikor a rendszernek „van rá ideje”. Ebből következően a statisztika nem mindig konzisztens a rendszer állapotával, de általában elég jól leírja a rendszerben zajló folyamatokat. Így akár egy régebbi és/vagy hiányos statisztika is jó kiindulási alapja lehet egy elfogadható becslésnek.

A statisztikákat az adatbázis-kezelők egyéb rendszerleíró paraméterek mellett egy ún. katalógusban tárolják. A következőkben bemutatjuk, hogy a relációkról és az indexekről tipikusan milyen információk találhatók meg a katalógusban.

6.2.1. Katalógusban tárolt egyes relációkra vonatkozó információk

n_r**:** az r relációban levő rekordok (elemek) száma (number)

b_r**:** az r relációban levő rekordokat tartalmazó blokkok (blocks) száma

s_r**:** az r reláció egy rekordjának nagysága (size) bájtokban

f_r**:** mennyi rekord fér az r reláció egy blokkjába (blocking factor)

$\mathrm{V}(A, r)$**:** hány különböző értéke (Values) fordul elő az A attribútumnak az r relációban. $\mathrm{V}(A, r) = |\pi_A(r)|$. Speciálisan, ha az A kulcs, akkor $\mathrm{V}(A, r) = n_r$.

$\mathrm{SC}(A, r)$**:** azon rekordok várható száma, amelyek kielégítenek egy egyenlőségi feltételt az A attribútumra (Selection Cardinality), feltéve, hogy legalább egy rekord kielégíti ezt az egyenlőségi feltételt.

Például, ha az A egyediséget biztosít, akkor $\mathrm{SC}(A, r) = 1$. Ha az A nem biztosít egyediséget és eloszlása nem ismert, akkor a becsléshez feltesszük, hogy a $\mathrm{V}(A, r)$ különböző érték egyenletesen oszlik el a rekordok között. Ekkor $\mathrm{SC}(A, r) = \frac{n_r}{\mathrm{V}(A,r)}$.

Az utóbbi két mennyiség definiálható tetszőleges A attribútumhalmazra is: $\mathrm{V}(A, r)$ illetve $\mathrm{SC}(A, r)$. Megfigyelés: Ha a relációk rekordjai fizikailag együtt vannak tárolva, akkor (optimális blokk-kihasználtság mellett): $b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$. A továbbiakban, ha külön nem hangsúlyozzuk, mindig optimális blokk-kihasználtságot tételezünk fel.

6.2.2. Katalógus információk az indexekről

Az indexek – leggyakrabban B* fák – olyan segédstruktúrák, amelyek gyorsíthatják a relációkban adott attribútum vagy attribútumhalmaz szerinti keresést. Az indexek létrehozása és karbantartása a rendszer számára nagyobb adminisztrációs költséggel jár, ami csak a használatuk során térülhet meg. Az egységes tárgyalás érdekében a hash-állományokat a továbbiakban speciális „indexnek” fogjuk tekinteni.

Az indexeket az alábbi paraméterekkel jellemezzük:

f_i**:** az átlagos pointer-szám a fa struktúrájú indexek csomópontjaiban, mint pl. a B* fáknál, azaz a csomópontokból induló ágak átlagos száma.

HT_i**:** az i index szintjeinek a száma, azaz az index magassága (Height of Tree). Az r relációt tartalmazó heap-szervezésű állományra épített B* fa esetén $\mathrm{HT}_i = \left\lceil \log_{f_i} b_r \right\rceil$, ill. hash-állománynál $\mathrm{HT}_i = 1$.

LB_i**:** az i index legalsó szintű blokkjainak a száma, azaz a levélszintű indexblokkok száma (Lowest level index Block).

6.2.3. A lekérdezés költsége

A bevezetőből kiderült, hogy a különböző végrehajtási tervek költsége más és más lehet. Definiáljuk most pontosabban, hogy költség és optimalitás alatt mit kell ér-

teni. A lekérdezés kiértékelés költségének meghatározása történhet az igényelt és a felhasznált erőforrások alapján, ez lehet például a felhasznált processzoridő, a háttértárhoz fordulás ideje vagy elosztott rendszerekben a kommunikációra fordított idő. Egy másik kézenfekvő lehetőség a válaszidő alapján történő költségbecslés. Azonban, ha jobban végiggondoljuk, ez nem is olyan jó alternatíva, hiszen a válaszidő erősen függ a környezet állapotától is (egy túlterhelt rendszer valószínűleg lassabban generálja le a végeredményt, mint egy kevésbé leterhelt). A válaszidő érezhetően nem biztosít elemi költségmeghatározási szempontot.

A nagy adatbázis-kezelőkben a költség becslésére alapesetben a *háttértár blokkműveletek számát* használják, mivel ez lényegében független a rendszer terhelésétől és mert ennek időigénye nagyságrenddel nagyobb, mint a processzor- és memóriaműveletek időigénye. A használható költségmérték megalkotásához azonban szükséges a probléma megfelelő szintű egyszerűsítése. Nem szabad különbséget tennünk az egyes blokkok elérési ideje között, azaz alapfeltételezés, hogy a diszken elhelyezkedő minden blokkhoz azonos idő alatt férünk hozzá. Nem vesszük figyelembe a lemez forgási irányát, a fej mozgását sem. Nem tudunk különbséget tenni továbbá az egyes írások és olvasások között sem. Ez alapján legyen a *költség* a diszk blokkok olvasásának és írásának a száma azzal a további megszorítással, hogy *az írásba csak a köztes blokkírások számát számítjuk bele*, hiszen a végeredmény kiírása mindenképpen szükséges.

Jelölés: $E_{\text{alg.}}$ = az algoritmus becsült költsége (estimate).

6.3. Műveletek költsége

A relációs adatbázis-kezelők világában szükséges a relációkon végezhető alapműveletek definiálása, ezek lesznek azok az elemi építőkövek, amelyek segítségével minden más lekérdezés megszerkeszthető. Jelen anyagban csak az alapesetekkel foglalkozunk.

6.3.1. Szelekció

Egy lekérdezés végrehajtása (pontosabban: annak kiértékelése) során egy reláció végigolvasása a legalacsonyabb szintű művelet. Egy adott érték megtalálását az adott szelekciós feltételek figyelembevétele mellett valamilyen keresési algoritmus alapján kell elvégezni. A relációk egyszerű esetben egyetlen állományban tárolódnak, ezért a keresési művelet csupán egy fájlra korlátozódik.

6.3.1.1. Alap szelekciós algoritmusok

A kiválasztás művelet megvalósítását lehetővé tevő két alapalgoritmus a következő:

A1: Lineáris keresés („full table scan”): Minden rekordot beolvasunk, és megvizsgáljuk, hogy kielégíti-e a szelekció feltételét. $E_{A1} = b_r$

A2: Bináris keresés: Csak akkor tudjuk végrehajtani, ha a blokkok folyamatosan helyezkednek el a diszken, a fájl az A attribútum szerint rendezett és a szelekció feltétele az egyenlőség az A attribútumon. Átlagosan

$\mathrm{SC}(A,r)$ darab ilyen rekordunk van, ezért az algoritmus becsült költsége: $E_{A2} = \lceil \log_2 b_r \rceil + \left\lceil \frac{\mathrm{SC}(A,r)}{f_r} \right\rceil - 1$.

(Az első ilyen blokk megtalálása a relációban lévő rekordokat tartalmazó blokkok logaritmusával arányos, az összeg második tagja pedig a szelekció feltételét kielégítő összes rekord tárolásához szükséges blokkok átlagos száma. A -1 azért szükséges, mert az összeg előbbi két tagja egyaránt tartalmazza az első blokk olvasásának költségét.)

Ha az A attribútum egyediséget biztosít, akkor a keresés költsége: $\lceil \log_2 b_r \rceil$.

6.3.1.2. Indexelt szelekciós algoritmusok

A szakirodalom megkülönbözteti az elsődleges és másodlagos indexeket. Az elsődleges index a rekordok olyan sorrendben való olvasását teszi lehetővé, amely megfelel a rekordok fizikai tárolási sorrendjének. Minden egyéb indexet másodlagos indexnek tekintünk. A legfontosabb indexelt szelekciós algoritmusok ezek alapján a következők[1]:

A3: Elsődleges index használatával, egyenlőségi feltételt a kulcson vizsgálunk: Az algoritmus költsége $E_{A3} = \mathrm{HT}_i + 1$, az index szintek plusz az adatblokk olvasása.

A4: Elsődleges index használatával egyenlőségi feltétel nem a kulcson: $E_{A4} = \mathrm{HT}_i + \left\lceil \frac{\mathrm{SC}(A,r)}{f_r} \right\rceil$. Az egyenlőségi feltételt $\mathrm{SC}(A,r)$ rekord elégíti ki, amely végigolvasásához $\left\lceil \frac{\mathrm{SC}(A,r)}{f_r} \right\rceil$ blokkművelet szükséges.

A5: Másodlagos index használatával, egyenlőségi feltétel alapján: $E_{A5} = \mathrm{HT}_i + \mathrm{SC}(A,r)$ (a második tag mutatja, hogy mennyi különböző blokkon lehetnek). Ha az A egyediséget biztosít, akkor $E_{A5} = \mathrm{HT}_i + 1$.

6.3.1.3. Összehasonlítás alapú szelekció

Tekintsük a $\sigma_{A \leq v}(r)$ alakú lekérdezéseket. Ha v értékét nem ismerjük, azt mondhatjuk, hogy átlagosan $\frac{n_r}{2}$ rekord elégíti ki a feltételt. Ha ismerjük v értékét, és egyenletes eloszlást feltételezünk az A attribútum maximális ($\max(A,r)$) és minimális ($\min(A,r)$) értéke között, akkor átlagosan $n_r \cdot (\frac{v - \min(A,r)}{\max(A,r) - \min(A,r)})$ rekord elégíti ki a feltételt.

A6: Elsődleges index használatával: $E_{A6} = \mathrm{HT}_i + \frac{b_r}{2}$ (a keresési feltételt átlagosan a rekordok fele kielégíti). Ha a v-t ismerjük, és c jelöli azon rekordok számát, ahol $A \leq v$, akkor: $E_{A6} = \mathrm{HT}_i + \left\lceil \frac{c}{f_r} \right\rceil$

A7: Másodlagos index használatával: $E_{A7} = \mathrm{HT}_i + \frac{\mathrm{LB}_i}{2} + \frac{n_r}{2}$ A második tag azt írja le, hogy a levélszintű indexblokkok átlagosan felét kell bejárni, hogy elérjük a feltételt kielégítő rekordokra mutató index-bejegyzéseket. Az utolsó

[1] A becslések elkészítése során elhanyagoljuk, hogy felhasznált indexrekordok hány indexblokkban vannak.

tagra azért van szükség, mert ha a rekordok átlagosan fele elégíti ki a feltételt, akkor ezeket a másodlagos index jellegéből következően csak egyesével, azaz egy-egy további blokkművelettel tudjuk elérni.

A másodlagos indexek használatával kapcsolatban meglepő következtetésre juthatunk. Az összehasonlítás alapú lekérdezéseknél szerencsétlen esetben kifizetődőbb egy egyszerű lineáris keresést alkalmazni, mert az kevesebb blokkműveletet igényel.

6.3.2. Join operáció

A join (illesztés vagy összekapcsolás) művelet általános értelemben két reláció Descartes-szorzatának adott feltétel (predikátum) szerinti szelekciója (theta-join):

$$r_1 \underset{\theta}{\bowtie} r_2 = \sigma_\theta(r_1 \times r_2)$$

A továbbiakban bemutatjuk a legfontosabb join típusokat, megbecsüljük, hogy két reláció illesztéséhez várhatóan mekkora tárterület szükséges, majd áttekintjük a join megvalósítását lehetővé tevő fontosabb algoritmusokat.

6.3.2.1. A join fontosabb típusai

- *természetes illesztés* (natural join): ld. 5.2.6. alszakasz.
- *Θ-illesztés* (Θ-join, theta join): ld. 5.2.7. alszakasz.
- *külső illesztés* (outer join):
  A természetes illesztés veszélye, hogy általában a kapcsolt táblák nem minden sora szerepel az eredménytáblában. A *külső illesztés* garantálja az összekapcsolt két tábla egyikénél vagy mindkettőnél az összes rekord megőrzését. Egy elterjedt implementáció jelölési konvenciója (+) alapján megkülönböztetjük:
  - *Bal oldali külső illesztés:* $t_1 * (+)t_2$. Azt jelenti, hogy az eredménytáblában t_1 azon sorai is szerepelnek, amelyek t_2 egyetlen sorával sem párosíthatók. Ezen sorokban a t_2-beli attribútumok értéke NULL.
  - *Jobb oldali külső illesztés:* $t_1(+) * t_2$. Hasonlóan a t_2 táblára.
  - *Teljes külső illesztés:* $t_1(+) * (+)t_2$. Itt mindkét tábla nem párosított rekordjai megőrződnek.

6.3.2.2. Egymásba ágyazott ciklikus illesztés (nested loop join)

Az egymásba ágyazott ciklikus illesztés egy általános algoritmus két reláció (r és s) theta-join műveletének implementálására. Az algoritmus logikája könnyen végigkövethető a megadott pszeudokód alapján. (A $*$ művelet a konkatenációt jelöli.)

Láthatóan ez egy elég költséges eljárás, hiszen a legrosszabb esetben $b_r + n_r \cdot b_s$ blokkműveletre van szükségünk a teljes algoritmus lefuttatására (az r reláció végigolvasása b_r blokkművelet, és minden egyes r-beli rekordhoz az összes s-beli blokk végignézése pedig b_s blokkművelet).

```
for minden t_r ∈ r rekordra do
    for minden t_s ∈ s rekordra do
        if a (t_r, t_s) pár kielégíti az illesztés θ feltételét then
            a t_r * t_s rekordot az eredményhez adjuk
        end
    end
end
```

6.3. ábra. A nested loop join-algoritmus pszeudokódja

Ha a két reláció befér a memóriába, akkor $b_r + b_s$ blokkműveletre van szükség a beolvasáshoz. Ha a rendelkezésre álló memória csupán az egyik reláció tárolását teszi lehetővé, akkor is $b_r + b_s$ lesz a költség. Legyen ugyanis az algoritmus szerinti s reláció az, amely elfér a memóriában. Olvassuk be s-et (b_s költség), így minden r-beli rekordhoz az összehasonlítást gyorsan, azaz költség nélkül megtehetjük, ehhez járul még az r-beli rekordok b_r beolvasási költsége.

6.3.2.3. Blokkalapú egymásba ágyazott ciklikus illesztés (block nested loop join)

A blokkalapú egymásba ágyazott ciklikus illesztés algoritmusa „okosabb”, mint az előző algoritmusunk, mert kihasználja a tárolás fizikai sajátosságait. A gyorsítást azáltal éri el, hogy blokkalapú rekord-összehasonlítást végez: a belső két ciklus összehasonlítja a két reláció egy-egy beolvasott blokkjának minden rekordját, a külső kettő pedig végigmegy a két reláció összes blokkján.

Az algoritmus „worst-case” költsége $b_r + b_r \cdot b_s$, kedvezőbb esetben (az első algoritmusnál bemutatott gondolatmenet alapján) $b_r + b_s$.

```
for minden b_r ∈ r blokkra do
    for minden b_s ∈ s blokkra do
        for minden t_r ∈ b_r rekordra do
            for minden t_s ∈ b_s rekordra do
                if a (t_r, t_s) pár kielégíti az illesztés θ feltételét then
                    a t_r * t_s rekordot az eredményhez adjuk
                end
            end
        end
    end
end
```

6.4. ábra. A block nested loop join-algoritmus pszeudokódja

6.3.2.4. Indexalapú egymásba ágyazott ciklikus illesztés (indexed nested loop join)

Az indexelt egymásba ágyazott ciklikus illesztés algoritmus kihasználja, hogy az egyik relációhoz van indexünk. Ha az első esetben bemutatott algoritmus belső ciklusába az indexelt relációt tesszük, akkor nem szükséges minden egyes s-beli rekordot végigvizsgálnunk, hogy megfelel-e a feltételnek, hiszen a keresés index alapján kisebb költséggel is elvégezhető. Az eljárás költsége $b_r + n_r \cdot c$, ahol c a szelekció költsége s-en, amely nyilván a konkrét indexstruktúra és indexelt szelekciós algoritmus függvénye.

6.3.2.5. Összefésülés alapú illesztés (merge join)

Az illesztés úgy is elvégezhető, ha mindkét relációt először rendezzük az illesztési feltételnek megfelelő attribútum szerint. Ezután már elég csak (szinkronban) végigolvasni mindkét relációt, hiszen az illeszkedő elemek a rendezés következtében egymás után kerültek. Az ilyen módon végzett illesztés költsége $b_r + b_s +$ *a rendezések költsége*. Ha a relációk igen nagyok, és hatékonyan tudunk rendezni, akkor gyakran ez a legkisebb költségre vezető algoritmus.

6.3.2.6. Hash-illesztés (hash join)

Az egyik relációt hash-táblán keresztül érjük el, miközben a másik, „külső” reláció egy adott rekordjához illeszkedő rekordokat keressük. Ebben az esetben a join algoritmus belső ciklusát a hash-állomány segítségével történő keresés váltja fel.

Megjegyzés. A join megvalósítására számos egyéb algoritmus is létezik.

6.3.3. Egyéb műveletek

- *Ismétlődés kiszűrése:* (Ha ugyanabból a rekordból több példány van, akkor csak egy maradjon.) Először rendezést hajtunk végre. Az azonos rekordok közvetlenül egymás után fognak megjelenni, ekkor már könnyen törölhetők. Költség: a rendezés költsége.
- *Vetítés:* Minden rekordra végrehajtjuk, aztán kiküszöböljük a másodpéldányokat a fenti módszerrel. Ha a rekordok eleve rendezettek, akkor a költség b_r, általános esetben $b_r +$ a rendezés költsége.
- *Egyesítés:* Először mindkét relációt rendezzük, majd összefésülésnél kiszűrjük a duplikációkat.
- *Metszetképzés:* Mindkét relációt rendezzük, az összefésülésnél csak a közös rekordokat vesszük figyelembe.
- *Különbségképzés:* Mindkét relációt rendezzük, összefésülésnél csak azok a rekordok maradnak, amelyek csak az első relációban szerepelnek.

6.4. Kifejezés kiértékelés

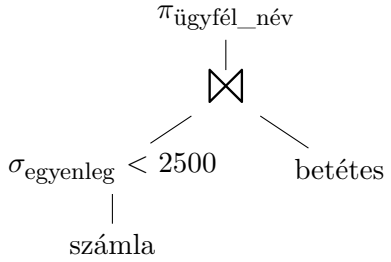
Áttekintettük az elemi műveletek néhány algoritmikus megvalósítását. Nem foglalkoztunk azonban még az összetett, több elemi műveletből álló kifejezések kiértékelésének módjával.

A legkézenfekvőbb stratégia, hogy az összetett kifejezésnek egyszerre egy műveletét értékeljük ki valamilyen rögzített sorrend szerint (materializáció (megtestesítés, létrehozás)). Ezzel azonban van egy nagy probléma, minden művelet végrehajtása után a keletkezett eredményt a későbbi felhasználás miatt a háttértárra kell kiírni, ezért a módszer rengeteg blokkműveletet igényel. Egy másik kiértékelési alternatíva: egy „csővezetékben" egyszerre több elemi művelet szimultán kiértékelése folyik, egy művelet eredményét – a háttértár bevonása nélkül – azonnal megkapja a sorban következő művelet operandusként (pipelining).

6.4.1. Materializáció (megtestesítés, létrehozás, materialization)

A $\pi_{\text{ügyfél\_név}}(\sigma_{\text{egyenleg}<2500}(\text{számla}) \bowtie \text{betétes})$ kifejezést a műveletek mentén a 6.5. ábra szerinti relációs algebrai fába transzformálhatjuk.

A fa leveleiben vannak a relációink, a belső csomópontokban pedig a műveletek. A fa alapján a materializációs stratégia lépései könnyen nyomon követhetőek. Az első lépésként hajtsunk végre egy olyan műveletet, amelyekhez az operandusaink rendelkezésre állnak. Ez a példánkban csak a szelekciós műveletre teljesül. Az így kapott ideiglenes relációt ezután illesszük a *betétes* relációval, majd hajtsuk végre a projekciót.



6.5. ábra. Relációs algebrai fa kifejezés kiértékelési stratégiák illusztrációjához

A stratégia minden relációt kiszámít a fában, miközben létrejönnek közbülső relációk is. A materializáció költsége tehát a végrehajtott operációk költségének összege, plusz a közbülső relációk tárolásának és visszaolvasásának a költsége.

6.4.2. Pipelining

A kiértékelés hatékonysága növelhető, ha redukáljuk az ideiglenesen tárolásra kerülő rekordok számát. Ha megszervezünk egy olyan munkafolyamatot, amelyben a részegységek az előttük álló elemtől kapott részeredményekből a sorban következő számára állítanak elő részeredményeket, akkor kiküszöbölhetjük az ideiglenes

tárolás szükségességét. A fenti példán illusztrálva a lépéseket, mindhárom relációt egy pipeline-ba tesszük. A szelekció eredményét azonnal átadjuk a joinnak, nem számítjuk ki előre az egész relációt. További előnye, hogy kicsi a memóriakövetelmény, mert az eredményeket nem tároljuk sokáig, hanem továbbadjuk. Hátránya, ha nincs közbülső reláció, nem tudunk rendezni sem (nem történik materializáció).

A feldolgozás vezérlése alapján kétféle pipeline-t különböztetünk meg: igényirányított és termelőirányított.

- Az igényirányított esetben maga a rendszer fordul a csővezeték tetejéhez és kér rekordokat. Minden alkalommal, ha a csővezeték megkapja ezt a kérést, akkor kiszámítja és átadja a rendszernek.
- Termelőirányított pipeline esetén a csővezeték mentén elhelyezkedő műveletek nem várnak kérésre. A csővezeték legalsó szintjén minden művelet folyamatosan generálja a rekordokat és egy pufferbe teszi, amíg a puffer meg nem telik (ugyanígy tesz minden szint). Minden szinten minden művelet „egymástól függetlenül" dolgozik.

6.4.2.1. Pipeline kiértékelési algoritmus

Tekintsük a példánkban a 6.5. ábra join műveletét, amelynek bal oldala (a szelekció eredménye) csővezetéken érkezik. Az egész baloldali reláció nem áll rendelkezésre, a rekordok egyenként jönnek, ezért nem használhatjuk bármelyik join algoritmust (pl. a *összefésülés alapú illesztés* rendezés alapú illesztési algoritmus nem alkalmazható, ha a baloldali input nincs rendezve az illesztési attribútumok szerint). Az *indexalapú egymásba ágyazott ciklikus illesztés* azonban használható, mert ahogy beérkeznek a baloldali rekordok, az illesztési attribútumértékek alapján a jobboldali reláció feletti index segítségével kikeressük a jobboldali relációból az illeszkedő rekordokat és összeillesztjük őket egymással.

6.5. Relációs kifejezések transzformációi

Láttuk az elemi műveletek és egy adott lekérdezés kiértékelésének lépéseit. A bevezetőben azonban említettük, hogy egy adott lekérdezéshez formális úton több, egymással ekvivalens relációs algebrai alakot is konstruálhatunk, amelyek általában különböző költséggel hajthatók végre.

6.5.1. Ekvivalens kifejezések

Tekintsük a következő, természetes nyelven megfogalmazott kérdést: „Add meg azoknak a betéteseknek a nevét, akiknek van számlájuk Bázelben!"

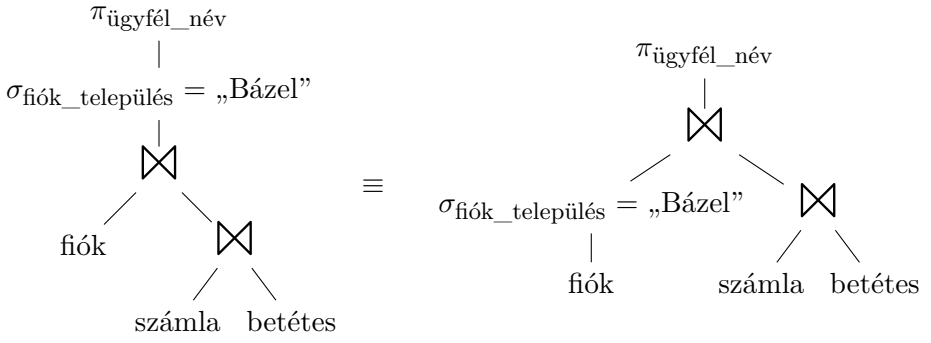
$$\pi_{\text{ügyfél\_név}}(\sigma_{\text{fiók\_település}=\text{"Bázel"}}(\text{fiók} \bowtie (\text{számla} \bowtie \text{betétes})))$$

A fenti relációs algebrai alak végrehajtása rengeteg erőforrást pazarolna, hiszen három reláció illesztése után végezné csak el a szelekciót. Egy sokkal hatékonyabb

alak a következő:

$$\pi_{\text{ügyfél\_név}}(\sigma_{\text{fiók\_település}=\text{"Bázel"}}(\text{fiók}) \bowtie (\text{számla} \bowtie \text{betétes}))$$

A második forma takarékosabban bánik az erőforrásainkkal, először kiválasztja a *fiók* relációból a Bázelben lévő fiókokat, majd csak ezt illeszti a maradék kifejezéssel. A kezdeti és az átalakított kifejezés relációs algebrai fáját mutatja a 6.6. ábra.



6.6. ábra. Ekvivalens kifejezések relációs algebrai fája

A legkisebb költségű végrehajtási terv megtalálása a lekérdezés optimalizáló feladata. Az optimalizáló első teendője adott lekérdezéshez ekvivalens algebrai alakok megtalálása. Majd az algebrai alakokhoz alternatív végrehajtási terveket kell készítenie.

6.5.2. Ekvivalencia szabályok

Az ekvivalens algebrai alakok legenerálásához az optimalizálónak szüksége van olyan szabályokra, amelyek mentén a feladat algoritmikusan elvégezhető, most ezeket a szabályokat vesszük sorra. Jelölések: θ, θ_1, θ_2 predikátumok, L_1, L_2, …L_n attribútumhalmazok, E, E_1, E_2 relációs algebrai kifejezések.

Megjegyzés. Emlékeztetünk rá, hogy az 5.1. szakaszban tárgyaltaknak megfelelően egy séma attribútumait *halmazként* kezeljük.

1. Szelekció kaszkádosítása:

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. A szelekció kommutativitása:

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Projekció kaszkádosítása, ha $L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n$:

$$\pi_{L_1}(\pi_{L_2}(\ldots \pi_{L_n}(E)\ldots)) = \pi_{L_1}(E)$$

4. A Θ-illesztés és a Descartes-szorzat kapcsolata:

$$\sigma_\theta(E_1 \times E_2) = E_1 \underset{\theta}{\bowtie} E_2$$

$$\sigma_{\theta_1}(E_1 \underset{\theta_2}{\bowtie} E_2) = E_1 \underset{\theta_1 \wedge \theta_2}{\bowtie} E_2$$

5. A Θ-illesztés kommutativitása:

$$E_1 \underset{\theta}{\bowtie} E_2 = E_2 \underset{\theta}{\bowtie} E_1$$

6. A természetes illesztés asszociativitása:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

7. A szelekció művelet disztributivitása a Θ-illesztés felett, ha a θ_1 csak E_1-beli attribútumokat tartalmaz:

$$\sigma_{\theta_1}(E_1 \underset{\theta_2}{\bowtie} E_2) = \sigma_{\theta_1}(E_1) \underset{\theta_2}{\bowtie} E_2$$

8. A projekció disztributív a Θ-illesztés felett, ha L_1 és L_2 E_1, illetve E_2-beli attribútumokat tartalmaz, és az illesztés feltételében csak $L_1 \cup L_2$-beli attribútumok vannak:

$$\pi_{L_1 \cup L_2}(E_1 \underset{\theta}{\bowtie} E_2) = (\pi_{L_1}(E_1)) \underset{\theta}{\bowtie} (\pi_{L_2}(E_2))$$

   Az imént megfogalmazott disztributivitás általánosításaképpen tekintsük azt az esetet, amikor a join feltételben szereplő attribútumokra nincs megkötés. Ekkor legyen L_3 és L_4 rendre a θ feltételben szereplő E_1 és E_2-beli attribútumok halmaza. L_1 és L_2 ismét csak E_1 illetve E_2-beli attribútumokat tartalmaz:

$$\pi_{L_1 \cup L_2}(E_1 \underset{\theta}{\bowtie} E_2) = \pi_{L_1 \cup L_2}((\pi_{L_1 \cup L_3}(E_1)) \underset{\theta}{\bowtie} (\pi_{L_2 \cup L_4}(E_2)))$$

9. A metszet és az unió kommutativitása:

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

10. Az unió és a metszet asszociativitása:

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. A szelekció disztributív az unió, a metszet és a különbség műveletek felett:

$$\sigma_\theta(E_1 \setminus E_2) = \sigma_\theta(E_1) \setminus \sigma_\theta(E_2) = \sigma_\theta(E_1) \setminus E_2$$
$$\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2) = \sigma_\theta(E_1) \cap E_2 = E_1 \cap \sigma_\theta(E_2)$$
$$\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$$

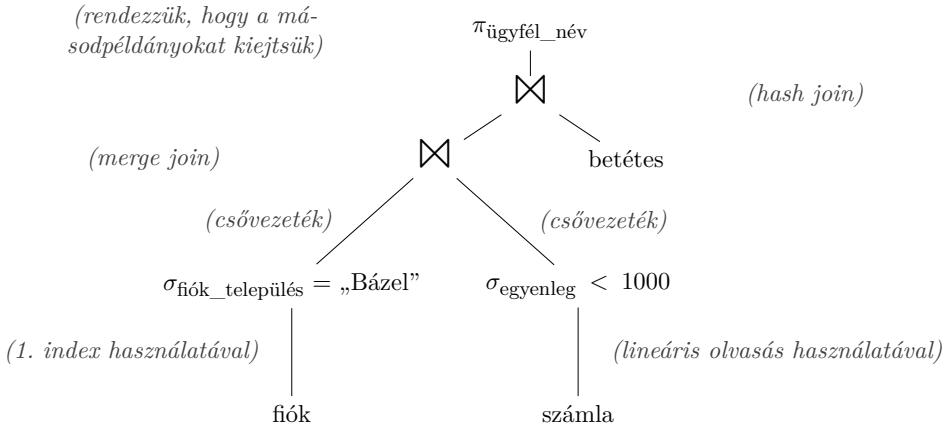
12. A projekció disztributív az unió művelet felett:

$$\pi_L(E_1 \cup E_2) = \pi_L(E_1) \cup \pi_L(E_2)$$

A felsorolt szabályok nem tartalmazzák az összes ekvivalenciaszabályt, hanem csak ízelítőt adnak belőlük. Számos egyéb, nem csak alapműveletekre kihegyezett átalakítási lehetőség is létezik.

6.6. A végrehajtási terv kiválasztása

Az ekvivalens kifejezések generálása csak az első lépése az optimalizálásnak. A második lépésben minden egyes kifejezéshez konkrét algoritmusokat kell rendelnünk. Meg kell mondanunk, hogy a műveleteket milyen sorrendben, milyen algoritmus szerint, milyen munkafolyamatba szervezve hajtjuk végre. Ennek egy grafikus alakban megadott példa reprezentációját mutatja a 6.7. ábra.



6.7. ábra. Végrehajtási terv reprezentációja

6.6.1. Költségalapú optimalizálás

A fordító az előző fejezetben látott azonosságok alkalmazásával először felsorol minden, az eredeti kifejezéssel ekvivalens kifejezést (véges sokat). Ezután minden kifejezéshez hozzá tudunk rendelni végrehajtási terve(ke)t. A megfelelő végrehajtási terv kiválasztásának folyamata tulajdonképpen a költségoptimalizálás.

Minden végrehajtási tervre kiszámítjuk a költséget, és kiválasztjuk a legolcsóbbat (becslések, statisztikák alapján). Hátránya, hogy túl sokféle végrehajtási terv lehet, amely rengeteg munkát ró a rendszerre. Tekintsük például három reláció illesztését. Három relációt hatféleképpen állíthatunk sorrendbe, és ezt még be kell szorozni 2-vel attól függően, hogy a zárójelet az első kettő relációhoz vagy a második kettőhöz tesszük (ne feledjük, hogy a végeredmény ugyan minden esetben azonos lesz, azonban a join elvégzésének módja, erőforrásigénye még két reláció esetén sem azonos akkor, ha a sorrendjüket felcseréljük). Általános esetben n reláció joinja $\frac{(2\cdot(n-1))!}{(n-1)!}$ különböző ekvivalens alakot jelent (és ebben nincs is benne az algoritmus hozzárendelés). Ez már kis n esetén is rengeteg ekvivalenst produkál, pl. $n = 7$ esetén 665 280-at, $n = 10$ esetén már több mint 17,6 milliárdot!

Szerencsére azonban nincs is szükség az összes ekvivalens kifejezésre. Némi heurisztikával kezelhetőbb méretűvé csökkenthetjük a problémateret. Tekintsük a következő kifejezést:

$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

Keressük meg először azt az optimális alakot, amely csak az első három relációt illeszti, majd utána az eredményt a maradék két relációval illeszti. Az első három reláció illesztésére 12 lehetőség adódik, majd az eredmény és a maradék két reláció illesztése ismét 12 lehetőséget kínál. Ha értelmetlenül minden verziót kipróbálunk, akkor összesen 144 lehetőséget nézünk végig. Ha azonban először megkeressük az első három reláció optimális kiértékelését és a 4. és az 5. relációval már ezt az optimális eredményt illesztjük, akkor a kiértékelés csupán 12 + 12 lépést próbálgat végig.

A bemutatott algoritmus egyik legnagyobb problémája, hogy elméletileg is rossz, mert nem minden esetben képes megtalálni az optimális megoldást. Ennek oka, hogy az algoritmus mohó lévén mindig a lokálisan legjobb megoldást választja, és nem mérlegeli azt, hogy egy kicsit rosszabb lokális megoldás globálisan esetleg jobb eredményre vezetne. Sajnos nem létezik olyan algoritmus, amely kis komplexitás mellett képes az optimális megoldást legenerálni, ezért be kell érnünk szuboptimális megoldásokkal.

6.6.2. Heurisztikus optimalizálás

A költségalapú optimalizálás legnagyobb hátránya magának – mint láttuk – az optimalizációs algoritmusnak a költsége. A legtöbb kereskedelmi rendszer ezért valamilyen heurisztikát használ a megfelelő kifejezés kiválasztásához. Egy heurisztika alapú optimalizációs stratégia szabályai lehetnek például a következők:

1. Induljunk ki a lekérdezés kanonikus alakjából (Descartes-szorzatokat, szelekciókat és projekciókat tartalmaz ilyen sorrendben).
2. Bontsuk szét a szelekciók konjunkcióját szelekciók szorzatára, hogy minden szelekciónak csak egy tényezője legyen. Ez lehetővé teszi, hogy a fában a szelekciókat lefelé vándoroltassuk, ezáltal a közbenső műveletek jóval kevesebb rekordot adjanak végeredményként.

3. A kiértékelési fában vándoroltassuk lefelé a szelekciókat.
4. Határozzuk meg, hogy mely szelekció és természetes illesztés eredményezi a legkisebb (= legkevesebb rekordot tartalmazó) relációkat. Használjuk fel a természetes illesztés asszociativitását, alakítsuk át a fát úgy, hogy ezek a szelekciós és join műveletek hajtódjanak végre először.
5. Előfordulhat, hogy a join megegyezik a Descartes-szorzattal. Ha ezt egy szelekció követi, akkor vonjuk össze a kettőt egy Θ-illesztés műveletté, így kevesebb rekordot kell generálni.
6. Törjük szét a projekciólistákat. Az egyes vetítéseket lökjük lefelé a fán, amennyire tudjuk. Ehhez új vetítéseket is létrehozhatunk, ha szükséges.
7. Keressük meg azokat a részfákat, ahol csővezetéket lehet alkalmazni.

A szabályok alkalmazásával több relációs algebrai fát kapunk, amelyek csomópontjaihoz további heurisztikák alapján algoritmusokat és pipelining-stratégiát rendelünk. Ezeknek meghatározzuk a költségét, és vesszük közülük a legolcsóbbat.

Mint láttuk, a költségalapú és a heurisztikus optimalizálás is jelentős terhelést jelent a rendszer számára. Nem elég tehát a legjobb megoldást megtalálni, hanem a keresés költségét is optimalizálni kell. Az az érdekes helyzet adódik, hogy az optimális optimalizáló a saját működését (tehát a végrehajtási terv keresését) és magának a megtalált megoldásnak a végrehajtását egyszerre kell, hogy optimalizálja.

6.7. A fejezet új fogalmai

szintaktikai elemzés, kiértékelési terv, szabály alapú optimalizálás, relációs algebrai fa, költség alapú optimalizálás, katalógus információk, attribútum kardinalitása, kiválasztási kardinalitás, elsődleges/másodlagos index, (blokk alapú) nested loop join (index alapú és anélkül), hash join, merge join, materializáció, pipelining

7. fejezet

A hálós adatmodell

7.1. 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 pedig *adatlekérdező és -manipulációs nyelv* (data manipulation language, DML) néven vált ismertté. Bár a XXI. századra az elterjedtsége marginálissá vált, még mindig vannak üzemben nagyméretű rendszerek bankokban, államigazgatásban.

7.2. 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 sethez 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 – rekord típus *(record type)*. Az R *rekord típus* egy olyan $A_1, A_2, \ldots, A_n$, n-es (tuple), ahol A_i-k az attribútumnevek és minden A_i-hez egy D_i halmaz, az attribútum doménje is hozzátartozik, amely halmazból az A_i attribútum értéket vehet fel.

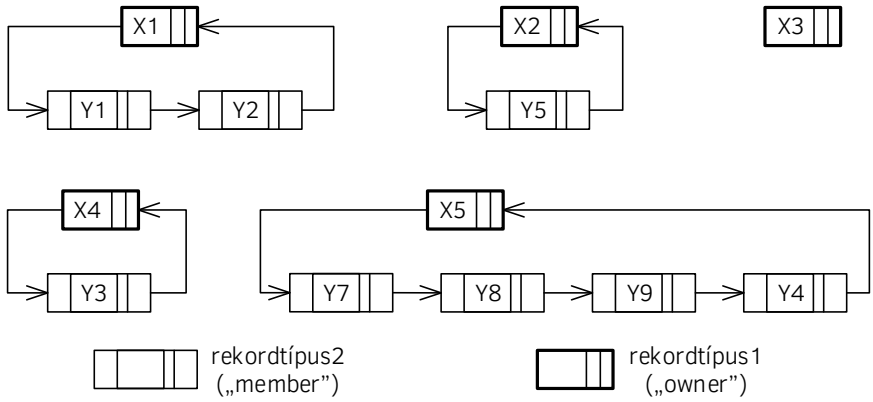
Egy R típusú, n komponensű r rekord a $D_1 \times D_2 \times \cdots \times D_n$ Descartes-szorzatnak egy eleme, szintén egy n-es: $r(d_1, d_2, \ldots, d_n)$.

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 7.1. 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.



7.1. ábra. Példa setek ugyanazon set típusból

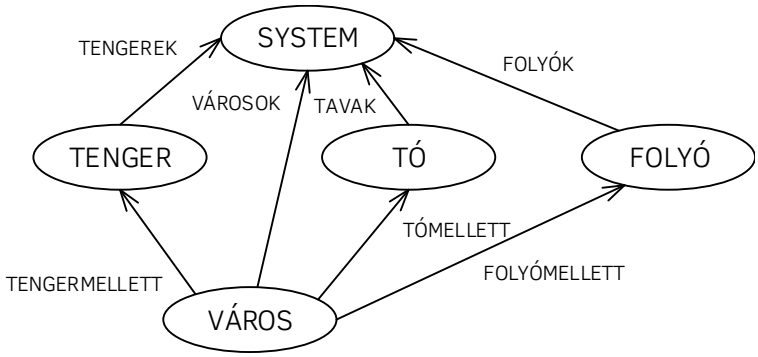
Definíció – set típus *(set type)*. Legyen R_1 és R_2 két rekord típus és legyenek $\Im(R_1)$ és $\Im(R_2)$ a konkrét eseteik halmazai. Ekkor az S *set típus*t az $S := R_1 \times R_2$ művelettel definiálhatjuk, ami egy $\Im(R_2) \to \Im(R_1)$ függvényszerű kapcsolatot ír le. R_1 az owner típus, R_2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

7.2. ábra. A hálós adatmodell egy grafikai jelölésrendszerének elemei

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

Egy hálós adatbázis tehát a legmagasabb szinten set típusok összességéből áll. Gyakran létezik egy „SYSTEM” rekord típus is, amelynek egyetlen példánya van csupán, azonban ownerévé tehető akár minden – az adatbázisban található – rekord típusnak. Ezzel az azonos típusú rekordok elérését könnyítik meg, amelyek általában csak több setből lennének összegyűjthetők (ld. 7.5.2. alszakasz).

Példa. A 7.3. ábra egy egyszerű kartográfiai adatbázis set típusait mutatja be. Feltételeztük, hogy egy város mindig csak egy tenger (tó, folyó) mellett fekszik.



7.3. ábra. Egy egyszerű kartográfiai adatbázis hálós modellje

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

7.3. Implementációs kérdések

Elsősorban a setek megvalósítása említésre méltó. Kézenfekvőnek tűnik, ha az egy ownerhez 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 setnek felelnek meg. A változó hosszúságú rekordok implementálására a fizikai szervezésnél megismert módszerek alkalmazhatók.

Probléma akkor adódik, ha R_1 membere más set típusnak is, pl. R_3 is ownere. Ekkor ugyanis R_1 példányait fel kellene R_2 és R_3 után is sorolni, ami közvetlenül nyilván nem tehető meg.

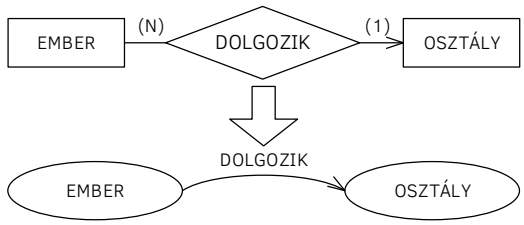
Jobbnak tűnik egy olyan megoldás, amelyben nem kell különböző típusú rekordoknak szomszédoknak lenniük. Ilyen tulajdonságúak a láncolt listák. Legegyszerűbb formáját a 7.1. ábra már bemutatta: ilyenkor rekordonként egy-egy mutatót kell a rekordokba járulékosan elhelyeznünk, amelyek a set (logikailag) következő tagjára mutatnak. A mutatók körbe érnek, így tetszőleges, a sethez tartozó rekordról tetszőleges másik (akár owner, akár member) elérhető. Az owner ill. member rekordok pl. a típusuk alapján különböztethetők meg.

Ha a seten 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 ownerre mutatnak. Mindezek természetesen a setek karbantartását teszik költségesebbé.

7.4. Hálós adatbázis logikai tervezése ER-diagramból

Ha egy adott problémakörnek már elkészült az ER-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.



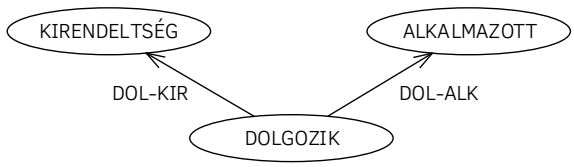
7.4. ábra. Egy bináris több-egy kapcsolat megfelelője egy set típus

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 ER-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.).

Példa. A teljesség igénye nélkül, példaképpen nézzük meg, hogyan alakítható át a 4.3. ábra ER-diagramja hálós modellbe.



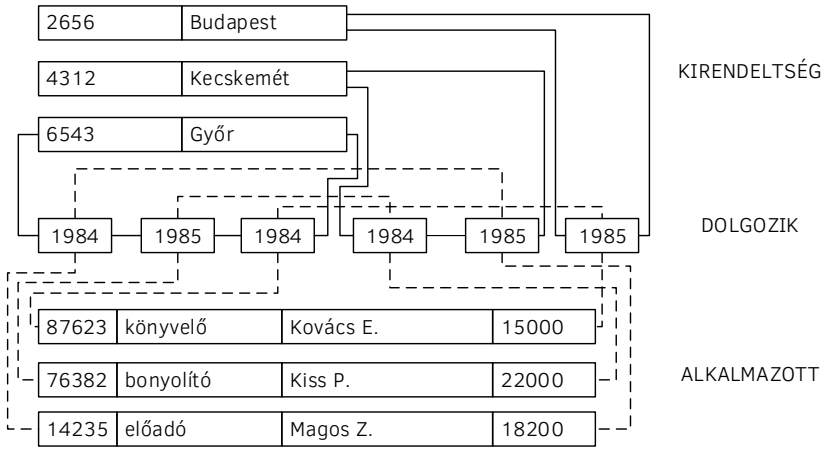
7.5. ábra. Hálós adatbázis séma a 4.3. ábra. ER-diagramjából

Az egyes rekord típusokban az alábbi mezők fognak megjelenni a setek kialakításához szükséges mutatókon kívül:

- KIRENDELTSÉG: k_kód, hely;

– ALKALMAZOTT: a_kód, beosztás, név, fizetés;
– DOLGOZIK: dátum.

Végül álljon itt ennek az adatbázisnak néhány mintarekordja.



7.6. ábra. Néhány rekord és set a 7.5. ábra modelljéhez

7.5. Adatkezelés lehetőségei a hálós adatmodellben

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

7.5.1. A hálós sémaleíró nyelv (DDL) elemei

A hálós *adatdefiníciós nyelv* (data definition language, 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 tip. név>` típusú setek 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 ún. *navigáció* (ld. 7.5.2. alszakasz) 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 seten belül, de lassú a keresés, ha nem ismerjük a kérdéses rekord ownerét.

A `<mezők leírása>` mezőben kell felsorolni a rekord típus mezőit és meghatározni a típusukat, hosszukkal együtt. Lehetőség van többértékű mezők (ún. vektormezők) 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. típ. név2> <insert opciók> <retention opciók>
```

A `<rendezési inf.>` mezőben azt írhatjuk elő, hogy egy setben 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 setbe.

Pl.

– `ORDER IS FIRST|LAST|NEXT` esetén az újonnan felvett rekord a setben az első | utolsó | következő lesz. A következő az ún. kurrencia mutatóhoz képest

értelmezett (ld. 7.5.2. alszakasz).

– `ORDER IS SORTED ASCENDING|DESCENDING BY KEYS` esetén a kulcsmezők által meghatározott sorrendnek megfelelően kerülnek bele a setbe a rekordok.

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

– `MANUAL` esetén „kézzel”, esetleg a rekord felvétele után jóval később kell rendelkezni a setbe rendezésről.
– `AUTOMATIC` mód esetén a rekord valamely mező értékétől függően automatikusan bekerül meghatározott setbe.

A `<retention opciók>` helyén egy rekord „egyedüli” létezését határozhatjuk meg:

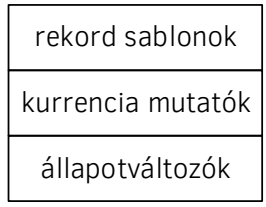
– `OPTIONAL` esetén, ha a rekord kikerül egy setből, akkor létezhet önállóan is,
– `MANDATORY` esetén ha kikerül egy setből, akkor egyúttal egy másikba be kell kerülnie. Pl.: DOLGOZÓ-OSZTÁLY mellett praktikus.
– `FIXED` esetén a rekord ownere rögzített, így egyáltalán nem kerülhet ki adott setből Pl.: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

7.5.2. Hálós DML

A hálós adatlekérdezés alapvetően *rekordorientált*, azaz egyszerre egyetlen rekord kiválasztása (majd kiolvasása) támogatott. (A relációs lekérdezések (ld. 5.3. szakasz) ezzel szemben *halmazorientáltak*).

Rekordcsoportok eléréséhez az *adatlekérdező és -manipulációs nyelv* (data manipulation language, 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 ún. *User Work Area* (UWA), amely egy pontosan definiált adatstruktúra. Felépítése a 7.7. ábrán látható.



7.7. 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 setek által meghatározva. A navigációt ún. *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 adatbázis-kezelő 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 setben 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`

Példa. 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');
```

7.5.2.1. 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.

7.5.2.2. A FIND parancsok

Több formája is létezik, céljuk a navigáció támogatása azzal, hogy minden kurrencia mutató értékét – a találat függvényében – beállítja.

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 setben

Egy set végigkereséséhez – egyik módszerrel – először az ownert kell megkeresni a

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

paranccsal, amely beállítja a kurrencia mutatókat: a CRU a `<set típus>` kurrens setének 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 setben a „current of set type" kurrencia mutató által azonosított rekordtól kiindulva (ezt az előbb az ownerra állítottuk be).

Példa navigációra egyik setről egy másikra. Mondjuk meg, hogy Kovács Ernőnek melyik évben hol voltak a cégen belül a munkahelyei (ld. 7.5., 7.6. 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 settől, 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 ownere vagy membere.

Annak a módja, hogy hogyan kerül be az új rekord valamely setbe, 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 az összes membert, így szerencsétlen esetben akár az egész adatbázist.

7.6. A fejezet új fogalmai

rekord, rekord típus, set, set típus, owner, member, link, system rekordtípus hálós séma, navigáció, rekord sablon, kurrencia mutatók (current of run-unit, current of record type, current of set type), UWA

8. fejezet

Objektumorientált adatbázis-kezelő rendszerek

A programozási nyelvek, módszertanok területén az objektumorientált paradigma nem új, jól bevált. Ugyanakkor az *objektumorientált* (object-oriented, OO) adatbázis-kezelés területén hosszú ideje nem történt meg az az áttörés, ami annak idején a relációs rendszerek megjelenését jellemezte. Bevezetésül vizsgáljuk azt meg, hogy milyen előnyöket nyújthat egyáltalán az OO szemlélet, ill. milyen hátrányokra kell számítanunk.

- Az OO adatbázis-kezelőktől az OO tervezési, fejlesztési módszerek összes előnye elvárható: jól megtervezett struktúrák, osztályszerkezetek kialakításával robusztus, könnyen továbbfejleszthető rendszereket, könnyen újrafelhasználható részegységeket kaphatunk, a fejlesztés produktívabbá válik, a létrejövő szoftver minősége javul. Mindezen tulajdonságok – mint minden nagy szoftverprojektben – az adatbázis-kezelésben 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.

8.1. A relációs adatmodell gyengeségei

Az OO koncepciók megjelenése az adatmodellek – adatok és rajtuk végezhető műveletek – területén elsősorban a relációs adatmodell számára jelent konkurenciát, mivel a 90-es évektől kezdődően a relációs rendszerek szinte egyeduralkodónak

tekinthetők, ha az iparban, ill. gazdasági életben alkalmazottakat tekintjük, nem pedig a kísérleti vagy speciális célokra fejlesztetteket.[1]

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 egy relációs adatmodellre. Erre néhány egyszerű példát mutatunk be.

Példa *(1).* Hogyan írhatjuk le objektumorientált 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 adatbázis-kezelőnek 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. Objektumorientált 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 felmerül az igény tetszőleges számú töréspontot tartalmazó vonalak eltárolására. Ezt a struktúrát egy relációban pl. a következő módon tárolhatjuk:

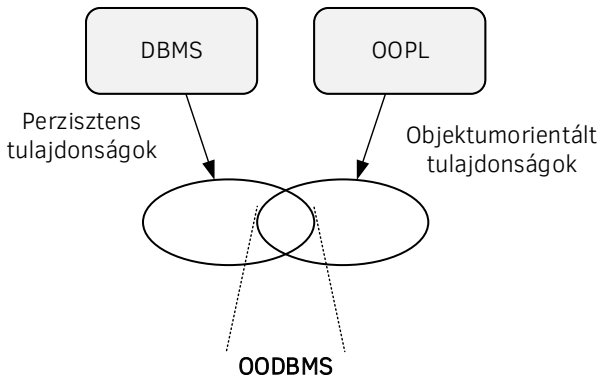
[1] Ez 2010 után kezd változni úgy, hogy egyre több (főleg webes) rendszer mögött NoSQL adatbázis-kezelő van (leginkább MongoDB, Cassandra, Redis, ld. `http://db-engines.com/en/ranking`).

VonalPontok(*VonalID*, *PosX*, *PosY*, *Pozíció*), ahol *VonalID* azonosítja, hogy a *PosX* és *PosY* koordinátákkal megadott pontok melyik vonalhoz 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 a tárolás meg is oldható, kényelmesnek, kézenfekvőnek nem nevezhető. Megállapíthatjuk tehát, hogy a relációs adatmodell nem támogatja a lista jellegű információ tárolását sem (hiszen halmazszemléletű).

8.2. Objektumorientált adatbázis-kezelők

Az OO adatbázis-kezelő rendszerek történelmileg két irányból fejlődtek. Az egyik az *objektumorientált programozási nyelvek* (object-oriented programming languages, OOPLs), a másik az adatbázis-kezelők iránya. Ahhoz, hogy adattárolásra képesek legyenek, az OO programozási nyelveket a *perzisztencia* (persistence) tulajdonságával kell felruházni; azaz az objektumok ne csak a program futása alatt létezzenek, hanem a futás befejeződése után is. A hagyományos adatbázis-kezelőket OO tulajdonságokkal – *osztályhierarchiák* (class hierarchies), *öröklődés* (inheritance), *polimorfizmus* (polymorphism), *egységbezárás* (encapsulation) stb. – kell ellátni[2] (8.1. ábra).

Az OO adatbázis-kezelők 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 adatbázis-kezelő mely OO tulajdonságokat és milyen formában valósít meg. Azonban vannak olyan alapvető jellemzők, melyek a legtöbb adatbázis-kezelőben megtalálhatók. Most csak ezekre összpontosítunk.



8.1. ábra. Az OODBMS-ek származtatása

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

A könnyebb megértés érdekében bizonyos analógiákra érdemes rámutatni az OO adatábrázolás és a relációs adatmodell között – a jelentős különbségek ellenére is. Az OO rendszerek *objektuma* egy *n-esnek* (a reláció egy elemének, ill. „a tábla egy sorának"), az *osztály* pedig a *relációs séma*nak feleltethető meg. Az *OO adatmodellben* (object-oriented data model) 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 (*asszociáció*, association) összessége képezi magát az OO adatbázist.

8.2.1. Típuskonstruktorok

Minden programozási nyelv és adatbázis-kezelő 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 adatbázis-kezelők egy része közvetlenül vagy közvetve a *típuskonstruktorokat* (type constructor) nyújtja.

Példa *(4).* 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.

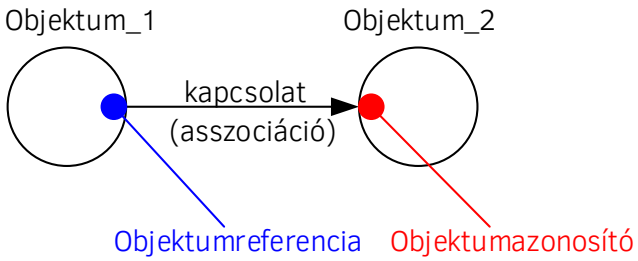
Listakonstruktor: T_{list} = **LIST** OF(A: T), ahol A attribútum, T pedig az attribútum típusa. A lista típusra jellemző, hogy elemei szekvenciálisan rendezettek. A programozási nyelvek láncolt listájával analóg struktúra kialakítására ad lehetőséget.

Tuplekonstruktor: T_{tuple} = **TUPLE** OF($A_1 : T_1, \ldots, A_n : T_n$), ahol A_i attribútum, T_i pedig az A_i attribútum típusa. A tuple típus a relációs modellben a reláció egy elemének 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.

8.2.2. Kapcsolatok – asszociációk

Az OO adatbázis-kezelők nagy újítása, hogy tetszőlegesen bonyolult kapcsolatokat is átláthatóan tudnak kezelni. Ezeket a kapcsolatokat a szakirodalom – a más „kapcsolatoktól" való megkülönböztetés érdekében – gyakran *asszociációnak* (association) nevezi (8.2. ábra). Az asszociációkat osztályokra definiáljuk, így az asszociációk az osztályok példányai, az objektumok között teremtenek kapcsolatot.



8.2. 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 is elérhető A, akkor 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 a lekérdezések során van elsősorban jelentősége. Egy objektumból kifelé irányított asszociáció mentén elérhetünk egy újabb objektumot, melynek nyilvános adatmezőihez így hozzáférhetünk. Az asszociáció kiindulási oldalán *objektumreferencia* (object reference), azaz a célobjektum *objektumazonosítója* (object identifier) található. Az objektumazonosító szolgál arra, hogy – még ha az objektumok tartalma teljesen azonos is – az objektumokat meg tudjuk különböztetni.

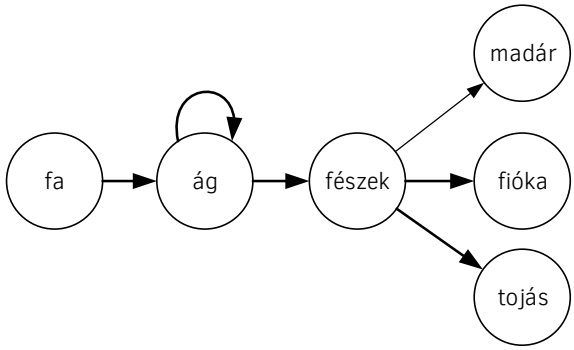
Az asszociációk mentén *terjedési tulajdonságokat* (cascade relationship) is meghatározhatunk, mint pl. az objektum törlése, zárolása vagy zár felszabadítása. Ez azt jelenti, hogy ha egy olyan A objektumot törlünk le, melynek egy másik B objektum felé törléssel terjedő asszociációja volt, akkor ez a bizonyos másik B objektum is törlődik. Ha B objektumnak is van ilyen asszociációja C objektum felé, akkor ugyanez történik tovább korlátlan mélységben.

Példa *(5).* 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 melyben lehetnek madarak, fiókák, ill. tojások. Mivel a fák nagyon öregek, előfordul, hogy ha villám csap beléjük, akkor kidőlnek. Ilyenkor a tojások és a fiókák odavesznek, a szülőmadarak pedig elrepülnek.

Hogyan ábrázoljuk ezt a struktúrát az asszociációk és a terjedési tulajdonságok felhasználásával? (8.3. 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[3] elpusztul, az összes ága is elpusztul.

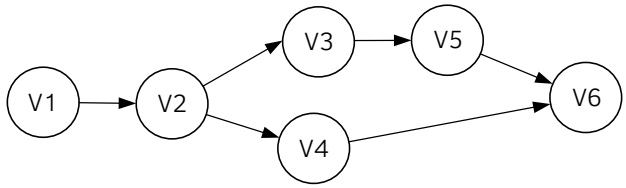


8.3. ábra. Terjedő asszociációk

8.2.3. Verziókezelés (version control)

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* (versions), melyek kezelését a legtöbb OO adatbázis-kezelő rendszer támogatja (ld. még 10.9.4. alszakasz).

A verziók fejlődése lehet lineáris és elágazó. Nem kizárt különböző verziók újbóli „egymásra találása” sem.



8.4. ábra. Verziók

Példa *(6).* Nézzük meg egy autómodell fejlesztését (8.4. 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 tovább 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 jó néhány tulajdonsága.

8.2.4. Nyelvek

Hasonlóan az OO adatbázis-kezelőkhöz, a lekérdező nyelvek is két irányból fejlődtek.

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

Az egyik út – melyet elsősorban a relációs adatbázis-kezelők 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. 8.3. szakasz).

A másik út, ami előttünk áll, egy OO nyelv perzisztens tulajdonságokkal való ellátása. A legtöbb napjainkban működő OO adatbázis-kezelő megalkotásánál a C++ perzisztens kiterjesztését választották. Egyre jelentősebb azonban a (Sun által kifejlesztett[4]) Java nyelv, melyhez egyre több rendszer kínál elérési felületet.

Míg a relációs adatbázis-kezelőknél 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 adatbázis-kezelő nyelvének más nyelvbe történő beágyazására pedig értelemszerűen nincs szükség, hiszen például az SQL-t legtöbbször éppen C-be ágyazzák be.[5] Azzal, hogy nem csak az adatbázis-alkalmazás fejlesztése, hanem maga az adatbázis-kezelés is egy OO programozási nyelven történik, az absztrakciós távolság a két terület között lecsökkent, ezáltal a teljes rendszer implementációjának a hatékonysága nőtt.

8.3. Az objektum-relációs technológia

Az OO szemlélet hosszabb idő alatt, fokozatosan hatolt be a relációs adatbázis-kezelés világába. Először az OO alkalmazásfejlesztő eszközök jelentek meg, amelyekkel klasszikus relációs adatbázis-alkalmazásokat lehet hatékonyan fejleszteni. Az *objektum-relációs* (object-relational, OR) alapgondolat szerint a relációs rendszerek összes előnyét megtartva (lekérdezésoptimalizálási lehetőség, kiforrott technológia stb.), OO tulajdonságokkal látják el a relációs adatbázis-kezelőt. Ezt a szemléletet követi még napjainkban is a relációs adatbázis-kezelők 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.

[4] A Sun 2009-től már az Oracle tulajdonában van.

[5] Azonban az adatbázis-kezelőt leggyakrabban valamilyen alkalmazásprogramozási interfészen (application programming interface, API) keresztül (pl. ODBC, JDBC) érik el.

8.4. Összegzés

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

- Mivel az OO adatbázis-kezelők mögött nem áll olyan hatékony matematikai modell, mint ami a relációs rendszereket támogatja, csak korlátozott lehetőségek mutatkoznak a lekérdezések optimalizálására és/vagy a séma – nagyrészt – automatikus egyszerűsítésére. Mivel OR esetben az objektumhierarchia leképezése táblákra automatikusan történik, az bonyolult esetekben pazarló, ésszerűtlen 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 adatbázis koncepciók újítása a hagyományos – elsősorban relációs – adatbázis-kezelőkhöz 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 adatbázis-kezelők esetén erre a sebességnövekedésre – a háttérben lévő relációs motor miatt – nem számíthatunk.

Felmerül tehát a kérdés: Mikor érdemes OO adatbázis-kezelőt 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 *számítógéppel segített tervezés* (computer-aided design, CAD), a *számítógéppel segített szoftvertervezés* (computer-aided software engineering, CASE), a térképészet-geográfia, a multimédia stb. területének problémái. Ezeken a területeken az OO, ill. OR adatbázis-kezelő rendszerek egyre intenzívebb előretörése várható.

8.5. A fejezet új fogalmai

objektum, osztály, öröklődés, polimorfizmus, rekurzív lekérdezés, asszociáció, objektumreferencia, objektumazonosító, típuskonstruktor (halmaz, lista, rekord), terjedési tulajdonság, verziók, SQL3, objektum-relációs technológia

9. fejezet

Relációs adatbázisok logikai tervezése

Bár a relációs adatmodell nem is az egyedüli, nem is a legjobb minden szempontból, mégis, a legelterjedtebb adatbázis-kezelő rendszerek még ma (2019) is a relációs adatmodellen alapulnak. Emiatt kitüntetett jelentősége van a relációs adatbázisok tervezésének. Jelen jegyzetben ezért szántunk a problémának külön fejezetet.

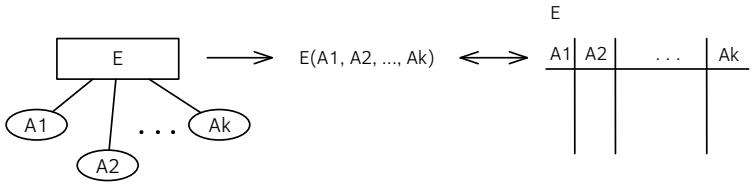
A tervezés első lépésben az adatbázis logikai tervezésére terjed ki (csak ezután szokás a fizikai tervezést végezni, tehát pl. a segédstruktúrákat kialakítani, tárolási paramétereket meghatározni), hiszen konkrét feladat megoldásához általában valamely megvásárolható relációs adatbázis-kezelő rendszert használnak fel. Ilyenkor nincsen sem mód sem szükség az adatbázis-kezelő működésének számos részletét megváltoztatni. A tervezés ekkor tehát arra korlátozódik, hogy meg kell határozni az adatbázis logikai szerkezetét (fogalmi (logikai) adatbázis: a relációs sémákat, definiálni kell az egyes adattípusokat), fizikai szerkezetének a paramétereit, a célszerű/szükséges segédstruktúrákat (fizikai adatbázis), majd meg kell írni magát az adatokat manipuláló programot/programokat, az *adatbázis alkalmazásokat*. Az alkalmazásokban az adatbázishoz való hozzáférés alapulhat pl. a szabványosított SQL nyelven, amelyet beágyazhatnak valamely magas szintű procedurális nyelvbe, de jellegzetesen valamilyen API-n (*alkalmazásprogramozási interfész*) keresztül hívják meg. Az alkalmazásfejlesztésnek számos, különböző szinten automatizált fejlettebb módszere is létezik.

Bárhogyan is hozzuk létre az alkalmazásainkat, az első lépés mindig az adatbázis logikai (koncepcionális) megtervezése. A tervezésnek két karakterisztikusan különböző módszerét ismertetjük a továbbiakban, amelyek azonban kombinálhatók is, és adott esetben jól kiegészítik egymást.

9.1. Tervezés ER-diagramból

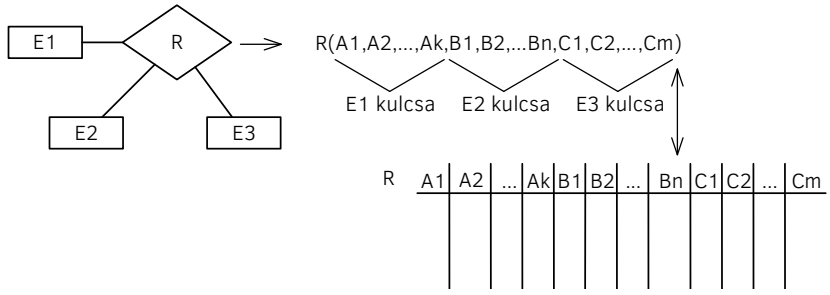
A 4.2. szakaszban megismerkedtünk az ER-diagrammal, amely szemléletes ábrázolásmódja következtében hatékonyan támogatja a valóság modellezésének folyamatát. Az 5. fejezetben megismerkedtünk a relációs adatmodellel. Természetes az az igény, hogy ER-diagramokat relációs sémákká kíséreljünk meg átalakítani,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

1. Az egyedhalmazokat olyan relációs sémával ábrázoljuk, amely tartalmazza az entitáshalmaz összes attribútumát. A reláció minden egyes n-ese az entitáshalmaznak pontosan egy példányát fogja azonosítani (ld. 9.1. 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.



9.1. ábra. Entitáshalmaz transzformációja relációs sémába

2. A kapcsolattípusokat olyan relációs sémákká alakítjuk, amelyek attribútumai között szerepel a kapcsolatban résztvevő összes entitáshalmaz kulcsa is (ld. 9.2. 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 szóban forgó kapcsolatban vannak egymással.



9.2. 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.

Ha pl. a 4.2. ábrán a *DOLGOZIK*: EMBER, OSZTÁLY kapcsolattípust akarjuk relációs sémákba transzformálni, akkor kihasználhatjuk a kapcsolat N:1 jellegét (függvényszerűségét), és az általános módszerből adódó három relációs séma helyett már kettővel is kifejezhetjük a kapcsolatot:

- OSZTÁLY(OSZT_ID, OSZT_NÉV, ÉPÜLET, EMELET)
- EMBER(SZEM_SZÁM, NÉV, ANYJA_NEVE, SZÜL_DÁTUM, OSZT_ID)[1]

Vegyük észre, hogy így ráadásul a relációs sémák struktúrájába sikerült beleépítenünk a kapcsolat függvényszerűségét biztosító kényszert is, amit elveszítettünk volna akkor, ha az általános módszer szerint transzformáltuk volna a kapcsolatot.

Megjegyzés. Egy több-egy kapcsolat két relációs sémába történő leképzésekor a „több" oldalon álló egyedhalmaznak megfelelő relációban az idegen kulcsnak megfelelő attribútum(ok) NULL értéket vehetnek fel (pl. a fenti példában ha egy ember nem dolgozik egyik osztályon sem, akkor az OSZT_ID attribútuma NULL lesz). Azzal, hogy két relációs sémát hozunk létre, az eredeti információ kinyeréséhez eggyel kevesebb természetes illesztés művelet szükséges.

Vegyük észre továbbá azt is, hogy az ER-diagram relációs sémákba alakításával elveszítettük az egyedek és kapcsolatok formális megkülönböztethetőségét.

Példa. Transzformáljuk relációs sémákká a 4.3. ábra ER-diagramját! Relációs sémák az entitáshalmazokból:

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

Relációs séma az egyetlen kapcsolatból:

- DOLGOZIK(KKÓD, AKÓD, DÁTUM)

Strukturálisan ezek a relációs sémák pontosan ugyanúgy néznek ki, mint a

- K(KK, H)
- A(A, N, B, F)
- D(KK, A, D)

sémák, azonban legfeljebb a kényszerek (kulcsok, idegen kulcsok) segíthetnek abban, hogy a sémák eredetére következtetni lehessen.

[1] Azokat az attribútumokat, amelyek egy adott relációs sémában nem, de egy másikban kulcsattribútumok, szokás szerint kétszeres aláhúzással jelöljük. Ennek az elnevezése: *idegen (vagy külső) kulcs* (ld. még a 9.2.3.2.3. bekezdést).

9.2. Tervezés sémadekompozícióval

Ha az ER-modellezés segítségével jutunk el a relációs sémáinkhoz, akkor a sémákban található attribútumokat és a relációk (valamint a teljes, az adatokon műveleteket végző adatbázis alkalmazás) további tulajdonságait az fogja meghatározni, hogy milyen „ügyesek" voltunk az ER-diagram megalkotása során. A relációk azonban tetszőleges számú attribútumot tartalmazhatnak. Így akár a rendszerben található valamennyi adatot beépíthetjük egyetlen sémába (ún. *univerzális sémába*), 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 – pl. tárolási és adatmanipulációs 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. relációban 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 fordul elő többször is a relációban. Ezt a hétköznapi nyelvben is redundanciának nevezzük, mert máshonnan is tudhatjuk, hogy az illető szeme színe barna. Általánosabban:

Definíció – *redundáns reláció.* Ha egy relációban valamely attribútum értékét a relációban található más attribútum(ok) értékéből ki tudjuk következtetni valamely ismert következtetési szabály segítségével, akkor a relációt *redundánsnak* nevezzük.

A kikövetkeztethető adatokra gyakran azt mondjuk, hogy *származtatott* adatok.

Példa. Az alábbi reláció szállítók adatait tartalmazza, ki milyen árut (*TÉTEL*) mennyiért (*ÁR*) szállít és a szállító (*NÉV*) hol lakik (*CÍM*).[2]

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

[2] Vegyük észre, hogy az áruszállítás ténye, ill. az, hogy egy adott szállítónak mi a lakcíme, valamint, hogy egy adott árucikknek mi az ára, mind azáltal van kifejezve, hogy a megfelelő adatértékeket egyazon sorba írva összerendeltük őket, ugyanakkor az összerendelés pontos szemantikájáról az adatstruktúra elemei – szigorúan véve – semmit nem mondanak.

Ha feltételezzük, hogy egy szállítónak csak egyetlen címe lehet, akkor ebben a relációban a címek többszörös tárolása teljesen felesleges, redundanciát okoz.

Vegyük észre, hogy az egyes relációk redundanciáját csökkenthetjük, ha az adatbázist alkalmas módon több, egyenként kevesebb attribútumot tartalmazó relációból alakítjuk ki.

9.2.1. Anomáliák (data anomalies)

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

9.2.1.1. Módosítási anomália (update anomaly)

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 nemcsak *többletmunkát* jelent, hanem a logikai ellentmondások keletkezésének lehetőségét is magában hordozza.

9.2.1.2. Beszúrási anomália (insertion anomaly)

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 elemet 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.

Másrészt, ha a fenti példában Tóth István elkezd egy harmadik tételt is szállítani, de ennek az adatbázisba írásánál a lakcímére már nem a Bp. Fa u. 5.-öt adjuk meg, hanem pl. az új lakcímét, Bp. Rönk u. 17.-et, akkor elveszítjük Tóth István lakcímét, hiszen ezután már nem tudhatjuk, hogy mi a lakcíme valójában.

9.2.1.3. Törlési anomália (deletion anomaly)

Ha csak egy attribútum értékét szeretnénk törölni, akkor előfordulhat, hogy ez valamiért nem lehetséges (pl. mivel része valamely kulcsnak). A kérdéses attribútumértéktől ilyenkor úgy szabadulhatunk meg, ha az egész sort töröljük, de ilyenkor elveszíthetünk olyan adatokat/információkat is, amelyekre még szükségünk lehet.

Ha pl. a cement tételt akarjuk törölni, akkor az egész sort kell, de ekkor elveszítjük Nagy Géza címét is.

A fenti problémákat megoldhatja a relációk függőleges felbontása[3] (*vertikális dekompozíció*ja). Az építési anyag szállítók relációját például két részre célszerű felbontani:

[3] Az ún. sémafelbontást később, a 9.2.5. alszakaszban definiáljuk.

szállító

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

építőanyagok

SZÁLLÍTÓ	*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, az adataink mélyebb ismerete szükséges.

9.2.2. Adatbázis kényszerek (data constraints)

Adatbázis kényszerek alatt azokat a szabályokat értik, amelyek segítségével az adatbázisunk tartalmát olyan módon lehet jellemezni/korlátozni, hogy az valamely tervezésnek, ill. elképzelt/elvárt feltételeknek megfeleljen. A leggyakrabban használt kényszerek az alábbiak (vö. 5.4.5.1. alalszakasz):

- értékfüggő kényszerek (pl. $0 <$ TESTMAGASSÁG < 300)
- értékfüggetlen kényszerek
  - Tartalmazási függőség (pl. az idegen kulcsok értékeinek halmaza részhalmaza a neki megfeleltethető kulcsértékek halmazának)
  - Funkcionális függőség (ld. 9.2.3. alszakasz)
  - Többértékű függőség (ld. 9.2.8. alszakasz)

9.2.3. Funkcionális függőségek (functional dependencies)

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 ahhoz, hogy tudjuk, hol lakik a szállító.

Ez utóbbi tényt úgy is kifejezhetjük, hogy azt mondjuk: a *NÉV* attribútum értéke egyértelműen 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 – funkcionális függés *(functional dependency)*. Legyen adott az $R(A_1, A_2, \ldots, A_n)$ relációs séma, ahol A_i-k, $i = 1, 2, \ldots, n$ (alap)attribútumok. Legyen X és Y a reláció attribútumainak két részhalmaza: $X \subseteq R$ és $Y \subseteq R$. Ha *bármely*, az R sémára illeszkedő r reláció *bármely* két $t, t' \in r(R)$ sorára 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. Más megfogalmazásban azt is mondhatjuk, hogy az X értékei meghatározzák az Y értékeit. Mindezt így jelöljük: $X \to Y$.

A definíció szoros kapcsolatban van a jegyzetben már korábban is többször felbukkant implikáció műveletével (ld. A. függelék).

Példa. Vegyük a következő két állítást:

- A állítás: „az R sémára illeszkedő *bármely* r reláció *bármely* két t, t' sorára $t[X] = t'[X]$"
- B állítás: „$t[Y] = t'[Y]$"

Ekkor az $A \to B$ implikáció teljesülése esetén definíció szerint funkcionális függésről beszélünk az X és az Y attribútum(halmaz)ok között. Formálisan erre vezettük be az $X \to Y$ jelölést.

Megjegyzések.

1. A funkcionális függőség definíció szerinti teljesüléséhez nem követelmény, hogy egy R sémára illeszkedő bármely r reláción valóban legyen két olyan t, t' sor, amelyre fennáll, hogy $t[X] = t'[X]$ (azaz nem követeli meg, hogy az A állítás igaz legyen). Ha tehát egy konkrét R sémára illeszkedő r relációban nincsen ilyen két t, t' sor, az $X \to Y$ függőség akkor is fennállhat. Ekkor nyilván nem lehet ellenőrizni, hogy $t[Y] = t'[Y]$ igaz-e (azaz a B állítás igaz-e), miközben a funkcionális függés fennáll (ez a helyzet, ha pl. egy relációnak csak egyetlen eleme van). Ez összhangban van az implikáció definíciójával: az implikáció értéke akkor is lehet igaz, ha az A állítás nem igaz, azaz nincsenek az X attribútum(halmaz)on megegyező sorok.
2. Érdemes külön kitérni a definíciónak arra a feltételére is, hogy bármely $r(R)$-re (amit a gyakorlatban úgy is lefordíthatunk, hogy „bármely időpillanatban"). Gyakori az, hogy *egy adott* $r(R)$ 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, a rendszer *tervezésének* fázisában), az természetesen az érdemi függőségek segítségével történhet. Ugyanakkor egy működő rendszer *használata során*, az adatainak elemzéséhez az eseti függőségeknek lehet nagyobb szerepe. Az eseti függőségek összessége csak bővebb lehet, mint az érdemi függőségeké. *A továbbiakban, ha nem hangsúlyozzuk külön, akkor – a jelen jegyzet céljaival összhangban – csak érdemi függőségekről lesz szó.*

3. Az előzőekből az is következik, hogy a funkcionális függőségek meghatározása egyfajta modellezési kérdés. Ez azt is jelenti, hogy egy reláció egyetlen pillanatnyi állapotából sohasem dönthető el egy (érdemi) függőség fennállása, legfeljebb arra következtethetünk, hogy mely függőségek nem állnak fenn.
4. Láthatóan a funkcionális függések egy relációban csak akkor okoznak redundanciát, amennyiben valamely $X \to Y$ funkcionális függés mellett a relációnak valóban van legalább két olyan eleme, amelyek X-ben azonosak. Ha viszont az X (szuper)kulcs, akkor ez a feltétel garantáltan soha nem fog teljesülni (ld. Boyce–Codd normálforma, 9.2.4.5. alalszakasz), ill. természetesen az is előfordulhat, hogy az R sémára („véletlenül") csak olyan elemeket illesztünk, amelyek mellett az X értékek különbözőek.

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

- $CÍM, VÁROS \to IRÁNYÍTÓSZÁM$ (ha ismerjük a címet és a város nevét, akkor ehhez egyértelműen tartozik egy irányítószám)
- $IRÁNYÍTÓSZÁM \to VÁROS$ (egy irányítószámhoz csak egy város tartozik)
- $NÉV \to CÍM$
- $NÉV \to VÁROS$
- $NÉV \to IRÁNYÍTÓSZÁM$
- $NÉV \to TELEFON$ (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:

9.2.3.1. Determináns

Definíció – determináns *(determinant set)*. Ha $X, Y \subseteq R$ és $X \to Y$, de $\nexists X' \subset X$, hogy $X' \to Y$, akkor X-et Y *determinánsának* nevezzük.

Definíció – teljes függés *(full dependency)*. Ha $X, Y \subseteq R$ és $X \to Y$, de $\nexists X' \subset X$, hogy $X' \to Y$, akkor azt mondjuk, hogy Y *teljesen függ* (funkcionálisan) X-től.

Megjegyzés. Vegyük észre, hogy az imént definiált teljes függés és a determináns valójában ugyanannak a két megközelítése: ha $X, Y \subseteq R$ esetén X determinánsa Y-nak, akkor és csak akkor Y teljesen függ X-től.

Definíció – részleges függés *(partial dependency)*. Ha $X, Y \subseteq R$ és $X \to Y$ mellett $\exists X' \subset X$, hogy $X' \to Y$, akkor Y *részlegesen függ* X-től.

9.2.3.2. Relációs sémák kulcsai

Az ER-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 relációs sémán értelmezett kulcs matematikai definícióját is. Az előző szakaszban bevezetett jelöléseket alkalmazzuk.

Definíció – kulcs (relációs sémáé) *(key)*. X-et pontosan akkor nevezzük *kulcsnak* az R relációs sémán, ha

1. $X \to R$ és
2. $\nexists X' \subset X$, hogy $X' \to R$.

Más szavakkal akkor, ha R teljesen függ X-től.

9.2.3.2.1. Szuperkulcs, kulcs

Definíció – szuperkulcs *(superkey)*. X-et *szuperkulcs*nak nevezzük, ha igaz, hogy $X \to R$. Más szavakkal akkor, ha X tartalmaz kulcsot.

Néha hangsúlyozandó egy kulcs minimális voltát *minimális kulcs*ról beszélünk.

Ha egy kulcs csak egy attribútumból áll, akkor *egyszerű kulcs* (simple key), egyébként *összetett kulcs* (composite key) a neve.

Példa. Az építőanyagok (9.2. szakasz) relációs sémájának a $\{NÉV, TÉTEL\}$ attribútumhalmaz (összetett) kulcsa, ha az ottani funkcionális függőségeket értelmezzük a relációs sémán.

A jelen szakaszban definiált S relációs sémának a $\{NÉV\}$ (egyszerű) kulcsa. Egy szuperkulcsa lehet pl. a $\{NÉV, TELEFON\}$ attribútumhalmaz.

A kulcsok néhány fontos tulajdonsága:

1. a kulcs a relációnak egy és csakis egy elemét („sorát") határozza meg,
2. egy kulcs attribútumai nem lehetnek NULL-értékűek (azaz értékük meghatározatlan).

Tétel. Minden relációs sémának van kulcsa.

Bizonyítás. 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 relációs séma minden attribútumának értékét. Ha a második feltétel is teljesül, akkor kulcs, ha pedig nem, akkor szuperkulcs, tehát tartalmaz kulcsot.

9.2.3.2.2. Elsődleges kulcs

Definíció – elsődleges kulcs *(primary key)*. Ha X és Z az R relációs sémának egyaránt kulcsai, miközben $X \neq Z$, akkor az R relációs sémának több kulcsa is van.

Ezek közül kiválasztunk egyet, amelyet *elsődleges* kulcsnak (primary key) nevezünk. A többi kulcsot *kulcsjelöltnek* (candidate key) hívjuk.

9.2.3.2.3. Idegen (külső) kulcs

Definíció – idegen kulcs *(foreign key)*. Adott egy R és egy R' relációs séma. Tételezzük fel, hogy $R' \neq R$. Ha $\exists D \subseteq (R \cap R')$, hogy $D \to R'$ és D 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.

9.2.3.3. 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 az adatok egy valóságos helyzetbeli jelentését írjuk le funkcionális függőségekkel, de attól még fennáll. Ha egy R séma (akár egyelemű) attribútumhalmazai X és Y, akkor mindig fennállnak ezen a sémán az $X \cup Y \to Y$ vagy az $X \cup Y \to X$ függőségek is. Hiszen minden R sémára illeszkedő r relációban $\forall t, t' \in r(R)$ elemekre, ha $t[X \cup Y] = t'[X \cup Y]$, akkor $t[Y] = t'[Y]$, valamint $t[X] = t'[X]$ egyaránt törvényszerűen fennáll.

Jelölés: a továbbiakban az $X \cup Y$ helyett egyszerűen XY-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, ún. 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" minden „igaz" (funkcionális) függőséget, amelyek egy adott függőséghalmazban található függőségekkel nincsenek ellentmondásban.

Az „igaz"[4] és a „meg lehet kapni" kifejezések alatt pontosan az alábbiakat értjük:

Definíció – igaz (funkcionális függés) *(sound (functional dependency))*. Egy adott R sémán az attribútumain értelmezett F_R függéshalmaz mellett egy $X \to Y$ függőség pontosan akkor igaz, ha minden olyan $r(R)$ reláción fennáll, amelyeken F_R összes függősége is fennáll.

Jelölése: $F_R \vDash X \to Y$.

Definíció – *„meg lehet kapni", azaz* levezethető (funkcionális függés) *(deducible (functional dependency))*. Egy $W \to Z$ funkcionális függőség pontosan akkor vezethető le adott F_R függőségekből, ha az axiómák ismételt alkalmazásával F_R-ből kiindulva megkaphatjuk $W \to Z$-t.

Jelölése: $F_R \vdash W \to Z$.

Az imént definiált tulajdonságokkal bíró – valójában következtetési – szabályok *Armstrong „axiómái"* néven váltak ismertté.

[4] Eredetileg „sound", de a magyar szakirodalomban ennek az „igaz" fordítása terjedt el.

9.2.3.4. Armstrong axiómái a funkcionális függőségekről

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

a) Ha $X \subseteq Y$, akkor $Y \to X$ (reflexivitás vagy triviális függőség).
b) Ha $X \to Y$ és $Y \to Z$, akkor $X \to Z$ (tranzitivitás).
c) Ha $X \to Y$, akkor $XZ \to YZ$ (bővíthetőség).

Megjegyzés. Az axiómák tömören:

a) $X \subseteq Y \vDash Y \to X$
b) $X \to Y \wedge Y \to Z \vDash X \to Z$
c) $X \to Y \vDash XZ \to YZ$

Tétel – igazság tétel *(soundness theorem)*. Az Armstrong axiómák igazak (helyesek), alkalmazásukkal csak igaz függőségek állíthatók elő adott függéshalmazból.[a]

Formálisan: $F_R \vdash X \to Y \Rightarrow F_R \vDash X \to Y$

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 $X \to Y$ fennáll. Ez azt jelenti, hogy $\exists t, t'$ n-esek valamely $r(R)$ relációban, melyekre $t[XZ] = t'[XZ]$, de $t[YZ] \neq t'[YZ]$.

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 $t[Y] \neq t'[Y]$. De ez nem lehetséges, mert a kiindulási $X \to Y$ függőség miatt ha $t[X] = t'[X]$, akkor $t[Y] = t'[Y]$. Tehát ellentmondásra jutottunk *c)* tagadásával, így *c)* igaz kell, hogy legyen.

[a] Eredetileg: soundness theorem. Emiatt talán jobb lenne helyesség tételnek nevezni, de nem ez honosodott meg.

Tétel – teljesség tétel *(completeness theorem)*. Az Armstrong axiómák teljesek, azaz belőlük minden igaz függőség levezethető. Formálisan: $F_R \vDash X \to Y \Rightarrow F_R \vdash X \to Y$ (Nem bizonyítjuk.)

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

9.2.3.5. Az axiómák következményei

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

Fentiek most már az Armstrong axiómák felhasználásával is bizonyíthatóak, példaképpen nézzük meg *d)* bizonyítását!

Bizonyítás. $X \to Y \vDash X \to XY$ (*c)* miatt)

$X \to Z \vDash XY \to ZY$ (*c)* miatt)

$X \to XY$ és $XY \to ZY \vDash X \to ZY$ (*b)* miatt, ami éppen a bizonyítandó állítás.)

Példa. Vegyük elő újra az $R(NÉV, TÉTEL, CÍM, ÁR)$ sémát a rajta értelmezett $NÉV, TÉTEL \to ÁR$ és $NÉV \to CÍM$ függőségekkel! Keressük meg R kulcsát!

$NÉV, TÉTEL \to ÁR \vDash NÉV, TÉTEL \to ÁR, NÉV, TÉTEL$

$NÉV \to CÍM \vDash NÉV, TÉTEL \to CÍM, TÉTEL.$

Az egyesítési szabályt alkalmazva adódik, hogy $NÉV, TÉTEL \to ÁR, NÉV, TÉTEL, CÍM.$

Mivel $NÉV, TÉTEL$ együttesen meghatározza R mindegyik attribútumát, ezért $NÉV, TÉTEL$ együtt (szuper)kulcsot alkotnak. Láthatóan bármelyik attribútumot is hagyjuk el, már nem fogja R mindegyik attribútumát meghatározni, tehát a $\{NÉV, TÉTEL\}$ attribútumhalmaz minimális is, azaz kulcs.

9.2.3.6. Attribútumhalmaz lezártja

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 – attribútumhalmaz lezártja *(attribute closure)*. Az X *attribútumhalmaz lezárása* adott F függéshalmaz mellett az a legbővebb $W \subseteq R$ halmaz, amelyre az $X \to W$ függőség az adott F függéshalmaz mellett fennáll. Jelölése: $X^+(F)$.

Formálisan: $X^+(F) = \{A | A \in R \text{ és } F \vDash X \to A\}$

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

Algoritmus: Egy attribútumhalmaz lezárása csaknem lineáris időben meghatározható az alábbi algoritmus segítségével, melyet addig folytatunk, mígnem az $X^{(i)}$ ún. *lezárt-kezdemény* már nem nő tovább (azaz $X^{(i+1)} = X^{(i)}$):

$X^{(0)} = X$
$X^{(i+1)} = X^{(i)} \cup \{A | \exists V \subseteq X^{(i)}, V \to U \in F \text{ és } A \in U\}$

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

$X^{(0)} = AC$
$X^{(1)} = AC \cup \{B\}$ ($A \to B$ miatt)
$X^{(2)} = ACB \cup \{D\}$ ($B \to D$ miatt)

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

9.2.3.7. Függéshalmaz lezártja

Láttuk, hogy (funkcionális) függőségeket definiálva Armstrong axiómái segítségével továbbiakat is meg tudunk határozni, sőt mindegyiket, amelyik logikailag a megadottakból következik. Gyakran hasznos, ha ezekre a függőségekre közösen tudunk hivatkozni.

Definíció – függéshalmaz lezártja *(closure of functional dependency set)*. Az F *függéshalmaz lezárása* mindazon függőségek halmaza, amelyek az F függéshalmaz elemeiből az Armstrong axiómák alapján következnek.

Formálisan: $F^+ = \{X \to Y | F \vDash X \to Y\}$.

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

Tanulság. Könnyen belátható, hogy – az adott függőségek szerkezetétől függően – a lezárt halmaznak 2^n számú eleme is lehet (sőt, néha még több is), 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 $A \to B$ –, hogy következik-e adott F függőséghalmazból. A feladat direkt megoldása lehetne, hogy kiszámítjuk F^+-t és megvizsgáljuk, hogy elemei között van-e $A \to B$. Függéshalmaz lezártjának költséges számítása helyett gyorsabban eredményre jutunk, ha (lineáris időben!) A^+-t határozzuk meg a megismert algoritmussal. Amennyiben $B \subseteq A^+$, akkor $A \to B \in F^+$. Ez az attribútumhalmaz lezártja definíciójának közvetlen következménye.

Ha függőségeknek egy bonyolult rendszere adott, gyakran szeretnénk könnyebben áttekinthető, 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 ugyanazokat a függőségeket lehessen előállítani. A függéshalmaz lezárása segítségével lehetőségünk van arra, hogy egyszerű definíciót adjunk két függéshalmaz egyenlőségére.

Definíció – függéshalmazok ekvivalenciája *(equivalence of functional dependency sets)*. Két függéshalmaz pontosan akkor *ekvivalens*, ha lezártjaik megegyeznek.

Ezt így jelöljük: $F \equiv G \Leftrightarrow F^+ = G^+$ 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 $X \to Y \in F$ függőségre a tartalmazás hatékonyan eldönthető $X^+(G)$ kiszámításával: ha $Y \subseteq X^+(G)$, akkor $X \to Y \in G^+$. $G \subseteq F^+$ eldöntése nyilván azonos elven történhet.

Definíció – minimális függéshalmaz *(minimal set of functional dependencies)*. F *minimális függéshalmaz* (beszélünk F *minimális fedés*éről is) akkor, ha

1. a függőségek jobb oldalán csak egyetlen attribútum van,
2. a függőségek bal oldaláról nem hagyható el attribútum,
3. nincs olyan függőség, amely elhagyható.

Tétel. Adott függéshalmazzal ekvivalens minimális függéshalmaz mindig előállítható.

Bizonyítás. Konstruktív, ami egyben algoritmust is ad a minimális függéshalmaz előállítására.

Adott egy F függéshalmaz.

1. A felbontási szabály alapján minden $X \to Y \in F$ függőség helyettesíthető $X \to A_1, X \to A_2, \ldots, X \to A_n$ függőségekkel (ahol $Y = \{A_1, A_2, \ldots, A_n\}$), így egy F' függéshalmazt kapunk. Nyilvánvaló, hogy F és F' ekvivalensek.
2. Minimalizáljuk a függőségek bal oldalát. Ehhez vizsgáljuk meg $\forall S \to B \in F'$ függőségre, ahol $S = \{D_1, D_2, \ldots, D_n\}$, hogy elhagyható-e valamely D_i attribútum. Definíció szerint ehhez az kell, hogy $(F'')^+ = (F')^+$ fennálljon

(ahol $F'' = F' \setminus \{S \to B\} \cup \{\{D_1, D_2, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n\} \to B\}$), ami ekvivalens az $F' \subseteq (F'')^+$ és $F'' \subseteq (F')^+$ egyidejű fennállásával. A függéshalmazok csak egyetlen függésben különböznek:

- $F' = \{X_1 \to A_1, X_2 \to A_2, \ldots, \{D_1, D_2, \ldots, D_{i-1}, D_i, D_{i+1}, \ldots, D_n\} \to B\}$
- $F'' = \{X_1 \to A_1, X_2 \to A_2, \ldots, \{D_1, D_2, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n\} \to B\}$

$F' \subseteq (F'')^+$ ezért ekvivalens $B \in S^+(F'')$, vagyis

$$B \in S^+(F' \setminus \{S \to B\} \cup \{\{D_1, D_2, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n\} \to B\})$$

vizsgálatával, $F'' \subseteq (F')^+$ pedig

$$B \in (D_1, D_2, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n)^+(F')$$

vizsgálatával. Mivel

$$B \in S^+(F' \setminus \{S \to B\} \cup \{\{D_1, D_2, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n\} \to B\})$$

triviálisan teljesül $S = \{D_1, D_2, \ldots, D_n\}$ miatt, elegendő csupán a másik irányt megvizsgálni. Eredményül egy F'' függéshalmazt kapunk, amely továbbra is ekvivalens F-fel.

3. Vizsgáljuk meg $\forall T \to C \in F''$ függésre, hogy elhagyható-e. Definíció szerint ehhez az kell, hogy $(F'' \setminus \{T \to C\})^+ = (F'')^+$ fennálljon, aminek az eldöntése költséges. Célszerűbb ezért azt vizsgálni, hogy $C \in T^+(F'' \setminus \{T \to C\})$ vajon teljesül-e – amint már korábban beláttuk, ez ekvivalens, ugyanakkor hatékonyabb. Végül egy olyan F^* függéshalmazt kapunk, amely továbbra is ekvivalens F-fel, de a minimális függéshalmaz mindegyik tulajdonságának megfelel.

Megjegyzés. Vegyük észre, hogy a 2. és 3. lépésekben az egyes attribútumok, ill. függőségek 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: $F = \{AB \to CD, AC \to BD, C \to AB\}$, mi egy minimális fedése?

1. $F' = \{AB \to C, AB \to D, AC \to B, AC \to D, C \to A, C \to B\}$
2. $C \to A$ miatt az $AC \to B$ függőség $C \to B$-re, $AC \to D$, $C \to D$ -re redukálható: $F'' = \{AB \to C, AB \to D, C \to B, C \to D, C \to A\}$. (A függőségek száma azért csökkent, mert $C \to B$ szerepelt az eredeti függéshalmazban is.)

3. $AB \to C$ és $C \to D$ miatt (a tranzitivitási axióma következtében) $AB \to D$ elhagyható. A kapott függéshalmaz minimális:
   $F_1^* = \{AB \to C, C \to D, C \to A, C \to B\}$.
   A minimális fedés nem egyértelmű, mert F''-ből $C \to A, C \to B$ és $AB \to D$ miatt $C \to D$ elhagyható, ekkor
   $F_2^* = \{AB \to C, AB \to D, C \to B, C \to A\}$.

9.2.4. Relációs sémák normálformái

Ahhoz, hogy a 9.2.1. alszakaszban ismertetett anomáliákat elkerülhessük, a relációink sémái meghatározott feltételeket kell, hogy teljesítsenek. Ezeket a feltételeket *normálformák*nak nevezik (Codd, 1970). A normálformák tehát megszorítások a relációs séma tulajdonságaira vonatkozóan annak érdekében, hogy a sémákra illeszkedő relációkkal végzett műveletek során egyes nemkívánatos jelenségeket elkerülhessünk.

9.2.4.1. A nulladik normálforma (0NF)

Ilyen alakúnak tekintünk minden olyan 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. Nulladik normálforma (0NF)-ben van az a relációs séma, amelyik pl. ismétlődő csoportot tartalmaz az attribútumai között.

Példa. Adottak az alábbi attribútumok:

- *EGYETEM_NÉV*
- *REKTOR*
  - *KAR*
  - *DÉKÁN*
    - *TANSZÉK*
    - *VEZETŐ*

Ekkor az alábbi séma 0NF alakú:

$$UNI(EGYETEM\_NÉV, REKTOR(KAR, DÉKÁN(TANSZÉK, VEZETŐ)*)*)$$

A $*$ jellel az ismétlődő csoportokat jelöltük.

9.2.4.2. Az első normálforma (1NF)

Definíció – *1NF*. Egy relációs séma 1NF alakú (vagy más szóval *normalizált*, normalized), ha csak atomi attribútum-értékek szerepelnek benne.

Megjegyzés. Az 1NF alak definíciója nyilvánvalóan nem a redundancia csökkentést szolgálja. Egyszerűen csak kiindulási alapot teremt a további normalizálás számára. Hacsak nem hangsúlyozzuk külön az ellenkezőjét, a továbbiakban mindig feltételezni fogjuk, hogy a sémáink normalizáltak a fenti értelemben.

9.2.4.3. A második normálforma (2NF)

Definíció – *elsődleges és másodlagos attribútumok.* Egy R relációs séma $A \in R$ attribútuma *elsődleges attribútum* (primary attribute), ha A eleme a séma valamely K kulcsának. Egyébként A *másodlagos attribútum* (secondary attribute).

Ezek szerint egy relációs séma kulcsai az attribútumokat két diszjunkt halmazba sorolják: ha R kulcsainak halmaza $\{K_1, K_2, \ldots, K_n\}$, ahol $K_i \subseteq R$, akkor $\cup_i K_i$ az elsődleges attribútumok, $R \setminus \cup_i K_i$ pedig a másodlagos attribútumok halmaza.

Definíció – *2NF.* Egy 1NF relációs séma 2NF alakú, ha benne minden másodlagos attribútum a séma bármely kulcsától teljesen függ.

Más szavakkal: másodlagos attribútum nem függ egyetlen kulcs egyetlen valódi részhalmazától (részkulcstól) sem.

A 2NF definíciójának célja már nyilvánvalóan a redundanciacsökkentés. Ha ugyanis megsértjük a 2NF definícióját, és lehetővé tesszük másodlagos attribútumnak részkulcstól való függését is, akkor ebben a másodlagos attribútumban az attribútumértékek redundáns tárolása megvalósulhat, amint azt az alábbi példa szemlélteti.

Példa. Az $R(A, B, C, D)$ atomi attribútumokat tartalmazó séma az $F = \{AB \to C, B \to D\}$ függéshalmaz mellett 1NF-ben van, ugyanis egyetlen kulcsa AB, elsődleges attribútumai A és B, másodlagos attribútumai C és D. A D attribútum a $B \to D$ függés miatt az AB kulcs részétől is függ, tehát nem lehet 2NF-ben. Az (R, F) sémára illeszkedik pl. a

A	B	C	D
a1	b1	c1	d1
a2	b1	c2	d1

reláció, amely a D attribútumban redundáns tárolást valósít meg a $B \to D$ függés következtében.

A definíció közvetlen következményei:

– Ha minden kulcs egyszerű, akkor 1NF $\Rightarrow$ 2NF.

– Ha nincsenek másodlagos attribútumok, akkor 1NF ⇒ 2NF.

Tétel. Minden 1NF relációs séma felbontható 2NF sémákba úgy, hogy azok felhasználásával az 1NF sémára illesztett, „eredeti" relációk helyreállíthatók (torzulás nélkül, ld. a veszteségmentes sémafelbontásról szóló 9.2.5. alszakaszt).

Példa. Az 5.2.9. alszakaszban ismertetett feladat relációs sémáinak megtervezéséhez a *NAPI_HELYZET* sémából indulunk ki, amely minden attribútumot tartalmaz:

NAPI_HELYZET(*DÁTUM*, *ÖSSZEG*, *ÁRUNÉV*, *DB*, *ÁRUKÓD*, *EGYSÉGÁR*, *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*, *EGYSÉGÁR* (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*, *EGYSÉGÁR*

A *NAPI_HELYZET* séma 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 sémára az alábbiak szerint:

– *ÁRU*(*ÁRUKÓD*, *ÁRUNÉV*, *EGYSÉGÁR*)
– *MENNYISÉG*(*DÁTUM*, *ÁRUKÓD*, *DB*)
– *BEVÉTEL*(*DÁTUM*, *ÖSSZEG*, *BEFIZ*)

Ahhoz, hogy ezen sémák normálformáiról mondhassunk valamit, szükségünk van a

sémákon értelmezett függőségekre. (Ehhez részletesen ld. a vetített függőségekről szóló részt a 9.2.6. alszakaszban). Itt csak a végeredményt közöljük:

- $F_{ÁRU} = \{ÁRUKÓD \to ÁRUNÉV, EGYSÉGÁR\}$
- $F_{MENNYISÉG} = \{DÁTUM, ÁRUKÓD \to DB\}$
- $F_{BEVÉTEL} = \{DÁTUM \to \{ÖSSZEG, BEFIZ\}, ÖSSZEG \to BEFIZ, BEFIZ \to ÖSSZEG\}$

Könnyen belátható, hogy most már mindhárom relációs séma 2NF-ben van: az első és utolsó azért, mert kulcsuk egyszerű kulcs, *MENNYISÉG* pedig azért, mert *DB*-t az összetett kulcs egyik valódi része sem határozza meg.

9.2.4.4. A harmadik normálforma (3NF)

A definícióhoz szükségünk lesz néhány új fogalomra.

Definíció – triviális függés *(trivial functional dependency)*. Ha az X, Y attribútumhalmazokra igaz, hogy $Y \subseteq X$, akkor az $X \to Y$ függőséget *triviális függőségnek* nevezzük, egyébként a függőség *nemtriviális*.

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

Példa. Adott $R(Járat, Dátum, Pilótakód, Név) = R(J, D, P, N)$, $F = \{JD \to P, P \to N, N \to P\}$.

Az N attribútum tranzitívan függ JD-től, mert $JD \to P, P \not\to JD, P \to N$ és nyilván $N \notin P$ az egyszerű attribútumok miatt.

Definíció – *3NF, definíció 1.* Egy 1NF R séma 3NF, ha $\forall A \in R$ másodlagos attribútum és $\forall X \subseteq R$ kulcs esetén $\nexists Y$, hogy $X \to Y$, $Y \not\to X$, $Y \to A$ és $A \notin Y$.

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

Definíció – *3NF, definíció 2.* Egy 1NF R séma 3NF, ha $\forall X \to A$, $X \subseteq R$, $A \in R$ nemtriviális függőség esetén

- X szuperkulcs vagy
- A elsődleges attribútum.

A 3NF első definíciója szemléletesebb, ha a redundanciacsökkentést tartjuk szem előtt. Felesleges ugyanis 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 értékekre is. Ilyenkor az $Y \to A$ függőségnek megfelelő (y, a) értékeket többszörösen tároljuk, nyilvánvaló redundanciát „építve" a relációba.

Egyes szerzők jobban kedvelik a 3NF második definícióját, talán azért, mert gyakran könnyebben ellenőrizhető ebben a formában, hogy egy séma teljesíti-e a 3NF kritériumait. A definíció szemléletes tartalma ugyanakkor ezesetben már nem nyilvánvaló.

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

Bizonyítás. Előre: Def. 1. $\Rightarrow$ Def. 2.

Indirekt: Def. 1. feltételei mellett t. f. h. $\exists Z \to B$ nemtriviális függőség, ahol Z nem szuperkulcs, és B nem elsődleges attribútum.

Viszont minden relációs sémának létezik kulcsa, legyen ez X. Igaz tehát, hogy $X \to Z, Z \not\to X$ (különben Z szuperkulcs lenne), $Z \to B, B \notin Z$ (különben $Z \to B$ triviális függőség lenne). Ez pedig éppen egy másodlagos attribútum 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 $\exists A$ másodlagos attribútum, hogy $X \to Y, Y \not\to X, Y \to A$ és $A \notin Y$. $X \to Y$, mivel X kulcs, ezért nincs ellentmondásban Def. 2-vel, $Y \not\to X$, tehát Y nem lehet szuperkulcs, $Y \to A$ és $A \notin Y$ miatt tehát létezik egy nemtriviális függőség, melyben Y nem szuperkulcs, A nem elsődleges attribútum, ellentmondásban Def. 2. feltételeivel.

Tehát Def. 2. $\Rightarrow$ Def. 1.

Megjegyzés. A definíció szerint minden F^+-beli függőségen ellenőrizni kellene a feltételt.

Tétel. Ahhoz, hogy egy (R, F) sémáról (Def. 2. alkalmazásával) eldöntsük, hogy 3NF-e, elég az F-beli funkcionális függőségek vizsgálata. (Nem bizonyítjuk.)

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

Bizonyítás. Ld. 9.2.7. alszakasz: veszteségmentes dekompozíció 3NF-be.

Példa. Az előző szakasz *ÁRU*, *MENNYISÉG* és *BEVÉTEL* relációs sémáit vizsgáljuk meg, hogy teljesítik-e a 3NF kritériumait:

Az $\acute{A}RU(\acute{A}RUK\acute{O}D, \acute{A}RUN\acute{E}V, EGYS\acute{E}G\acute{A}R)$ séma nemtriviális függősége csupán $\acute{A}RUK\acute{O}D \rightarrow \acute{A}RUN\acute{E}V, EGYS\acute{E}G\acute{A}R$, tehát *ÁRUKÓD* kulcs, így *ÁRU* 3NF alakú.

A $MENNYIS\acute{E}G(D\acute{A}TUM, \acute{A}RUK\acute{O}D, DB)$ séma egyetlen nemtriviális függősége $D\acute{A}TUM, \acute{A}RUK\acute{O}D \rightarrow DB$, tehát $(D\acute{A}TUM, \acute{A}RUK\acute{O}D)$ kulcs, így *MENNYISÉG* is 3NF alakú.

A $BEV\acute{E}TEL(D\acute{A}TUM, \ddot{O}SSZEG, BEFIZ)$ séma nemtriviális függőségei:

- $D\acute{A}TUM \rightarrow \ddot{O}SSZEG, BEFIZ$
- $\ddot{O}SSZEG \rightarrow BEFIZ$
- $BEFIZ \rightarrow \ddot{O}SSZEG$

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:

- $BEV\acute{E}T(D\acute{A}TUM, \ddot{O}SSZEG)$
- $BEFIZ(D\acute{A}TUM, BEFIZET\acute{E}S)$

Könnyen belátható, hogy mindkettő 3NF alakú[5].

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

Megjegyzés. Az A. függelékben definiált implikáció műveletével ezt így is megfogalmazhatjuk: 3NF(R) $\rightarrow$ 2NF(R).

Bizonyítás. Indirekt. T. f. h. az R séma nem 2NF, ekkor ellentmondásra kell jutnunk a 3NF definíciójával (azaz $\neg$2NF(R) $\rightarrow \neg$3NF(R)). 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 lenne minimális,

[5] Noha ez egy jó megoldás, de a felbontás nem függőségőrző (ld. később). Ha viszont az egyik sémát *ÖSSZEG, BEFIZETÉS*-re cseréljük, akkor függőségőrző is lesz.

tehát nem lehetne kulcs. Továbbá $A \notin K$, mert A másodlagos attribútum, tehát egyetlen kulcsnak sem lehet eleme. Tehát: $K \to K'$, $K' \not\to K$, $K' \to A$, $A \notin K'$, ami éppen azt jelenti, hogy az A attribútum tranzitívan függ a K kulcstól, ellentmondásban a 3NF definíciójával.

A bizonyításból az is következik, hogy egy másodlagos attribútum részleges függése kulcstól egyúttal a másodlagos attribútum tranzitív függését is jelenti a kulcstól.

9.2.4.5. A Boyce–Codd normálforma (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 normálforma, a *Boyce–Codd normálforma (BCNF)* bevezetése. Be fogjuk látni, hogy a BCNF sémákra illeszkedő relációk már mentesek a funkcionális függés-alapú redundanciától.

Definíció – *BCNF, definíció 1.* Egy 1NF R séma BCNF, ha $\forall A \in R$ attribútum és $\forall X \subseteq R$ kulcs esetén $\nexists Y$, hogy $X \to Y$, $Y \not\to X$, $Y \to A$ és $A \notin Y$.

Szavakkal: egyáltalán nincs tranzitív függőség kulcstól.

Definíció – *BCNF, definíció 2.* Egy 1NF R séma BCNF, ha $\forall X \to A, X \subseteq R, A \in R$ 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.

Előre: Def. 1. $\Rightarrow$ Def. 2.

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

Viszont minden relációs sémának létezik kulcsa, legyen X ezek közül egy. Igaz tehát, hogy $X \to Z, Z \not\to X, Z \to B, B \notin Z$. 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 $X \to Y, Y \not\to X, Y \to A$ és $A \notin Y$.

$X \to Y$: mivel X kulcs, ezért nincs ellentmondásban Def. 2-vel, $Y \not\to X$, tehát Y nem lehet szuperkulcs,

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

Tehát Def. 2. $\Rightarrow$ Def. 1.

Tétel. Ahhoz, hogy egy (R, F) sémáról (Def. 2. alkalmazásával) eldöntsük, hogy BCNF-e, elég az F-beli funkcionális függőségek vizsgálata. (Nem bizonyítjuk.)

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 funkcionális függőségek következtében).

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

Bizonyítás. Indirekt. T. f. h. a séma BCNF, de mégis van egy rá illeszkedő relációban redundancia. Ez azt jelenti, hogy van 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, 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 $Z \to A$ függőségnek, ahol nyilván $Z \subseteq X$. Viszont Z nem lehet szuperkulcs, mert akkor a t és t' soroknak azonosaknak

kellene lenniük, ami ellentmond annak a feltételezésnek, hogy $y1 \neq y2$. Ha pedig Z nem szuperkulcs, akkor a $Z \to A$ függőség ellentmond a BCNF definíciójának.

A normálforma definíciókból és a viszonyukból az következik, hogy egy reláció redundanciáját (funkcionális függések következtében) az okozhatja, ha az attribútumai tranzitívan függenek a kulcs(ok)tól. A 3NF a másodlagos attribútumok tranzitív (és egyúttal részleges) függését zárja ki, a BCNF pedig az elsődleges attribútumokét is.

Ez és az előző tétel tehát azt sugallja, hogy a relációs adatbázisok tervezése során célszerű BCNF sémákat kialakítani. Hiszen ekkor – ha az attribútumaink között csak funkcionális függőségekkel leírható függőségi kapcsolatok vannak – a relációink redundanciamentesek lesznek, ami lényegesen megkönnyíti az egyes relációk (táblák) tartalmának módosítását végző alkalmazások megírását, ill. a hatékonyság biztosítását. Tegyük fel ugyanis, hogy nem bízhatunk a redundanciamentességben: ekkor minden egyes érték bevitele előtt ellenőrizni kell(ene), hogy a relációban már meglévő elemek és az új elem együttesen nincs-e ellentmondásban valamely ismert kényszerrel (jelen esetben funkcionális függőséggel). Az ellenőrzés ugyan elvégeztethető egy adatbázis alkalmazással, kliens oldalon vagy magával az adatbázis-kezelő rendszerrel is, de bármelyik is igen költséges lehet, különösen, ahogy egyre nagyobb mennyiségű adatot szeretnénk kezelni. 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 ún. tranzakciókezelő (manapság leginkább *on-line tranzakciókezelő*, OLTP) rendszereknél.[6] Ennek tipikus példái a repülőtéri helyfoglaló rendszer, banki átutalásokat végző rendszerek vagy éppen egy hallgatói tanulmányi rendszer (pl. Neptun).

A valóság ezzel szemben az, hogy még további szempontok is léteznek a relációs sématervezésnél és a relációs adatbázisok üzemeltetésénél, amelyeket – bár még nem ismerünk – majd figyelembe kell vennünk. 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 normálformában található. Nem meglepő módon:

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

[6] Az OLTP rendszerekkel szemben az ún. döntéstámogató rendszerekre (decision support system, DSS) viszont a ritkán módosuló adatok, nagy tömegű, bonyolult és változatos lekérdezések jellemzőek. Ilyen esetekben a lekérdezés sebességére kell optimalizálni, és emiatt kifejezetten ellenjavallt lehet a normalizált sémák alkalmazása. Ld. analitikus alkalmazások, dimenziós modellezés, adattárházak.

9.2.5. Veszteségmentes sémafelbontás (lossless schema decomposition)

Definíció – sémafelbontás *(schema decomposition)*. Egy R relációs sémának $\rho(R_1, R_2, \ldots, R_n)$ egy *felbontás*a, ha $R_1 \cup R_2 \cup \ldots \cup R_n = R$.

A felbontás célja általában az, hogy ezáltal a részsémák valamely magasabb normálformába 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 részrelációkból (azaz a részsémákra vetített relációkból) az eredeti relációt – és ezzel együtt annak eredeti információtartalmát – helyreállítani. A helyreállítás itt a részrelációk 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: $C \to A$. Vizsgáljuk meg a $\rho_1(AB, BC)$ és a $\rho_2(AC, BC)$ 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 $R' \neq R$, miközben $R'' = R$. Tehát az R' -höz tartozó ρ_1 felbontásnak gyakorlati haszna aligha van, hiszen a felbontás 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. Utóbbi esetben ρ_2-re azt mondjuk, hogy *veszteségmentes*.

Tanulság. Amikor relációs adatbázis sémákat tervezünk, nem elegendő csupán a minél magasabb normálformára, azaz minél kevesebb redundanciára törekedni. Egyidejűleg a veszteségmentesség biztosítása feltétlen követelmény, hiszen semmit sem ér egy kevés redundanciát tartalmazó adatbázis, amelyből a modellezett valósággal nem összhangban levő, ill. annak ellentmondó adatokat/információt is ki lehet olvasni!

Definíció – project-join mapping *(project-join mapping)*. Legyen az R relációs séma ρ felbontása mellett $m_\rho(r) \equiv \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \ldots \bowtie \pi_{R_n}(r)$.

Definíció – veszteségmentes sémafelbontás *(lossless schema decomposition)*. Az R relációs séma egy $\rho(R_1, R_2, \ldots, R_n)$ *felbontását veszteségmentesnek* mondjuk, ha $\forall r(R)\ m_\rho(r) = r$.

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

Bizonyítás. Vegyünk egy tetszőleges $t \in r$ sort. Képezzük t vetületeit az R_i részsémákra, legyen ez $t\,[R_i]$, amely nyilván eleme az i-edik részrelációnak. Ez a vetület nem változik $m_\rho(r)$ -ben sem, és a természetes illesztés tulajdonságai miatt $m_\rho(r)$ valamelyik sorában mindegyik $t\,[R_i]$, $i = \{1, \ldots, n\}$ meg fog jelenni. Így $t \in m_\rho(r)$.

A tétel állítása más szavakkal: tetszőleges (tehát nemcsak veszteségmentes!) felbontás után a természetes illesztés eredményeképpen sorok nem tűnhetnek el, csak újak keletkezhetnek.

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

Bizonyítás. A természetes illesztés asszociativitását kihasználva triviális.

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

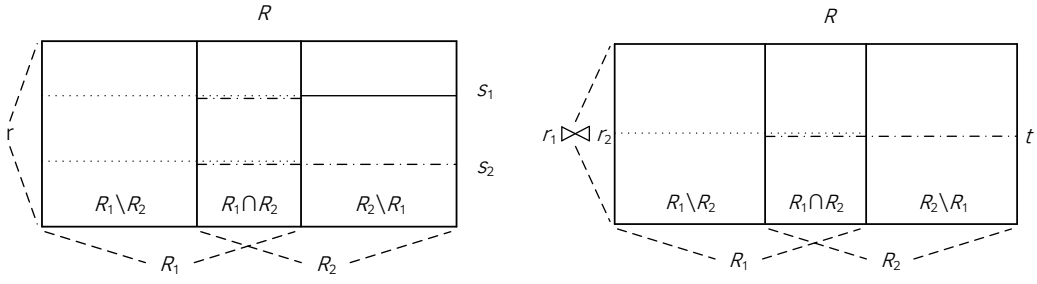
Bizonyítás. Ismét a természetes illesztés asszociativitását használjuk fel. τ veszteségmentes, ha $\forall r(R)$ relációra $r = m_\tau(r)$. $m_\tau(r) = m_\rho(r) \bowtie \pi_{R_k}(r) \bowtie \pi_{R_l}(r) \bowtie \ldots \bowtie \pi_{Rm}(r)$, ahol $R_k, R_l, \ldots, R_m \in \tau$, de $\notin \rho$. Készítsük 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 $m_\rho(r) \bowtie \pi_{R_i}(r) \equiv r \bowtie \pi_{R_i}(r) \equiv r$. Emiatt $m_\rho(r) \bowtie \pi_{R_k}(r) \bowtie \pi_{R_l}(r) \bowtie \ldots \bowtie \pi_{Rm}(r) = r$, tehát $m_\tau(r) = r$, azaz τ veszteségmentes.

Tudjuk most már, 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 –, sématervezéskor azonban ez a megközelítés nyilván használhatatlan, hiszen tervezési időben nem ismerhetjük a sémára a jövőben illeszkedő példányokat.

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 $\Leftrightarrow (R_1 \cap R_2) \to (R_1 \setminus R_2) \in F^+$ vagy $(R_1 \cap R_2) \to (R_2 \setminus R_1) \in F^+$.

Bizonyítás. Visszafelé: Azt akarjuk belátni $(R_1 \cap R_2) \to (R_1 \setminus R_2) \in F^+$ vagy $(R_1 \cap R_2) \to (R_2 \setminus R_1) \in F^+$ 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 $s_2[R_2] = t[R_2]$ (szaggatott vonal).

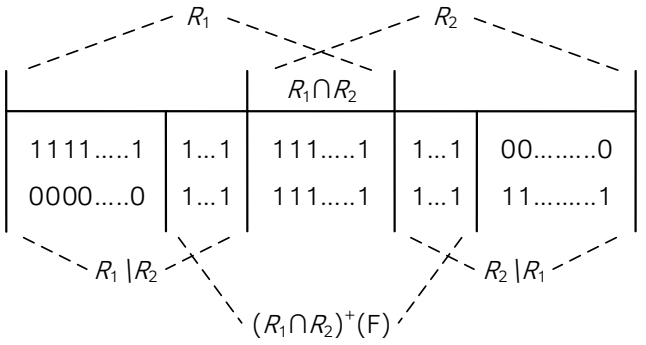
Tudjuk továbbá, hogy $t[R_1 \cap R_2] = s_1[R_1 \cap R_2] = s_2[R_1 \cap R_2]$.

T. f. h. a kétféle lehetőség közül $(R_1 \cap R_2) \to (R_1 \setminus R_2)$ áll fenn, amely szerint, ha

$s_1[R_1 \cap R_2] = s_2[R_1 \cap R_2]$, akkor kötelezően $s_1[R_1 \setminus R_2] = s_2[R_1 \setminus R_2]$. 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 $(R_1 \cap R_2) \to (R_1 \setminus R_2) \in F^+$ sem $(R_1 \cap R_2) \to (R_2 \setminus R_1) \in F^+$ 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 $(R_1 \cap R_2) \to (R_1 \setminus R_2)$ sem $(R_1 \cap R_2) \to (R_2 \setminus R_1)$ nem áll fenn. Azt kell még belátni, hogy kielégíti F minden függőségét. Valamely $V \to W \in F$-re:

- ha V „kilóg” $(R_1 \cap R_2)^+$-ből, akkor nincs két azonos sor, tehát $V \to W$ teljesül bármely W-re.
- ha $V \subseteq (R_1 \cap R_2)^+$, akkor $(R_1 \cap R_2) \to V$ és $V \to W$ miatt $(R_1 \cap R_2) \to W$, melynek feltétele, hogy $W \subseteq (R_1 \cap R_2)^+$. Így W értékei is azonosak, tehát $V \to W$ ekkor is teljesül.

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

Példa. Tekintsük újra a 9.2.5. alszakaszban szereplő $R(A, B, C)$ sémát, amelynek F függéshalmaza csak a $C \to A$ függőséget tartalmazza. Vizsgáljuk meg rendre a

$\rho_1(AB, BC)$ és a $\rho_2(AC, BC)$ felbontásokat a fenti tétel alapján!

ρ_1:

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

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

Tehát ρ_1 veszteséges.

ρ_2:

$AC \cap BC = C, AC \setminus BC = A, C \to A \in F^+$ (nyilvánvaló), tehát ρ_2 veszteségmentes.

$(AC \cap BC = C, BC \setminus AC = B, C \to B \notin F^+)$

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

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

Könnyen ellenőrizhető, hogy ekkor az $(R_1 \cap R_2) \to (R_1 \setminus R_2)$ pontosan $X \to Y$ 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, a $\rho(XY, XZ)$ felbontás pontosan akkor veszteségmentes, ha $X \to Y \in F^+$ vagy $X \to Z \in F^+$.

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 $R(A_1, A_2, \ldots, A_k)$ séma, a sémán értelmezett F függőségek és R-nek egy $\rho(R_1, R_2, \ldots, R_n)$ 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 $X \to Y \in F$ 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_l-t tegyük egyenlővé tetszőlegesen.

Példa. Legyen $R(S, A, I, P)$, $F = \{S \to A, SI \to P, P \to S\}$, $\rho(SA, SI, IP, PS)$.

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 $\Leftrightarrow$ van csupa a-ból álló sor.

Bizonyítás. Előre:

A táblázatot tekintsük olyan $r(R)$ relációnak, ahol a-kat és b_i-ket DOM(A_j)-ből választottuk. A végállapotban r kielégíti F összes 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 $m_\rho(r) \equiv \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \ldots \bowtie \pi_{R_n}(r)$, 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 végállapotbeli táblázat $\equiv m_\rho(r)$, azaz tartalmazza a csupa a-ból álló sort.

Visszafelé: nem bizonyítjuk.

9.2.6. Függőségőrző felbontások

Vizsgáljuk meg a következő példát! Adott az $R(VÁROS, ÚT, IR\_SZÁM) = R(V, U, I)$ séma, amelyen – a valósággal jó összhangban – az $F = \{VU \to I, I \to V\}$ függőségeket értelmeztük. Készítsük el a séma $\rho(UI, VI)$ felbontását! (Könnyen ellenőrizhető az előző tétel segítségével, hogy ρ veszteségmentes, tehát tetszőleges, (R, F)-re illeszkedő r reláció esetén r helyreállítható természetes illesztéssel a részsémákra vonatkozó vetületeiből.)

Példa. Egy adatrögzítő az ennek alapján készített adatbázisba 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:

$r_1 \bowtie r_2$

ÚT	*IR_SZÁM*	*VÁROS*
Kossuth	2142	Baja
Kossuth	2144	Baja

Az eredménnyel az a probléma, hogy nyilvánvalóan nincs összhangban a feltételezett $VU \to I$ függőséggel, amely szerint a valóságunk „működik", tehát azzal, hogy egy városban egy utcának csak egyetlen irányítószáma lehet. A részrelációk természetes illesztése „hamis" eredményre vezetett. A hiba azonban nem az illesztésnél van, hiszen a felbontás veszteségmentes, tehát az „eredeti" relációt kell visszakapnunk. A valódi ok az, hogy *nem is létezett* „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.

Egy „jó" felbontás esetén biztosak akarunk lenni abban, hogy ha a részrelációkba csak szemantikailag helyes (adott esetben tehát a felvett funkcionális függéseknek megfelelő) adatok kerülnek bele, akkor a relációk valamely illesztése eredményeként sem fogunk majd az adatbázisból olyan sorokat kiolvasni, amelyek a felvett – és az adatok jelentését valamilyen szinten leíró – funkcionális függéseknek ellentmondanak.

Másrészt mindaddig, amíg a séma egyben van, minden egyes sor bevitele előtt elvileg lehetőség van széleskörűen ellenőrizni, hogy a bevitt adatok bizonyos *kényszerfeltételeknek* (integrity constraints) megfelelnek-e. Ilyenek lehetnek pl. meghatározott függőségi feltételek is. Sajnos, amint az R sémát felbontottuk, már nem feltétlenül tudjuk az eredeti függőségeket alkalmazni a részrelációkban annak elkerülésére, hogy „hamis" adatok kerüljenek be az adatbázisba, legfeljebb a függőségeknek a részsémákra való „vetületeit". Ez egyes esetekben elegendő lehet (ezt részesítjük előnyben), más esetekben pedig nem.

Definíció – vetített függéshalmaz *(projected dependency set).* 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 $\pi_Z(F) = \{X \to Y | X \to Y \in F^+ \text{ és } XY \subseteq Z\}$.

Fontos, hogy a vetített függőséghalmazban benne legyen minden olyan függés, amely F-ből következik, ill. levezethető.

Példa. Egy $R(A, B, C)$ sémán értelmezett $F = \{A \to B, B \to C\}$ függőséghalmaz esetén $\pi_{AC}(F) = \{A \to C\}$, mert a tranzitivitási axióma miatt $A \to C \in F^+$.

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

Definíció – függőségőrző felbontás *(dependency-preserving decomposition).* Adott egy R relációs séma és egy, a sémán értelmezett F függéshalmaz. A séma $\rho = \{R_1, R_2, \ldots, R_k\}$ felbontása *függőségőrző*, ha $\bigcup_{i=1}^{k} \pi_{R_i}(F) \vDash F$.

Egy relációs séma veszteségmentes, de nem függőségőrző felbontása eredményezheti tehát, hogy a részsémák összességén sem tudjuk többé az eredeti sémában érvényes mindegyik függőséget alkalmazni. Ennek következtében a részrelációink illesztésével kapott lekérdezési eredményekbe 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 függőségőrzők. Ebben az esetben ugyanis ha a részrelációkba bevitt adatok valóban megfelelnek a tervezett funkcionális függéseknek (pl. nincs téves adatbevitel), akkor garantálható, hogy a lekérdezések eredményei sem fognak a függéseknek (tehát az adatszemantikának) ellentmondani.

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

1. elkészítjük F^+-t,
2. vetítjük F^+ mindegyik függőségét minden részsémára,
3. meghatározzuk a vetített függőséghalmazok uniójának lezárását,
4. ha ez azonos F^+-tal, akkor a felbontás függőségőrző, ellenkező esetben biztosan nem az.

Megjegyzés. Egy felbontás lehet

- veszteségmentes és függőségőrző,
- veszteségmentes és nem függőségőrző,
- veszteséges és függőségőrző,
- veszteséges és nem függőségőrző.

9.2.7. 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 normálforma 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 $G = \{X_1 \to A_1, X_2 \to A_2, \ldots, X_n \to A_n\}$ alakú, akkor a felbontás legyen $\rho(X_1A_1, X_2A_2, \ldots, X_nA_n, K)$, 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. $\exists Y \to B \in G^+, YB \subseteq X_iA_i$, 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 $Y \to B$ miatt $Y \to A_i$ ugyanazt fejezi ki, mint $X_i \to A_i$, így G-ben $X_i \to A_i$ helyett $Y \to A_i$-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 $X_i \to A_i$ függőség helyettesíthető G-ben $Z \to A_i$-vel.

Meg kell még vizsgálnunk, hogy a K kulcsként (esetleg) megjelenő $n+1$-edik séma is 3NF-e. Ha $Y \to B$ 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 mindegyik 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 $B_1, B_2, \ldots, B_k$ attribútumokkal bővítjük $K^{(0)} = K$-t, ebben a sorrendben. Nyilván igaz, hogy $K \cup \{B_1, B_2, \ldots, B_k\} = R$. 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 $Y \to B_i \in G$ függőség miatt. K^+ konstrukciója miatt biztos, hogy $Y \subseteq K \cup \{B_1, B_2, \ldots, B_{i-1}\}$. 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, $B_1, B_2, \ldots, B_{i-1}$ oszlopai pedig már a-k,
- YB_i sorában Y attribútumainak oszlopaiban a táblázat konstrukciója miatt mindenkor a-k állnak.

Mivel YB_i sorában B_i oszlopában a áll, ezért jogosan írhatunk K sorába B_i oszlopába is a-t.

Megjegyzés.

1. Vegyük észre, hogy a K kulcs csak a veszteségmentes tulajdonság biztosításához 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 minden mechanizmus ekkor is működik.

Példa. Egyetemi kurzusok adatait szeretnénk tárolni, az $R(\mathit{Kurzus}, \mathit{Tanár}, \mathit{Időpont}, \mathit{Helyszín}, \mathit{Diák}, \mathit{Jegy}) = R(K, T, I, H, D, J)$ sémában. A függőségek:

- $K \to T$: minden kurzushoz csak egy tanár tartozhat.
- $IH \to K$: egy időpontban egy helyszínen legfeljebb egy kurzus lehet.
- $IT \to H$: egy időpontban egy tanár csak egy helyszínen lehet.
- $KD \to J$: egy kurzusban egy diák csak egy jegyet kaphat.
- $ID \to H$: egy időpontban egy diák csak egy helyszínen lehet.

A $\rho(KT, IHK, ITH, KDJ, IDH)$ 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 ID attribútumhalmaz kulcs. Határozzuk meg ezért a lezárását:

- $ID^{(0)} = ID$,
- $ID^{(1)} = IDH$,
- $ID^{(2)} = IDHK$,
- $ID^{(3)} = IDHKJ$,
- $ID^{(4)} = IDHKJT$,

tehát $ID^+ = IDHKJT$, az attribútumok teljes halmaza, így ID legalábbis szuperkulcs. Ugyanakkor minimális is, mert F-ben sem I, sem D egyedül semmilyen más attribútumot nem határoz meg, $I^+ = I$, $D^+ = D$, vagyis az ID valóban kulcs.

Ha hozzávesszük a ρ felbontáshoz az ID sémát a 2. megjegyzés szerint, akkor az eredmény ρ-val azonos lesz, hiszen ρ utolsó sémája tartalmazza az ID 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 $\exists X \to A \in F^+$, ami megsérti a BCNF tulajdonságokat, azaz $A \notin X$ és X nem szuperkulcsa R-nek. Legyen ekkor a felbontás $\rho_1(R_1, R_2)$, $R_1 = XA, R_2 = R \setminus A$, 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 $X \to A$ 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:

- $DIÁKKÓD \to DIÁKNÉV$
- $DIÁKKÓD \to OFŐNÖK$
- $OFŐNÖK \to OFTEL$
- $OFTEL \to OFŐNÖK$
- $DIÁKKÓD \to OFTEL$

– $DIÁKKÓD, NYELV, FÉLÉVKÓD \rightarrow OSZTÁLYZAT$

Bontsuk fel BCNF alakú sémákba veszteségmentes dekompozícióval!

A séma egyetlen kulcsa a $DIÁKKÓD, NYELV, FÉLÉVKÓD$ attribútumhalmaz. Válasszuk ki az $OFŐNÖK \rightarrow OFTEL$ függőséget, és bontsuk fel a $NYELV$ sémát 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:

– $DIÁKKÓD \rightarrow DIÁKNÉV$
– $DIÁKKÓD \rightarrow OFŐNÖK$
– $DIÁKKÓD, NYELV, FÉLÉVKÓD \rightarrow OSZTÁLYZAT$

tehát R_2 kulcsa továbbra is $DIÁKKÓD, NYELV, FÉLÉVKÓD$.

R_2 még mindig nem BCNF alakú, mert a $DIÁKNÉV, OFŐNÖK$ nem-üres attribútumhalmaz determinánsa $DIÁKKÓD$, de nem szuperkulcs.

Az első két függőséget az alábbi formában is felírhatjuk:

– $DIÁKKÓD \rightarrow DIÁKNÉV, OFŐNÖK$

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. Mivel a függőségőrzés megtartása mellett nem mindig érhető el a BCNF felbontás (részletesebben ld. a megjegyzések után), ezért 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, ld. lekérdezés-intenzív/analitikus rendszerek.)
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 részsémára megvizsgáljuk, hogy az BCNF-e. Ha egy séma nem BCNF, akkor tovább 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 lehet. Szerencsére számos egyszerűsítésre nyílik lehetőség, melyek 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 $R(VÁROS, ÚT, IR\_SZÁM) = R(V, U, I)$ sémát, amelyen most is az $F = \{VU \to I, I \to V\}$ függőségeket értelmeztük. Így R kulcsai UI és VU, az $I \to V$ függőség miatt R nem BCNF. Készítsük el a séma egy valódi, veszteségmentes felbontását! Könnyen ellenőrizhető, hogy $\rho(R_1(UI), R_2(VI))$ az egyetlen lehetőség. A részsémák nyilván BCNF-ek. A vetített függőségek: $\pi_{R_1}(F)$ csak triviális függőségeket tartalmaz, $\pi_{R_2}(F) = \{I \to V, \text{triviális függőségek}\}$. Mivel a $VU \to I$ 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 feltétlenül tudunk készíteni. Ezért célszerű egy enyhébb normálforma 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.

9.2.8. 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 $r(R)$ reláció adódott.

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 (módosítási anomália). Próbáljuk meg R-et felbontani és a redundanciát csökkenteni!

$r(R)$

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

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 természetes illesztése többé nem adja vissza r-et!)

Mindezt fogalmazzuk meg pontosabban:

Definíció – többértékű függőség *(multivalued dependency, MVD).* 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 $t_3[R \setminus XY] = t_2[R \setminus XY]$,
– $t_4[Y] = t_2[Y]$ és $t_4[R \setminus XY] = t_1[R \setminus XY]$,

akkor Y többértékűen függ X-től (X multi-determinálja Y-t). Jelölése: $X \twoheadrightarrow Y$.

Megjegyzés. Vegyük észre, hogy a definíció szimmetriája miatt egyidejűleg fennáll $X \twoheadrightarrow R \setminus XY$ is.

$X \twoheadrightarrow Y$ tehát „ekvivalens” $X \twoheadrightarrow R \setminus XY$-nal, és mindkettő azt jeleni, hogy X minden egyes értéke meghatározza Y és $R \setminus XY$ lehetséges értékeinek egy halmazát, és ezen értékek minden egyes kombinációja (tehát minden megengedett Y érték mindegyik megengedett $R \setminus XY$ értékkel) elő is kell, hogy forduljon ilyen X mellett.

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

Bizonyítás. 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 $(R_1 \cap R_2) \twoheadrightarrow (R_1 \setminus R_2)$ (vagy $(R_1 \cap R_2) \twoheadrightarrow (R_2 \setminus R_1)$) következik a D függőségekből.

Speciálisan, ha $R(A, B, C)$ és $\rho(AB, AC)$ alakú, akkor ρ veszteségmentességének szükséges és elégséges feltétele, hogy $A \twoheadrightarrow B$ (vagy $A \twoheadrightarrow C$) fennálljon. Ennek elégséges feltétele, hogy $A \to B$ vagy $A \to C$ igaz legyen, összhangban a 9.2.4.5. alalszakasz 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 normálformának (4NF)* neveznek.

Definíció – *4NF.* 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.

Megjegyzések.

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

9.3. A fejezet új fogalmai

adatbázis kényszerek (értékfüggő, értékfüggetlen), tartalmazási függés, funkcionális függés (eseti, érdemi), implikáció, relációs séma determinánsa, (minimális) kulcs, szuperkulcs, idegen kulcs, egyszerű/összetett kulcs, elsődleges kulcs, kulcsjelölt, elsődleges/másodlagos attribútum, teljes/részleges függés, tranzitív függés, reláció redundanciája, atomi attribútum, 1NF, 2NF, 3NF, BCNF, funkcionális függés igazsága/levezethetősége adott függéshalmaz mellett, az Armstrong axiómák, az axiómák igazsága és teljessége, függéshalmaz lezártja, attribútumhalmaz lezártja, sémafelbontás, veszteségmentes sémafelbontás, vetített függéshalmaz, függőségőrző sémafelbontás, többértékű függőség, 4NF

10. fejezet

Tranzakciók adatbázis-kezelő rendszerekben

Mindeddig arról volt szó – hallgatólagosan –, hogy valamely adatbázis-kezelő rendszer egyidőben egyetlen felhasználó egyetlen programját szolgálja ki, egyéb igények csak a korábbiak kielégítése után következhetnek. A 9. fejezetben megmutattuk azt, hogy milyen módszerekkel lehet és célszerű a relációs adatbázis-kezelő logikai adatstruktúráit annak figyelembevételével kialakítani, hogy az adatbázis-kezelő tranzakciófeldolgozó képessége (praktikusan az adatbázis tartalmának módosítási képessége) minél magasabb lehessen. A továbbiakban annak a problémakörét vizsgáljuk meg, hogy milyen nehézségek merülnek fel akkor, ha több felhasználó vagy program „egyidejűleg" (ld. concurrency) kerül kiszolgálásra a DBMS által.

10.1. Bevezető

Definíció – tranzakció *(transaction).* (Tágabb értelemben is): Egy program egyszeri futása, amelynek vagy minden művelete hatásos, vagy belőle semmi sem (ld. atomicitás, oszthatatlanság).

Ahhoz, hogy egy tranzakció egy DBMS konkurens környezetében is értelmezhető legyen, hagyományosan további tulajdonságokat is, ill. ezek közül minél többnek a teljesülését elvárják tőle. Ezeket az angol elnevezések kezdőbetűiből alkotott betűszó után ACID tulajdonságoknak nevezik.

- *atomicitás* (atomicity): ld. fentebb.
- *konzisztencia* (consistency): csak sikeresen (teljes egészében) lefutott tranzakcióknak van hatása az adatbázis tartalmára, ekkor a tranzakciók az adatbázist egyik konzisztens állapotból egy másikba viszik át[1].

[1] A fogalom több más módon is értelmezhető, de fontos, hogy jelen jegyzet kontextusában ezt a definíciót fogadjuk el.

– *izoláció* (isolation), másnéven elszigetelés: minden tranzakció úgy fut le (egy konkurens környezetben is), mintha közben más tranzakció nem futna.
– *tartósság* (durability): ha egy tranzakció már sikeresen lefutott, akkor annak hatása „nem veszhet el”.[2]

Az adatbázis-kezelőkben a tranzakciókezelés megvalósítása alapvetően arról szól, hogy milyen feltételek mellett, milyen módszerekkel, milyen hatékonyan lehet az ACID tulajdonságokat biztosítani.[3]

A tranzakciók maguk is elemi lépésekből állnak: írás/olvasás + kiegészítő lépések: ezek szintén oszthatatlannak tekintettek.

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

Problémák a tranzakciókkal:

– idő előtti befejeződés (ld. 10.7. szakasz)
  ◦ nullával osztás/illegális művelet végzése
  ◦ nem fér hozzá adathoz
  ◦ kívülről lövik ki
– konkurens rendszerekben a tranzakciók műveletei összefésülődhetnek, mert a tranzakció általában nem tud befejeződni, mielőtt az adatbázis-kezelőnek el kell kezdenie egy másik tranzakciót kiszolgálni, vagy folytatnia egy másik tranzakció kiszolgálását.

A tranzakciók egymásba fésülődése csak akkor probléma, ha közös erőforráshoz akarnak hozzáférni. A közös erőforrások jelen vizsgálatainkban csupán az adatok lesznek.

10.2. Ütemezések (schedules)

Ha a tranzakciók egy rendszerben szigorúan egymás után futnak le úgy, hogy egyidejűleg mindig csak egyetlen tranzakció fut, tehát időben nem lapolódnak át, akkor ez egy *soros ütemezés* (serial schedule). Egy soros ütemezés mindig megvalósítható, problémamentes, amennyiben a tranzakciók külön-külön megvalósíthatók, és az izolációjuk természetes módon teljesül.[4]

Minden egyéb ütemezés neve: *nem soros ütemezés* (non-serial schedule). Ilyen ütemezés megvalósulhat egyetlen CPU-n (időosztással) vagy több CPU-n is. A nem

[2] Néha igen szélsőséges feltételek mellett kell ezt biztosítani, ami jelentős költségekkel járhat.
[3] Az ún. NoSQL adatbázis-kezelőkben (ld. C. függelék) egészen más elvárások is megfogalmazódhatnak a konkurencia biztosítása kapcsán, ezért az ACID tulajdonságok elsődlegesen a „klasszikus” rendszerekre igazak.
[4] Vegyük észre, hogy a soros ütemezés egy elégséges, de nem feltétlenül szükséges feltétel a tranzakciók izolációjához. A cél éppen az, hogy a tranzakciók minél nagyobb fokú konkurenciája valósulhasson meg, és ezzel együtt a gépi erőforrások jobb kihasználhatósága.

soros ütemezések lehetnek *sorosíthatóak* (serializable schedule) vagy *nem sorosíthatóak* (non-serializable).

Definíció – sorosíthatóság *(serializability)*. Egy ütemezés pontosan akkor sorosítható, ha létezik olyan soros ütemezés (ez lesz a *soros ekvivalens ütemezés*, serial equivalent schedule), amelynek minden hatása a módosított adatokra azonos az adott ütemezésével.

Nem soros ütemezés esetén a tranzakciók egymásba fésülődése/átlapolódása a következő problémákat okozhatja:

***Piszkos olvasás* (dirty read):** egy T_2 tranzakció olyan – ún. *piszkos* (ld. *Tranzakcióhibák kezelése*, 10.7. szakasz) – adatot olvas, melyet egy másik, T_1 tranzakció azelőtt írt az adatbázisba, hogy sikeresen befejeződött volna. Ha a T_1 tranzakció végül valóban sikertelennek bizonyul, akkor a piszkos adat az adatbázisból mihamarabb eltávolítandó.

Példa.

T_1	T_2
READ A	
$A = A + 1$	
WRITE A	
	READ A
	$A = A + 1$
	WRITE A
	COMMIT
ABORT	

***Elveszett módosítás* (lost update):** több tranzakció ugyanazon az adategységen végez módosításokat úgy, hogy egy T_1 tranzakció felülírja a másik, T_2 által végzett műveletek eredményét.

Példa.

T_1	T_2
READ A	
	READ A
$A = A + 1$	
	$A = A + 1$
	WRITE A
WRITE A	

***Nem megismételhető olvasás* (non-repeatable read):** egy T_1 tranzakció különböző eredményeket kap egy adategység többszöri olvasásakor, mert egy másik, T_2 tranzakció időközben módosította azt.

Példa.

T_1	T_2
READ A	
READ B	
$B = A + B$	
WRITE B	
	READ A
	$A = A + 1$
	WRITE A
READ A	
READ C	
$C = A + C$	
WRITE C	

Megjegyzés. Vegyük észre, hogy a nem megismételhető olvasás annyiban tér el az elveszett módosítástól, hogy itt az egyik tranzakció (a példákban T_2) által elvégzett módosítás nem veszik el.

***Fantom olvasás* (phantom read):** egy T_1 tranzakció többször is végrehajtja ugyanazt a lekérdezést, miközben egy másik, T_2 tranzakció olyan rekordokat szúr be vagy töröl, melyek kielégítik a T_1 lekérdezésének szelekciós feltételét. Így a korábbi lekérdezés más rekordhalmazt adhat vissza, mint az utána következő(k).

Példa.

T_1	T_2
READ τ	
	INSERT Y TO τ
READ τ	

Megjegyzés. τ az adatok egy (bizonyos szelekciós feltételnek megfelelő) halmaza, Y pedig egy (a szelekciós feltételt (részben) szintén kielégítő) rekordhalmaz.

Fentiek elkerülése, ill. a kapcsolódó problémák megoldása a tranzakciókezelés egyik központi kérdése. Egyik lehetőség, ha egyenként keresünk megoldásokat. Észrevehetjük azonban, hogy a soros ütemezéseknél a fenti problémák fel sem merülnek. Ezért:

Definíció – izolációs elv *(isolation principle)*. Feltételezzük, hogy egy tranzakció elvárt, korrekt eredménye az, amit akkor kapunk, ha a tranzakció futása közben más tranzakció nem fut.

Ha tehát nem lehet egy nem soros ütemezésnél biztosítani, hogy ugyanazt az eredményt produkálja, mintha a tranzakcióit *valamely* sorrendben sorosan egymás után futtatnánk le, akkor az ütemezést nem tekintjük helyesnek, korrektnek.

Definíció – korrekt *(correct)*. Egy ütemezés pontosan akkor *korrekt*, ha sorosítható.

Példa. Soros, sorosítható és nem sorosítható ütemezések:

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	
$A = A - 10$	
WRITE A	
	READ B
READ B	
$B = B + 10$	
	$B = B - 20$
	WRITE B
	READ C
WRITE B	
	$C = C + 20$
	WRITE C

c) nem sorosítható ütemezés

A sorosíthatóság számos módszerrel és feltétel mellett biztosítható. Általában az adathozzáférés módjának alkalmas korlátozása jelenti a probléma megoldását.

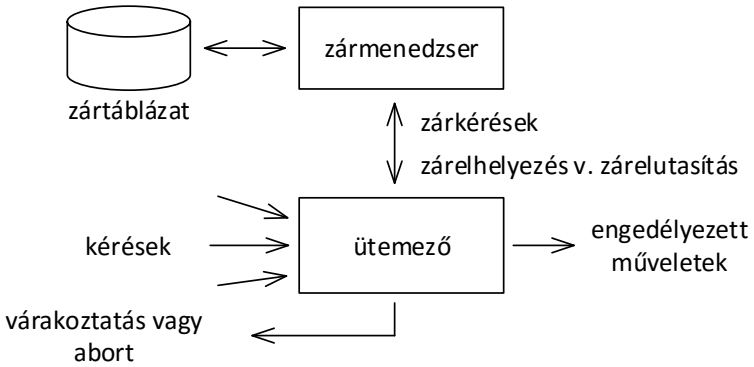
Az adathozzáférés

- módjának szabályozása pl. *zárakkal* (lock) (ld. 10.3. szakasz), és/vagy időbélyegekkel (ld. 10.9. szakasz) történhet,
- egységeinek megválasztása (*granularitás*, granularity) kritikus rendszertechnikai kérdés.

Megjegyzések.

1. A definíció alapján 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. A sorosíthatóság kérdését a gyakorlatban folyamatosan („on-the-fly”) kell tudni megítélni, minden egyes új adathozzáférési igény kapcsán (ld. *ütemező*).

4. Tipikus gyakorlati megoldás: olyan szabályok (*protokollok*) megalkotása, amelyeknek a betartása garantálja a sorosíthatóságot.



10.1. ábra. A konkurens működés elemei zár alapú tranzakciókezelés esetén

Definíció – ütemező *(scheduler)*. A DBMS azon része, amely az adatelérési igények megítélése felett dönt (pl. a sorosíthatóság biztosítása és a pattok (ld. alább) feloldása érdekében). Ennek során

- engedélyezheti az egyes műveleteket, vagy
- ha ennek feltételei nem állnak fenn, akkor
  - várakoztathatja, ill.
  - abortálhatja, újraindíthatja a tranzakciókat.

Az ütemező bonyolultsága jelentősen függ az alkalmazott protokolltól. Zárkezelésen alapuló tranzakció menedzsment esetén szorosan együttműködik a zármenedzserrel. A zármenedzser kezeli a zártáblázatot, mely az alábbi szerkezetű bejegyzéseket tartalmazza:

$$\langle \text{adategység} \rangle \langle \text{zártípus} \rangle \langle \text{tranzakció} \rangle$$

10.3. Tranzakciókezelés zárakkal

Definíció – zár *(lock)*. Hozzáférési privilégium egy adategységen, amely adható és visszavonható.

Példa. Hogyan kezelik a zárak a konkurenciát?

Tekintsük az előbbiekben ismertetett példát az elveszett módosítás problémára! Bár T_1 és T_2 egyaránt 1-gyel növelték A értékét, az végeredményben csak 1-gyel

nőtt. Ennek oka, hogy T_1 anélkül módosította A-t, hogy tudott volna a T_2 által végzett módosításról. Ha T_2 módosításait meg akarjuk akadályozni, akkor helyezzünk el zárakat (legkésőbb) az adatmódosító műveletek előtt (és célszerű, ha a felszabadításaikról is gondoskodunk (legkésőbb) a tranzakció végén):

T_1	T_2
LOCK A	
READ A	
	LOCK A
	READ A
$A = A + 1$	
	$A = A + 1$
	WRITE A
	UNLOCK A
WRITE A	
UNLOCK A	

10.1. táblázat

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.

Ennek hatására a T_2 tranzakció a harmadik sorban valójában nem fogja tudni végrehajtani a zár elhelyezését az A adategységen, hanem meg kell várnia annak felszabadítását. A vezérlés a másik tranzakcióra adódik át, ezért a fenti T_1 és T_2 tranzakciókból álló ütemezés valójában így fog lefutni:

T_1	T_2
LOCK A	
READ A	
$A = A + 1$	T_2 várakozik
WRITE A	
UNLOCK A	
	LOCK A
	READ A
	$A = A + 1$
	WRITE A
	UNLOCK A

10.2. táblázat

Ekkor már helyesen, 2-vel fog nőni A értéke T_1 és T_2 lefutása után.[5]

[5] A példában egy soros ütemezés valósult meg a zárak alkalmazása miatt, de hangsúlyozzuk, hogy ez messze nem törvényszerű.

Definíció – legális ütemezés *(legal schedule)*. Legális az az ütemezés, amelyben

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

A fenti 10.1 táblázatban látható ütemezés tehát nem legális, a 10.2 táblázatban látható viszont legális.

10.4. 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 *pattról, holtpontról* (deadlock) beszélünk.

Patt elképzelhető természetesen kettőnél több tranzakció részvételével is.

Megoldási lehetőségek:

1. A tranzakciók lockoljanak mindent egyszerre, amire a futásukhoz szükségük lehet. Ha valamely zárat nem kaphatják meg, akkor ne is próbálkozzanak egyetlen művelet elvégzésével sem.
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 *(wait-for graph)*. 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.[a]

[a] Kifejezőbb lenne a *várakoztatási gráf* elnevezés.

Tétel. Adott időpillanatban nincs patt $\Leftrightarrow$ a várakozási gráfban nincs kör (azaz a gráf irányított körmentes gráf (DAG))

Bizonyítás. Előre: (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.

Visszafelé: 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 egy akár legális ütemezésben szereplő tranzakció ne tudjon lefutni. Ha egy tranzakció egy adategység lockolására vár, de közben más tranzakciók mindig lockolják 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 (ami először megy be, az jön ki először (FIFO) stratégia).

10.5. Tranzakció modellek

A szabályos tranzakciókat jellemző tulajdonságok gyűjteménye.

Definíció – egyszerű tranzakció modell *(simple transaction model)*. Egyszerű tranzakció modellről beszélünk, ha

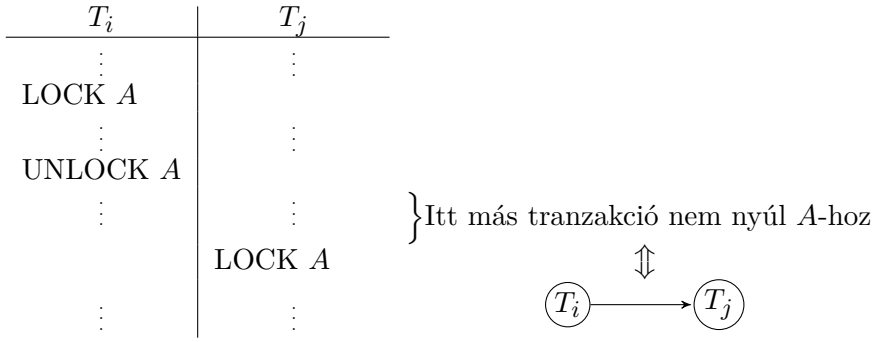
- csak egyfajta zár létezik
- egy adatelemen egyidőben csak egyetlen zár lehet.

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 gráf segítségével.

Definíció – sorosítási gráf, precedenciagráf *(precedence graph)*. 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.

A definíció háttere 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 T_i meg kell, hogy előzze T_j-t az ütemezés minden soros ekvivalensében, ezt fejezi ki a nyíl.



10.2. ábra. Precedenciagráf rajzolása egyszerű tranzakció modellben

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

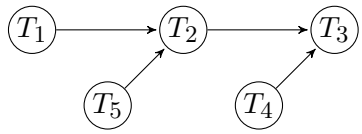
Bizonyítás. **Előre (indirekt):** t. f. h. S sorosítható, de tartalmaz kört a sorosítási gráfja. Ekkor a kört alkotó tranzakciók közül egyik sem előzi meg a másikat, tehát egy soros ekvivalensben egyik sincs legelöl, azaz az ütemezés nem sorosítható, ellentmondásban a feltétellel.

Bizonyítás. **Visszafelé:** ha a gráf DAG, akkor létezik topologikus rendezése. Belátjuk, hogy ebből a rendezésből legalább egy soros ekvivalens előállítható. Ugyanis 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 ($T_{\text{első}}$) 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 $T_{\text{első}}$ után lefuthatnak, mert vagy egyáltalán nem olvas(nak) olyan adategységet, amin egy másik tranzakció korábban már zárat helyezett el, vagy csak $T_{\text{első}}$ által már korábban lockolt adategységen helyeznek el zárat. Mindez addig folytatható, amíg a gráf minden 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 minden tranzakció egy adott protokollt követ, amely protokoll mellett a sorosíthatóság vagy automatikusan teljesül, vagy egyszerű módszerekkel biztosítható.

10.5.1. Kétfázisú zárolás (2PL)

Definíció – kétfázisú zárolás *(two-phase locking, 2PL)*. Egy tranzakció a kétfázisú zárolás protokollt követi[a], ha az első zárfelszabadítást megelőzi mindegyik zárkérés.

[a] Röviden azt mondjuk, hogy a tranzakció kétfázisú.

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

Példa. Az előző példában T_3 kivételével minden tranzakció kétfázisú.

Tétel. Ha egy legális ütemezés minden 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:

$$T_{i_1} \to T_{i_2} \to T_{i_3} \to \ldots \to T_{i_k} \to T_{i_1}$$

- A $T_{i_1} \to T_{i_2}$ él megléte azt jelenti, hogy az ütemezésben van egy

  $$T_{i_1} : \text{LOCK } A_{j_1} \ldots \text{UNLOCK } A_{j_1} \ldots T_{i_2} : \text{LOCK } A_{j_1}$$

  szekvencia.
- Hasonlóan, a $T_{i_2} \to T_{i_3}$ él megléte azt jelenti, hogy az ütemezésben van egy

  $$T_{i_2} : \text{LOCK } A_{j_2} \ldots \text{UNLOCK } A_{j_2} \ldots T_{i_3} : \text{LOCK } A_{j_2}$$

  szekvencia.
- Végül a $T_{i_k} \to T_{i_1}$ él megléte azt jelenti, hogy az ütemezésben van egy

  $$T_{i_k} : \text{LOCK } A_{j_k} \ldots \text{UNLOCK } A_{j_k} \ldots T_{i_1} : \text{LOCK } A_{j_k}$$

  szekvencia.

Láthatóan így a T_{i_1} 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 *(synchronization point)*. Az az időpont, amikor egy kétfázisú protokoll szerinti tranzakció az utolsó zárját is megkapja.

Tétel. A fenti tétel a zárpont segítségével is bizonyítható.

Bizonyítás. 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 minden 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 ekvivalens(eke)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	T_2
LOCK A	
UNLOCK A	
	LOCK A, B
	UNLOCK A, B
LOCK B	
UNLOCK B	

   Ennek a (legális) ütemezésnek a sorosítási gráfja:

   $$T_1 \rightleftarrows T_2$$

   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 – RLOCK-WLOCK modell *(shared and exclusive lock model)*. Az RLOCK–WLOCK modellben

- kétfajta zár létezik:
  - RLOCK (*megosztható*, shareable; *puha*, soft): ha T: RLOCK A érvényes, akkor más tranzakció is olvashatja A-t, de senki nem írhatja.
  - WLOCK (*nem megosztható*, unshareable; *kemény*, hard): ha T: WLOCK A érvényes, akkor T-n kívül semmilyen más tranzakció nem fér hozzá A-hoz, sem írásra, sem olvasásra.
- Az UNLOCK művelet kiadása mind az RLOCK-ot, mind a WLOCK-ot felszabadítja, ami legális ütemezések esetén a tranzakció vége előtt bekövetkezik.

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 gráf segítségével eldönteni egy ütemezés sorosíthatóságát.

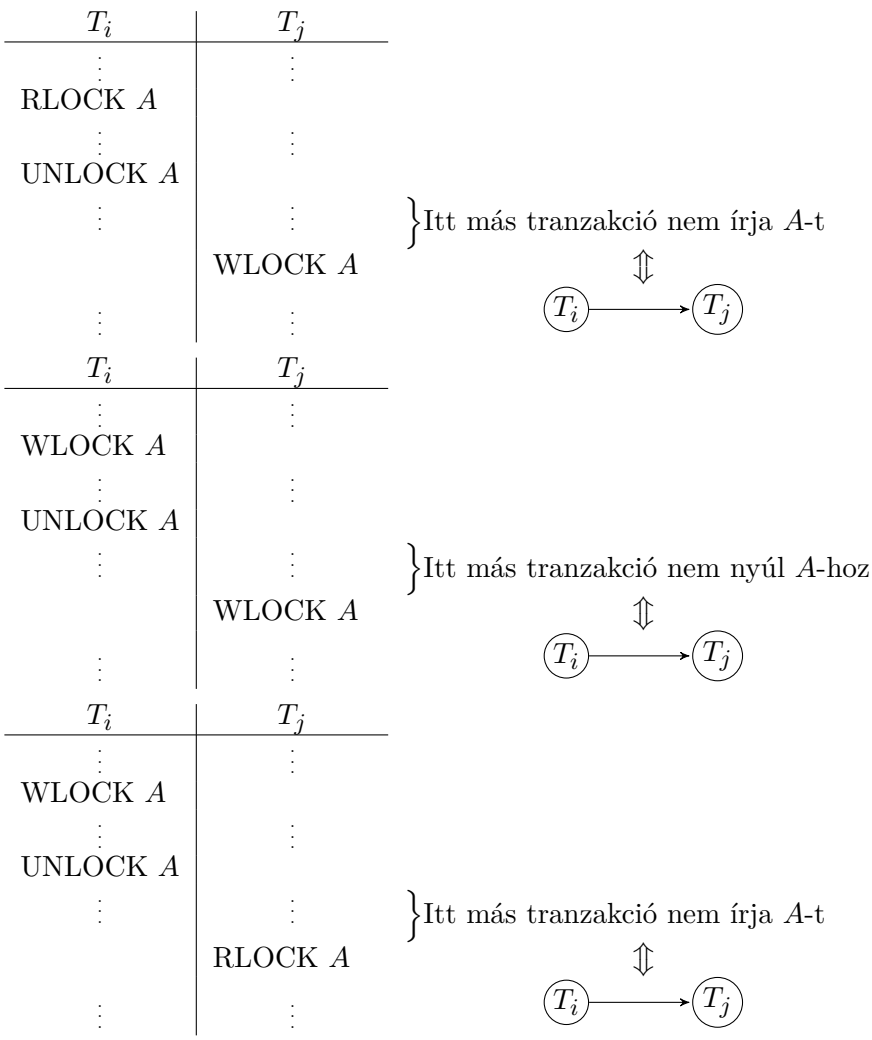
A 10.3. ábra mutatja azt a három szekvenciát, amelyek esetén a precedenciagráfban $T_i \to T_j$ él rajzolandó:

- Az első esetben T_j feltétlenül T_i után kell, hogy következzen, mert T_i-nek T_j-től független A értéket kell olvasnia, és ez igaz az összes olyan T_i-re, amit T_j előtt nem követett az A értékét író tranzakció.
- 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, és ez – az első esethez hasonlóan – igaz az összes olyan T_j-re, amit T_i óta nem előzött meg az A értékét író tranzakció.

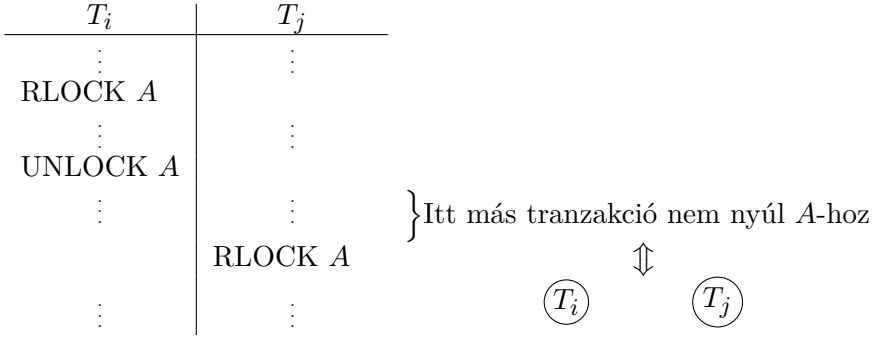
Vegyük észre: valójában a hatékonyságnövekedést az okozza, hogy a 10.4. ábrán bemutatott $READ \dots READ$ szekvenciák esetén nem kell élet húzni T_i és T_j között (ha az egyszerű tranzakció modellben gondolkodunk, akkor itt is lenne él, tehát nagyobb lenne a kör keletkezésének valószínűsége; természetesen az ütemezés másik szekvenciája miatt keletkezhet él T_i és T_j között). Az A-t csak olvasó tranzakciók sorrendje a sorosíthatóság szempontjából tehát mellékes, ha mindegyik a legutóbbi, A-t író tranzakció után és az első, őt követő, A-t író tranzakció előtt következik a soros ekvivalensben.

Tétel. Egy RLOCK-WLOCK modellbeli S ütemezés sorosítható $\Leftrightarrow$ a fenti szabályok szerint rajzolt sorosítási gráf DAG.

Bizonyítás. Analóg az egyszerű tranzakció modell hasonló tételével.



10.3. ábra. Precedenciagráf rajzolása az R/W tranzakciómodellben



10.4. ábra. Precedenciagráf $READ \ldots READ$ szekvencia esetén az R/W tranzakciómodellben – természetesen az ütemezés másik szekvenciája miatt keletkezhet él T_i és T_j között

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 zármódok is definiálhatók. Ettől nyilván akkor várhatunk további hatékonyságnövekedést, ha az értelmezett zármódok között minél több olyan van, amely egy adategységen egyidejűleg tartható fenn, azaz összeférhetők, kompatibilisek. A zármódok közötti összeférhetőséget a zárkompatibilitási mátrixban *(lock compatibility matrix)* szokás ábrázolni. 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.

Példa *(1)*. Az RLOCK-WLOCK modell kompatibilitási mátrixa:

		meglévő zár az adategységen	
		RLOCK	WLOCK
zárkérés	RLOCK	igen	nem
	WLOCK	nem	nem

Példa *(2)*. 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	igen	nem	nem
WLOCK	nem	nem	nem
INCR	nem	nem	igen

Ha ismerjük az alkalmazott zárak kompatibilitási mátrixát, akkor fel tudjuk rajzolni valamely ütemezés sorosítási gráfját.

10.6. 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.:

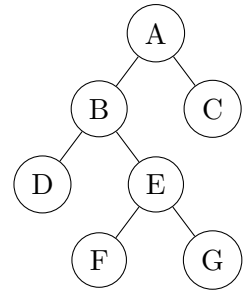
1. hierarchikus adatbázis rekordjai
2. B*-fa elemei
3. egymásba ágyazott adatelemek: ezek közül a legfontosabb a relációs adatbázis–reláció–blokk–tuple hierarchia.

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 minden 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*.

10.6.1. A fa protokoll (tree protocol)

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 ugyanaz 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 sem:

1. LOCK A
2. LOCK B
3. UNLOCK A
4. LOCK E

Ez a szekvencia a valóságban is gyakran előfordul, pl. amikor egy tranzakció beszúrás műveletet hajt végre egy B*-fán. Ha B egy olyan csomópont, amiben van hely egy újabb indexrekord számára, akkor a fa átstruktúrálódása a beszúrás következtében a B csomópont őseit 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.

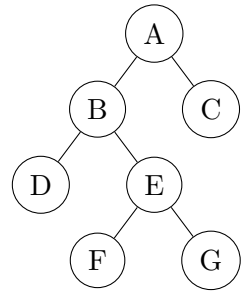
Ha $E(T_i)$ és $E(T_j)$ közül egyik sem őse a másiknak, akkor nem rajzolunk élt T_i és T_j között, hiszen a protokoll garantálja, hogy ekkor T_i és T_j sohasem fog közös csúcsot (adategységet) zárolni.

Ezért t. f. h. $E(T_i)$ őse $E(T_j)$-nek.

- Ha T_i zárolja először $E(T_j)$-t, mielőtt T_j zárolná, akkor egy $T_i \to T_j$ élt 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 $T_j \to T_i$ élt rajzolunk a gráfba.

10.6.2. A figyelmeztető protokoll (warning protocol)

Az egyszerű tranzakció modellt követjük, de egy csomópont zárolása a gyerek és az összes leszármazott csomópontok zárolását is jelenti (utóbbi neve: *implicit zár*, implicit lock).



Ez utóbbi feltételezés azonban azt is eredményezheti, hogy két tranzakció tart fenn egyidejűleg zárat ugyanazon adategységen. Tekintsük az alábbi példát:

1. T_1: LOCK E
2. T_2: LOCK A

Ezután F-en és G-n T_1 és T_2 is (implicit) zárat tart fenn (valamint E-n T_2 implicit, T_1 explicit zárat tart fenn). A jelenség neve *zárkonfliktus* (lock conflict), amit alkalmas szabályok megtartásával el kell kerülni.

A figyelmeztető protokoll zárműveletei:

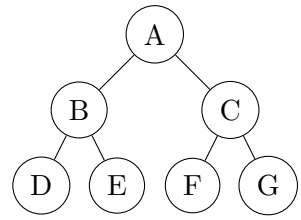
- LOCK A: zárolja A-t és az összes leszármazott csomópontot is. Két különböző tranzakció nem tarthat fenn egyidejűleg zárat ugyanazon adategységen.
- WARN A: A-ra *figyelmeztetést* (warning) rak. Ekkor A-t más tranzakció nem zárolhatja.
- UNLOCK A: eltávolítja a zárat vagy az UNLOCK-ot kiadó tranzakció által elhelyezett 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 ugyanaz a tranzakció már helyezett el WARN-t,
3. UNLOCK A akkor lehetséges, ha A gyerekein már ugyanaz a tranzakció nem tart fenn sem LOCK-ot, sem WARN-t,
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



10.5. ábra. Az adategységek hierarchiája

Tétel. A figyelmeztető protokollt követő legális ütemezések zárkonfliktusmentesek (1) és sorosíthatók (2).

Bizonyítás.

1. A protokoll 1-3. szabályai biztosítják, hogy bármely T_1 tranzakció csak akkor tehessen zárat egy adategységre, ha figyelmeztetés van annak minden ősén. Emiatt egyidejűleg más T_2 tranzakció nem tehet zárat egy lockolt adategységnek egyetlen ősére sem. Ahhoz pedig, hogy egy leszármazott adategységre tehessen zárat T_2 az kellene, hogy ettől az adategységtől a gyökérig T_2 figyelmeztetéseket helyezzen el. Azonban a figyelmeztetés biztosan nem helyezhető el arra az adategységre, amelyiken már T_1 helyezett el zárat, hiszen ezek a zárműveletek nem kompatibilisek. 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-et tehát úgy állítjuk elő, hogy

1. eltávolítjuk az összes figyelmeztetést és UNLOCK párjait R-ből,
2. LOCK X esetén explicit zárat helyezzünk X minden leszármazottjára is,
3. UNLOCK X esetén eltávolítjuk a zárat X minden leszármazottjáról.

Ezek után 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 az átalakítás során a kétfázisú tulajdonság megmaradt. Ezek elégséges feltételek S sorosíthatóságához.

Megjegyzések.

1. A WARN–LOCK kompatibilitási mátrix

	LOCK	WARN
LOCK	nem	nem
WARN	nem	igen

   formailag 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.

10.7. 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* (system failure) lép fel, emiatt az adatbázis-kezelő hibásan kezd működni (szoftver és/vagy hardver eredetű hibák egyaránt tartozhatnak ide),
5. a háttértár tartalma (is) megsérül (*médiahiba*, medium failure).

Az 1-3. hibákban közös az, hogy a memória- és a háttértár-struktúrák érintetlenek, ellentétben a 4-5. pontokhoz tartozó hibákkal. Rendszerhibáról beszélünk akkor, ha az operatív 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 milyen változások történtek az adatbázison a legutóbbi mentés óta.

Első lépésben csak azokkal a kérdésekkel foglalkozunk, hogy mi a teendő az 1-3. hibaokok esetén (ún. *tranzakcióhiba*, transaction failure).

Definíció – konzisztens állapot *(consistent state)*. Az adatbázisnak olyan állapota, amely csak teljesen lefutott tranzakciók hatását tükrözi (ld. ACID tulajdonságok, 10.1. szakasz).

Mivel a tranzakciók idő előtt, ill. irregulárisan is befejeződhetnek, az adatbázis konzisztens állapotának fenntartása automatikusan nem fog megvalósulni. Olyan módszerekre van szükség, amelyek az egyes tranzakcióknak ezt a tulajdonságát biztosítják. Tranzakcióhibák esetén ezek a módszerek a készpont fogalmának bevezetésén keresztül érthetők meg.

Definíció – készpont, commit pont *(commit point)*. Az az időpillanat, amikor egy tranzakció futása során már minden befejeződött, ami a tranzakció 1-3. 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 (adatértékek visszaállítása, zárak felszabadítása).

Definíció – piszkos adat *(dirty data)*. Olyan adat, amit az előtt írt valamely tranzakció az adatbázisba, mielőtt commitált volna (ld. még a 10.2. szakaszban a piszkos olvasás jelenségét).

A piszkos adat az adatbázisból mihamarabb eltávolítandó. Azonban előfordulhat, hogy egy másik tranzakció olvassa a piszkos adatot, és az a tranzakció már sikeres. Ennek ellenére, eredménye nem tekinthető helyesnek, így az adatbázisból ez is eltávolítandó. A jelenség iteráltan is előfordulhat, ekkor a jelenség neve *lavina* (cascading aborts).

Példa.

T_1	T_2
LOCK A	
READ A	
$A = A - 1$	
WRITE A	
LOCK B	
UNLOCK A	
	LOCK A
	LOCK C
	READ A
	$C = A \times 2$
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 adatbázis-kezelő 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 commitált 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).

10.7.1. Szigorú kétfázisú protokoll (strict two-phase locking protocol)

A tranzakcióhibák kezelésének igen gyakori módszere. Egy tranzakció ezt a protokollt követi, ha kétfázisú, továbbá

1. nem ír az adatbázisba, amíg a készpontját 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*, journaling, logging, ld. később).

Ezután az alábbi tétel állítása csaknem triviális:

Tétel. A szigorú kétfázisú protokollt követő tranzakciókból álló legális ütemezések sorosíthatók és lavinamentesek.

Bizonyítás. Az ütemezés sorosítható, mert kétfázisú; továbbá lavinamentes, mert nincs lehetőség piszkos adat olvasására.

10.7.2. 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 az összes zárját előre kéri, és csak akkor indul el, ha már mindet megkapta).

Ezen alternatívák elsősorban annak alapján értékelhetők, hogy mennyire segítik elő az adatbázis-kezelő rendszer „hatékony” konkurens működését.

„Hatékonyság” mércéje lehet pl.:

- egy adott tranzakció minél hamarabb fusson le,
- adott idő alatt minél több tranzakció fusson le sikeresen (*tranzakciós teljesítmény*, transactional performance).

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 zártáblák 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 – agresszív protokoll *(aggressive protocol),* optimista konkurenciakezelés *(optimistic concurrency control).* Egy protokoll *agresszív*, ha megpróbál olyan gyorsan lefutni, amennyire csak lehetséges, nem törődve azzal, hogy ez esetleg aborthoz is vezethet (keveset foglalkozik *a)*-val, azaz a zárakkal).

Definíció – konzervatív protokoll *(conservative protocol),* pesszimista konkurenciakezelés *(pessimistic concurrency control).* Egy protokoll *konzervatív*, ha megkísérli elkerülni az olyan tranzakciók futtatását, amelyek nem biztos, hogy eredményesek lesznek (sokat fordít *a)*-ra).

Példa *(1).* 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 kap meg egy zárat, ha az előtte senkinek nem kell az adatelemhez tartozó várakozó sorban.

Vegyük észre: 1. miatt sorosítható és lavinamentes, 2. miatt pattmentes, 3. miatt éhezésmentes az ütemezés. 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) minden zárat.

Példa *(2).* Másik véglet: nagyon agresszív egy protokoll, ha a zárakat közvetlenül írás vagy olvasás előtt kéri. Amennyiben a zárakat csak azután engedi el, hogy mindet megkapta, akkor egyszerűen sorosítható, egyébként pedig a sorosíthatóság biztosítása is külön probléma lehet. 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.

10.8. 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 (tranzakciós) *naplózás.* [6] A naplózásnak számos módja lehet, itt csak a leggyakoribbakra térünk ki.

Definíció – napló *(journal, log).* 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: az abort művelet naplózása).

[6] Több, különböző célú napló is létezhet egy DBMS-ben, mint pl. a felhasználói bejelentkezések naplója, (egyes) adatelemekhez való hozzáférések naplója (audit log), üzemeltetési műveletek naplója, stb. Itt kifejezetten csak arról a naplóról van szó, amelyik az adatbázison végrehajtott változtatásokat tartalmazza (transaction log, redo log).

10.8.1. Hatékonysági kérdések

A helyreállítás képességének igénye miatt a napló stabil tárban (háttértáron, diszken) tárolandó. Ennek költsége – mint ismert, általában – a háttértárra írt naplóblokkok számával arányos. A gyakorlatban a naplóblokkokat 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, hiszen ezáltal válik a tranzakció tartósan – akár megismételhetően – végrehajtottá (az ACID-ból a tartósság így teljesül).

Az adatbázis oldalakat lehet a naplótól függetlenül is, egy másik lapozási stratégia szerint cserélni.

10.8.2. A redo protokoll (redo protocol)

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*.

10.8.2.1. A redo naplózás (redo logging)

A redo naplózás a szigorú 2PL finomítása. Lépései:

1. (T, begin) naplóba,
2. (T, A, $\langle A$ új értéke$\rangle$) 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 (operatív tárba),
6. a piszkos DB blokkok diszkre írása egyéb szempontok szerint,
7. zárak elengedése.

Megjegyzések.

1. Az a tranzakció lett véglegesítve, amelyik eljutott a 4. pont végéig, hiszen a napló alapján az adatbázison végzett műveletei már bármikor megismé-

telhetők. A 4. pont végéig el nem jutott tranzakciók hatása pedig részben vagy egészben elveszhet.
2. 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.
3. A 7. pont után férnek hozzá csak más tranzakciók a T által megváltoztatott adatokhoz, így nincs lavinaeffektus sem.

10.8.2.2. A redo helyreállítás (redo recovery)

Az adatbázist egy konzisztens állapotba viszi át a helyreállítás végére.

Lépései:

1. az összes zár felszabadítása,
2. napló vizsgálata visszafelé: feljegyezzük azon tranzakciókat, amelyekre találunk (T, commit) bejegyzést,
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* (idempotent).[7]

Példa. Egy redo protokoll szerinti tranzakcióra:

tranzakció	napló		
	(T, begin)	⎫	
LOCK A		⎪	Ha itt jön a rendszerhiba, akkor a
LOCK B		⎪	helyreállítás során A és B értéke
	(T, A, v)	⎪	változatlan marad.
	(T, B, w)	⎭	
	(T, commit)	←	Ezen a ponton a napló stabil tárba
WRITE A			írandó. Ha ezután jön a rendszerhiba,
WRITE B			akkor később a tranzakció már bármikor
UNLOCK A			helyreállítható.
UNLOCK B			

[7] Az informatikában idempotensnek nevezünk egy műveletet, ha tetszőleges bemenetre (adatra) egyszer vagy többször lefuttatva ugyanazt az eredményt adja.

10.8.3. 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* (checkpoint) tényét naplózzuk,
5. naplót kiírjuk a háttértárra.

Előnyök:

- redo helyreállításnál csak a legutóbbi ellenőrzési pontig kell a naplóban visszamenni,
- emiatt a napló korábbi részei eldobhatók (amennyiben más érv nem szól ez ellen),
- 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.

10.8.4. Médiahibák elleni védekezés

A tranzakcióhibák tárgyalása során az utolsó eset a *médiahibák* (medium failure) kezelése, melyre a következő megoldások terjedtek el:

- Rendszeres mentések (backup). Modern adatbázis-kezelő rendszerek ezt úgy hajtják végre, hogy egy pillanatnyi üzemszünetet sem okoz.
- Az adatbázis fájlok duplikálása lehetőleg egy másik fizikai eszközön, néha más földrajzi helyen (ld. 11. fejezet).

Érdemes a napló fájlokat is védeni: jó gyakorlat, ha az adatbázis és a napló más-más fizikai eszközön található. Szóba jön itt is a duplikálás.

10.9. Időbélyeges tranzakciókezelés (timestamp-based concurrency control)

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, így ez alapvetően egy agresszív protokoll.

Definíció – időbélyeg *(timestamp)*. 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, amelyek hatása a tranzakciók növekvő időbélyegei által meghatározott soros ütemezés hatásaival egyezik meg, akkor a tranzakciók indulási sorrendje valóban egy soros ekvivalens ütemezés lesz.

Megjegyzés. Érdemes összehasonlítani: soros ekvivalens ütemezés kétfázisú zárkezeléssel: a zárpontok szerinti növekvő sorrend a(z egyik) soros ekvivalens. Soros ekvivalens ütemezés időbélyegesen: a $t(T_{i_1}) < t(T_{i_2}) < \ldots < t(T_{i_n})$-nek megfelelő $T_{i_1}, T_{i_2}, \ldots, T_{i_n}$ a soros ekvivalens.

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.

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 (vagy csomópont) azonosítója is hozzáveendő (ld. 11.3. szakasz).

A sorosítási feltételek megsértése többféle modell alapján is detektálható. Itt is létezik:

- egyszerű modell,
- read/write modell,
- és további műveletek is megkülönböztethetők.

10.9.1. Időbélyeges tranzakciókezelés R/W modellben

Definíció – *olvasási és írási idő.* egy adott t pillanatban

- $R(A)$: az adategység olvasási ideje az A adategységet t-nél nem később olvasó tranzakciók időbélyegei közül a legnagyobb.
- $W(A)$: az adategység írási ideje az A adategységet t-nél nem később író tranzakciók időbélyegei közül a legnagyobb.

Az alábbi táblázat alapján eldönthető, hogy egy tranzakció műveleti igénye az A adategységre vonatkozóan engedélyezhető-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) ugyanaz, mint (2),

(5) ugyanaz, mint (1),

(6) egy később indult tranzakció már írta A-t, így T nem írhatja felül, azaz T-nek abortálnia kell.[8]

Megjegyzések.

1. Az időbélyeg tesztelése és maga a művelet oszthatatlan kell, hogy legyen.
2. A fenti táblázatban a $t(T) = W(A)$ vagy $t(T) = R(A)$ teljesülése nyilván akkor következhet be, ha maga T írta vagy 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 $t(T) \geq W(A)$,
3. WRITE elvégzendő, ha $t(T) \geq R(A)$ és $t(T) \geq W(A)$.

(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		

10.9.1.1. Az időbélyeges R/W modell és a 2PL összehasonlítása

Elképzelhető, hogy egy ütemezés sorosítható

- időbélyegesen, de kétfázisú zárakkal nem (Példa 1),
- időbélyegesen is és zárakkal is (pl. minden olyan ütemezés, amelyben a tranzakciók nem használnak közös adatokat),
- kétfázisú zárakkal, de időbélyegesen nem (Példa 2),
- időbélyegesen sem és zárakkal sem (Példa 3).

Példa *(1)*. Az időbélyeges protokoll szerint sorosítható (a megadott időbélyegekkel), de a(z RLOCK–WLOCK zárakat használó) 2PL protokoll szerint nem.

Az ütemezés nem sorosítható a 2PL szerint, mert ahhoz, hogy a megadott sorrendben fusson le, T_2-nek a (2) és (3) lépések között el kell engednie az A-n tartott zárat, majd a (4) és az (5) lépések között zárolnia kell B-t, így T_2 mindenképpen sérti a kétfázisú protokollt.

[8] A konkurencia növelése további ötletekkel lehetséges, ld. pl. 67. feladat (254. oldal)

	T_1 $t(T_1) = 10$	T_2 $t(T_2) = 20$	T_3 $t(T_3) = 30$
(1)	WRITE A		
(2)		WRITE A	
(3)			WRITE A
(4)	WRITE B		
(5)		WRITE B	

Példa *(2).* A 2PL protokoll szerint sorosítható, de időbélyegesen (a megadott időbélyegekkel) nem.

	T_1 $t(T_1) = 10$	T_2 $t(T_2) = 20$
(1)	READ A	
(2)		WRITE B
(3)	READ B	

Időbélyeges protokoll szerint történő futtatás esetén T_1 a (3) pontban abortál, mert $t(T_1) < W(B) = t(T_2)$.

Az ütemezés kétfázisú zárakkal sorosítható:

	T_1	T_2
(1)	RLOCK A	
(2)	READ A	
(3)		WLOCK B
(4)		WRITE B
(5)		UNLOCK B
(6)	RLOCK B	
(7)	READ B	
(8)	UNLOCK A	
(9)	UNLOCK B	

Soros ekvivalense a T_2, T_1 ütemezés.

	T_1	T_2
(1)		WRITE B
(2)	READ A	
(3)	READ B	

Példa *(3).* Nem sorosítható sem időbélyegesen, sem a 2PL protokoll szerint.

Kétfázisú zárakkal nem sorosítható, mert T_2-nek az (1) és (2) lépések között el kell engednie az A-n tartott (RLOCK) zárat, majd a (2) és (3) lépések között zárolnia kell (WLOCK). Ez sérti a kétfázisú protokollt. Időbélyeges futtatás esetén a (2) pontban abortál, mert $t(T_1) < R(A) = t(T_2)$.

	T_1 $t(T_1) = 10$	T_2 $t(T_2) = 20$
(1)		READ A
(2)	WRITE A	
(3)		WRITE A

Tanulság: sem a zárakkal, sem az időbélyegekkel való sorosítás nem jobb egyértelműen a másiknál.

10.9.2. Időbélyegek kezelése

Zár alapú tranzakciókezelésnél praktikus megközelítés, ha

- külön tároljuk a zárinformációt az adategységektől (*zártábla*, lock table), és
- feltételezzük, hogy az adategységek alapértelmezett állapota „nem zárolt". Tehát a zártáblában csak azokról az adategységekről tárolunk információt, amelyeken van zár.

Analóg módon: időbélyeges esetben az adategységek alapértelmezett időbélyege lehet $-\infty$, és csak az ettől eltérő értékeket kell explicit módon tárolni. Ezek az értékek megjelenhetnek közösen egy időbélyegtáblában, ami lehetőség szerint a memóriában tárolandó. A tábla nem nő a végtelenségig, mert a táblából kitörölhetők azok az időbélyegek, amelyek kisebbek, mint a futó tranzakciók legkisebb időbélyegének az értéke.

10.9.3. Tranzakcióhibák és az időbélyegek

Elképzelhető, hogy T_2 olvas T_1 által előállított értéket, majd később T_1 abortál bármely (korábban 1-3. pontok alatt hivatkozott) ok miatt. Ez a piszkos adat olvasásának esete, amikor tehát lavinaveszéllyel kell számolni.

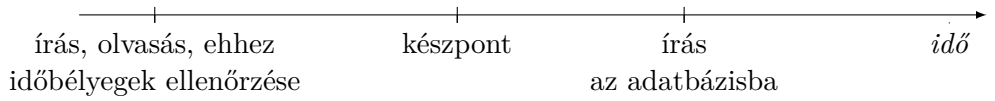
A megoldás:

1. elfogadjuk a lavinákat, hiszen az időbélyeges tranzakciókezelést tipikusan olyan környezetben használjuk, ahol kevés az abort, tehát a lavina még kevesebb, vagy
2. megakadályozzuk a piszkos adat olvasását pl. azzal, hogy nem írunk az adatbázisba, amíg a tranzakció el nem érte a készpontját (ld. *időbelyeges szigorú protokoll*, timestamp-based strict protocol).

Lépései az alábbiak *lennének*:

1. módosítások csak munkaterületen elvégezve
2. tranzakció eléri a készpontját
3. írások véglegesítése az adatbázison

Az írásokkal azonban baj van:



10.6. ábra. Lavina megelőzése időbélyeges tranzakciókezelés esetén

Az időbélyeg ellenőrzése a készpont 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 készpont 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 írási/olvasási igény megjelenik,

b) közvetlenül a készpont 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.

10.9.4. Verziókezelés időbélyegek mellett (MVCC) (multi-version concurrency control)

Feltételezés: minden adatelem írásakor a régi értéket is megőrizzük.

Kézenfekvő megoldás, ha idősor jellegű adatokat kívánunk tárolni (betegek adatai, tőzsdei árfolyamváltozások, szoftver projektek 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	
		$R(A_0)$	$W(A_0)$	$R(A_1)$	$W(A_1)$	$R(B_0)$	$W(B_0)$	$R(B_1)$	$W(B_1)$
$t(T_1) = 10$ READ A	$t(T_2) = 20$	0 10	0			0	0		
	READ A (3) WRITE B	20							20
READ B (1) WRITE A (2)					10	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 (pesszimista stratégia),
- hatékonyan támogatja a sok/hosszú olvasási tranzakciókat,
- több háttértárat igényel,
- bonyolultabb 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).

10.9.5. 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

10.10. A fejezet új fogalmai

többfelhasználós, ill. konkurens működés, fantom olvasás, nem megismételhető olvasás, elveszett adatmódosítás, piszkos adat olvasása, ACID, atomicitás, adat-

bázis konzisztencia, tranzakciók izolációja, tartósság (durability), zár (privilégium, szinkronizációs primitív), patt (deadlock, holtpont), éhezés, várakozási gráf, zármódok, tranzakció modell, ütemezés, soros/nem soros, ill. sorosítható/nem sorosítható ütemezések, soros ekvivalens, sorosítási (precedencia) gráf, zár kompatibilitási mátrix, (zár)protokollok, 2PL, időbélyeg, soros ekvivalens időbélyeges tranzakciókezelés esetén, tranzakciós teljesítmény, agresszív protokoll, konzervatív protokoll, tranzakcióhiba, rendszerhiba, médiahiba, kész (commit) pont, helyreállítás (recovery), tranzakciós napló, redo, undo, checkpoint, szigorú protokoll, redo naplózás, redo helyreállítás, MVCC, szigorú időbélyeges tranzakciókezelés

11. fejezet

Elosztott adatbázisok

Definíció – elosztott adatbázis *(distributed database)*. *Csomópontok* (nodes, sites) összessége, amelyekben egy-egy számítógép üzemel saját CPU-val, háttértárral és adatbázis-kezelővel.[a] A csomópontok egymással össze vannak kötve adatcserére alkalmas módon úgy, hogy az adatbázis-kezelők felhasználói csupán egyetlen, logikailag egységes adatbázist érzékelnek.

[a] Az adatbázis-kezelők természetesen autonóm módon is képesek működni

Sebezhetőbb az elosztott rendszer, mert a csomópontokon kívül a linkek változatos hibái is befolyásolhatják a működést. Ezért 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.

Az 1. esetén problémaként merül fel a másolati példányok írásának kérdése, amely atomi kell, hogy legyen minden másolatot beleértve. Külön megfontolást igényel az az eset, amikor egy csomópont elérhetetlen, ilyenkor ugyanis az adatbázis működése nem állhat meg.

Definíció – globális (logikai) adat *(global data unit)*. 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.

Példa *(1).* Egy (globális) adatelem zárolása egyetlen logikai művelet (aminek ráadásul oszthatatlannak kellene lennie), amely a lokális adategységekre visszavezetett fizikai zárakon keresztül valósul(hat) meg. Eközben jelentős idő telhet el! A felmerülő problémákat ld. 11.1. szakasz. Jelölés: ha A a logikai adatelem, akkor A_i-k a fizikai adatelemek.

Példa *(2).* Hasonlóan: egy *globális (logikai) tranzakció* a valóságban különböző csomópontokban futó *lokális (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. 11.2. szakasz). Jelölés: ha T a globális tranzakció, akkor T_i-k a lokális tranzakciók.

11.1. Elosztott zárak

A lokális példányokon a zárakat úgy kell elhelyezni, hogy támogassák a globális példányok zárolását.

Az 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 teljesíthetőségét 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

- kontrollüzenetek, melyek rövidebbek, olcsóbbak vagy
- adatüzenetek, melyek hosszabbak, így drágábbak.

11.1.1. A WALL (write locks all) protokoll

A zárak és szemantikájuk a lokális példányokon pontosan megegyezik az 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.

A WALL protokoll globális zárkompatibilitási mátrixa az alábbi

	RLOCK	WLOCK
RLOCK	igen	nem
WLOCK	nem	nem

Azaz formailag megegyezik a lokális adategységekre értelmezett zárkompatibilitási mátrixszal. Ezt a továbbiakban bizonyítjuk.

Tétel. A WALL protokoll globális zárkompatibilitási mátrixa megegyezik a lokális adategységekre értelmezett zárkompatibilitási mátrixszal.

Bizonyítás.

A fenti protokoll következtében két különböző tranzakció nem tarthat fenn WLOCK-ot ugyanazon logikai adategységen. Ha T_k már zárolta A-t, akkor az összes A_i-n T_k: WLOCK A_i érvényes kell, hogy legyen, így T_l nem tud egyetlen A_i-re sem (sem író, sem olvasó) zárat rakni. Emiatt T_k: WLOCK A nem kompatibilis sem T_l: WLOCK A-val, sem T_l: RLOCK A-val. Másrészt, ha T_k: RLOCK A érvényes, akkor T_k: RLOCK A_i áll fenn valamely A_i-re, amely kompatibilis T_l: RLOCK A_i-vel, tehát T_k: RLOCK A kompatibilis T_l: 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,
– 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 készpont képzése során (ld. 11.2.3. alszakasz).

A globális RLOCK-hoz elég egyetlen csomópontba kontrollüzenetet küldeni (tudjuk, hol vannak a példányok, ezekből egy tetszés szerint kiválasztható), innen pozitív válasz esetén egyetlen adatüzenet küldendő.

11.1.2. Többségi zárolás (majority locking)

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 zárkompatibilitási mátrix azonos a WALL protokoll mátrixával.

Bizonyítás.

- T_k: WLOCK A_i (lokálisan) nem kompatibilis T_l: WLOCK A_i-val, így egyidőben nem lehet A lokális példányainak többségét zárolni. Emiatt (a globális) T_k: WLOCK A nem kompatibilis T_l: WLOCK A-val.
- T_k: WLOCK A nem kompatibilis T_l: RLOCK A-val, mert T_k: WLOCK A-hoz A példányainak többségén WLOCK van, ami nem kompatibilis RLOCK-kal, így a többségen már nem lehet RLOCK. Emiatt T_k: WLOCK A nem kompatibilis T_l: RLOCK A-val és T_k: RLOCK A nem kompatibilis T_l: WLOCK A-val.
- T_k: RLOCK A_i kompatibilis T_l: RLOCK A_i-vel, így egyidőben több tranzakció is zárolhatja az A_i-k többségét, és globálisan T_k: RLOCK A is kompatibilis T_l: RLOCK A-val.

A többségi zárolás hatékonysága:

WLOCK A létesítéséhez legalább az n példány többségének, $\lceil (n+1)/2 \rceil$ példánynak kell kontrollüzenetet küldeni, legalább $\lceil (n+1)/2 \rceil$ válasz- (kontroll-) üzenet jön vissza, majd n adatüzenettel mindegyik lokális példányt írni kell.

RLOCK A-hoz legalább $\lceil (n+1)/2 \rceil$ csomópontnak kell kontrollüzenetet küldeni, amelyre $\lceil (n+1)/2 \rceil$ válasz érkezik. A válaszokból (legalább) az egyik adatüzenet, ez tartalmazza a kért adategység értékét.

Hozzávetőleges összehasonlító táblázat a WALL-lal (számos részlettől eltekintettünk):

	adat üzenet	kontroll üzenet
WALL írás	n	$2n$
többségi írás	n	$\geq n+1$
WALL olvasás	1	1
többségi olvasás	1	$\geq n+1$

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).

11.1.3. k az n-ből protokoll

Ez a módszer az előző kettő általánosítása.

n = csomópontok száma, továbbá $\left\lceil \frac{n+1}{2} \right\rceil \leq k \leq n$

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.
3. $k = n$ esetén megfelel a WALL protokollnak, $k = \left\lceil \frac{n+1}{2} \right\rceil$ esetén pedig a többségi protokollnak.

k alkalmas megválasztásával a protokoll „hangolható”.

11.1.4. 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 minden példányának elérhetőségét egyetlen X_A csomópont zármenedzsere felügyeli. A különböző adategységekre ez a csomópont általában különböző, és tipikusan olyan csomópont, amelyen van a kiválasztott adategységnek másolata (ez lesz az A adategységnek az *elsődleges példánya*).

Hatékonysága:

1. WLOCK A-hoz kell egy kérés- (kontroll-) üzenet X_A-ba, erre jön egy válasz, majd (jó esetben) írandó A-nak mindegyik másolata.
2. RLOCK A-hoz kell egy kontrollüzenet 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. Az elsődleges példány így tehát *egyszeres hibapont* (Single Point of Failure, SPOF).

Az is elképzelhető, hogy mindegyik adategység felett ugyanaz a csomópont rendelkezik (*centrális csúcs módszere*). Ebben az esetben az adategység elsődleges

példánya nem biztos, hogy a centrális csúcson helyezkedik el, ekkor a centrális csúcs és az elsődleges példány között íráskor és olvasáskor plusz egy kontroll üzenet küldése szükséges.

11.1.5. 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 RT(A) tokent, 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 *mindegyik* (m db) csomópontba, amely az elosztott adatbázishoz tartozik $\rightarrow$ m db kontrollüzenet

A csúcsok válaszolnak *a)* vagy *b)* üzenettel ($\rightarrow$ m db kontrollüzenet), 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 mindegyik 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.

11.1.6. Összefoglaló

A zárkezelési protokolloknál a kontrollüzenetek szükséges száma a 11.1 táblázatban látható, ahol m az elosztott rendszer csomópontjainak száma, n csomópontban van példány az A adategységből. Az ismertetett módszerek csak a zároláshoz szükséges kontrollüzenetek számában különböznek, az adatüzenetek száma mindegyik esetén íráskor n, olvasáskor 1.

	kontrollüzenet		megjegyzés
	íráskor	olvasáskor	
WALL	$2n$	1	jó, ha sok az olvasás
többségi zárolás	$\geq n+1$	$\geq n+1$	jó, ha sok az írás
elsődleges példányok	2	1	hatékony, de sebezhető
elsődleges példányok tokennel	$0 \ldots 3m$	$0 \ldots 3m$	adaptív
centrális csúcs	3	2	nagyon sebezhető, centralizált hálózati forgalmat okoz

11.1. táblázat. A zárkezelési protokolloknál a kontrollüzenetek szükséges száma

11.2. 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 *(distributed serializability)*. Tranzakciók egy ütemezése egy elosztott adatbázison sorosítható, ha hatása a logikai adategysé-

geken 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.

csomópont 1

T_{11}	T_{21}
WLOCK A	
UNLOCK A	
	WLOCK A
	UNLOCK A

csomópont 2

T_{12}	T_{22}
	WLOCK B
	UNLOCK B
WLOCK B	
UNLOCK B	

A lokális sorosítási gráfok:

$T_1 \to T_2$ $\qquad$ $T_1 \leftarrow T_2$

A globális sorosítási gráf:

$$T_1 \rightleftarrows T_2$$

Azaz a sorosítási gráfban kör van, a globális ütemezés nem sorosítható, mégha a lokális sorosíthatóság fenn is áll a lokális kétfázisúság következtében.

11.2.1. Elosztott kétfázisú zárolás

A globális sorosíthatósághoz nyilván elégséges feltétel, hogyha a globális tranzakciók *globálisan kétfázisú*ak, azaz egy T_i tranzakció egyetlen T_{ij} lokális tranzakciója sem engedhet el egyetlen zárat sem, ameddig bármelyik T_{ij} még kérhet új zárat.

A megvalósítás úgy képzelhető el, ha a lokális tranzakciók üzenetváltásokkal informálják egymást arról, hogy elérték a zárpontjukat, több zárra nincs szükségük. Azaz kell egy *közös zárpont*.

11.2.2. Szigorú kétfázisú zárolás

A lavinák problémája az elosztott környezetben is megmaradt. Elkerülése itt is lehetséges, ha biztosítani tudjuk, hogy egyetlen lokális tranzakció se írjon az adatbázisba mindaddig, amíg minden lokális tranzakció el nem érte a készpontját. A lavinák ellen tehát *közös készpont*ot kell létrehozni, amihez az kell, hogy minden lokális tranzakció minden számítását befejezze, mindegyik zárját megkapja. Ha ez minden lokális tranzakcióra teljesül, akkor a tranzakció folytatható, egyébként pedig az összes lokális tranzakciónak abortálnia kell. Tehát a közös készpontot

megelőzi a közös zárpont elérése, vagy akár egybeeshet vele. Ha valamely protokoll tehát biztosítja a közös készpont képzését, akkor ez közös zárpontnak is megfelelő. A közös készpont megvalósítása különböző hibalehetőségeket is figyelembe véve egy *elosztott megegyezés*i feladatra vezet, amely költséges művelet. A következő szakasz részletesebben ismerteti.

11.2.3. Elosztott készpont képzése

Elsőként egy olyan (alap-)protokoll bemutatása következik, amely nem számol semmilyen hibával, tehát az üzenetek nem vesznek el, a csomópontok nem esnek ki. A későbbiekben ezt a protokollt fogjuk tovább bővíteni.

Feltételezések:

Adott egy T logikai tranzakció, amely számos T_i lokális tranzakcióból áll az N_i csomópontokban. *Koordinátornak (főnöknek)* nevezzük azt a csomópontot, ahol a tranzakciót kezdeményezték, a többi csomópont neve *résztvevő*. A protokoll célja, hogy közös döntésre jussanak a résztvevők, azaz vagy minden résztvevő abortálja a lokális tranzakcióját, vagy minden lokális tranzakció commitáljon.

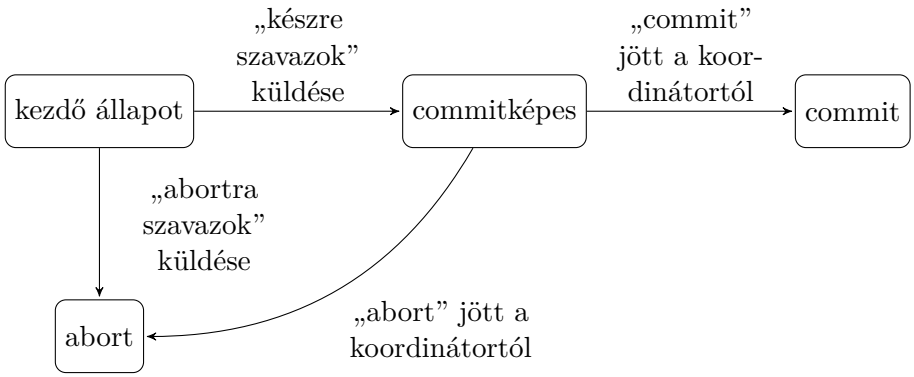
A résztvevők a lokális szavazatukat a koordinátornak egy üzenet formájában küldik el, amely ezekből a szavazatokból egy globális döntést hoz. A lokális szavazatok a következőképpen alakulnak:

- a csomópont commitra szavaz akkor, ha a lokális tranzakció elérte a készpontját,
- a csomópont abortra szavaz akkor, ha bármely ok miatt a lokális tranzakció abortálni kényszerült.

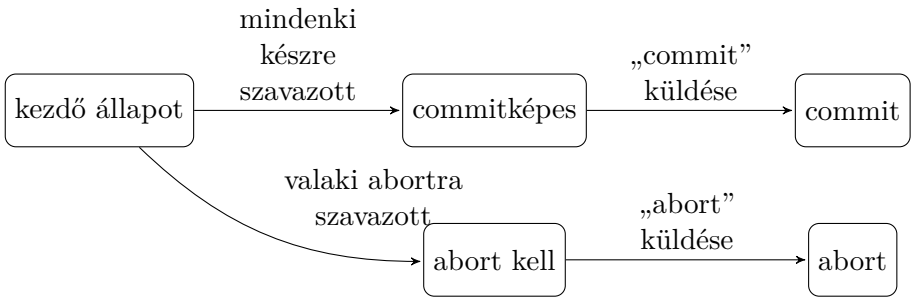
A koordinátor ezután két dolgot tehet:

- Ha minden résztvevőtől „készre szavazok” üzenetet kapott, akkor „commit” üzenetet küld minden résztvevőnek. Ez alapján a résztvevők egységesen tudnak egy globális készpontot kialakítani, hiszen minden résztvevő értesül a többi résztvevő commit döntéséről.
- Ha bárhonnan is „abortra szavazok” üzenetet kap, akkor a tranzakciónak is abortálnia kell, ezért „abort” üzenetet küld minden résztvevőnek, akik ennek hatására abortálják a lokális tranzakciókat.

A protokoll segítségével így egységes döntésre juthatnak a résztvevők. A csomópontok különböző állapotai és állapotátmenetei a következő ábrákon láthatóak.



11.1. ábra. Egy résztvevő állapotai



11.2. ábra. A koordinátor állapotai

Megjegyzések.

1. A koordinátor is résztvevő egyben, ő is küld magának üzeneteket, csak éppen ezek nem költséges üzenetek.
2. Ha nem lenne koordinátor, csak résztvevők, akkor mindenki mindenkinek kellene, hogy küldjön üzenetet. Ilyenkor az üzenetek száma n^2-tel lenne arányos.
3. A most bemutatott protokoll nem kezeli a hibákat és könnyű azt is belátni, hogy egy csomópont kiesése esetén a protokoll nem garantálja az egységes döntést. Ha például egy csomópont commitra szavazott, és nem kap üzenetet a koordinátortól, akkor nem léphet tovább, hiszen
   - commit állapotba nem léphet, mert jöhet „abort” üzenet,
   - abort állapotba sem léphet, mert jöhet „commit” üzenet.

   Ilyenkor a csomópont még a zárait sem engedheti el, ellenkező esetben sérülhet a globális kétfázisúság.
   A fent említett jelenség neve *blokkolás*, kiküszöbölése a gyakorlatban is használható protokollok egyik alapproblémája.

11.2.3.1. A kétfázisú kész protokoll (2PC protokoll)

Az előző pontban bemutatott protokoll egy továbbfejlesztett változata a *kétfázisú commit* (two-phase commit, 2PC) protokoll néven vált az iparban is elterjedtté, mert az alábbi hibalehetőségeket képes kezelni:

- Elvesző üzenetek hálózati hiba miatt.
- Kieső csomópont, amely később esetleg újraindulhat, de meghibásodása esetén hallgat, nem küld hamis üzenetet.

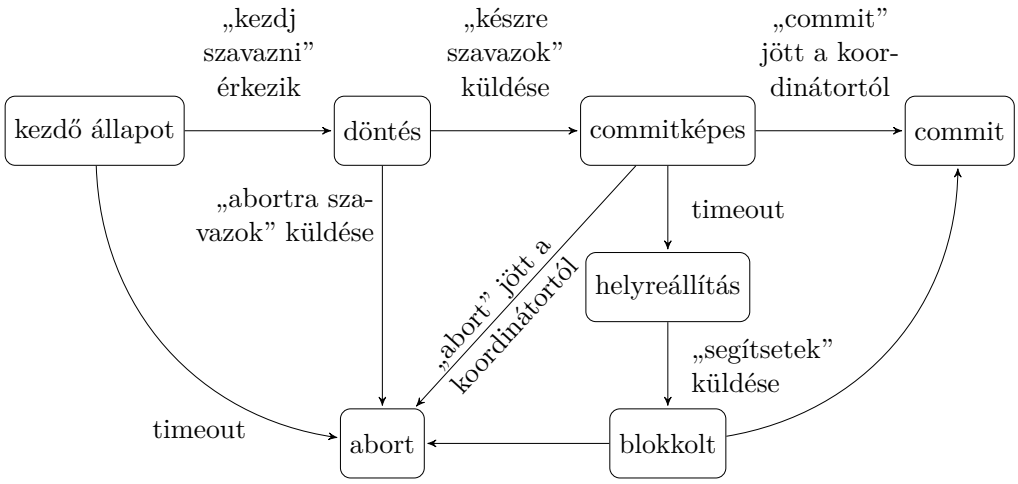
Az alábbi hibák fellépésével nem kell számolnunk:

- Üzenetek sorrendje felcserélődik.
- Nem valós üzenetek generálódnak.

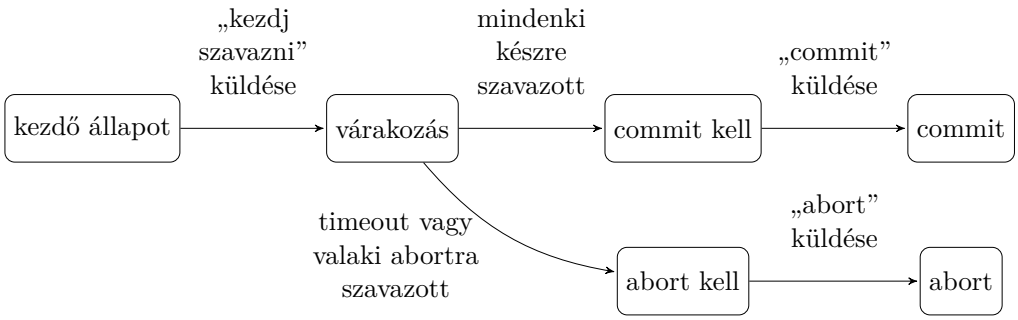
A protokoll ezen felül feltételezi, hogy a csomópontok valamilyen stabil tárba képesek naplózni, amelyek a kiesések következtében nem vesznek el. Ez által garantálható, hogy a csomópont a kiesés utáni újraindításkor ismeri az utolsó állapotát és képes helyreállító lépéseket tenni.

A hibamentes esethez képest a következő jelentősebb változtatások jelennek meg:

1. A lokális tranzakciók mérik azt az időt, hogy mióta várják a választ a szavazatukra. Ha túl sok idő telik el (timeout), akkor a rendszerben hiba történt, amelyet valamilyen módon kezelni kell. Ha ez a commitképes állapotban történne meg, akkor egy helyreállító állapotba kell átlépni.
2. A koordinátor üzenetküldéssel szólítja fel a résztvevőket a szavazás megkezdésére.
3. Ha egy résztvevő meghibásodik, és ezért ideiglenesen kiesik, akkor újraindulása során egy stabil tárba mentett naplóból képes emlékezni, hogy hozott-e már korábban döntést. Ha korábban abort vagy commit döntést hozott, akkor a megfelelő állapotba lép újraindulása után. Amennyiben még nem hozott döntést, akkor az 1. pontban említett helyreállító állapotba lép.



11.3. ábra. Egy résztvevő állapotai a 2PC protokollban



11.4. ábra. A koordinátor állapotai a 2PC protokollban

A helyreállítás folyamata a következőképpen zajlik:

- Ha a résztvevő belép a „helyreállítás” állapotba, akkor „segítsetek” üzenetet küld az összes többi résztvevőnek (ehhez célszerűen ismernie kell őket). Erre az alábbi válaszok jöhetnek:
  - Commit üzenet, mert ha a résztvevő már commit állapotban van, akkor ezt az üzenetet kell küldenie.
  - Abort üzenet, abort állapotban lévő résztvevőtől vagy olyantól, aki még kezdő állapotban van.
- Ha a résztvevő commitképes állapotban van, akkor nem tud hasznos üzenetet küldeni, így nem is küld semmit. Ennek oka az, hogy ebben az állapotban nem tudja, hogy más résztvevő szavazott-e már esetleg abortra.
- A „segítsetek” üzenetek elküldése után, a résztvevő várakozik a válaszra. Ilyenkor a résztvevő blokkolt állapotban van, amelyből a beérkező üzenetek alapján helyes döntést fog hozni, hiszen:
  - nem kaphat commit és abort választ is a „segítsetek” üzenetére,

- nem kaphat abortot, ha a koordinátor már globális commitot küldött,
- nem kaphat commitot, ha a koordinátor már globális abortot küldött.

– Amennyiben nem jön semmilyen üzenet a segítségkérésre – mert pl. a többi résztvevő még commitképes állapotban van, vagy szintén segítségre várnak –, akkor a blokkolás a 2PC-nél is bekövetkezhet. Ez például megtörténhet akkor, ha a koordinátor kiesik, közvetlen miután minden résztvevő commit szavazatot küldött (vagy ha minden a döntésről már értesült résztvevő is kiesik).
– A koordinátor a várakozás állapotból timeouttal abort állapotba kerülhet, ha valamely résztvevő hosszú idő után sem válaszol. Ekkor ugyanis joggal feltételezhető, hogy a résztvevő nem elérhető vagy meghibásodott. Ilyenkor minden aktív résztvevő abort állapotba kerül. Amennyiben a meghibásodott résztvevő később újraindulna, akkor a többi, már abort döntést hozott résztvevőtől erről értesülni fog.
– A résztvevő, ha hosszú idő után sem kap „kezdj szavazni” üzenetet, akkor abort állapotba kerül, mert azt feltételezi, hogy a főnök elérhetetlenné vált vagy meghibásodott. Ebben az állapotában már elengedheti a zárait, ha pedig mégis megjön a „kezdj szavazni” üzenet, akkor egyszerűen abortra szavaz.

Megjegyzés. Egy lokális tranzakció kész vagy abortált állapotában is jöhetnek kérések, ha más lokális tranzakció segítséget kér. Ilyenkor a napló alapján van lehetőség korrekt választ adni.

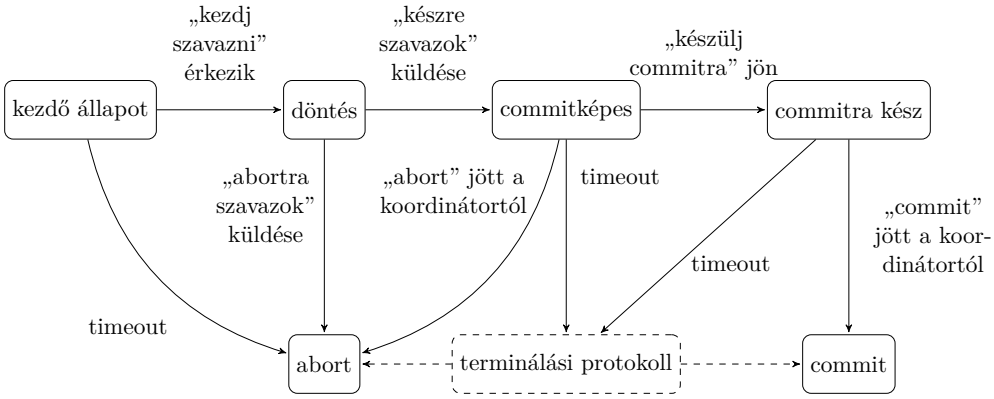
Összefoglalva tehát a 2PC protokoll egyes üzenetek elvesztését, egy csomópont ideiglenes kiesését, majd újraindulását képes helyesen kezelni, azonban blokkolás mégis előfordulhat, így van létjogosultsága a következőkben bemutatott 3PC protokollnak.

11.2.3.2. Egy „blokkolásmentes” kész protokoll – 3 fázisú kész protokoll (3PC)

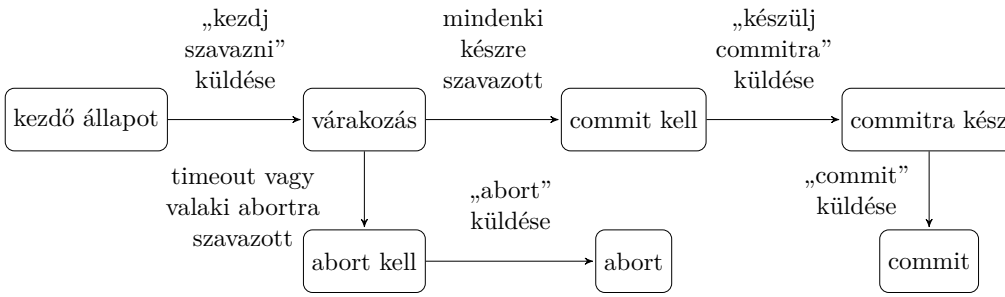
A háromfázisú commit (3PC) protokoll is csak bizonyos hibák esetén blokkolásmentes. Valójában semmilyen protokoll nem képes minden hibára felkészülni, hiszen ha a kapcsolatok véglegesen megszakadnak, vagy hamis üzenetek jelennek meg a rendszerben, akkor a protokollok elvileg sem tudnak működni. A 2PC-nél a blokkolást az okozza, ha valamely résztvevő nem tudja, hogy mindenki készre szavazott, így előfordulhat, hogy minden résztvevő a készre szavaz, de ezt követően a koordinátor kiesik. Így a működő résztvevők akár a commit állapotba is léphetnének, de ezt nem teszik meg – blokkolódik a rendszer.

Ezért a 3PC-nél commit állapotba csak akkor kerülhet egy résztvevő, ha már tudja, hogy mindenki tudja, hogy mindenki commitra szavazott. Mindez egy újabb üzenet-körrel realizálható (ez lesz a harmadik fázis), az ehhez tartozó újabb állapot

a „commitra kész”. Ennek a jelentése, hogy a rendszer globális commit döntést fog hozni, ha senki nem hibásodik meg.



11.5. ábra. Egy résztvevő fontosabb állapotai a 3PC protokollban



11.6. ábra. A koordinátor állapotai a 3PC protokollban

A három fázis értelmezése:

- Az első fázis a 2PC-ből már ismert, szavazásra felszólítás, majd szavazás.
- A második fázisban a koordinátor – amennyiben minden résztvevő commit szavazatot küldött – egy commitra készülj üzenetet küld a résztvevőknek, majd ezt követően nyugtára vár a résztvevőktől. Egy nyugta azt jelzi, hogy a résztvevő tudja, hogy mindenki commitra szavazott. Amennyiben minden résztvevő nyugtázta a commitra készülj üzeneteket, akkor *a koordinátor ebből már megtudja*, hogy már minden résztvevő tudja, hogy mindenki commitra szavazott.
- Ezt követően a harmadik fázisban a koordinátor commit üzenetet küld a résztvevőknek, amiből *már minden résztvevő is megtudja*, hogy mindenki tudja, hogy globális commit lesz.

A korábbi blokkolás megoldására egy ún. *terminálási protokollt* vezettek be a 3PC-be, amely a hibakezelést valósítja meg. Amennyiben commitképes állapot-

ban vagy commitra kész állapotban timeout történik (azaz nem érkezik üzenet a koordinátortól), akkor a hibakezelés az alábbi módon zajlik:

1. A működő csomópontok egy új koordinátort választanak.
2. Az új koordinátor bekéri a résztvevők aktuális állapotait.
3. A beérkező információk alapján döntést hoz az új koordinátor (és ezt közli a résztvevőkkel):
   a) Ha van abortált vagy még nem szavazott résztvevő, akkor globális abort döntés születik.
   b) Ha van commitált résztvevő, akkor globális commit döntés születik.
   c) Ha mindenki commitképes állapotban van, akkor abort döntés fog születni.
   d) Ha van commitképes és commitra kész állapotú résztvevő is, akkor a commitképeseknek commitra készülj üzenetet kell küldeni, majd a nyugták után minden résztvevőnek commit üzenet kell küldeni.

Megjegyzés. Belátható, hogy csak a fenti négy eset egyike történhet meg, nem érkezhetnek vegyesen abort és commit üzenetek.

11.3. 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 minden 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 adat-hozzáférési 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 $\mathrm{R}(A_i)$, $\mathrm{W}(A_i)$ vizsgálatából, míg WRITE A esetén az összes $\mathrm{R}(A_i)$, $\mathrm{W}(A_i)$ 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.

11.4. 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 mindegyik 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 folytatja 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).

11.5. Elosztott pattok keletkezése és kezelése

Patthelyzetek természetesen elosztott környezetben is előfordulhatnak. A keletkezésük analóg az elosztott tranzakciók sorosíthatóságának alakulásával (ld. a 11.2. 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 a „k az n-ből” 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.

11.6. A fejezet új fogalmai

elosztott adatbázis, ill. adatbázis-kezelő, lokális vagy fizikai adat, globális vagy logikai adat, globális vagy logikai tranzakció, lokális vagy fizikai (rész-) tranzakció, globális zár, lokális zár, adatüzenet, kontrollüzenet, globális zárkompatibilitási mátrix, elsődleges adatpéldány, centrális csúcs, olvasási token, írási token, elosztott sorosíthatóság, elosztott 2PL, globális zárpont, globális készpont, 2PC protokoll, blokkolás, 3PC protokoll, elosztott időbélyeg, elosztott patthelyzet

12. fejezet

Gyakorló feladatok

A példatár után a feladatok mintegy feléhez található megoldás. A megoldások szándékosan elkülönülnek a feladatok szövegétől, mert a példatár javasolt és kívánatos felhasználása a feladatok megoldásainak önálló megkeresése, majd csak ezután a közölt megoldás tanulmányozása.

Egyes feladatokhoz részletes megoldási útmutatót is adunk, másoknál csak végeredményt. Jelmagyarázat: ● részletes megoldás, ◐ csak végeredmény, ○ nem adtunk meg megoldást.

12.1. ER-diagramok

1. Adott a következő informális leírás:
   Egy páciensnek számos betegsége is lehet, vannak betegségek, amikben pillanatnyilag senki sem szenved. Minden pácienst egyetlen mentőállomáson kezelnek, akár több orvos is. Az orvosoknak több páciensük 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! ●
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 informális leírás:

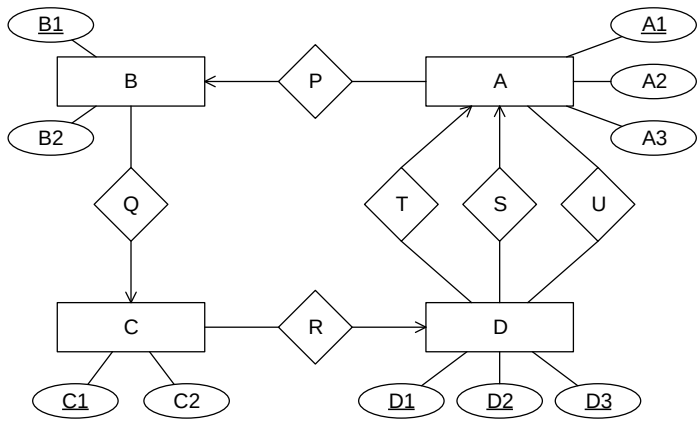
Egy kórház számos osztályból áll, mindegyiknek van egy osztályvezető főorvosa és akárhány főorvosa. Ha nincs osztályvezető főorvos, akkor van megbízott osztályvezető. Ők valamennyien a kórház – orvosdiplomával is rendelkező – alkalmazottai, másik kórházban nincs állásuk. Egy kórháznak ezen kívül még 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 az 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! ●

4. Hogyan alakítható át egy ternáris kapcsolatot tartalmazó ER-diagram ekvivalensen csak bináris kapcsolatokat tartalmazó ER-diagrammá? ●

12.2. Relációs sémaábrázolás, relációs algebra

5. Alakítsa át az alábbi ER-diagramot relációs sémákba! Törekedjen minél kevesebb séma kialakítására! ○



6. A feladat során az 1., ill. a 3. feladatban kapott ER-diagramokon dolgozzon.
   a) A diagramokat alakítsa át 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 (f, c) és (d, e) is

elemei. Tudjuk továbbá, hogy a természetes illesztésükkel kapott relációnak pedig elemei az alábbiak is:

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) A kulcs R-ben,
   b) B kulcs R-ben,
   c) B kulcs R-ben és S-ben is,
   d) 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?
    a) $\pi_X(r \cup s) = \pi_X(r) \cup \pi_X(s)$
    b) $\pi_X(r \setminus s) = \pi_X(r) \setminus \pi_X(s)$ ○
11. Adott a következő sémaleírás, adjon relációs algebrai kifejezéseket a kérdésekre!
    – *TERMÉK*(<u>*GYÁRTÓ*</u>, <u>*MODELL*</u>, <u>*TÍPUS*</u>) (az összes attribútum együtt alkotja a kulcsot)
    – *PC*(<u>*MODELL*</u>, *CPU_SEBESSÉG*, *MEMÓRIA*, *MEREVLEMEZ*, *CD*, *ÁR*)
    – *LAPTOP*(<u>*MODELL*</u>, *CPU_SEBESSÉG*, *MEMÓRIA*, *MEREVLEMEZ*, *KÉPERNYŐ*, *ÁR*)
    – *NYOMTATÓ*(<u>*MODELL*</u>, *SZÍNES*, *TÍPUS*, *ÁR*)

    A *CPU_SEBESSÉG* mértékegysége MHz, a *MEMÓRIA* és a *MEREVLEMEZ* attribútumoké GB.
    Kérdések:
    a) Melyek azok a PC modellek, amelyeknek sebessége legalább 2 GHz?
    b) Mely gyártók készítenek legalább 1000 gigabájtos merevlemezű laptopot?
    c) Adjuk meg a „B” gyártó által gyártott összes termék modellszámát és árát típustól függetlenül!
    d) Melyek azok a gyártók, akik laptopot gyártanak, de PC-t nem?

e) Melyek azok a gyártók, amelyek gyártanak legalább két, egymástól különböző, legalább 3 GHz-en működő PC-t vagy laptopot? (Nincs két azonos modellszám!)

f) Adjuk meg az összes 3 GHz-nél gyorsabb PC és laptop gyártóját!

g) Adjuk meg azon gyártókat, amelyek olyan laptopokat hoznak forgalomba, melyekkel megegyező tulajdonságú PC-ket is árulnak! ◐

12. Tekintsük az alábbi csillagflotta adatbázissémát:
    - *CSILLAGHAJÓ*(<u>*HAJÓNÉV*</u>, *ÉV*, *FAJ*)
    - *DOLGOZÓ*(*DOLGOZÓNÉV*, <u>*AZONOSÍTÓ*</u>, *SZÜLETÉS*)
    - *BEOSZTÁS*(<u>*AZONOSÍTÓ*</u>, <u>*HAJÓNÉV*</u>, *RANG*)

    Az egyes attribútumok jelentései rendre a következők:
    - *CSILLAGHAJÓ*: a hajó neve, gyártási éve, és az hogy melyik faj tervei alapján készült
    - *DOLGOZÓ*: a dolgozó neve, Csillagflotta-azonosítója, mikor született
    - *BEOSZTÁS*: melyik dolgozó, melyik hajón, milyen rendfokozatban dolgozik

    Adjunk relációs algebrai kifejezést, amely 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*)

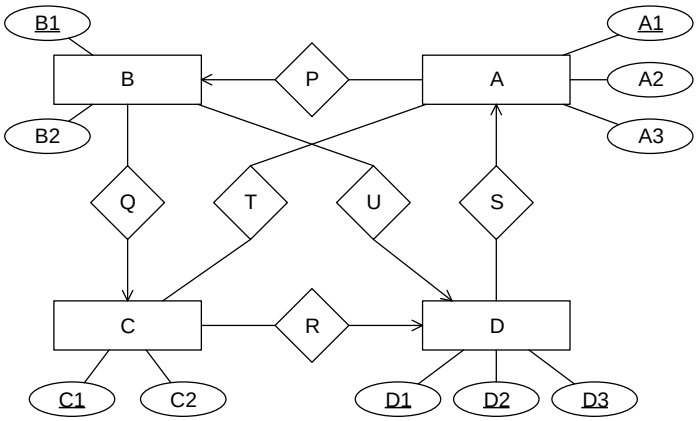
    Fejezze ki relációs algebra segítségével

    a) azon sörök összességét, amelyeket minden látogató kedvel azokban a sörözőkben, ahol kaphatók!

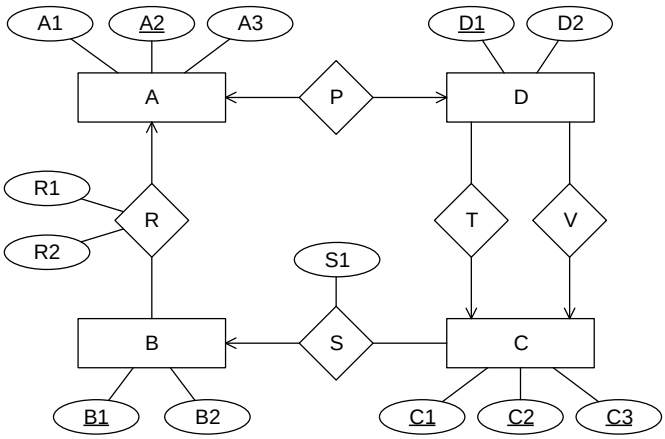
    b) azon személyek összességét, akik minden sört kedvelnek azokban a sörözőkben, amelyeket látogatnak! ◐

12.3. Hálós sémaábrázolás

14. Adott egy ternáris kapcsolat az attribútumaival. Milyen mezőket tartalmaz(nak) az ekvivalens hálós modellben a member típusú rekord(ok)? ◐
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-diagramot hálós sémába úgy, hogy minimális számú set típust definiáljon! Definiálja a rekord típusokat is, a definíciókat rendezze ábécé sorrendbe! ◐



17. Alakítson ki egy hálós sémát, amely 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 projekt(ek) amiben használják.
    - Cég: név, cím, vezető, dolgozók, projektek.
    - Projekt: elnevezés, vezető, érintett cég(ek), határidő, résztvevők. ○
18. Adott az alábbi ER-diagram.



a) Alakítsa át a diagramot hálós sémákba úgy, hogy a lehető legkevesebb set típust definiálja! A hálós séma elemeit a megfelelő ER-diagrambeli elemmel azonosan nevezze el! Rajzolja le a hálós sémát a tanult jelölésrendszert használva! Definiálja a rekordtípusok szerkezetét is!

b) 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 Zrt. 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 a 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 Zrt.-re is hat az információs társadalom, így szabványba foglalták, hogy a gyerekek jósága $1 \ldots 10$-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.

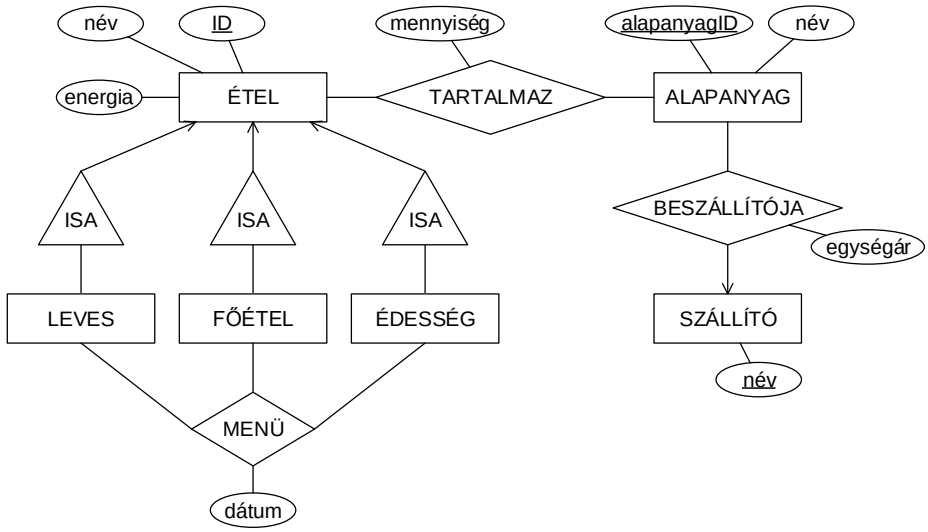
    A leírás alapján tervezzen a Nyusziadatbázishoz

    – ER-diagramot,
    – relációs sémát,
    – hálós sémát.

    Adjon relációs algebrai kifejezést a következő meghatározásához: Tapsi Hapsi saját maga szeretné megajándékozni a 10-es faktorú gyerekeket, és kíváncsi, hogy melyik körzetben laknak. ●

12.4. Relációs lekérdező nyelvek

20. A következő ER-diagramhoz illeszkedő relációs sémát definiáljon SQL-ben. (Hozzon létre megfelelő táblákat SQL utasításokkal.) Ne feledkezzen meg a különféle kényszerfeltételekről 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.)

    a) Fejezze ki sorkalkulussal azokat a tárgykódokat, amely tárgyakból csak olyan diákok szereztek jegyet, akik legalább egy tárgyból szereztek legalább elégségest.

    b) 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 $S(C, D, E)$ sémák és legyenek $\pi_{AC}(\sigma_{E=2}(r \bowtie s))$, illetve $\pi_{CD}(r) \cap \pi_{CD}(s)$ rájuk illeszkedő relációk. Fejezze ki ez utóbbiakat sorkalkulussal, oszlopkalkulussal! ○

23. Vizsgálja meg, hogy biztonságos-e az alábbi sorkalkulus kifejezés, ha $alma^{(1)} = \{2, 3\}$. ●

$$\left\{ x^{(1)} \middle| (\exists t^{(1)}) x^{(1)}[1] = t^{(1)}[1] \wedge t^{(1)}[1] > 2 \wedge ALMA^{(1)}(x^{(1)}) \right\}$$

24. Az R séma attribútumai (A, B, C, D, E), az S séma attribútumai pedig (A, B, F, G). Fejezze ki oszlopkalkulus segítségével $r \bowtie s$-et! ◐

25. Az r reláció attribútumai (A, B, C, D), az s-é pedig (C, D). Ekkor $r \div s$, r és s hányadosa azon (A, B) attribútumú t sorokból áll, melyekre igaz, hogy bármely s-beli u sor esetén a tu sor (a t és u összefűzésével kapott sor) r-ben van. Fejezze ki $r \div s$-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 autótípust szereti; a vevő melyik cégnél dolgozik, milyen beosztásban.

a) 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!

b) Vizsgálja meg, hogy a fent leírt kifejezés biztonságos-e! ○

12.5. Objektumorientált adatmodell

27. Az objektumorientált adatmodellnél megismert típuskonstruktorok segítségével készítse el az egyszerűsített „újság” típust! Tárolnunk kell az újság címét, a főszerkesztőt, a főszerkesztő-helyetteseket (sorrendhelyesen), a kiadót, a rovatokat a rovatvezetőkkel és rovatújságírókkal. ●

12.6. Fizikai szervezés

28. Milyen módszerekkel támogatható a több kulcs szerinti keresés? ○
29. Egy 25 000 rekordból álló állományt szeretnénk ritka index szervezéssel tárolni. A rekordhossz 850 bájt, egy blokk kapacitása (a fejrészt nem számítva) 4000 bájt. A kulcs 50 bájtos, egy mutató tárolásához 18 bájt kell.

    a) Legalább hány blokkra van szükség a teljes struktúra tárolásához?

    b) Mennyi ideig tart legfeljebb egy rekord tartalmának kiolvasása, ha az operatív tárban rendelkezésünkre álló szabad hely 6000 bájt? (Egy blokkművelet ideje 5 ms.)

    c) Segít-e a rekordhozzáférési idő csökkentésében, ha 10-szer ennyi szabad memóriával gazdálkodhatunk, melybe előzetesen tetszőleges blokkokat tölthetünk be? Mi a helyzet 100-szor ennyi szabad memória esetén? Hogyan célszerű a többletmemóriát felhasználni? ○

30. Egy 15 525 rekordból álló állományt szeretnénk ritka index szervezéssel tárolni. A rekordhossz 850 bájt, egy blokk kapacitása (a fejrészt nem számítva) 4000 bájt. A kulcs 50 bájtos, egy mutató tárolásához 18 bájt kell.

    a) Legalább hány blokkra van szükség?

    b) Mennyi ideig tart legfeljebb egy rekord tartalmának kiolvasása, ha az operatív tárban rendelkezésünkre álló szabad hely 5000 bájt? (Egy blokkművelet ideje 5 ms.)

    c) Segít-e a rekordhozzáférési idő csökkentésében, ha 10-szer ennyi szabad memóriával gazdálkodhatunk, melybe előzetesen tetszőleges blokkokat tölthetünk be? Mi a helyzet 100-szor ennyi szabad memória esetén? Hogyan célszerű a többletmemóriát felhasználni? ○

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 bájt,
    - egy blokk mérete 1000 bájt,
    - a kulcshossz 45 bájt,
    - egy mutató hossza 5 bájt. ◐

32. Egy 270 000 rekordból álló állományt akarunk tárolni. Két lehetőség közül választhatunk: vagy sűrű indexre épített egyszintes ritka indexet használunk, vagy 3 szintes ritka indexet. Melyik megoldást lehet kevesebb blokk felhasználásával megvalósítani, ha még azt is el szeretnénk érni, hogy sem az indexállományban, sem a főállományban ne legyenek 80%-nál telítettebb blokkok? Tudjuk, hogy egy blokk mérete 1900 bájt, egy rekord hossza 300 bájt, a kulcs hossza 35 bájt, a mutató hossza pedig 15 bájt. (Optimális tárolást feltételezünk, azaz azt, hogy az adatok a lehető legkevesebb blokkban helyezkednek el a fenti feltételek figyelembevétele mellett.) ◐

33. Egy adatbázisban egymilliárd rekordot akarunk tárolni. Egy rekord mérete 100 bájt, a blokkméret 4000 bájt. Egy blokkművelet hossza 5 ms. Két kulcs van, mindkettő 10 bájtos. A mutatók 32 bitesek. Az egyszerűség kedvéért feltételezzük, hogy egyszerre csak egy blokk fér el a memóriában, valamint a rekordokat a lehető legkompaktabb formában tároljuk.
    a) 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!
    b) Egy konkrét 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 annak érdekében, hogy az adatelérési idő megfeleződjön? ○

35. Egy adatstruktúrában hash-alapú tárolást építünk ki egy CD-lemezeket tároló adatbázisban. Minden CD-ről nyilvántartjuk, hogy képeket, zenéket, videót vagy adatot tárol, mindezt egy karakter típusú mezőben: `K`, `Z`, `V`, `A`. Milyen hash-függvényt célszerű választani, ha ezen mezőre szeretnénk alapozni a hash alapú adattárolást? Mi a mező kardinalitása? ○

36. Egy adatbázisban szeretnénk 1 000 000 rekordot tárolni vödrös hash szervezéssel. Egy rekord mérete 110 bájt, egy blokk 3000 bájt, egy kulcs 25 bájt, egy mutató pedig 64 bit méretű. Egy blokk elérésének ideje 5 ms. A rekordelérési idő max. 20 ms lehet. A vödörkatalógus befér a memóriába, a hash-függvény egyenletesen szór.
    a) Mennyi az átlagos rekordelérési idő?
    b) A vödörkatalógus hány bájtot foglal el a memóriában?

c) Mennyi többletmemóriára lenne szükség, hogy a rekordelérési idő a felére csökkenjen? ●

12.7. Funkcionális függések

37. Bizonyítsa be, hogy az alábbi három szabályból következnek Armstrong axiómái. (Azaz csak ezen három szabályt használva axiómaként levezethetők az Armstrong axiómák, mint „tételek”.)
Ha X, Y, Z, C egy relációséma attribútumhalmazai, akkor:

B1. $X \to X$

B2. Ha $X \to YZ$ és $Z \to C$, akkor $X \to YZC$.

B3. Ha $X \to YZ$, 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 $R(A, B, C, G, W, X, Y, Z)$ és $F = \{XZ \to BGYZ, AY \to CG, C \to W, B \to G\}$. Adja meg 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) $A \to B; C \to D \Rightarrow A \cup (C \setminus B) \to BD$,
    b) $A \to B; C \to D \Rightarrow C \cup (D \setminus A) \to BD$. ◐
44. Adott egy $R(A, B, C)$ sémára illeszkedő r reláció, amelynek 3 sora van. Bizonyítsa be, hogy meg lehet adni olyan nemtriviális funkcionális függést, amit r kielégít! ●

12.8. Normálformák

45. Mutassa meg, hogy egy 2NF sémára illeszkedő reláció 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 normálformában 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 normálformában 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 $R \setminus AB \to A$! ●
50. Állapítsa meg, hogy tartalmazhat-e redundanciát (funkcionális függések következtében) az (R,F) sémára illeszkedő reláció, és ha igen, akkor milyen jellegűt? $R(X,Y,Z,W)$, $F = \{XY \to Z, YZ \to W, X \to W, WY \to X\}$. ○

12.9. Relációs sémafelbontás

51. Adott egy (R,F) séma, ahol $R(A,B,C,D,E)$ és $F = \{AB \to C, D \to A, AE \to B, CD \to E, BE \to D\}$.
    a) BCNF-ben van-e ez a séma?
    b) 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 $R_1(L,M,O)$ legyen.
    a) Adjon meg egy ilyen felbontást!
    b) Van olyan megoldás, ami két részsémára bont? ◐
53. Adott az $R(A,B,C,D,E,F)$ relációs séma és az $F = \{A \to B, AC \to DB, C \to AD, AF \to ECB\}$ függéshalmaz. Adja meg a séma egy veszteségmentes, függőségőrző felbontását 2NF sémákba, törekedve minél kevesebb relációs séma definiálására! ●
54. Adott az $R(G,H,I,J,K,L)$ 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 meg, 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 $\sigma = (ACEFG, BCDE)$ 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, ahol $R(A,B,C,D,E)$ 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 relációs sémánk és annak egy nem függőségőrző felbontása. Ha a felbontásra illeszkedő 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 $\pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \pi_{R_3}(r) = r$, ahol $R(A, B, C, D, E, G, H)$ relációs séma, ha $F = \{AB \to C, AC \to D, C \to H, D \to B, E \to G\}$ függéshalmaz és $\sigma = \{R_1(ACE), R_2(EHG), R_3(BCDE)\}$ egy sémafelbontás. ○

12.10. Tranzakciókezelés

62. Tekintsük a T_1, T_2, T_3, T_4 tranzakciók alábbi ütemezését (az utasítások sorfolytonosan szerepelnek):
    - T_2: RLOCK A; T_3: RLOCK A; T_2: WLOCK B; T_2: UNLOCK A;
    - T_3: WLOCK A; T_2: UNLOCK B; T_1: RLOCK B; T_3: UNLOCK A;
    - T_4: RLOCK B; T_1: RLOCK A; T_4: UNLOCK B; T_1: WLOCK C;
    - T_1: UNLOCK A; T_4: WLOCK A; T_4: UNLOCK A; T_1: UNLOCK B;
    - T_1: UNLOCK C.

    Rajzolja meg az ütemezés precedencia (sorosítási) gráfját, és döntse el, hogy az ütemezés sorosítható-e! ◐
63. Rajzolja fel a precedenciagráfot! Használjon zárakat! Hogyan változik a gráf, ha kétfázisú a rendszer? ●

T_1	T_2
	WRITE A
WRITE B	
	WRITE B
WRITE A	

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, $\langle A$ értéke$\rangle$, $\langle A$ új értéke$\rangle$)
	naplóba(T, $\langle B$ értéke$\rangle$, $\langle B$ új értéke$\rangle$)
(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
$A = A \times 2$
WRITE A
COMMIT
UNLOCK A

66. Miért lehet előnyös zárakat is használni időbélyeges tranzakciókezelésnél? ○
67. *Fakultatív feladat:* Sorosítható-e az alábbi ütemezés időbélyeges (R/W) tranzakciókezelés használatával? ●

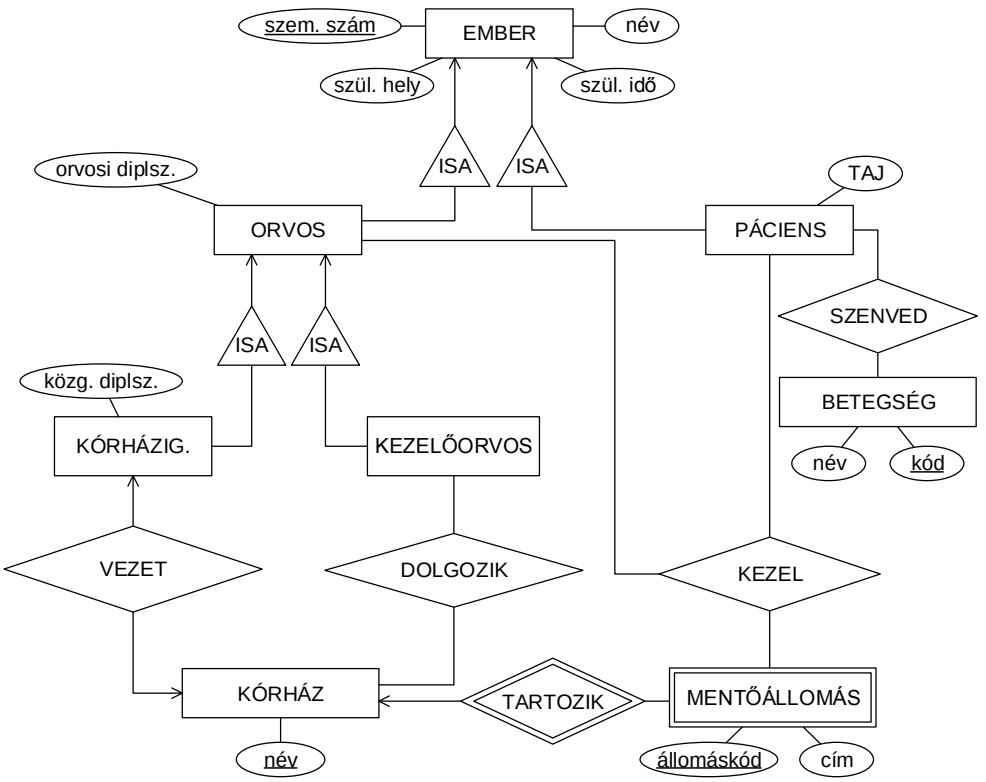
	T_1 $t(T_1) = 10$	T_2 $t(T_2) = 20$
(1)	READ A	
(2)		WRITE A
(3)	WRITE A	

13. fejezet

Gyakorló feladatok megoldásai

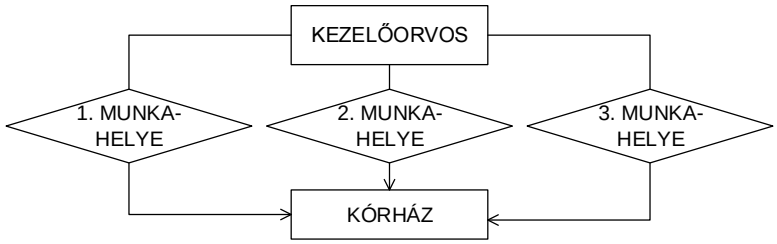
13.1. ER-diagramok

1. Az ER-diagram:



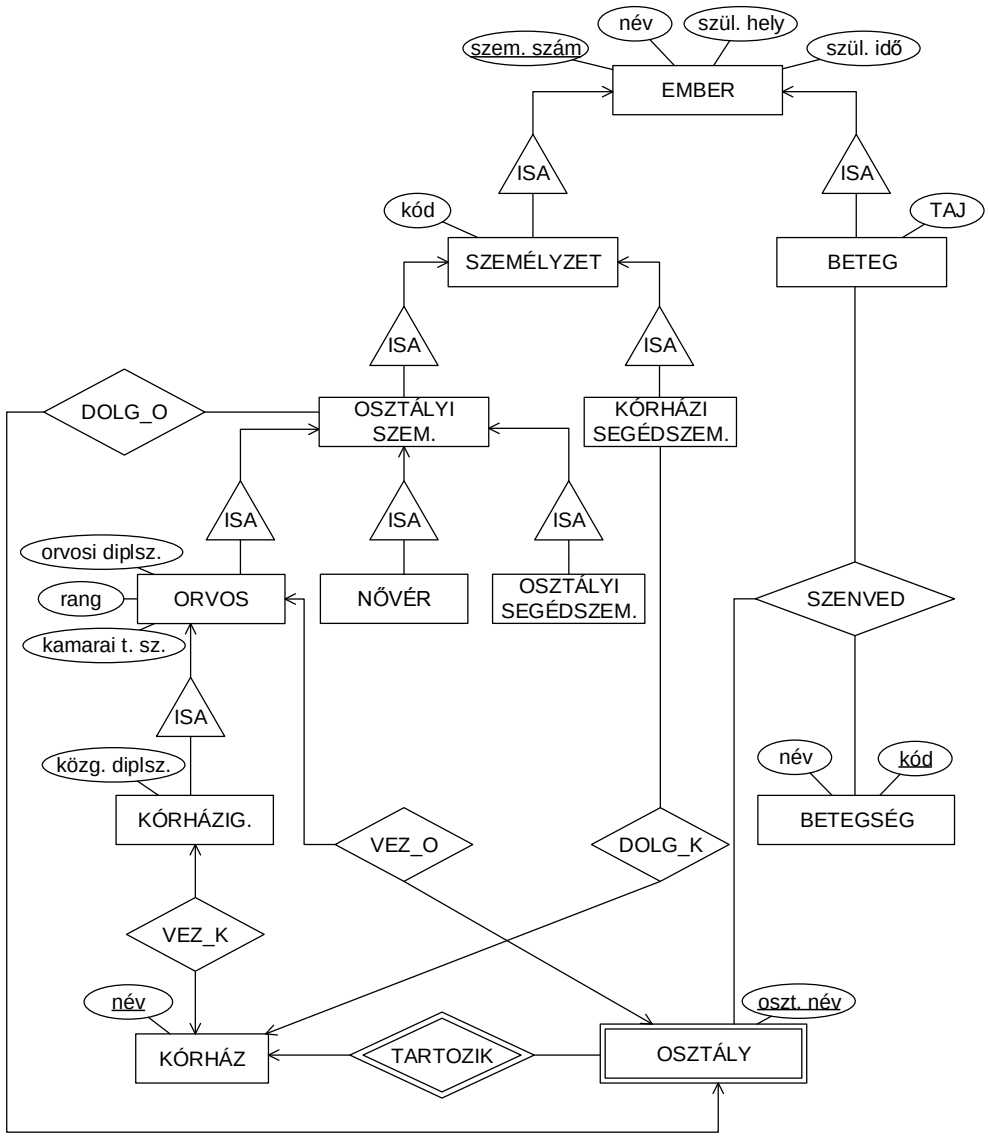
Megjegyzések.

1. A feladat nem specifikálta, hogy a mentőállomások azonosítója globálisan vagy csak kórházanként egyedi. A megoldásban a *MENTŐÁLLOMÁS* azért lett gyenge egyedhalmaz, mert így garantálhatjuk, hogy egy mentőállomás pontosan egy kórházhoz tartozzon.
2. A megoldás nem tudja ábrázolni azt, hogy a kórházigazgató a kórház egy dolgozója. Egy lehetséges megoldás: ahelyett, hogy az *ORVOS* egyedhalmazból származtatnánk kórházigazgatót és kezelőorvost, a *DOLGOZIK* kapcsolattípusból gyenge egyedhalmazt készítünk és ebből egy *IGAZGATÓ* – szintén gyenge – egyedhalmazt származtatunk, ami rendelkezik „közg. diplsz." attribútummal. A megoldás hátránya, hogy nem köti meg a kórházigazgató egyediségét.
3. A ternáris kapcsolattípusok kardinalitását nem definiáltuk, ezért a *KEZEL* kapcsolattípus nem tudja ábrázolni azt, hogy egy pácienst pontosan egy mentőállomáson kezelnek.
4. Azt, hogy egy orvost legfeljebb 3 kórház alkalmaz, nem tudjuk ábrázolni a diagramon, ezt kiegészítésként mellékelnünk kell. Egy relációs adatmodellt használó adatbázisban a korlátozásokat elsősorban adatbázis kényszerek, másodsorban ún. triggerek alkalmazásával garantálhatjuk. Egy másik lehetséges megoldás 3 több-egy kardinalitású kapcsolattípus felvétele.



Ez a módszer akkor használható jól, ha csak néhány kapcsolattípust kell felvenni. A módszer előnye, hogy a több-egy kapcsolatok kevesebb sémára képezhetők le, ld. a 6 feladat megoldásában.

3. Az ER-diagram:



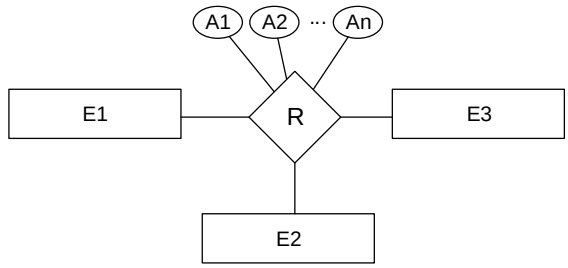
A rövidített kapcsolattípusok jelentése:

- *DOLG_O*: osztályon dolgozik,
- *DOLG_K*: kórházban dolgozik,
- *VEZ_O*: osztályt vezet,
- *VEZ_K*: kórházat vezet.

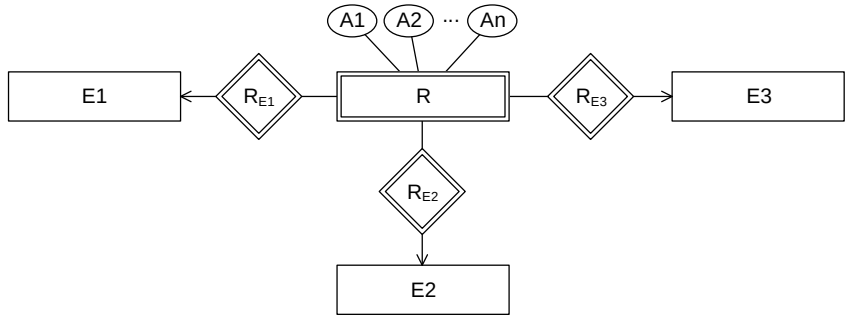
A főorvosokat és a megbízott osztályvezetőket a rang attribútummal azonosítjuk.

Az 1. feladat megoldásához hasonlóan ez a megoldás sem tudja ábrázolni azt, hogy a kórházigazgató a kórház egy dolgozója. Az ott vázolt megoldáshoz hasonló alkalmazható itt is.

4. Egy ternáris kapcsolat:



A ternáris kapcsolatot három bináris kapcsolatra transzformáljuk. Ehhez egy új, a kapcsolatot reprezentáló `R` egyedhalmazt veszünk fel. Annak érdekében, hogy ne három független bináris kapcsolatot kapjunk – amelyek nem (feltétlenül) ugyanazokat a példányokat kapcsolnák össze, mint amelyeket egy ternáris kapcsolatpéldány, ezért `R` olyan gyenge egyedhalmaz lesz, amelynek három determináló kapcsolata van. A kapcsolat attribútumai egyébként önmagukban nem is feltétlenül biztosítanának egyediséget, de az `E1`, `E2`, `E3` egyedhalmazok kulcsaival együtt már igen.



A ternáris kapcsolat kardinalitásának elemzése túlmutat a jegyzet keretein, ezért ezzel nem foglalkozunk.[1]

[1] A mélyebb részletek iránt is érdeklődők számára ld. pl. Trevor H. Jones, Il-Yeol Song, *Binary Equivalents of Ternary Relationships in Entity-Relationship Modeling: a Logical Decomposition Approach*, Journal of Database Management, April-June, 2000, pp. 12-19, `http://www.ischool.drexel.edu/faculty/song/publications/p_JDB99.PDF`.

13.2. Relációs sémaábrázolás, relációs algebra

6. Azoknál a sémáknál, ahol nem egyértelmű, hogy mely attribútumok alkotnak kulcsot, ill. idegen kulcsot, külön felsoroltuk a kulcs attribútumait. Azoknál a sémáknál, ahol minden attribútum idegen kulcs, az összes attribútum együtt alkotja a kulcsot.
   Ha a *b)* feladatrészben nem soroljuk fel külön egy séma attribútumait, akkor azok megegyeznek az *a)* feladatrészben találhatókkal.

   1. feladat

   a) Az egyedhalmazoknak és az egy-egy kapcsolatoknak megfelelő sémák:
   - *EMBER*(*szem_szám*, *szül_hely*, *szül_idő*, *név*)
   - *ORVOS*(*szem_szám*, *szül_hely*, *szül_idő*, *név*, *orvosi_diplsz*)
   - *PÁCIENS*(*szem_szám*, *szül_hely*, *szül_idő*, *név*, *TAJ*)
   - *KÓRHÁZIG*(*szem_szám*, *szül_hely*, *szül_idő*, *név*, *orvosi_diplsz*, *közg_diplsz*, *vezet_kórháznév*)
   - *KEZELŐORVOS*(*szem_szám*, *szül_hely*, *szül_idő*, *név*, *orvosi_diplsz*)
   - *KÓRHÁZ*(*név*)
   - *BETEGSÉG*(*kód*, *név*)
   - *MENTŐÁLLOMÁS*(*állomáskód*, *kórháznév*, *cím*)

   A bináris több-több és a ternáris kapcsolattípusoknak megfelelő sémák:
   - *SZENVED*(*szem_szám*, *betegség_kód*)
   - *KEZEL*(*orvos_szem_sz*, *páciens_szem_sz*, *kórház_név*, *állkód*)
   - *DOLGOZIK*(*szem_szám*, *kórháznév*)

   Az 1. feladat megoldásának 4. megjegyzése – azt, hogy egy orvost legfeljebb 3 kórház alkalmaz, nem tudjuk ábrázolni az ER-diagramon – a leképzés után is érvényes. Ha az ott javasoltaknak megfelelően 3 többegy kardinalitású kapcsolattípust veszünk fel, a *DOLGOZIK* séma helyett az alábbi sémákat kapjuk:
   - *MUNKAHELYE_1*(*szem_szám*, *kórháznév*)
   - *MUNKAHELYE_2*(*szem_szám*, *kórháznév*)
   - *MUNKAHELYE_3*(*szem_szám*, *kórháznév*)

   b) Az egyedhalmazoknak és az egy-egy kapcsolattípusoknak megfelelő sémák: az *EMBER* sémát megszüntetjük (minden, a kórház adatbázisa szempontjából releváns ember orvos vagy páciens[2]). A kórház igazgatójának személyi számát és közgazdászdiplomája számát a *KÓRHÁZ* sémában tároljuk (ez garantálja azt is, hogy egy kórháznak csak egy igazgatója lehessen). Így a *KÓRHÁZIG* séma feleslegessé válik.
   - *PÁCIENS*
   - *ORVOS*
   - *KEZELŐORVOS*

[2] Az *EMBER* séma az objektumorientált programozás terminológiája szerint absztrakt.

- *KÓRHÁZ*(*név*, *igazgató_szem_száma*, *igazgató_közg_diplszáma*)
- *BETEGSÉG*
- *MENTŐÁLLOMÁS*

A bináris több-több és a ternáris kapcsolattípusoknak megfelelő sémák változatlanok:

- *SZENVED*
- *KEZEL*
- *DOLGOZIK*

Az 1. feladat megoldásának 4. megjegyzésében javasolt 3 kapcsolattípust leképezhetjük a *KEZELŐORVOS* sémára is, ekkor az *a)* megoldásban szereplő *MUNKAHELYE_1* stb. sémákat kiválthatjuk a *KEZELŐORVOS* sémába felvett attribútumokkal:

- *KEZELŐORVOS*(*személyi_szám*, *sz_hely*, *sz_idő*, *név*, *orvosi_diplomaszám*, *munkahelye_1*, *munkahelye_2*, *munkahelye_3*)

Megjegyzés. Természetesen ekkor a *DOLGOZIK* sémára nincs szükség.

3. feladat

a) Az egyedhalmazoknak és az egy-egy kapcsolattípusoknak megfelelő sémák:

- *EMBER*(*szem_szám*, *név*, *szül_hely*, *szül_idő*)
- *SZEMÉLYZET*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*)
- *BETEG*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *TAJ*)
- *OSZTÁLYI_SZEM*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*)
- *KÓRHÁZI_SEGÉDSZEM*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*)
- *ORVOS*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*, *orvosi_diplsz*, *rang*, *kamarai_t_sz*)
- *NŐVÉR*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*)
- *OSZTÁLYI_SEGÉDSZEM*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*)
- *KÓRHÁZIG*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*, *orvosi_diplsz*, *rang*, *kamarai_t_sz*, *közg_diplsz*, *vez_k_kórháznév*)
- *BETEGSÉG*(*kód*, *név*)
- *KÓRHÁZ*(*név*)
- *OSZTÁLY*(*osztálynév*, *kórháznév*, *vez_o_szem_szám*)

A több-egy kapcsolattípusoknak megfelelő sémák:

- *DOLG_O*(*szem_szám*, *kórháznév*, *osztálynév*)
- *DOLG_K*(*szem_szám*, *kórháznév*)

A ternáris kapcsolattípusnak megfelelő séma:

- *SZENVED*(*szem_szám*, *kórháznév*, *osztálynév*, *betegségkód*)

> **Megjegyzés**. Az *OSZTÁLY* gyenge egyedhalmaz, ezért a *TARTOZIK* több-egy kapcsolattípust nem képezhetjük le külön sémára, mert az *OSZTÁLY*-nak nem lenne kulcsa. A *DOLG_O*-ban idegen kulcsként az *OSZTÁLY* séma teljes kulcsát meg kell adnunk.

b) Az egyedhalmazoknak és az egy-egy kapcsolattípusoknak megfelelő sémák: a 2. feladat sémáihoz hasonlóan itt is megszüntethetjük az *EMBER* sémát és összevonhatjuk az orvosokat és a kórházigazgatót. Szintén megszüntethetjük a *SZEMÉLYZET* sémát. A több-egy kapcsolattípusoknak megfelelő *DOLG_O* és *DOLG_K* kapcsolattípusokhoz nem veszünk fel külön sémát. Előbbihez az *OSZTÁLYI_SZEM* sémába és leszármazottaiba, utóbbihoz a *KÓRHÁZI_SEGÉDSZEM* sémába vesszük fel a megfelelő idegen kulcsokat.

- *BETEG*(...)
- *OSZTÁLYI_SZEM*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*, *dolg_o_kórháznév*, *dolg_o_osztálynév*)
- *KÓRHÁZI_SEGÉDSZEM*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*, *dolg_k_kórháznév*)
- *ORVOS*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*, *orvosi_diplsz*, *rang*, *kamarai_t_sz*, *dolg_o_kórháznév*, *dolg_o_osztálynév*)
- *NŐVÉR*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*, *dolg_o_kórháznév*, *dolg_o_osztálynév*)
- *OSZTÁLYI_SEGÉDSZEM*(*szem_szám*, *név*, *szül_hely*, *szül_idő*, *kód*, *dolg_o_kórháznév*, *dolg_o_osztálynév*)
- *BETEGSÉG*(...)
- *KÓRHÁZ*(*név*, *igazgató_szem_száma*, *igazgató_közg_diplszáma*)
- *OSZTÁLY*(...)

A ternáris kapcsolattípusnak megfelelő séma:

- *SZENVED*(...)

8. A természetes illesztésnek mindenhol legalább 0 sora lesz – akkor és csak akkor 0, ha $r(R)$-ben és $s(S)$-ben nincs azonos B attribútumérték.

a) A kulcs R-ben.
Legfeljebb $n_r \cdot n_s$, ha r és s minden sorának B mezője megegyezik.

b) B kulcs R-ben.
Készítsük az illesztést úgy, hogy az s reláció soraihoz keresünk megfelelő sort az r-ben. Mivel B kulcs R-ben, r-nek a B attribútumon felvett

értékei egyediek. Ezért s soraihoz legfeljebb egy-egy illeszkedő sor találhatunk r-ben, vagyis legfeljebb n_s sora lesz a kapott relációnak.

c) B kulcs R-ben és S-ben is.
Az illesztés menti attribútum kulcs, tehát relációnként minden sorban egyedi. Ezért az illesztés során legfeljebb annyi párt alkothatnak, ahány sora a kevesebb sorból álló relációnak van, tehát a kapott relációnak legfeljebb $\min\{n_r, n_s\}$ sora lesz.

d) A kulcs R-ben, B kulcs S-ben.
Az illesztéssel kapható sorok szempontjából nem releváns, hogy A kulcs-e. Így a megoldás a B kulcs R-ben eset tükörképe: legfeljebb n_r sora lesz a kapott relációnak.

11. A megoldás során a várt eredményt tartalmazó relációt r-rel jelöljük.

a) Melyek azok a PC modellek, amelyeknek sebessége legalább 2 GHz?

$$r = \pi_{MODELL}(\sigma_{CPU\_SEBESSÉG \geq 2000}(pc))$$

b) Mely gyártók készítenek legalább 1000 gigabájtos merevlemezű laptopot?

$$r = \pi_{GYÁRTÓ}(\sigma_{MEREVLEMEZ \geq 1000}(laptop) \bowtie termék)$$

c) Adjuk meg a B gyártó által gyártott összes termék modellszámát és árát típustól függetlenül!

$$\begin{aligned} r = \pi_{MODELL,ÁR}(\sigma_{GYÁRTÓ=\text{'B'}}(& \\ & \pi_{GYÁRTÓ,MODELL,ÁR}(termék \bowtie pc) \cup \\ & \pi_{GYÁRTÓ,MODELL,ÁR}(termék \bowtie laptop) \cup \\ & \pi_{GYÁRTÓ,MODELL,ÁR}(termék \bowtie nyomtató))) \end{aligned}$$

d) Melyek azok a gyártók, akik laptopot gyártanak, de PC-t nem?

$$\pi_{GYÁRTÓ}(termék \bowtie laptop) \setminus \pi_{GYÁRTÓ}(termék \bowtie pc)$$

e) Melyek azok a gyártók, amelyek gyártanak legalább két, egymástól különböző, legalább 3 GHz-en működő PC-t vagy laptopot? (Nincs két azonos modellszám!)

$$\begin{aligned} s =& \sigma_{CPU\_SEB \geq 3000}(\pi_{MODELL,CPU\_SEB}(pc) \cup \pi_{MODELL,CPU\_SEB}(laptop)) \\ & \bowtie termék \\ t =& \pi_{GYÁRTÓ,MODELL}(s) \\ r =& \pi_{GYÁRTÓ}(t \bowtie_{GYÁRTÓ1=GYÁRTÓ2 \wedge MODELL1 \neq MODELL2} t) \end{aligned}$$

f) Az összes 3 GHz-nél gyorsabb PC és laptop gyártója.

$$s = \sigma_{CPU\_SEB>3000}(\pi_{MODELL,SEB}(pc) \cup \pi_{MODELL,SEB}(laptop)) \bowtie termék$$
$$r = \pi_{GYÁRTÓ}(s)$$

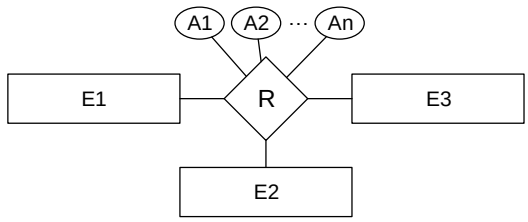
g) Azon gyártók, melyek olyan laptopokat hoznak forgalomba, melyekkel megegyező tulajdonságú PC-ket is árulnak.

$$l = \pi_{GYÁRTÓ,CPU\_SEBESSÉG,MEMÓRIA,MEREVLEMEZ}(laptop \bowtie termék)$$
$$p = \pi_{GYÁRTÓ,CPU\_SEBESSÉG,MEMÓRIA,MEREVLEMEZ}(pc \bowtie termék)$$
$$r = \pi_{l.GY}(l \bowtie_{l.GY=p.GY \wedge l.SEB=p.SEB \wedge l.MEM=s.MEM \wedge l.ML=p.ML} p)$$

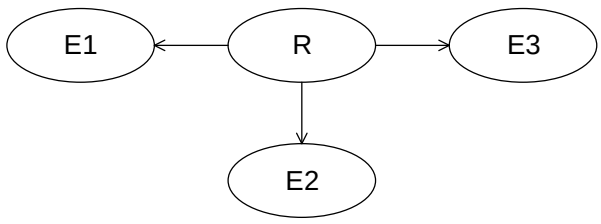
13. *a)* $\pi_{sör}(kapható) \setminus \pi_{sör}(\pi_{személy,sör}(kapható \bowtie látogat) \setminus kedvel)$
b) A feladat megoldása nagyon hasonlít az *a)* feladat megoldására.

13.3. Hálós sémaábrázolás

14. Egy ternáris kapcsolat ER-diagramon ábrázolva:

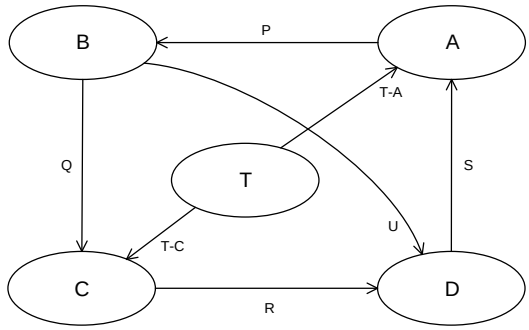


A hálós modell közvetlenül csak bináris több-egy kapcsolatokat képes implementálni, ezért minden mást kapcsolattípust ilyenekre kell visszavezetni. Ennek érdekében fel kell vennünk egy új virtuális rekord típust, amely tartalmazza a kapcsolat attribútumait és membere lesz mindhárom set típusnak. Ezzel a módszerrel minden R példányhoz egyértelműen tudunk egy-egy E1, E2, E3-beli példányt rendelni, hiszen egy set típuson belül a member rekordok egyértelműen azonosítják a hozzájuk tartozó owner rekordot.



Az R rekord típusban szerepelnek az $A1, A2, \ldots, An$ attribútumoknak megfelelő mezők.

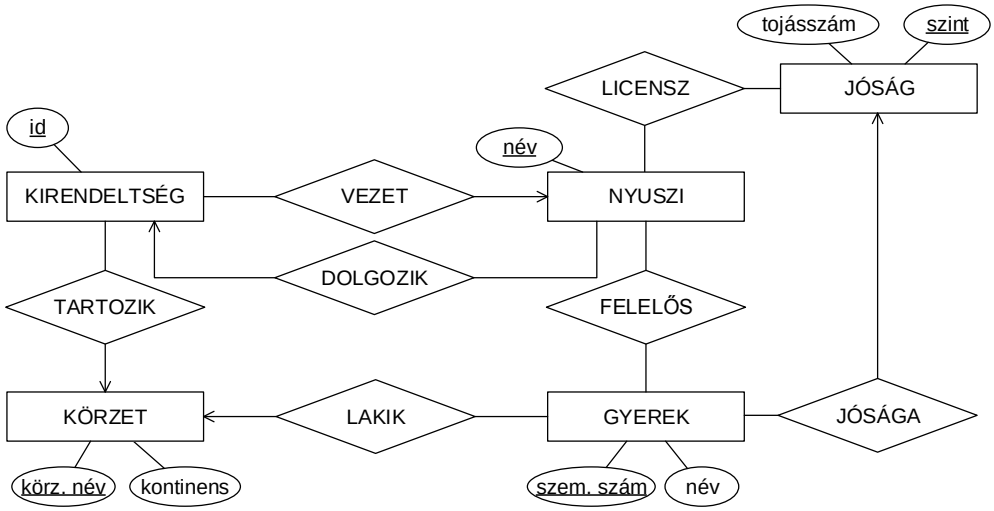
16. A hálós modell:



Az egyes rekord típusokban az alábbi mezők szerepelnek (a setek kialakításához szükséges mutatókon kívül):

- $A(\underline{A1}, A2, A3)$
- $B(\underline{B1}, B2)$
- $C(\underline{C1}, C2)$
- $D(\underline{D1}, \underline{D2}, \underline{D3})$
- $T()$

19. ER-diagram:



Az ER-diagram eszközeivel nem ábrázolhatók az alábbi megkötések:

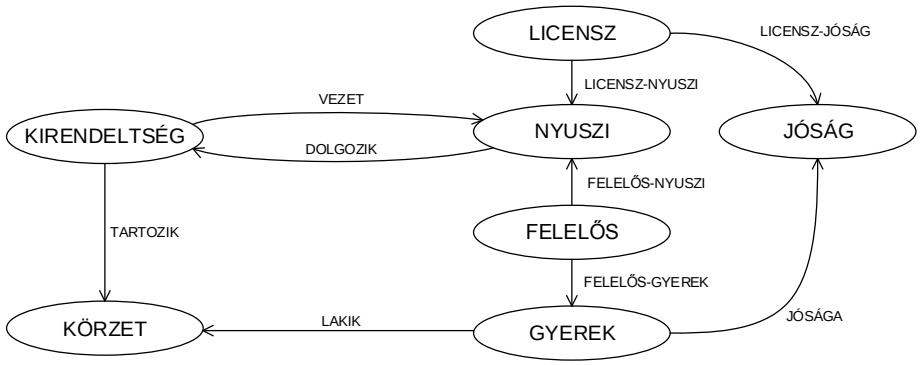
- a jóság értéke $1 \ldots 10$ közötti egész szám lehet.

- egy nyuszi csak olyan gyerekért lehet felelős, aki olyan körzetben lakik, ami ahhoz a körzethez tartozik, ahol a nyuszi dolgozik

Relációs sémák:

- *JÓSÁG*(<u>*szint*</u>, *tojásszám*)
- *KIRENDELTSÉG*(<u>*id*</u>, <u>*körz_név*</u>, <u>*vezető_név*</u>)
- *NYUSZI*(<u>*név*</u>, <u>*kirendeltség_id*</u>)
- *KÖRZET*(<u>*körz_név*</u>, *kontinens*)
- *GYEREK*(<u>*szem_szám*</u>, *név*, <u>*jóság_szint*</u>, <u>*körz_név*</u>)
- *FELELŐS*(<u>*nyuszi_név*</u>, <u>*gyerek_szem_szám*</u>)
- *LICENSZ*(<u>*jóság_szint*</u>, <u>*nyuszi_név*</u>)

Hálós sémák:



Tapsi Hapsi saját maga szeretné megajándékozni a 10-es faktorú gyerekeket, és kíváncsi, hogy melyik körzetben laknak:

$$\pi_{név,körzet\_név}(\sigma_{jóság\_szint=10}(gyerek))$$

13.4. Relációs lekérdező nyelvek

21. *a)* Olyan $u^{(4)}$ sorok *Tárgykód*ját keressük ($t[1] = u[3]$), amelyeknél *minden*, a tárgyat felvett ($u[3] = s[3]$) hallgatónak *van* legalább egy elégséges osztályzata ($p[4] \geq 2$).

$$\Big\{t^{(1)}\Big|(\exists u^{(4)})JEGYEK(u) \wedge t[1] = u[3] \wedge$$
$$((\forall s^{(4)})(JEGYEK(s) \wedge u[3] = s[3]) \Rightarrow$$
$$((\exists p^{(4)})JEGYEK(p) \wedge s[1] = p[1] \wedge p[4] \geq 2))\Big\}$$

Az implikációt átírva és átalakítva az egyik De Morgan azonossággal:

$$\Big\{t^{(1)}\Big|(\exists u^{(4)})JEGYEK(u) \wedge t[1] = u[3] \wedge$$
$$((\forall s^{(4)})\neg JEGYEK(s) \vee u[3] \neq s[3] \vee$$
$$((\exists p^{(4)})JEGYEK(p) \wedge s[1] = p[1] \wedge p[4] \geq 2))\Big\}$$

b) A lekérdezés:

```
SELECT Tárgykód
FROM
       Jegyek,
       (SELECT Neptun
       FROM Jegyek
       GROUP BY Neptun
       HAVING AVG(Jegy) > 4) Jótanulók
WHERE Jegyek.Neptun = Jótanulók.Neptun (+)
GROUP BY Tárgykód
HAVING COUNT(*) = COUNT(Jótanulók.Neptun);
```

23. A kifejezés részformulái logikai „és” kapcsolatban állnak, ezért minden, a formulát kielégítő $x^{(1)}$ sorra igaz az $ALMA^{(1)}(x^{(1)})$ kifejezés, a biztonságosság definíciójának 1. pontja teljesül.
A Ψ kifejezés maga egy $(\exists t)\omega(t)$ alakú formula, melyben a szabad változó x, $\omega(t) = x^{(1)}[1] = t^{(1)}[1] \wedge t^{(1)}[1] > 2 \wedge ALMA^{(1)}(x^{(1)})$.
Az $x^{(1)}[1] = t^{(1)}[1]$ kifejezés és az ezzel konjunkcióban („és” kapcsolatban) álló $ALMA^{(1)}(x^{(1)})$ miatt csak $DOM(\Psi)$-beli elemek kerülhetnek az eredményhalmazba, melyet a $t^{(1)}[1] > 2$ tovább szűkít. Tehát a 2. pont is teljesül, a kifejezés biztonságos.
Fontos, hogy a kifejezés biztonságossága független attól, hogy milyen reláción értékeljük ki, ezért nem szükséges $DOM(\Psi)$ meghatározása a feladatban megadott *alma* reláció mellett.
24.
$$r \bowtie s \equiv \Big\{a, b, c, d, e, f, g\Big|R(a, b, c, d, e) \wedge S(a, b, f, g)\Big\}$$

13.5. Objektumorientált adatmodell

27. A típusok:

```
Ujsag: TUPLE OF (
       cim: String,
       foszerkeszto: Szemely,
       foszerkeszto_helyettesek: LIST OF (sz: Szemely),
       rovatok: SET OF (r: Rovat)
)

Szemely: TUPLE OF (
```

```
        nev: String
)

Rovat: TUPLE OF (
        vezeto: Szemely,
        ujsagirok: SET OF (sz: Szemely)
)
```

Látható, hogy a `Szemely` típus elhagyható lenne, hiszen csak egy attribútumot tartalmaz. Az objektumorientált tervezés miatt azonban célszerű külön típusként definiálnunk a szemantikai különválasztás és a későbbi bővíthetőség érdekében.

13.6. Fizikai szervezés

31. Az adatállomány $b_r = 10^6$ blokkból áll, a sűrű indexhez $1{,}5 \cdot 10^5$ blokkra, az erre épített ritka indexhez 7500 blokkra van szükség.
Összesen $10^6 + 1{,}5 \cdot 10^5 + 7{,}5 \cdot 10^3 = 1\ 157\ 500$ blokk.
32. A felhasználható blokkméret 1520 bájt, az adatállomány 54 000 blokkból áll.
Sűrű indexre épített 1 szintes ritka index: A sűrű indexhez 9000, a ritka indexhez 300 blokk szükséges. Ez összesen $54\ 000 + 9000 + 300 = 63\ 300$ blokk.
3 szintes ritka index: A ritka index egyes szintjeinek tárolásához 1800, 60 és 2 blokk kell. Összesen $54\ 000 + 1800 + 60 + 2 = 55\ 862$ blokk, így ez a takarékosabb megoldás.
Megjegyzendő, hogy a többszintes ritkaindex-struktúra nem gyökeres fa, hiszen a legfelső szinten nem egyetlen csomópont található.
33. $n_r = 10^9, s_r = 100$ bájt, $b = 4000$ bájt, $k_1 = k_2 = 10$ bájt,
$p = 32$ bit $= 4$ bájt, $t_{\text{blokkművelet}} = 5$ ms
Egy blokkba $f_i = \left\lfloor \frac{b}{k+p} \right\rfloor = \left\lfloor \frac{4000}{10+4} \right\rfloor = 285$ indexbejegyzés fér.
    1. Az intervallumkeresés támogatásának igénye kizárja a hash szervezés alkalmazását, valamilyen indexalapú szervezést kell használnunk. Annak érdekében, hogy a keresés maximum 40 ms-ig tartson, legfeljebb 8 blokkműveletet végezhetünk.
    Egy lehetséges megoldás, hogy az adatállományra a keresési kulcsoknak megfelelő sűrű indexeket építünk, majd ezekre egy-egy B*-fát.
    A sűrű indexnek 10^9 bejegyzése van, ehhez $\left\lceil \frac{10^9}{285} \right\rceil = 3\ 508\ 772$ blokkra van szükség.
    B*-fa esetén egy indexblokkba elhelyezhetünk plusz egy mutatót akkor is, ha a kulcsának már nem marad hely. Itt ezt ki tudjuk használni, mert $285 \cdot (10+4)+4 = 3994$, ezért a fa elágazási tényezője 286. Ahhoz, hogy a sűrű indexben keresni tudjunk, $\lceil \log_{286} 3\ 508\ 772 \rceil = 3$ szintű B*-fára

van szükség. Így egy rekord beolvasásához pontosan $3 + 1 + 1 = 5$ blokkművelet szükséges, vagyis a keresés csak 25 ms-ot vesz igénybe.

Vegyük észre, hogy a feladat megoldása során az adatállomány és a B*-fa között található sűrű index által nyújtott indirekció miatt eddig nem szükséges az adatállomány blokkszámának meghatározása.

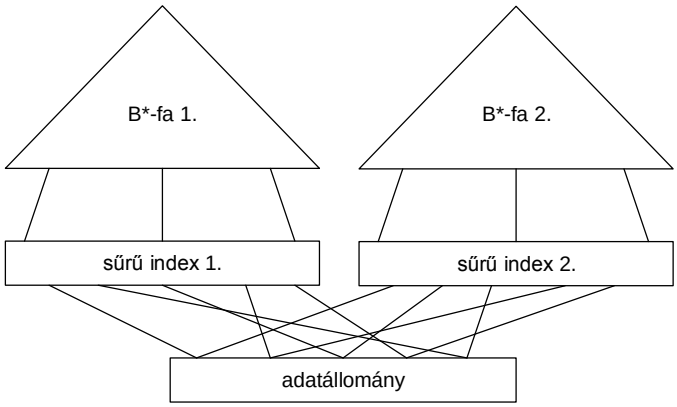
2. A rekordok 8%-a: $10^9 \cdot 0{,}08 = 8 \cdot 10^7$ rekord. Feltételezzük, hogy az állomány a keresési kulcs szerint *nem rendezett* (két kulcs esetén amúgy sem valószínű, hogy mindkettő szerint rendezhető). Ekkor mivel egy rekord kiolvasása 5 blokkműveletet igényel, az eredményül adandó rekordok kiolvasásához $(8 \cdot 10^7) \cdot 5 = 4 \cdot 10^8$ blokkműveletre van szükség. Ez 5 ms-es blokkműveletekkel számolva több mint 23 nap.

   A problémát az okozza, hogy mindig végig kell menni a teljes indexstruktúrán. Ötlet: ha a sűrű index blokkjait olvassuk végig (3 508 772 blokkművelet) és az alapján olvassuk ki az adatállományból a $8 \cdot 10^7$ rekordot, $(3\ 508\ 772 + 8 \cdot 10^7) \cdot 5$ ms $\approx 4{,}8$ nap alatt kiolvashatjuk a keresett rekordokat.

   Meglepő módon akkor járunk a legjobban, ha egyszerűen végigolvassuk az adatállomány blokkjait. Egy blokkba $f_r = \left\lfloor \frac{b}{s_r} \right\rfloor = \left\lfloor \frac{4000}{100} \right\rfloor = 40$ adatrekord fér, ezért az adatállomány $b_r = \left\lceil \frac{n_r}{f_r} \right\rceil = \left\lceil \frac{10^9}{40} \right\rceil = 2{,}5 \cdot 10^7$ blokkból áll. Ennek végigolvasásához „csak” $(2{,}5 \cdot 10^7) \cdot 5$ ms $\approx 1{,}5$ nap szükséges.

36. $n_r = 10^6, s_r = 110$ bájt, $b = 3000$ bájt, $k = 25$ bájt, $p = 64$ bit $= 8$ bájt, $t_{\text{rekord max.}} = 20$ ms, $t_{\text{blokkművelet}} = 5$ ms

    a) A feladat szövegéből tudjuk, hogy a vödörkatalógus belefér a memóriába, ezért egy rekord eléréséhez csak a hash függvény által kijelölt vödröt kell végigolvasnunk. A rekordelérési idő max. 20 ms lehet, egy blokkelérés 5 ms-ig tart, tehát egy vödör legfeljebb 4 blokkból állhat. Legjobb esetben csak a vödör első blokkját kell beolvasnunk (1 blokkművelet), legrosszabb esetben az összeset (4 blokkművelet). Az átlagos rekordelérési idő ez alapján: $t_{\text{átlag}} = \frac{1+4}{2} \cdot 5$ ms $= 12{,}5$ ms.



Megjegyzés. Nem vettük figyelembe, hogy a vödrök utolsó blokkjai nincsenek tele, ezért az átlagolásunk nem teljesen pontos. Nézzük az alábbi példát!

a) eset *b)* eset

Az *a)* esetben 5 rekordot tárolunk. Az első 4 rekord eléréséhez 1 blokkműveletre, míg az 5. rekord eléréséhez 2 blokkműveletre van szükség. Ez átlagosan $\frac{4\cdot 1+1\cdot 2}{5} = 1{,}2$ blokkműveletet jelent. A *b)* esetben 8 rekordot tárolunk, ezért átlagosan $\frac{4\cdot 1+4\cdot 2}{8} = 1{,}5$ blokkműveletre van szükség. A fenti módszerrel $\frac{1+2}{2} = 1{,}5$ blokkműveletet kapunk. Mivel ez elhanyagolható eltérést jelent, de sok többletszámolást megtakaríthatunk vele, megelégszünk a blokkonkénti számítással kapott pontossággal – egyébként is becslésekről van szó, a konkrét elérési idő sok más tényezőtől függ.

b) Egy adatblokkba $f_r = \left\lfloor \frac{b}{sr} \right\rfloor = \left\lfloor \frac{3000}{110} \right\rfloor = 27$ rekord fér, egy vödörben $4 \cdot 27 = 108$ rekord tárolható. Az állomány $B = \left\lceil \frac{n_r}{108} \right\rceil = \left\lceil \frac{10^6}{108} \right\rceil = 9260$ vödörből áll, melyek címzéséhez $9260 \cdot 8 = 74\,080$ bájt szükséges – ennyi helyet foglal a vödörkatalógus a memóriában.

c) Lásd a 34. feladatot. Itt $\frac{N}{B} = \frac{10^6}{9260} \approx 108 \gg 1$, tehát a vödrök számát a kétszeresére kell növelni, ehhez 74 080 bájt többletmemóriára van szükség – ennyivel lesz nagyobb a vödörkatalógus. Nézzük, hogyan hat ez a rekordelérési időre!
$B' = 9260 \cdot 2 = 18\,520$ vödör esetén egy vödörbe $\left\lceil \frac{n_r}{B'} \right\rceil$ rekord kerül, ehhez $\left\lceil \frac{54}{f_r} \right\rceil = \left\lceil \frac{54}{27} \right\rceil = 2$ blokkra van szükség vödrönként.
Az átlagos rekordelérési idő: $t_{\text{átlag}} = \frac{1+2}{2} \cdot 5 \text{ ms} = 7{,}5 \text{ ms}$.

13.7. Funkcionális függések

41. Vizsgáljuk meg, hogy A, B, C attribútumok esetén milyen nemtriviális függőségek állhatnak fenn!

A függőségek bal oldalán legfeljebb 3 attribútum állhat, de 0 és 3 attribútum esetén csak triviális függőségek írhatók fel: $\emptyset \to \emptyset$, ill. $ABC \to \ldots$, ahol a jobb oldalon ABC tetszőleges (akár üres) részhalmaza állhat. Így azokat az eseteket kell megvizsgálnunk, ahol a bal oldalon 1 vagy 2 attribútum áll.

Bal oldalon egy attribútum

- $A \to B, A \to C$
- $B \to A, B \to C$
- $C \to A, C \to B$

Ezen függőségek megsértésére léteznie kell olyan sorpároknak, melyeknek vannak olyan soraik, hogy az adott attribútumban megegyeznek, a többiben viszont különböznek.

A	B	C
0	1	1
1	0	1
1	1	0

Látható, hogy a példa első 3 sorában vannak A-ban, B-ben, ill. C-ben azonos sorok, melyek rendre megsértik a felsorolt függőségeket.

Bal oldalon két attribútum

- $AB \to C$
- $AC \to B$
- $BC \to A$

Az eddigi példában nem sérti ezeket a függőségeket egyik sorpár sem. Ahhoz, hogy ezt a 3 függőséget megsértsük, olyan sort kell felvennünk, amelynek van AB-ben azonos, de C-ben különböző, AC-ben azonos, de B-ben különböző, BC-ben azonos, de A-ban különböző párja:

A	B	C
0	1	1
1	0	1
1	1	0
1	1	1

Erre az $r(R)$ relációra nem teljesül semmilyen nemtriviális funkcionális függés.

42. *a)* 1. Felbontjuk a függőségeket úgy, hogy a jobb oldalakon egy attribútum legyen:

$$F' = \{XZ \to B, XZ \to G, XZ \to Y, XZ \to Z, AY \to C, \\ AY \to G, C \to W, B \to G\}$$

2. $XZ \to Z$ triviális, ezért elhagyható (ha precízen követjük az algoritmust, először a bal oldalát Z-re redukáljuk és csak a 3. lépésben hagyjuk el a $Z \to Z$ függőséget).

$$F'' = \{XZ \to B, XZ \to G, XZ \to Y, AY \to C, AY \to G, C \to W, B \to G\}$$

3. $XZ \to G$ elhagyható, mert $(XZ)^+(F' \setminus \{XZ \to G\}) = \{XZBGY\} \ni G$.

$$F''' = \{XZ \to B, XZ \to Y, AY \to C, AY \to G, C \to W, B \to G\}$$

b) Igaz-e, hogy $AXZ \to BY \in F^+$?
Attribútumhalmaz lezártjának meghatározásával.

$$(AXZ)^+(F) = \{AXZBGYCW\} \supseteq BY,$$

ezért $AXZ \to BY \in F^+$

Következtetéssel. Az eredeti függéshalmazzal dolgozunk:

$$F = \{XZ \to BGYZ, AY \to CG, C \to W, B \to G\}.$$

Az első függőséget a bővítési axióma alapján mindkét oldalon az A attribútummal bővítjük: $AXZ \to ABGYZ$. Ebből a dekompozíciós szabály felhasználásával adódik, hogy $AXZ \to BY$. Tekintve, hogy a kérdéses függést formálisan le tudtuk vezetni az Armstrong-axiómák, ill. arra visszavezethető szabály felhasználásával, ezért az Armstrong axiómák tulajdonságai (ami levezethető, az igaz, és viszont) miatt az állítás igaz.

43. *a)* igaz
b) nem igaz

44. A 41. feladat megoldásánál beláttuk, hogy az $R(A, B, C)$ relációs séma esetén az alábbi funkcionális függőségek állhatnak fenn.
Bal oldalon egy attribútum
- $A \to B, A \to C$
- $B \to A, B \to C$
- $C \to A, C \to B$

Bal oldalon két attribútum
- $AB \to C$
- $AC \to B$
- $BC \to A$

Próbáljuk meg kizárni a bal oldalon két attribútumos függőségeket. Ezt úgy tehetjük meg, hogy olyan sorokat veszünk fel, hogy lesznek olyan $t, t' \in r(R)$ sorpárok, melyek:

- AB-ben megegyeznek, de C-ben nem,
- AC-ben megegyeznek, de B-ben nem,
- BC-ben megegyeznek, de A-ban nem.

Ehhez szükség van (legalább) két olyan sorra, amelyek AB-ben megegyeznek, de C-ben nem. Ezután olyan sorpárt szeretnénk, melynek sorai AC-ben megegyeznek, de B-ben nem. Mivel az előző két sor C-ben különbözött, új sorpárt kell keresnünk. Összesen 3 sorból gazdálkodhatunk, ezért csak egy új sort vehetünk fel – ennek AC-ben egyeznie kell valamelyik korábbi sorral. Ebből viszont következik, hogy mindhárom sor ugyanazt azt értéket veszi fel az A attribútumon. Tehát a $BC \to A$ funkcionális függést sértő sorpárt nem tudunk felvenni.

A	B	C
1	1	0
1	1	1
1	0	1

Adott 3 soros r reláció tehát nem tudja az összes lehetséges funkcionális függőséget sérteni, így mindig meg lehet adni olyan nemtriviális funkcionális függést, amely az adott reláción fennáll.

Megjegyzés. Nem kell, hogy az adott funkcionális függőségnek megfelelő sorok megjelenjenek r-ben. Lásd a 9.2.3. *Funkcionális függőségek* alszakasz 1. megjegyzését: „Ha soha nincsen két t, t' sor, az $X \to Y$ függés akkor is fennállhat!”

13.8. Normálformák

47. Az R relációs séma legmagasabb normálformája az 1NF.
48. Az R relációs séma legmagasabb normálformája a 3NF.
49. R nem BCNF, tehát van egy olyan nemtriviális $X \to A$ függőség, amire X nem szuperkulcs, $A \notin X$. Mivel X nem szuperkulcs, van legalább egy olyan B attribútum, amit nem határoz meg. Ebből $X \subseteq R \setminus AB$, hiszen sem A, sem B nem lehet X eleme.

    Az $X \to A$ függőségben vegyük X-hez az összes olyan elemet $R \setminus AB$-ből, ami még nincs benne. Az így kapott függőség éppen az, amit kerestünk, hiszen a bal oldalt addig alakítottuk, amíg nem lett $R \setminus AB$-vel egyenlő.

    A kapott függőség igaz, hiszen X meghatározta A-t, ezt pedig nem tudjuk elrontani azzal, hogy új elemeket veszünk hozzá a függőség bal oldalához.

Megjegyzés. Az utolsó állítás a bővíthetőségi és a dekompozíciós szabály közvetlen következménye. $P \to Q$ esetén $PS \to Q$, hiszen először S-sel bővítünk mindkét oldalon, majd a kapott függőség jobb oldalát felbontjuk.

13.9. Relációs sémafelbontás

51. Egy séma BCNF-ben van, ha minden nemtriviális $X \to A$ függés esetén X szuperkulcs. Nem szükséges R minden szuperkulcsát meghatározni, elég megvizsgálni, hogy a függések bal oldali attribútumhalmazainak lezártja visszaadja-e a teljes relációs sémát.
    - $(AB)^+(F) = \{ABC\} \neq R$
    - $(D)^+(F) = \{DA\} \neq R$

    Látható, hogy $AB \to C$ és $D \to A$ sértik a BCNF feltételt, mert AB és D nem szuperkulcsok. Egy relációt BCNF-be veszteségmentesen úgy bonthatunk fel, hogy a BCNF tulajdonságot sértő $X \to Y \in F^+$ függőségek mentén $R_1 = XY$ és $R_2 = R \setminus Y$ sémákat hozunk létre.
    - Az R sémát $AB \to C$ mentén felbontva: $R_1(ABC)$, $R_2(ABDE)$. A függőségeknek egy $Z \subset R$ attribútumhalmazra való vetítése a $\pi_Z(F) = \{X \to Y | X \to Y \in F^+ \text{ és } XY \subseteq Z\}$. Fontos, hogy először készítsük el az függéshalmaz lezártját, ezután végezzük el a vetítést – fordított sorrendben csak a vetített függéshalmaz egy részhalmazát kapjuk.

      Az R_1-re vetített függőségek meghatározásához kiszámítjuk az attribútumhalmaz összes lehetséges valódi részhalmazának F függéshalmazon vett lezártjait: $(A)^+$, $(B)^+$, $(C)^+$, $(AB)^+$, $(AC)^+$, $(BC)^+$, $(ABC)^+$. Ez alapján $F_1 = \{AB \to C\}$. $(AB)^+(F_1) = \{ABC\} = R_1$, tehát AB szuperkulcs és R_1 BCNF.

      Az R_2-re vetített függőséghalmaz meghatározásához kiszámítjuk az attribútumhalmaz összes lehetséges részhalmazának F függéshalmazon vett lezártjait: $(A)^+$, $(B)^+$, $(D)^+$, $(E)^+$, $(AB)^+$, $(AD)^+$, $(AE)^+$, $(BD)^+$, $(BE)^+$, $(DE)^+$, $(ABD)^+$, $(ABE)^+$, $(ADE)^+$, $(BDE)^+$.

      Csak azokat a lezártakat fejtjük ki, amik bővítik a vetített F_2 függéshalmazt:
      1. $(D)^+(F) = \{DA\} \Rightarrow D \to A \in F_2$
      2. $(AE)^+(F) = \{AEBCD\} \Rightarrow AE \to BD \in F_2$
      3. $(BD)^+(F) = \{BDACE\} \Rightarrow BD \to E \in F_2$ ($BD \to A \in F_2^+$ az 1. pontban szereplő $D \to A$ miatt)
      4. $(BE)^+(F) = \{BEDAC\} \Rightarrow BE \to D \in F_2$ ($BE \to A \in F_2^+$ a 4. pontban szereplő $BE \to D$ és a 1. pontban szereplő $D \to A$ miatt)

      A vetített függéshalmaz $F_2 = \{D \to A, AE \to BD, BD \to E, BE \to D\}$. Vegyük észre, hogy ha először vetítettünk volna, és utána képzünk lezártat, csak a $\{D \to A, AE \to B, BE \to D\} \subset F_2$ függéshalmazt kapjuk, amiben nem szerepel a $BD \to E$ függés.

Vizsgáljuk meg, hogy a kapott séma BCNF-e. $(D)^+(F_2) = \{DA\} \neq R_2$, tehát a $D \to A$ függőség továbbra is sérti a BCNF tulajdonságot, ezért tovább bontjuk: R'. Megmutatható, hogy mindkét séma BCNF, tehát a $\rho_1(ABC, AD, BDE)$ felbontás minden sémája BCNF.

- Az R sémát $D \to A$ mentén felbontva az $\rho_2(AD, BCDE)$ felbontást kapjuk, melynek részsémái BCNF-ek (a részletes levezetéstől eltekintünk).

52. *a)* A felbontás $\rho_1(LMO, NOM, LNOP)$.

b) Csak a $\rho_2(LMO, LMNOP)$ felbontás megoldása a feladatnak. Ez azonban – bár a felbontás definícióját teljesíti – tartalmazza az eredeti sémát is.

53. A könnyebb számítás érdekében határozzunk meg egy minimális függéshalmazt.

1. $F' = \{A \to B, AC \to B, AC \to D, C \to A, C \to D, AF \to E, AF \to C, AF \to B\}$.
2. $C \to A$ miatt az $AC \to B$ függőség bal oldaláról elhagyható A, $A \to B$ miatt az $AF \to B$ függőség bal oldaláról elhagyható F, $C \to D$ miatt az $AC \to D$ bal oldaláról elhagyható A, így
   $F'' = \{A \to B, C \to B, C \to A, C \to D, AF \to E, AF \to C\}$.
3. $C \to A$ és $A \to B$ miatt $C \to B$ elhagyható, így
   $F''' = \{A \to B, C \to A, C \to D, AF \to E, AF \to C\}$

Az F attribútum csak függőségek bal oldalán szerepel, tehát mindenképpen eleme a kulcsnak. $F^+(F) = \{F\} \neq R$, ezért bővítenünk kell. A B, D és E attribútumokkal nem érdemes bővíteni, mert csak függőségek jobb oldalán szerepelnek, AF és CF viszont kulcsok.

Egy séma akkor 2NF, ha 1NF és minden másodlagos attribútuma a séma bármely kulcsától teljesen függ. Itt az $A \to B$ és a $C \to D$ függőségek sértik a 2NF tulajdonságot, mert $A \subset AF$ és $C \subset CF$.

A feladat többféle módon is megoldható, az egyes módszerek különböző mértékben jó megoldást adnak.

3NF-re bontó algoritmussal. Tudjuk, hogy ha egy séma 3NF, akkor 2NF is. Alkalmazzuk a 3NF-re bontó algoritmust az AF kulccsal, ami függőségőrző és veszteségmentes felbontást garantál:

$$R_1(AB), R_2(AC), R_3(CD), R_4(ACF), R_5(AEF), R_6(AF).$$

Látható, hogy R_6 elhagyható. Így a felbontás $\rho_1(AB, AC, CD, ACF, AEF)$, vagyis *öt* 3NF sémára bontottunk (amelyek egyben 2NF-ek is). A sémák számát a részsémák összekapcsolásával lehet csökkenteni.

Belátható, hogy ha egy $\mu(R_1, R_2, \ldots, R_n)$ felbontásból töröljük az R_i, R_j sémákat és hozzáadjuk az $R_i \cup R_j$ sémát, akkor a $\lambda(R_1, R_2, \ldots, R_{i-1}, R_{i+1}, \ldots, R_{j-1}, R_{j+1}, \ldots, R_n, R_i \cup R_j)$ felbontás megőrzi μ veszteségmentes, ill. függőségőrző tulajdonságát.

Veszteségmentesség bizonyítása: t. f. h. μ veszteségmentes, de λ nem az, vagyis a λ-beli sémákra illeszkedő relációk összekapcsolásakor új sorok keletkezhetnek. Vagyis $\exists t \in m_\lambda(r)$, $t \notin m_\mu(r)$. $\exists t \in m_\lambda(r)$ miatt $t[R_i \cup R_j] \in \pi_{R_i \cup R_j}(r)$, de $t[R_i \cup R_j] \notin \pi_{R_i}(r) \bowtie \pi_{R_j}(r)$. Ez viszont ellentmondás, mert a természetes illesztés tulajdonságai miatt $t[R_i \cup R_j]$ biztosan megjelenik $\pi_{R_i}(r) \bowtie \pi_{R_j}(r)$-ban is. Tehát λ veszteségmentes.

Függőségőrzőség bizonyítása: triviális. μ akkor függőségőrző, ha a részsémáira vetített függőségek uniójából levezethető az eredeti függéshalmaz minden függése. Az R_i, R_j-re vetített függőségek mindegyike szerepel az $R_i \cup R_j$-re vetített függéshalmazban (sőt, bővebb is lehet), tehát a λ felbontás függőségőrző, ha μ függőségőrző volt.

A fenti ötlet alapján lehet próbálkozni a részsémák unióját képezve csökkenteni a felbontásban szereplő sémák számát. Természetesen minden lépés után ellenőrizni kell, hogy a kapott részséma még 2NF-ben van-e. Látható, hogy öt részséma esetén a részsémák uniójának összes lehetséges kombinációjának kipróbálása rendkívül időigényes feladat.

Módosított 3NF-re bontó algoritmussal. A 3NF-re bontó algoritmus esetén a függéshalmaz minimalitására csak a 3NF tulajdonság biztosításához van szükség (ld. a tétel bizonyításában). Próbáljunk egy olyan függéshalmazt alkotni, amely ekvivalens az eredetivel, de minél kevesebb függőséget tartalmaz. Ilyen pl. az

$$\hat{F} = \{A \to B, C \to AD, AF \to CE\}.$$

Ez alapján a felbontás $R_1(AB), R_2(ACD), R_3(ACEF), R_4(AF)$, amiből R_4 elhagyható:

$$R_1(AB), R_2(ACD), R_3(ACEF).$$

Mivel nem minimális függéshalmazból indultunk ki, meg kell vizsgálnunk, hogy milyen normálformában vannak a felbontás részsémái. R_1 kételemű, tehát BCNF. Az R_2-re vetített függéshalmaz meghatározásához határozzuk meg az ACD attribútumhalmaz lehetséges részhalmazainak (A, C, D, AC, AD, CD, ACD) lezártját. Mivel D csak függőség jobb oldalán szerepel, elég az $A^+(\hat{F})$, $C^+(\hat{F})$, $AC^+(\hat{F})$ lezártak meghatározása. A vetített függéshalmaz: $\{C \to AD\}$, a séma BCNF.

Az R_3-ra vetített függéshalmazhoz az $ACEF$ lehetséges részhalmazait kell meghatározni. Tudjuk, hogy E csak függőség jobb oldalán szerepel, ezért elég az $\{A, C, F, AC, AF, CF, ACF\}$ attribútumhalmazok lezártjának meghatározása.

- $A^+(\hat{F}) = \{AB\}$
- $C^+(\hat{F}) = \{CABD\}$
- $F^+(\hat{F}) = \{F\}$
- $AC^+(\hat{F}) = \{ACD\}$

- $AF^+(\hat{F}) = \{AFBCDE\}$
- $CF^+(\hat{F}) = \{CFABDE\}$
- $ACF^+(\hat{F}) = \{ACFBDE\}$ (tartalmaz kulcsot).

A vetített függéshalmaz: $\{C \to A, AF \to CE, CF \to AE\}$, a kulcsok AF és CF, az egyetlen másodlagos attribútum az E. A séma 2NF, mert minden másodlagos attribútuma a séma bármely kulcsától teljesen függ.[3] A felbontás $\rho_2(AB, ACD, ACEF)$, tehát *három* sémára bontottunk. Próbáljuk „összeilleszteni" a sémákat az előző pontban leírtak alapján, a részsémák unióját véve.

1. $\rho_{2a}(AB, ACDEF)$: Vizsgáljuk $ACDEF$-et. D biztosan másodlagos attribútum, mert az eredeti függéshalmazban sem szerepel függés bal oldalán (de szerepel olyan nemtriviális függés jobb oldalán, melynek minden attribútuma szerepel a részsémában), CF (az egyik) kulcs, $C \to D$ eleme a vetített függéshalmaznak. D nem függ teljesen a CF kulcstól, ezért a séma nem 2NF.
2. $\rho_{2b}(ABCEF, ACD)$: Vizsgáljuk $ABCEF$-et. B biztosan másodlagos attribútum, mert az eredeti függéshalmazban sem szerepel függés bal oldalán (de szerepel olyan nemtriviális függés jobb oldalán, melynek minden attribútuma szerepel a részsémában), AF (az egyik) kulcs, $A \to B$ eleme a vetített függéshalmaznak. B nem függ teljesen az AF kulcstól, ezért a séma nem 2NF.
3. $\rho_{2c}(ABCD, ACEF)$: Vizsgáljuk $ABCD$-t. Mivel D a vetített függéshalmazban is csak C-től függhet, C minden szuperkulcsnak eleme kell, hogy legyen. C szuperkulcs és minimális, ezért kulcs is, következésképpen az egyetlen kulcs. Mivel minden kulcs egyszerű, a séma 2NF. $ACEF$-ről korábban beláttuk, hogy 2NF, a felbontás tehát megoldása a feladatnak. Mivel az eredeti R séma nem 2NF, ez a két részsémára történő felbontás minimális.

BCNF-re bontó algoritmussal. A BCNF-re bontó algoritmus veszteségmentes, de nem biztos, hogy a kapott felbontás függőségőrző. Az algoritmust úgy használjuk, hogy a 2NF tulajdonságot sértő $X \to Y$ függőségek mentén $R \setminus Y$ és XY sémákra bontjuk a relációt.

Az $A \to B$ függőség mentén a felbontás: $R_1(AB)$ és $R_2(ACDEF)$. Látható, hogy R_1 BCNF, mert két attribútuma van. Az R_2-re vetített függőségek halmazában az $\{C \to A, C \to D, AF \to C, AF \to E\}$ függőségek szerepelnek. AF és CF R_2 szuperkulcsai és minimálisak, mert F csak függés jobb oldalán szerepel, de önmagában nem kulcs. A felbontás függőségőrző, mert $A \to B$ R_1-re, a többi függőség R_2-re vetítve megjelenik.

A kulcsok részhalmazai közül C-től függ a D másodlagos attribútum, A elsődleges attribútum, ezért $C \to A$ nem, de $C \to D$ sérti a 2NF tulajdonságot. További felbontás: $R_3(CD)$, $R_4(ACEF)$. R_3 2NF (sőt

[3] Ellenőrizhető, hogy R_3 pontosan 3NF normálformájú.

BCNF), R_4-ről pedig korábban beláttuk, hogy 2NF. A felbontás függőségőrző, mert $C \to D$ R_3-re, a többi függőség R_4-re vetítve megjelenik. A felbontás: $\rho_3(AB, CD, ACEF)$, tehát *három* részsémára bontottunk. Az előző megoldáshoz hasonlóan itt is megkaphatjuk a ρ' felbontást.

Intuitív megoldás. Próbáljuk meg két 2NF részre felbontani a sémát! Ehhez alkossunk egy olyan részsémát, amelyben már nincs összetett kulcs. Legyen pl. $R_1 = C^+(F) = \{ABCD\}$. Ezen a sémán C az egyetlen kulcs, hiszen az eredeti függéshalmazban C csak AF-től függ, F nem szerepel függőség jobb oldalán, vagyis egy R_1-beli attribútumhalmaz lezártjaként nem állhat elő. Mivel minden másodlagos attribútum bármely kulcstól teljesen függ, a séma 2NF.

A másik részséma legyen olyan, hogy AF és CF egyik részétől sincs függés, azaz a 2NF tulajdonságot sértő függőségek jobb oldalán szereplő attribútumokat (B, D) elhagyjuk. Így $R_2 = \{ACEF\}$. Korábban beláttuk, hogy $ACEF$ 2NF.

Meg kell vizsgálnunk, hogy ez a felbontás veszteségmentes és függőségőrző-e.

Tudjuk, hogy egy felbontás akkor (és csak akkor) veszteségmentes, ha $(R_1 \cap R_2) \to (R_1 \setminus R_2) \in F^+$ vagy $(R_1 \cap R_2) \to (R_2 \setminus R_1) \in F^+$. Itt $R_1 \cap R_2 = \{AC\}$, $R_1 \setminus R_2 = \{BD\}$, $R_2 \setminus R_1 = \{EF\}$. Mivel F nem szerepel függés jobb oldalán, ezért $AC \to BD$-t érdemes vizsgálni. $(AC)^+(F) = \{ABCD\}$, ezért $AC \to BD \in F^+$, azaz a felbontás veszteségmentes.

Már csak azt kell ellenőriznünk, hogy a felbontás függőségőrző-e. Ehhez vegyük azon függések unióját, amelyek csak olyan attribútumokra vonatkoznak, amelyek a részsémában benn vannak. Fontos, hogy ekkor a vetített függéshalmaznak csak egy részhalmazát kapjuk – ha már ezek uniójából következik az eredeti függéshalmaz minde függése, a felbontás biztosan függőségőrző, ellenkező esetben további vizsgálatok szükségesek.

$$F_2^* = \{C \to A, AF \to EC\}$$

$$F_1^* \cup F_2^* = \{A \to B, AC \to DB, C \to AD, C \to A, AF \to EC\}$$

$$F_1^* = \{A \to B, AC \to DB, C \to AD\}$$

tartalmazza a minimális függéshalmaz összes függőségét, a felbontás függőségőrző.

A felbontás: $\rho_4(ABCD, ACEF)$, tehát egy lépésben sikerült megtalálni a *két* részsémából álló minimális megoldást.

54. A felbontások: $\rho(GHI, GJ, HKLG)$ vagy $\tau(GHI, GJ, HKLI)$.

55. Lehet.

57. A feladat megoldása többféle módon is lehetséges.

Táblázatos módszerrel. Vegyük fel a részsémáknak és az attribútumoknak megfelelő táblázatot.

	A	B	C	D	E	F	G
$ACEFG$	a	b_1	a	b_1	a	a	a
$BCDE$	b_2	a	a	a	a	b_2	b_2

$C \to F$ és $E \to G$ alkalmazása után az alábbi táblázatot kapjuk.

	A	B	C	D	E	F	G
$ACEFG$	a	b_1	a	b_1	a	a	a
$BCDE$	b_2	a	a	a	a	a	a

Ezen az $F = \{AB \to C, AC \to D, C \to F, D \to B, E \to G\}$ függőséghalmaz egyik függőségét sem tudjuk alkalmazni, ezért a táblázatnak nincs csupa a sora, tehát a felbontás nem veszteségmentes.

Tétel felhasználásával. Egy $\rho(R_1, R_2)$ felbontás akkor és csak akkor veszteségmentes, ha $(R_1 \cap R_2) \to (R_1 \setminus R_2) \in F^+$ vagy $(R_1 \cap R_2) \to (R_2 \setminus R_1) \in F^+$. Itt $R_1 \cap R_2 = CE$, $R_1 \setminus R_2 = AFG$ és $R_2 \setminus R_1 = BD$. $(CE)^+(F) = \{CEFG\}$, ezért egyik sem $CE \to AFG$, sem $CE \to BD$ nem eleme F^+-nak.

Ellenpéldával. Egy konkrét, az R sémára illeszkedő reláció:

A	B	C	D	E	F	G
a_1	b_1	c	d_1	e	f	g
a_2	b_2	c	d_2	e	f	g

(A reláción teljesülnek az F függéshalmaz funkcionális függései.)
Felbontása:

A	C	E	F	G
a_1	c	e	f	g
a_2	c	e	f	g

B	C	D	E
b_1	c	d_1	e
b_2	c	d_2	e

A két részreláció természetes illesztése:

A	B	C	D	E	F	G
a_1	b_1	c	d_1	e	f	g
a_1	b_2	c	d_2	e	f	g
a_2	b_1	c	d_1	e	f	g
a_2	b_2	c	d_2	e	f	g

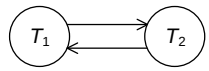
Látható, hogy az eredeti relációhoz képest új sorok keletkeztek, ezért a felbontás nem veszteségmentes.

13.10. Tranzakciókezelés

62. Az ütemezés sorosítható, soros ekvivalense: $T_2T_3T_1T_4$.
63. Az ütemezés nem sorosítható, mert nincs olyan soros ütemezés, amellyel minden hatása ekvivalens lenne. Ugyanis az ütemezés lefutásakor az A adategységet a T_1, a B adategységet a T_2 tranzakció módosította utoljára. A

soros ütemezések közül T_1T_2 lefutása után mindkét adategységet T_2, a T_2T_1 lefutása után mindkét adategységet T_1 módosította utoljára.

Az ütemezés precedenciagráfja bármilyen legális zárolással – pl. ha minden WRITE X utasítást a LOCK X, WRITE X, UNLOCK X utasítássorozatra cserélünk – az alábbi lesz:



Kétfázisú rendszer esetén: tudjuk, hogy ha egy legális ütemezés mindegyik tranzakciója 2PL protokollt követi, akkor az ütemezés sorosítható. Ebből következik, hogy ha egy ütemezés nem sorosítható, akkor nem tartozhat hozzá legális, 2PL tranzakciókból álló ütemezés.

65. A tranzakció 2PL, de nem szigorú. A két sor felcserélésével a tranzakció szigorú 2PL lesz. A protokoll sorosíthatóságot és lavinamentességet biztosít.

LOCK A	
	zárpont
READ A	
$A = A \times 2$	
COMMIT	
	készpont
WRITE A	
	írások vége
UNLOCK A	

67. Az ütemezés lefutása:

	T_1 $t(T_1) = 10$	T_2 $t(T_2) = 20$	R(A)	W(A)
(1)	READ A		10	0
(2)		WRITE A	10	20
(3)	WRITE A			

Az ütemezés első közelítésben nem sorosítható, mert a (3) lépésben a WRITE A abortot okoz $t(T_1) < W(A)$ miatt (ld. 10.9. *Időbélyeges tranzakciókezelés R/W modellben* alszakasz (6) megjegyzése).

Ennek ellenére, ha a T tranzakció írni szeretné az A adategységet $R(A) \leq t(T) < W(A)$ esetén, a tranzakciót nem feltétlenül kell abortálni, ekkor az időbélyegeket nem kell módosítani, elég csak az írást elhagyni. Ezt *Thomas írási szabálynak* (Thomas' Write Rule) nevezzük.

Az első pillanatban meglepő állítás azt a megfigyelést használja ki, hogy ebben az esetben az A adategységet már írta egy később indult U tranzakció ($t(T) < t(U)$). Ha később egy A-t *olvasni* próbáló V tranzakció $W(A) =$

$t(U)$-nál kisebb időbélyegű, akkor $t(V) < W(A)$ miatt abortálni kell, míg ha W(A)-nál nagyobb időbélyegű, akkor az U által írt értéket kell olvasnia – egyik olvasásnál sincs tehát szükség a T által írt értékre.

Fontos, hogy a Thomas írási szabályt csak akkor használhatjuk, ha a magasabb időbélyegű tranzakció már committált. Gondoljuk végig, hogy mi történik, ha a fenti példában T_2-nek további műveletei is vannak és a T_1 tranzakció A-n történő írásakor úgy alkalmazzuk a szabályt, hogy T_2 még nem committált. Ekkor lehetséges, hogy T_1 commitja után T_2 egy későbbi művelete miatt abortál, így az A adategységnek a (már commitált) T_1 tranzakció írását kellene tükröznie, amit azonban korábban kihagytunk.

A probléma egy lehetséges megoldása, hogy minden adategységhez (X) rendelünk egy $C(X)$ commit bitet, ami alapértelmezésben *igaz* értékű. A $C(X)$ bitet akkor állítjuk be *hamis*ra, ha egy írást a munkaterületen elvégeztünk, de a tranzakció még nem committált. $C(X) =$ hamis esetén az adategységre irányuló további olvasásokat/írásokat várakoztatjuk, amíg $C(X)$ igaz nem lesz vagy az X-et legutoljára író tranzakció nem abortál.

Ha az X-et legutoljára író tranzakció – ami miatt $C(X)$ hamis értékű – committál, az ütemező a $C(X)$ bitet igazra állítja; ha abortál, vissza kell állítani X korábbi értékét és a W(X) időbélyeget, majd az X-re váró tranzakcióknak újra meg kell ismételniük az olvasási/írási kísérletüket. A commit bit használata megszünteti a piszkos olvasások és a Thomas írási szabállyal felmerülő inkonzisztencia lehetőségét, mert így csak committált adategységre alkalmazhatjuk a szabályt, különben $C(X) =$ hamis miatt a tranzakció várakozásra kényszerül.

Vegyük észre, hogy a commit bit működése megegyezik a 10.9.3. *Tranzakcióhibák és az időbélyegek* c. részben használt zárakkal.

A fenti példában a Thomas írási szabályt alkalmazva – tehát a (3) lépésben az írás műveletet elhagyva, az időbélyegeket változatlanul hagyva és a tranzakciót nem abortálva – a kapott új ütemezés *hatása* bármely konzisztens adatbázison ekvivalens lesz a T_1, T_2 soros ütemezéssel:

	T_1	T_2
(1)	READ A	
(2)		WRITE A
(3)	–	

$\equiv$

	T_1	T_2
(1)	READ A	
(2)	WRITE A	
(3)		WRITE A

A Thomas írási szabállyal tehát találtunk egy olyan „trükköt”, amivel úgy transzformálhatunk bizonyos nem sorosítható ütemezéseket, hogy végül mégis legyen hozzá soros ekvivalens ütemezés.

Irodalomjegyzék

[1] Chris J. Date: *An Introduction to Database Systems*. 2000, Addison-Wesley-Longman. ISBN 978-0-201-68419-3.

[2] Ramez Elmasri – Shamkant B. Navathe: *Fundamentals of Database Systems*. 2013, Pearson Education, Limited. ISBN 978-1-292-02560-5.
URL `https://books.google.hu/books?id=x3HanQEACAAJ`.

[3] Joe Hellerstein – Eric Brewer: Query Processing, Advanced Topics in Computer Systems. `http://www.cs.berkeley.edu/~brewer/cs262/queryopt.html`, 2008. október.

[4] Kiss István: *Az SQL nyelv*. 1993, BME MMT. Oktatási segédanyag.

[5] Demetrovics János – Jordan Denev – Radiszlav Pavlov: *A számítástudomány matematikai alapjai*. Budapest, 1985, Tankönyvkiadó.

[6] Stefan M. Lang – Peter C. Lockemann: *Datenbankeinsatz*. 1995, Springer. ISBN 3-540-58558-3.

[7] Peter C. Lockemann – Gerhard Krüger – Heiko Krumm: *Telekommunikation und Datenhaltung*. 1993, Hanser. ISBN 978-3-446-17465-8.

[8] David Maier: *The Theory of Relational Databases*. Computer Software Engineering Series sorozat, vol. 1. köt. 1983, Computer Science Press. ISBN 978-0-914-89442-1.
URL `http://books.google.hu/books?id=INdQAAAAMAAJ`.

[9] Abraham Silberschatz – Henry F. Korth – S. Sudarshan: *Database System Concepts*. 2011, McGraw-Hill. ISBN 978-0-071-28959-7.
URL `https://books.google.hu/books?id=Hvn_QQAACAAJ`.

[10] Gajdos Sándor – Németh Gábor – Kiss Bálint: *Informatika II.* 1996, Műegyetemi Kiadó.

[11] Andrew S. Tanenbaum: *Számítógép-hálózatok*. Budapest, 1992, Novotrade Kiadó Kft.

[12] Jeffrey D. Ullman: *Principles of Database and Knowledge-Base Systems, Volume I*. 1988, Computer Science Press. ISBN 0-7167-8158-1.

[13] Jeffrey D. Ullman: *Principles of Database and Knowledge-Base Systems, Volume II*. 1989, Computer Science Press. ISBN 0-7167-8162-X.

[14] Simon Zoltán: *Lekérdezés feldolgozás és optimalizálás*. 2002, BME TMIT. Oktatási segédanyag.

A. függelék

Az implikáció műveletéről

Vizsgáljuk meg a következő állítást: *„ha esik a hó, akkor nulla foknál hidegebb van”*.

Ez az állítás nyilván igaz, ha minden hóeséskor nulla foknál hidegebb van. Nem állít ugyanakkor semmit a hőmérsékletről, amikor nem esik a hó, sem általában a hőmérsékletről a hóeséstől függetlenül. Az állítást egyedül úgy cáfolhatjuk meg, ha mutatunk egy olyan esetet, amikor esik a hó, de nincs nulla foknál hidegebb.

A vizsgált állítást pl. az alábbi módon formalizálhatjuk: *„esik a hó”* $\rightarrow$ *„nulla foknál hidegebb van”*. Jelöljük az *„esik a hó”* állítást A-val, a *„nulla foknál hidegebb van”* állítást B-vel. Ekkor az állítás az $A \rightarrow B$ alakban írható fel.

Az implikáció formálisan a logikai „és”, „vagy” és „kizáró vagy” mellett a kétváltozós logikai műveletek egyike. Az implikáció műveletét általában a $\rightarrow$ szimbólummal jelölik. Magát a műveletet az alábbi igazságtáblázat definiálja (az 1 az igaz, a 0 a hamis értéket jelöli):

A	B	$A \rightarrow B$
0	0	1
0	1	1
1	0	0
1	1	1

A táblázat alapján látható, hogy $A \rightarrow B$ igazságértéke $\neg A \vee B$ igazságértékével ekvivalens. Tehát $A \rightarrow B$ akkor hamis, ha $\neg(\neg A \vee B)$ igaz. Az egyik De Morgan-azonosság felhasználásával látható, hogy $\neg(\neg A \vee B) = A \wedge \neg B$. Ennek alapján a bevezető példa esetén az implikáció akkor hamis, ha az *„esik a hó”* $\wedge$ $\neg$*„nulla foknál hidegebb van”* kiértékelése igaz értéket ad. Vegyük észre, hogy ez megfelel az állítás bevezetőben említett egyetlen lehetséges szemantikus cáfolatának („esik a hó, de nincs nulla foknál hidegebb”). Kijelenthető tehát, hogy az implikáció műveletével logikailag helyesen formalizálhatunk „ha…akkor…” típusú állításokat.

Az implikáció nem kommutatív és nem asszociatív művelet. Az $A \to B$ implikációban az A kifejezést *premisszának,* a B kifejezést *konklúziónak* nevezzük.

A matematikában számos tétel is egy-egy implikációnak felel meg: ekkor az előtagot (A) feltételnek, az utótagot (B) állításnak nevezzük. Az ilyen tételek megfordítását kapjuk, ha felcseréljük a feltételt és az állítást (ami – mint tudjuk – vagy igaz lesz, vagy nem, A és B konkrét tartalmától függően). Egy ilyen (ha…, akkor…) típusú tétel bizonyításához gyakran az indirekt módszert használjuk, amikor az állítást (B) tagadjuk, és ekkor ellentmondásra kell jutnunk a feltétellel (A). Formálisan ezt $\neg B \to \neg A$ alakban írhatjuk, amit a formális logikában *kontrapozíciónak* neveznek. (A 9. és a 10. fejezetekben található indirekt bizonyításokon is megfigyelhető a kontrapozíció működése.)

Egyes szerzők az implikációt a $\Rightarrow$ vagy a $\supset$ szimbólumokkal jelölik. Az implikáció fontos szerepet játszik a mesterséges intelligenciában, a formális nyelvek elméletében, az érveléstechnikában, a deklaratív programozásban és sok más tudományterületen, köztük az adatbázis-kezelés elméletében is.

B. függelék

Szemistrukturált adatok[1]

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.

B.1. 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 (structured data) 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 objektumorientált adatmodellekkel, ill. hatékonyan kezelhetők az ezekre a modellekre épülő adatbázis-kezelő rendszerekben. A relációs és objektumorientált modelleket ezentúl mint „hagyományos adatmodelleket", a rájuk épülő rendszereket mint „hagyományos adatbázis-kezelőket" fogjuk emlegetni.

Strukturálatlan adatoknak (unstructured data) olyan adatokat tekintünk, melyeknek az aktuális alkalmazás szempontjából semmilyen használható szemantikája és emiatt felismerhető szerkezete nincs. Ezek alapján a *szemistrukturált adatok* (semi-structured data) olyan adatok, melyek az aktuális alkalmazás szempontjából hordoznak ugyan értékes szemantikus információt, de a reprezentációjuk, struktúrájuk eltér a hasznos szemantikus jelentés által meghatározottól.

[1] Megjelent: Gajdos–Nagypál: Hálószemantika címmel az InfoByte Magazin 1. számában (2001. december).

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.
- *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. tageket, 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 adatbázis-kezelő 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 adatbázis-kezelő rendszerekben használatos sémával,* több olyan tulajdonsága is lehet, amely a hagyományos adatbázis-kezelő 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 alterna-

tív 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.

B.2. 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 middlewareekkel).

B.3. 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 adatbázis-kezelő rendszerek nem (vagy nem hatékonyan) tudtak megfelelni. Néhány példa:

- a séma változékonyságának, ill. lazaságának tolerálása
- a séma lekérdezésének biztosítása
- az adatok szabad böngészésének biztosítása

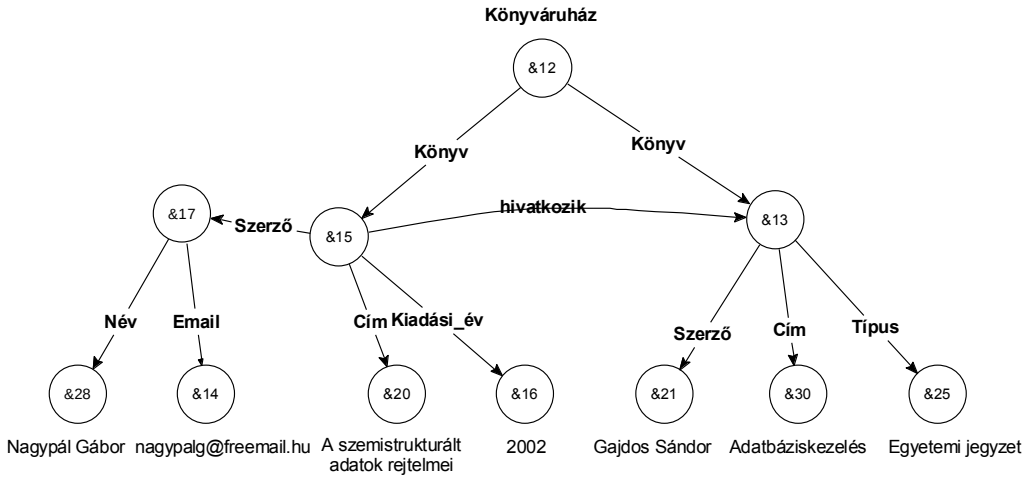
Új adatmodelleket, új adatmanipulációs nyelveket és ezek alapján működő adatbázis-kezelőket kellett alkotni, melyek képesek voltak a fenti követelményeknek megfelelni. A kutatások eredményeképpen megszületett néhány adatmodell, melyek lényegében azonos ötleten alapultak: a modell egy gráf, melynek bizonyos elemeit címkékkel látjuk el. Az adatmodellek közül a legismertebb és legsikeresebb az *OEM* (Object Exchange Model – objektum adatcsere modell) [1], a hozzá kifejlesztett *LOREL* adatmanipulációs nyelv [2] illetve *LORE* adatbázis-kezelő [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 művelethalmaz 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 objektumorientált 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. A B.1 ábrán egy képzeletbeli könyváruház OEM modelljének részlete látható.



B.1. ábra. Könyváruház OEM modelljének részlete

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.

B.4. A legelterjedtebb szemistrukturált formátum: az XML

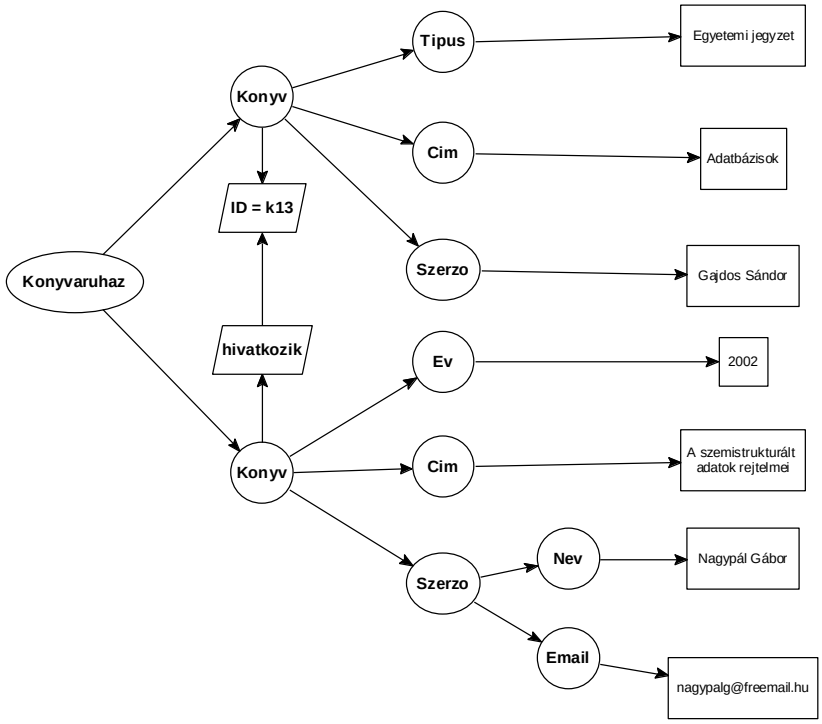
Napjainkban a szemistrukturált adatok egyik legnépszerűbb megjelenési formája az XML dokumentum. Adatmodellezési szempontból egy XML állomány az OEM modellhez hasonlóan egy irányított gráfot ír le. A gráf csomópontjai az XML elemek (element), melyek határát az XML dokumentumban a nyitó és záró címkék (tagek) 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 B.2 ábrán grafikus alakban.

B.1. Forrás. A könyváruház példa XML forrása

```
<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>
```

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.



B.2. ábra. A könyváruház XML modellje grafikusan

B.2. Forrás. A könyváruház példa DTD sémája

```
<!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)>
```

Fontos megjegyezni, hogy bár XML-ben is van lehetőség 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 melléktermékneknek 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.

B.5. 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.

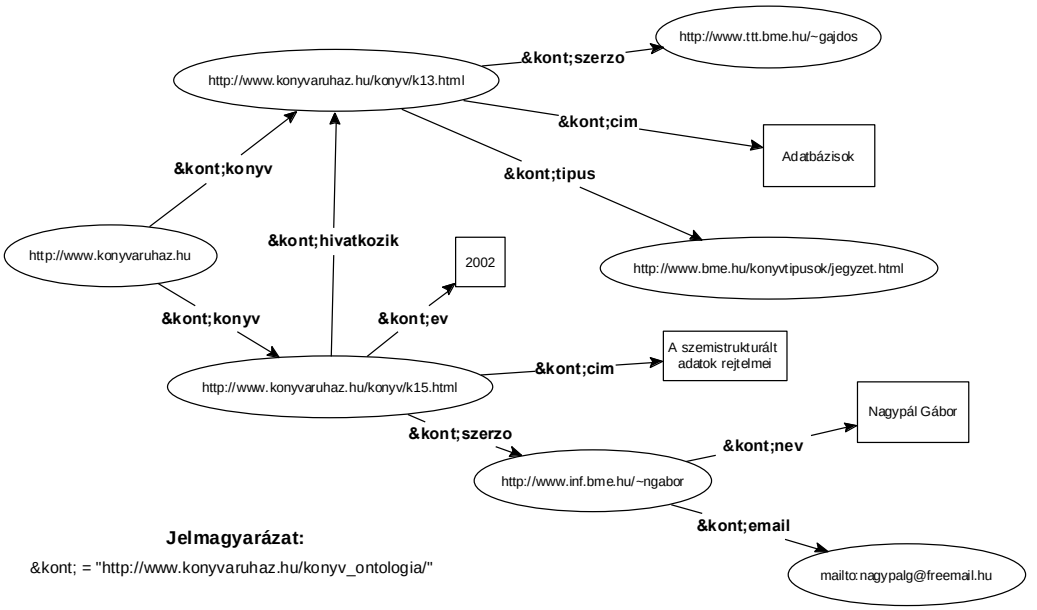
Egy ilyen, egyelőre csak álmainkban létező rendszer megvalósításához két út vezethet: vagy a gépi intelligencia színvonalát kell jelentősen növelni, vagy a rendelkezésre álló adatokat kell ellátni a megfelelő szemantikus információval. A Szemantikus Web létrehozói az utóbbi utat választották, és a jelenlegi 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 B.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.



B.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 tettünk 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.

B.6. 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 adatbáziskezelőkben tároljuk őket. Ennek megfelelően például a legtöbb modern relációs adatbázis-kezelő 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] és [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 objektumorientált 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 adatbázis-kezelőket érdemes alkalmazni, amennyiben ez a struktúra könnyen felderíthető és stabil.

B.7. 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.

B.8. Irodalomjegyzék a B 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 adatbázis-kezelő 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 adatbázis-kezelő 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

C. függelék

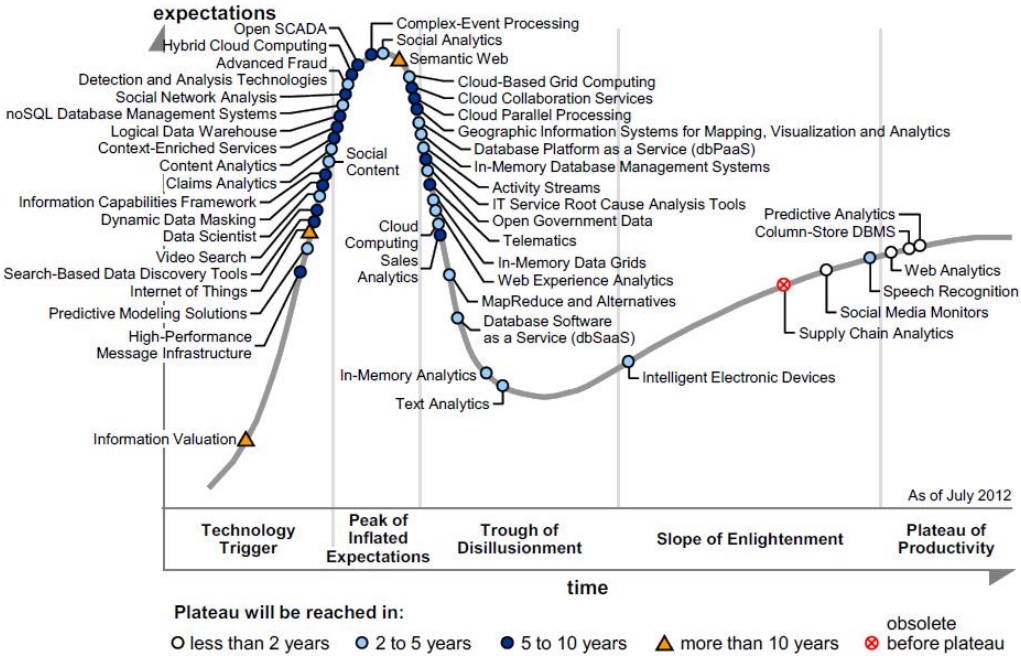
NoSQL adatbázis-kezelők[1]

Napjaink hatalmas adathalmazainak hatékony feldolgozása komoly kihívást jelent a mérnökök és az elemzők számára. Az IBM szerint napi 2,5 exabájt ($2{,}5 \cdot 10^{18}$ bájt) adat keletkezik – a növekedés olyan sebességű, hogy az elmúlt két évben jött létre a ma tárolt adatok 90%-a [1].

A relációs adatbázis-kezelő rendszerek évtizedeken át kielégítették a piac döntő részének műszaki és üzleti igényeit, azonban a 21. század elejétől olyan új igények fogalmazódtak meg (pl. web 2.0, számítási felhő, szenzorhálózatok, mobil eszközök által), melyek kiszolgálása túllépi a relációs rendszerek határait.

A világméretű internetes rendszerek rövid válaszidőket, magas rendelkezésre állást, nagy mennyiségű adat feldolgozását és horizontális skálázhatóságot kívántak meg. Ugyanakkor nem mindenhol követelmény a szigorúan vett adatbázis-konzisztencia, és számos esetben még valamennyi adatvesztés is tolerálható. Ezen igények kielégítésére jöttek – és jönnek – létre olyan új adatbázis-kezelő rendszerek, melyekre számos esetben NoSQL rendszerekként hivatkoznak. Az információs technológia ugyancsak támogatta a fejlődést: a számítógépes hálózatok adatátviteli sebessége nagyságrendekkel növekedett, fejlődtek az elosztott technológiák, jelentősen csökkentek a háttértárak és a memóriamodulok fajlagos költségei.

[1] Barabás Ákos, Szárnyas Gábor, Gajdos Sándor: NoSQL adatbázis-kezelők, BME-TMIT belső tanulmány, 2012. alapján.



C.1. ábra. A „big data” technológiák és trendek életciklusgörbéje [2]

A Gartner *életciklusgörbéken* (hype cycle), ábrázolja az egyes trendek, technológiák érettségét és elterjedtségét. A nagy adathalmazok tárolását, feldolgozását és elemzését jelentő „big data” kifejezés 2011-ben került fel a görbére. 2012-ben a big data olyannyira aktuálissá és népszerűvé vált, hogy külön életciklusgörbét készítettek számára, amelyen a NoSQL adatbázis-kezelő rendszereket a felfokozott várakozások csúcsának közelébe helyezték [2].

A big data kifejezésnek nincs egységesen elfogadott definíciója, általában az alábbi három tulajdonsággal jellemzik [3]:

- *Mennyiség* (volume): az eddig megszokottnál nagyságrendekkel megnövekedett adatmennyiség.
- *Sebesség* (velocity): az adatok nagy sebességgel generálódnak, a rendszertől minél gyorsabb feldolgozást és visszacsatolást várnak el.
- *Változatosság* (variety): az adatok többféle forrásrendszerből érkeznek, lazán struktúráltak és gyakran kérdéses minőségűek.

A NoSQL rendszerek alapjául szolgáló technikák többnyire nem új keletűek. A ’70-es években készítették az első ún. *kulcs-érték tárolót* (key-value store) és a hálós adatbázis-kezelőket.

A napjainkban (2012.) legelterjedtebb NoSQL rendszerek fejlesztése 2006 és 2009 között kezdődött. A modern NoSQL rendszerek elméletében a Google játszott úttörő szerepet: 2004-ben bemutatták a MapReduce keretrendszert [4], majd 2006-ban a BigTable adatbázis-kezelőt [5] és az elosztott zárolást megvalósító,

a BigTable-ben is alkalmazott Chubby protokollt [6]. Ezután sorra kerültek bejelentésre a nagy internetes vállalatok – így a Yahoo!, az Amazon, a Facebook és a LinkedIn – saját rendszerei.

C.1. Elnevezés

A NoSQL kifejezés a 2009-ben megtartott nyílt forráskódú, elosztott adatbázisokkal foglalkozó no:sql(east) konferenciát követően terjedt el [7].

Fontos megjegyezni, hogy az SQL (relációs) és a NoSQL rendszerek közötti határ nem éles, ugyanis számos NoSQL adatbázis-kezelő a relációs adatmodellen alapul, ill. ACID tranzakciókat kínál. A NoSQL közösség a nevet (általában) „not only SQL"-ként oldja fel, arra utalva, hogy ezek a rendszerek nem a relációs adatbázisok ellenpólusai, hanem azok mellett, azokat kiegészítve működnek. A NoSQL adatbázis-kezelőket gyakran hívják *poszt-relációs* (post-relational) vagy *nem-relációs* (non-relational) rendszereknek [8]. A NoSQL rendszereket jellemzően nem önmagukban, hanem többféle, különböző adatmodellen alapuló tárolási technológiákkal együtt használják, ezt *többnyelvű perzisztenciának* (polyglot persistence) nevezik [9].

A `http://nosql-database.org/` szerint a NoSQL mozgalom eredeti célja modern, jól skálázódó adatbázis-kezelő rendszerek készítése [10]. A legtöbb NoSQL rendszer az alábbi szempontok szerint készült:

- nem-relációs adatmodell,
- elosztott működés,
- nyílt forráskód,
- horizontális skálázhatóság.

A definíció nem szigorú: nem szükséges, hogy egy rendszerre mindegyik tulajdonság igaz legyen. A NoSQL rendszerek további gyakori tulajdonságai:

- sémamentesség vagy gyenge séma kényszerek,
- replikáció támogatása,
- *egyszerű alkalmazásprogramozási interfész* (application programming interface, API),
- *fokozatos konzisztencia* (eventual consistency).

A konzisztenciával és skálázhatósággal kapcsolatos feltételeknek a továbbiakban külön alfejezeteket szentelünk, és külön foglalkozunk a replikációval is.

A nyílt forráskódúság feltétele vitatott, hiszen ez alapján nem tartozhatna például a NoSQL adatbázis-kezelők közé a már említett úttörő, a Google BigTable. Bár a legtöbb rendszer egyszerű API-val rendelkezik, komplex lekérdezéseket jellemzően egyszerűbb megfogalmazni SQL nyelven, mint a NoSQL rendszerek sajátos keretei között. Ezért néha olyan rendszereknél is megjelenik az SQL vagy ahhoz hasonló lekérdezések támogatása, ahol ezt az adatmodell nem indokolja.

C.2. Skálázhatóság

A bevezetőben említett adatmennyiségek mellett az igények kiszolgálása már nem lehetséges (vagy nem praktikus) egyetlen szervergéppel, ezért az ilyen rendszerek tervezésénél már fő szempont, hogy azok több számítógépen elosztottan működjenek. Tekintettel arra, hogy esetenként több tízezer számítógépről van szó, kritikus kérdés, hogy hogyan viselkedik a rendszer újabb és újabb erőforrások (számítógépek) hozzáadása következtében.

A skálázhatóságnak nincs általánosan elfogadott definíciója [11]. Informálisan egy rendszert akkor nevezünk *skálázható*nak (scalable), ha az erőforrásait növelve, a rendszer teljesítménye a hozzáadott erőforrásokkal *arányosan* javul. A teljesítmény jelentheti a rendszer által egyszerre feldolgozható lekérdezések számát, a feldolgozott adathalmazok vagy adatelemek méretét, a késleltetést stb. [12].

A plusz erőforrások hozzáadásának másik oka a rendszer hibatűrésének, ill. rendelkezésre állásának növelése. Fontos követelmény, hogy ez ne jelentsen túl nagy többletterhelést a rendszer számára. A skálázhatóság két fő típusát különböztetjük meg.

- ***Vertikális skálázhatóság* (vertical scalability, scale up):** a rendszer kiválasztott elemét bővítjük új erőforrással, leggyakrabban erősebb/több processzorral vagy több memóriával. A módszer előnye, hogy egyszerű megvalósítani, és a megfelelő, a rendszer szűk keresztmetszeteit okozó erőforrások bővítése esetén biztosan teljesítménynövekedéssel jár. Ezt az elvet egy- és többszerveres elosztott rendszerekben is lehet alkalmazni.
- ***Horizontális skálázhatóság* (horizontal scalability, scale out):** a rendszert bővítjük új számítógéppel. Előnye, hogy sok, olcsó gépből nagy teljesítmény érhető el. Hátránya, hogy elosztottsága miatt többféle meghibásodás is felléphet, valamint bonyolult szoftvert igényel. Ezt támogatja a MySQL Scale-Out, az Oracle Real Application Cluster (RAC) és a legtöbb NoSQL rendszer is.

A skálázhatósággal szorosan összefüggő szempont, hogy a rendszer komponensei milyen közös adattárat használnak. Ez alapján három fő elosztott architektúrát különböztetünk meg [13], melyek egyre lazábban csatolt rendszereket definiálnak:

- ***Megosztott memória* (shared memory):** a komponensek közös memórián dolgoznak. Ilyen architektúrát egy számítógépen belül használnak, pl. többprocesszoros rendszerekben.
- ***Megosztott lemez* (shared disk):** a komponensek saját memóriával rendelkeznek és közös lemez(ek)re mentenek. Ilyenek a közös háttértáron dolgozó adatbázis-kezelők, pl. az Oracle RAC.
- ***Megosztás nélküli* (shared nothing):** a komponensek nem rendelkeznek sem közös memóriával, sem közös lemezzel.

Természetesen egy rendszer annál jobban skálázható, minél kevesebb megosztott

komponenssel rendelkezik. Ezért a NoSQL rendszerek jellemzően megosztás nélküli architektúrájúak.

C.3. A CAP-tétel

A NoSQL rendszerek tervezését befolyásoló egyik legfontosabb eredmény az ún. CAP-tétel, amely az elosztott rendszerek által nyújtott garanciákra ad formális korlátot.

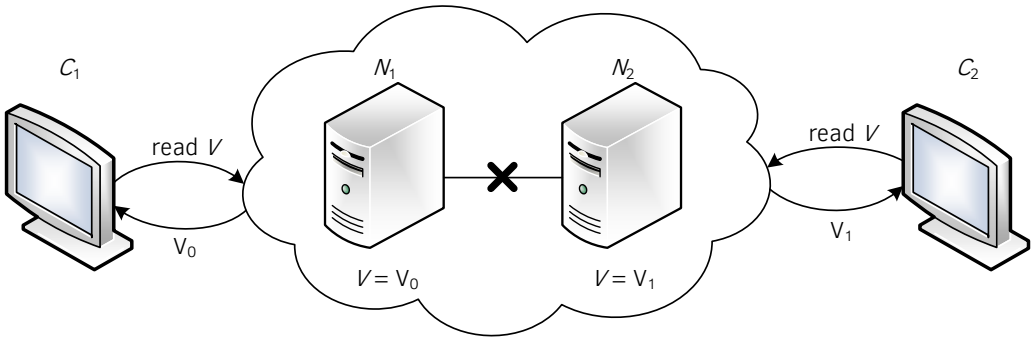
Eric Brewer, a Berkeley Egyetem professzora 1999-es publikációjában definiálta (informálisan) a CAP tulajdonságokat és a CAP alapelvet [14], amelyet 2000-ben a PODC (Principles of Distributed Computing) szimpóziumon ismertetett CAP-sejtés néven [15]. A sejtés szerint egy webszolgáltatás a CAP rövidítésben szereplő *konzisztencia* (consistency), *rendelkezésre állás* (availability), *partíció tolerancia* (partition tolerance) tulajdonságok közül egyidejűleg legfeljebb kettőt garantálhat teljesen.

A sejtést Seth Gilbert és Nancy Lynch, az MIT (Massachusetts Institute of Technology) kutatói formalizálták és bizonyították be 2002-ben [16]. Ebben a formájában a CAP-tétel nem csak webszolgáltatásokra, hanem minden elosztott rendszerre érvényes. Az alábbiakban definiáljuk a három tulajdonságot, majd ismertetjük a tétel bizonyításának vázlatát.

C.3.1. Konzisztencia

Egy elosztott rendszer akkor konzisztens, ha bármely időpillanatban egy adategység értékét bármely csomóponttól lekérdezve ugyanazt az értéket kapjuk.

Fontos, hogy az elosztott rendszerek konzisztenciája nem egyezik meg az tranzakciókezelésnél használt ACID (atomicity, consistency, isolation, durability) tulajdonságokban definiált konzisztenciával [16] [17]. Az ACID konzisztencia definíciója azt garantálja, hogy az egyes *tranzakciók* az adatbázist konzisztens állapotból konzisztens állapotba viszik át, ahol az adatbázison mindig csak a sikeresen lefutott tranzakciók eredménye látszik. Itt a konzisztencia a *rendszer* tulajdonsága. Gilbert és Lynch az említett bizonyításukban az *atomi* (atomic) kifejezést használták (szintén nem tévesztendő össze az ACID tranzakciókra vonatkozó azonos nevű tulajdonságával). Definíció szerint atomi konzisztencia esetén az elosztott rendszernek külső szemlélő számára úgy kell tűnnie, mintha a műveleteit egy csomóponton, sorban egymás után hajtotta volna végre, azok átlapolódása nélkül. Ehhez léteznie kell a műveletek egy olyan teljes sorrendezésének, ahol a külső szemlélő számára úgy tűnik, hogy minden művelet „egy pillanat alatt” végrehajtódott.

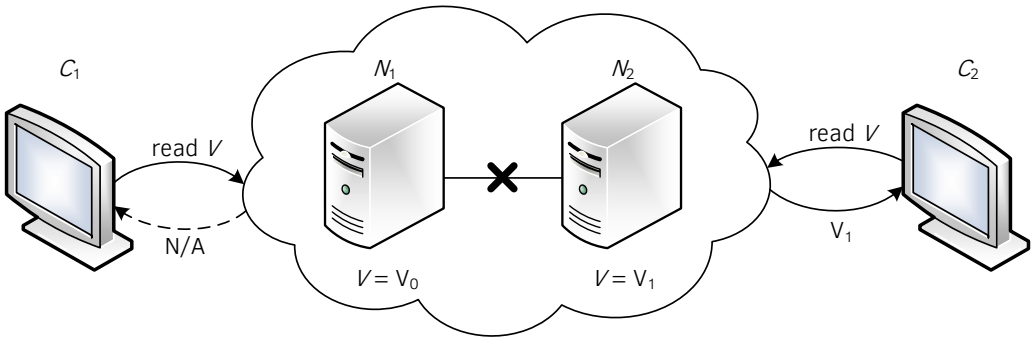


C.2. ábra. A V adategység nem konzisztens, a C_1 és a C_2 kliensek eltérő értéket látnak

C.3.2. Rendelkezésre állás

Egy elosztott rendszer rendelkezésre áll, ha minden működő csomóponthoz érkező kérésre válaszol, tehát a csomópontokon futtatott algoritmusoknak véges idő alatt be kell fejeződniük. A formális definíció nem korlátozza a futási időt, de később látni fogjuk, hogy a gyakorlati alkalmazások során a késleltetésnek kiemelt szerepe van.

A C.3 ábrán egy olyan rendszer látható, amely nem garantálja sem a konzisztenciát, sem a rendelkezésreállást. A hálózat két részre szakadt, ezért az N_1 csomópont nem tudja kiszolgálni a C_1 kliens kérését.



C.3. ábra. A C_1 kliens számára a V adategység nem elérhető

C.3.3. Partíció tolerancia

A hálózat partíciója annak alapján modellezhető, hogy a hálózat egyik csomópontjából a másikba küldött üzenetekből tetszőleges számú elveszhet. Egy nem összefüggő hálózatban a hálózat egyik komponenséből a másikba küldött minden üzenet elveszik. Minden üzenetvesztési szekvencia modellezhető a hálózat ideiglenes partíciójával, majd újraegyesülésével. A partíció oka lehet a hálózat hibája,

valamint a csomópontok hardveres vagy szoftveres hibái. A partíció tolerancia a másik két tulajdonság tükrében értelmezhető:

- A konzisztencia követelményhez azt kell teljesíteni, hogy a rendszer műveletei akkor is atomiak legyenek, ha az azokat megvalósító algoritmusok üzeneteiből tetszőleges számú elveszhet.
- A rendelkezésre álláshoz azt kell biztosítani, hogy a csomópontok *minden*, a kliensektől érkező *kérésre* érvényes választ adjanak, annak ellenére, hogy tetszőlegesen sok üzenet elveszhet.

Egy rendszer tehát akkor partíció toleráns, ha a kérésre hálózati partíció esetén is helyes[2] választ ad (kivéve a teljes hálózat kiesésének esetét). Ha egy rendszer nem partíció toleráns, a hálózat partíciója esetén semmilyen garanciát nem nyújt a konzisztenciára és a rendelkezésre állásra.

A partíció tolerancia elengedhetetlen a nagyméretű elosztott rendszerben, hiszen a hálózat partíciójának valószínűsége a gépek számának növekedésével egyre magasabb lesz.

C.3.4. A CAP-tétel formálisan

A CAP-tétel röviden úgy fogalmazható meg, hogy elosztott rendszerben a hálózat partíciója esetén a rendszer műveletei nem lesznek atomiak és/vagy az adategységei elérhetetlenek lesznek.

Formálisan: egy aszinkron vagy részben szinkron[3] hálózaton működő elosztott rendszerben nem biztosítható, hogy a rendszer *mindig* (üzenetek elvesztése esetén is) garantálja az alábbi tulajdonságokat:

- rendelkezésre állás,
- atomi konzisztencia.

Bizonyítás

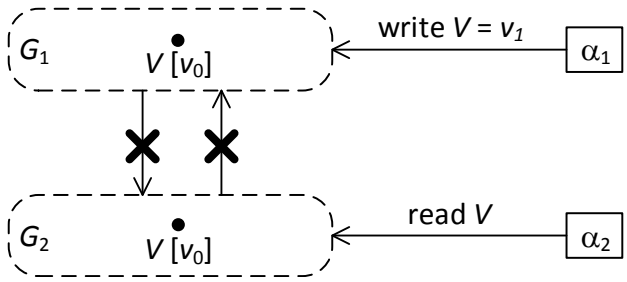
A bizonyítást aszinkron hálózatra mutatjuk meg, Gilbert és Lynch cikke tartalmazza a részben szinkron hálózatra vonatkozó bizonyítást is [16]. Az aszinkron hálózati modellben nincs óra és az egyes csomópontok döntéseit kizárólag a beérkezett üzenetek és a lokális számításaik befolyásolják [18].

Indirekt módon bizonyítunk. Tegyük fel, hogy létezik olyan A algoritmus, amely teljesíti a CAP tulajdonságokat. Feltételezhetjük, hogy a hálózat legalább két csomópontból áll, így felosztható két diszjunkt, nem üres halmazra. Legyen a felosztás (G_1, G_2). Tegyük fel, hogy hálózati partíció miatt G_1 és G_2 között minden üzenet elveszik, valamint G_1-hez és G_2-höz nem fordul más kliens. Megmutatjuk, hogy

[2] Érvényes (helyes) válasz alatt azt értjük, hogy a rendszer végrehajtja a kért írást vagy olvasást (a konzisztencia, tehát a művelet atomicitásának garantálása viszont nem kötelező). A „nem elérhető” jellegű hibaüzenetek nem számítanak érvényes válasznak.

[3] A *részben szinkron* hálózati modellben a csomópontok azonos ütemben növekvő órákkal rendelkeznek. Az órák azonban nem teljesen szinkronizáltak.

ha G_1-ben írunk, akkor egy későbbi, G_2-beli olvasás művelet nem térhet vissza a korábbi írás művelet értékével.



C.4. ábra. G_1 és G_2 részekből álló hálózati partíció

Mindkét halmazban tároljuk a V adategység értékét, ami kezdetben v_0.

Legyen α_1 az A algoritmus olyan lefutásának első lépése, amiben egy írás G_1-ben V értékét a v_0-tól különböző v_1 értékre állítja. A rendelkezésre állás tulajdonság miatt tudjuk, hogy az írás sikeresen befejeződik.

Hasonlóan, legyen α_2 egy olyan lefutás első lépése, amiben G_2-ben V értékét olvassuk. A rendelkezésre állás miatt az olvasásnak vissza kell térnie egy értékkel. Amennyiben α_2 előtt más műveletet nem futattunk, ez az érték biztosan v_0 lesz.

Legyen α az algoritmus egy olyan lefutása, ahol α_1 után lefut α_2. A G_1 és G_2 közötti kommunikáció hiánya miatt a G_2-beli csomópontok számára α megkülönböztethetetlen α_2-től, míg α_1 nem fordul G_2-beli csomópontokhoz.

A fentiek miatt α lefutása során az α_2-beli olvasás művelet v_0-t fog visszaadni. α_2 olvasása azonban csak akkor kezdődhet meg, ha α_1 írása befejeződött, így a konzisztencia tulajdonság nem teljesül, hiszen V értéke G_1-ben már v_1. Ezért nem létezhet olyan algoritmus, ami a CAP tulajdonságok közül mindhármat teljesíti.

Triviális esetek

A CAP-tétel alapján az alábbi esetek adódnak egy elosztott rendszer tulajdonságaira.

- *Konzisztens és rendelkezésre álló (CA):* megpróbáljuk megszüntetni a partíció lehetőségét, például úgy, hogy a rendszert egy gépen futtatjuk. Ezzel a hálózati partíció lehetőségét kizártuk, de részleges hibák előfordulhatnak. Természetesen a hálózati hibákra való érzékenység miatt ezek a rendszerek csak korlátozottan skálázódhatnak, ezért általában megbízható LAN hálózatokon üzemeltetik őket. Ezt az architektúrát alkalmazza a napjainkban használt RDBMS-ek többsége.
- *Konzisztens és partíció toleráns (CP):* hálózati partíció fellépése esetén azok adategységek elérhetetlenné válnak, amelyekre nem a rendszer nem tudja biztosítani a műveletek atomi végrehajtását. Ennek ellenőrzése és az egyes csomópontok újra beléptetése meglehetősen összetett lehet. Elméleti szem-

pontból CP az a triviális rendszer, amely nem dolgozza fel a beérkező üzeneteket, azaz a rendszer mindig elérhetetlen.

– *Rendelkezésre álló és partíció toleráns (AP):* a konzisztencia gyengítésével egyidejűleg elérhető partíció tolerancia és rendelkezésre állás is. Bizonyos alkalmazások (pl. keresők, közösségi oldalak) nem feltétlenül igényelnek erős konzisztenciát, más alkalmazásoknál a konzisztencia megvalósítható az alkalmazási rétegben is [17].

Fontos, hogy a fenti triviális esetekkel nem fedhető le az összes (elosztott) adatbázis-kezelő.

Mindhárom CAP tulajdonság garantálása

Brewer prezentációjában úgy fogalmazott, hogy egy rendszer a CAP tulajdonságokból legfeljebb kettőt garantálhat [15]. Ez a korlátozás azonban csak egy adott időpillanatra érvényes, nem a rendszer működésének egészére.

A CAP-tétel nem zárja ki ugyanis azt, hogy egy rendszer jól működő hálózat esetén konzisztens és rendelkezésre álló legyen, hálózati partíció esetén pedig valamelyik (esetleg mindkettő) tulajdonságra gyengébb garanciát nyújtson. Ezért a rendszereket célszerű úgy vizsgálni, hogy a hálózat állapotától függően milyen tulajdonságokat garantálnak.

A gyakorlati rendszerek vizsgálatánál célszerű a késleltetés csökkentésére irányuló kompromisszumokat is figyelembe venni [20].

C.3.5. A CAP-tétel kritikája

A CAP-tétel jelentős eredmény az elosztott rendszerek elmélete terén, ugyanakkor több szempontot figyelmen kívül hagy.

Ilyenek például olyan kritikus rendszertervezési kérdések, mint a rendszer teljesítménye (áteresztőképessége) és késletetése, valamint az *egyszeres hibapont*ok (SPOF, Single Point of Failure) jelenléte. A tétel szintén nem érvényes alkalmazáshibák, az adatbázis-kezelő összeomlását okozó tranzakciók esetén (amelyek a hálózat más csomópontján futtatva is összeomlást fognak okozni).

Késleltetés

A valós alkalmazások során kiemelt szerepe van a késleltetésnek is. Kísérletek kimutatták, hogy az Amazonnál a lapbetöltési idő minden 100 ms-os növekedése 1%-kal csökkentette az eladásokat, míg a Google-nél a keresési találatok megjelenési idejének fél másodperces növekedése 20%-os bevételcsökkenést okozott [19].

A késleltetés csökkentésének érdekében sok NoSQL rendszer (pl. az Amazon Dynamo [21]) összefüggő hálózaton futtatva sem garantál atomi konzisztenciát, cserébe a rendszer hálózati partíció kialakulása esetén is elérhető marad. Ezek a rendszerek tehát csak az *AP* tulajdonságokat garantálják. A megközelítés BASE (Basically Available, Soft-state, Eventually consistent) néven ismert [23]. A BASE kifejezést szintén Brewer alkotta meg [22], azonban ő is elismeri, hogy a kifejezés

– az ACID-hoz hasonlóan – nem precíz, formális definíció, hanem egy jól hangzó rövidítés a gyengébb konzisztencia követelményekre[4] [17].

Léteznek olyan rendszerek is (pl. a GenieDB [24]), amelyek normál működés során erős konzisztenciát nyújtanak, partíció esetén viszont gyengébb konzisztencia kritériumok mellett folytatják a működést, így a hálózat állapotától függően *CA* és az *AP* tulajdonságpárokat teljesítik.

Az elosztottan működő, ACID tranzakciókat garantáló rendszerek, többek között a VoltDB RDBMS és az OrientDB NoSQL rendszer, normál működés során konzisztensek és rendelkezésre állók, hálózati partíció esetén pedig a rendelkezésre állást csökkentik a konzisztencia megtartása érdekében. Ezek a rendszerek a *CA* és a *CP* tulajdonságpárokat teljesítik.

Eltérő megközelítést alkalmaz a Yahoo! több kontinensen elosztva működő PNUTS (Platform for Nimble Universal Table Storage) rendszere [25]. A PNUTS normál működés során az atominál gyengébb konzisztenciát[5] nyújt az alacsony késleltetés érdekében, hálózati partíció esetén viszont ugyanazt a konzisztencia garanciát nyújtja, és szükség esetén a rendelkezésre állást csökkenti. A PNUTS tehát partíció esetén sem az *A*, sem a *C* tulajdonságot nem teljesíti, annak érdekében, hogy a szerverek közötti fizikai távolság ellenére alacsonyan tarthassa a késleltetést.

C.4. Konzisztenciamodellek

A konzisztenciamodell az adattár és az ahhoz hozzáférő folyamatok között létrejött megállapodás, amely szerint ha a folyamatok betartanak bizonyos szabályokat, az adattár helyesen fog működni [26]. Másképp, a konzisztenciamodell meghatározza a frissítések láthatóságára és látszólagos sorrendjére vonatkozó szabályokat [31]. A konzisztenciamodelleket két fő csoportra oszthatjuk.

- A *kliensközpontú* (client-centric) modell a kliensek és a rendszer közötti üzenetváltások alapján ad garanciát a konzisztenciára, ezért a fejlesztők számára praktikusabb definíciót nyújt.
- Az *adatközpontú* (data-centric) modell leírja, hogy az adategységek frissítései milyen korlátozásokkal terjednek az egyes szerverek között (ezért gyakran szerveroldali konzisztenciaként hivatkoznak rá [27]).

C.4.1. Kliensközpontú konzisztenciamodellek

A konzisztenciát illetően többféle kompromisszum is megengedhető, az alábbiakban a leggyakrabban használtakat ismertetjük. Fontos, hogy az alábbi konzisztenciamodelleket *egy adategységre* értelmezzük. A több adategységre értelmezett

[4] Brewer a *sav-bázis* (acid–base) reakcióban szereplő kifejezések alapján választotta a BASE rövidítést, mert konzisztencia szempontból a BASE a megszokott, elvárt ACID ellentétének tekinthető.

[5] A PNUTS egy speciális, a sorosíthatónál gyengébb, a fokozatosnál erősebb garanciákat nyújtó konzisztenciamodellt használ (a konzisztenciamodellek definícióit ld. később).

konzisztencia tranzakciókkal valósítható meg. A tranzakciók elosztott rendszerekben költségesek, ezért sok NoSQL rendszer csak korlátozottan vagy egyáltalán nem támogatja őket.

C.4.1.1. Gyenge konzisztencia (weak consistency)

A rendszer nem garantálja, hogy az írást követő olvasások a legutoljára beírt adatot érik el. Az írás és azon pillanat között eltelt időt, amíg nem garantálható, hogy minden megfigyelő a frissített adatot látja, *inkonzisztenciaablaknak* (inconsistency window) nevezzük.

C.4.1.2. Fokozatos konzisztencia (eventual consistency)

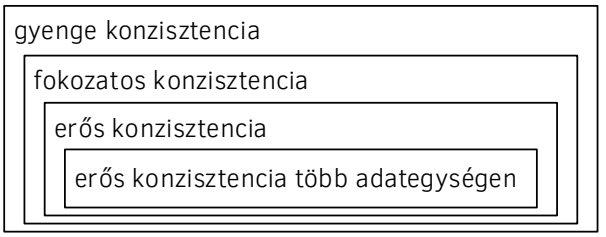
A gyenge konzisztencia egyik típusa. A rendszer garantálja, hogy ha nincsenek további frissítések, előbb-utóbb minden olvasás a legutóbbi írás értékét éri el. Ha nem lépnek fel hibák, az inkonzisztenciaablak mérete meghatározható bizonyos tényezők alapján (pl. kommunikációs késleltetés, a rendszer terheltsége, a replikák száma stb.). A legismertebb fokozatos konzisztenciára építő rendszer a DNS (Domain Name System), ahol a gyorsítótárak előre meghatározott időközönként frissülnek.

C.4.1.3. Erős konzisztencia (strong consistency)

Minden olvasás művelet az adategységen legutóbb befejezett írás művelet eredményével tér vissza, függetlenül attól, hogy az adategységet melyik csomóponton éri el. Ez a konzisztenciamodell könnyen érthető a rendszer felhasználói számára, és jelentősen egyszerűsíti a kliensalkalmazások fejlesztőinek munkáját.

Erős konzisztencia megvalósításához egy adott adategység összes műveletét egy csomóponton kell végrehajtani vagy megfelelő elosztott protokollt kell használni.

Több adategységen végzett műveletek esetén erős konzisztenciát tranzakciókezeléssel biztosíthatunk, ami azonban elosztott környezetben csak költségesen valósítható meg.



C.5. ábra. Kliensközpontú konzisztenciamodellek. A szűkebb halmazok modelljei erősebb garanciákat nyújtanak.

A fokozatos konzisztencia többféle módon gyengíthető, a gyakori konzisztenciamodellek az alábbiak [26] [27].

***Monoton olvasási konzisztencia* (monotonic read consistency, MR)**

Ha egy folyamat beolvas egy x adatelemet, akkor az x adatelem minden további olvasásának ugyanazt vagy frissebb értéket kell szolgáltatnia.

Példa. Legyen a rendszer egy elosztott e-mailadatbázis, amelyben a felhasználó postaládáját elosztott módon, többszörözve tárolják. Ha felhasználó az egyik városban elolvassa a leveleit, majd átmegy egy másik városba, a monoton olvasás biztosítja, hogy a korábban olvasott levelek a másik városból csatlakozva is a fiókjában lesznek.

***Monoton írási konzisztencia* (monotonic write consistency, MW)**

Adott folyamat által végrehajtott, x adatelemet módosító műveletnek be kell fejeződnie, mielőtt ugyanez a folyamat újabb írási műveletet hajtana végre az x adatelemen. Azaz a rendszer garantálja, hogy egy folyamat az írásai sorosan végzi.

Példa. Egy verziókezelés alatt álló programkód esetén az írások során csak a programkód egy része változik. Ilyenkor elengedhetetlen, hogy egy fejlesztő módosításai a megfelelő sorrendben kerüljenek be a rendszerbe. Az MW-t nem biztosító elosztott rendszerek komoly kihívások elé állítják az alkalmazásfejlesztőket [27].

***„Olvasd az írásod" konzisztencia* (read your writes consistency, RYW)**

Adott folyamat által az x adatelemen végrehajtott írási művelet eredményének mindig láthatónak kell lennie a folyamat által x adatelemen végrehajtott későbbi olvasási művelet számára.

Példa. Sok elosztott webes szolgáltatás (pl. a Gmail) RYW konzisztenciát nyújt. RYW konzisztencia hiányában a felhasználó számára hibásnak tűnhet a rendszer működése. Előfordulhat például, hogy a felhasználó megváltoztatja a jelszavát, de a változás nem terjed rögtön végig a rendszeren (pl. a jelszavakat egy erre a célra dedikált szerveren tárolják és időre van szükség ahhoz, hogy a többi szerverhez ez eljusson), így egy ideig nem tud belépni az új jelszavával.

***„Írás után olvasás" konzisztencia* (writes follow reads, WFR)**

Adott folyamat által az x adatelemen végrehajtott írási művelet, amely az x adatelemnek a folyamat által korábban történt olvasását követi, feltétlenül ugyanazt az értéket vagy annál frissebbet állít be, mint amit korábban olvasott.

Példa. Legyen az elosztott rendszer egy bibiliográfiai adatbázis, amelybe a felhasználók tudományos cikkek adatait tölthetik fel. Egy felhasználó olyan bejegyzést talál, ahol a cikk valamelyik adata (pl. az oldalszám) hibás és ezt a mezőt javítja. A WFR konzisztencia követelménye szerint az írás csak akkor jelenik meg a szerveren, ha az alapjául szolgáló bejegyzés már megjelent rajta. Így garantálható, hogy az írás minden szerveren az olvasott (és javított) bejegyzéshez tartozó értékeket állítja be [26] [28].

Az MR, MW, RYW és WFR modelleknek többféle kombinációja is elképzelhető, amelyekkel eltérő teljesítményű és konzisztenciájú rendszerek definiálhatók.

C.4.2. Adatközpontú konzisztenciamodellek

Az adatközpontú konzisztenciamodellek [26] alapján. Az erősebb garanciákat nyújtó modellektől haladunk a gyengébbek felé.

***Szigorú konzisztencia* (strict consistency)**

Adott adatelemen végrehajtott bármely olvasási művelet az ugyanezen az adatelemen végrehajtott legutolsó írási művelet értékével tér vissza. Ez a legerősebb adatközpontú konzisztenciamodell.

A definícióban szereplő „legutolsó" kitétel megvalósításához abszolút globális idő meghatározására lenne szükség, ezért elosztott rendszerben a szigorú konzisztencia megvalósítása gyakorlatilag lehetetlen. A többi adatközpontú konzisztenciamodell úgy enyhítjük a feltételeket, hogy definiáljuk az ütköző műveletek esetén még elfogadott viselkedést.

***Lineáris konzisztencia* (linearizability)**

A szigorú konzisztenciánál gyengébb modell. A modellben minden műveletet egy globális, de véges pontosságú óra által kiosztott időbélyeggel látunk el. $t_{OP}(x)$ az x adatelemen végrehajtott OP művelethez rendelt időbélyeg, ahol OP lehet írás (R) vagy olvasás (W). Az adattár akkor lineáris konzisztenciájú, ha az alábbi feltételeket teljesíti:

1. Bármely futás eredménye ugyanaz, mintha az adattáron dolgozó összes folyamat minden írási és olvasási műveletét meghatározott sorrendben hajtanánk végre, megőrizve bármely adott folyamat saját műveleteinek sorrendjét.
2. Ha bármely $OP1(x)$, $OP2(y)$ műveletpárra $t_{OP_1}(x) < t_{OP_2}(x)$, akkor a műveletsorban $OP1(x)$-nek meg kell előznie $OP2(y)$-t.

***Soros konzisztencia* (sequential consistency)**

Egy adattár sorosan konzisztens, ha a lineáris konzisztencia 1. feltételét teljesíti. A lineáris és a soros konzisztencia közötti különbség, hogy utóbbi nem tesz semmilyen megkötést a fizikai időre vonatkozóan. Soros konzisztenciánál egyidejűleg futó folyamatok esetén a konkurens írási és olvasási műveletek bármilyen sorrendje elfogadható, de mindegyik folyamatnak ugyanazt a végrehajtási sorrendet kell észlelnie.

Megjegyzés. A definíciókból következik, hogy ha egy rendszerben minden tranzakcióban csak egyetlen írás vagy egyetlen olvasás művelet szerepel, a tranzakciókezelés *sorosíthatóság* (serializability) fogalma megegyezik a soros konzisztenciával [32].

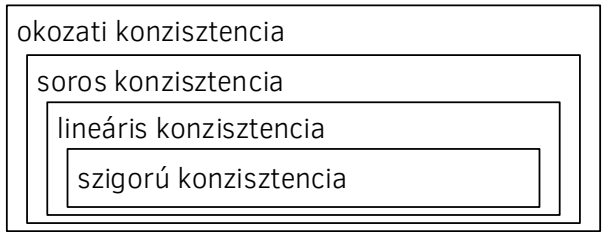
***Okozati konzisztencia* (causal consistency)**

Az okozati konzisztenciához be kell vezetnünk egy további definíciót. Két esemény *oksági viszonyban* (causally related, causality) van, ha:

- két írás ugyanazon az adategységen dolgozik,
- egy folyamatban egy olvasás művelet után írás következik (akár különböző adategységekre),
- egy olvasás egy írás eredményét adja vissza (az írás történhet bármely folyamatban) vagy
- két művelet a fentiek bármely kombinációjában (tranzitívan) függ egymástól.

Okozati konzisztencia: a potenciálisan oksági viszonyban lévő eseményeket a rendszer minden csomópontja ugyanabban a sorrendben látja. Azaz, ha egy B esemény egy korábbi A eseményből következik, vagy eredményét a korábbi A esemény befolyásolja, mindenki először az A eseményt látja és csak utána B-t.

Belátható, hogy ha egy rendszerben a (kliensközpontú) MR, MW, RYW és WFR konzisztenciamodellek feltételei teljesülnek, az okozati konzisztencia is fennáll [29]. Egyes szerzők ezért az okozati konzisztenciát kliensoldali modellként definiálják [27] [30].



C.6. ábra. Adatközpontú konzisztenciamodellek. A szűkebb halmazok modelljei erősebb garanciákat nyújtanak.

C.4.3. Elosztott tárolás

A *replikáció* (replication) azonos adategység többszörözését jelenti – ugyanazt az adategységet különböző szervereken, replikálva tárolják. A *sharding* különböző adategységek különböző szervereken tárolását jelenti. A két technológia tehát működhet külön-külön és együtt is.

Replikáció

Replikáció esetén fontos feladat a konzisztencia biztosítása. Például egy 3 szerveren replikált adategység esetén elég 2 csomóponton írni az adategységet. Így biztosítható, hogy a legutolsó írás művelet eredménye lesz látható a szerverek többségén. Az ún. *testületalapú protokollok* (quorum-based protocols) az írandó és az olvasandó adategységekre definiálnak küszöbértékeket, melyekkel képesek erős konzisztenciát nyújtani. Az adategységet N példányban tároljuk, a küszöbértékek:

- R: az olvasáshoz szükséges szerverek száma.

– W: az íráshoz szükséges szerverek száma.

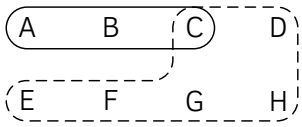
A küszöbértékekre a következő feltételeket definiáljuk [25] [26]:

1. $W > N \ / \ 2$
2. $W + R > N$

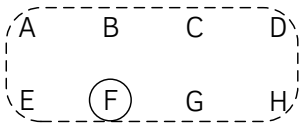
Az 1. feltétel miatt bármely két *írási testületnek* (write quorum) van közös szervere, ez lehetővé teszi az egymás utáni írások sorrendjének megfigyelését.

A 2. feltétel miatt az *olvasási* (read quorum) és az írási testületek mindig átlapolódnak, így erős konzisztencia biztosítható.

Megjegyzés. Az elosztott zárolásnál ismertetett „k az n-ből protokoll” is a fenti feltételeket kielégítő küszöbértékeket definiál a globális adategységek zárolására.



C.7. ábra. Írási (szaggatott vonal) és olvasási testületek ($N = 8,\ R = 3,\ W = 6$)



C.8. ábra. Írási (szaggatott vonal) és olvasási testületek ($N = 8,\ R = 1,\ W = 8$)

A C.7 és a C.8. ábrán a testületi taglétszámok helyes megválasztására láthatók példák. A C.8. ábra taglétszámválasztása „olvasáshoz egy, íráshoz mind” (Read One, Write All, ROWA) néven ismert.

Megjegyzés. A séma hasonlít az elosztott zárolás *write locks all* sémájához, ami a „k az n-ből protokoll” speciális esete $k = n$-re.

Fokozatos konzisztencia

A 2. feltétel sérülése, azaz $W + R \leq N$ esetén előfordulhat, hogy az olvasási és az írási halmazok diszjunktak. Ezekben a rendszerekben – mivel erős konzisztencia nem garantálható – az olvasási küszöbérték (R) tipikusan 1 és a frissítések valamilyen *lusta* (lazy) algoritmus szerint terjednek a rendszerben. Az inkonszitenciaablak ebben az esetben az az idő, amíg a frissítés az adategység minden replikáját el nem éri. Amennyiben garantálható, hogy egy kliens egy *munkameneten* (session) belül mindig ugyanahhoz a szerverhez fordul, az MR és az RYW

konszitenciamodellek feltételei viszonylag könnyen garantálhatók, de a *terheléselosztás* (load balancing) és a *hibatűrés* (fault tolerance) biztosítása nehezebbé válik [27].

C.5. A NoSQL rendszerek típusai

Az alábbiakban a NoSQL adatbázis-kezelők típusait ismertetjük, mely besorolás a NoSQL közösség által meghatározottnak felel meg – mindezt azért lényeges kiemelni, mert számos olyan rendszer létezik, mely nem sorolható be egyértelműen egyik csoportba sem [10] [35].

C.5.1. Kulcs-érték tárolók

A *kulcs-érték tárolók* (key-value stores) olyan egyszerű adatbázis-kezelők, melyek kulcsokat és a hozzájuk rendelt értékeket tárolják. Ennek megfelelően a kulcs-érték tárolók attribútum–attribútumérték párokat tartalmaznak. Az egyszerű adatmodell számos alkalmazási területen hasznos, azonban a lekérdezések korlátozottak; jellemzően csak kulcs szerint valósulhatnak meg. Bár az első kulcs-érték tárolót már a '70-es években megalkották, ezek a rendszerek is csak a többi NoSQL rendszer megjelenésének idején indultak fejlődésnek, lévén a megváltozott igényeket már nem lehetett hatékonyan kulcs-érték tárolóként használt relációs adatbázis-kezelőkkel kielégíteni. A kulcs-érték tárolás ötletét az oszlopcsaládok és a gráfadatbázisok (ld. alább) is alkalmazzák.

Példák kulcs-érték tárolókra: Berkeley DB, Memcached, Project Voldemort, Redis, Riak.

C.5.2. Oszlopcsaládok

Az oszlopcsaládok tárgyalásánál fontos megkülönböztetnünk az oszlopalapú tárolást az oszlopcsaládoktól.

Oszlopalapú adattárolás relációs adatbázis-kezelőkben

A napjainkban elterjedt *relációs* adatbázis-kezelők többsége a rekordokat fizikailag sorokba szervezve tárolják (ld. 3. fejezet). A *soralapú* (row-based, row-oriented) tárolás előnye, hogy egy sor beszúrásához, módosításához vagy lekérdezéséhez jellemzően csak néhány blokkművelet szükséges. A módszer hátránya, hogy nehezen tömöríthető struktúrákat képez, valamint kevés attribútumot érintő lekérdezések esetén sok felesleges adatot is be kell olvasni a háttértárról.

A soralapú szervezéssel ellentétes elgondolás, az *oszlopalapú* (column-based, column-oriented) szervezés ötlete a '80-as években merült fel. Az oszlopalapú tárolás a rekordok attribútumok szerinti csoportosítását jelenti, így az egyes attribútumok különböző értékei találhatók meg egy fizikai szervezési egységben. Ezzel a szervezéssel a kevés oszlopot és sok sort érintő elemzések hatékonyabban elvégezhetők, ezért előszeretettel alkalmazzák analitikus adatbázisokban, adattárházakban.

Az oszlopalapú rendszerek magas szelektivitású (kevés sort érintő) lekérdezések esetén kerülnek hátrányba, mert az oszlopok végigolvasása sok felesleges blokkművelettel jár. Az oszlopalapú tárolók az egyes csoportokban azonos típusú adatokat tárolnak, ezért általában hatékonyan tömöríthetők.

Az alábbiakban a *személy(ID, keresztnév, életkor, lakhely)* sémára illeszkedő relációt szemléltetjük először sor-, majd oszlopalapú tárolással.

ID	keresztnév	életkor	lakhely
1	Klemens	42	Stuttgart
2	Rajesh	29	Delhi
3	Francesco	30	Rome
4	Colin	51	Dublin

Soralapú tárolás:

1. blokk	1	Klemens	42	Stuttgart
2. blokk	2	Rajesh	29	Delhi
3. blokk	3	Francesco	30	Rome
4. blokk	4	Colin	51	Dublin

Oszlopalapú tárolás:

1. blokk	1	2	3	4
2. blokk	Klemens	Rajesh	Francesco	Colin
3. blokk	42	29	30	51
4. blokk	Stuttgart	Delhi	Rome	Dublin

Példák oszlopalapú relációs adatbázis-kezelőkre: Sybase IQ, Vertica.

A NoSQL oszlopcsaládok

Az elmúlt években megjelentek olyan NoSQL rendszerek, melyek az oszlopalapú alapelvet az adatmodellben alkalmazzák és kiegészítik más – jellemzően kulcs-érték – módszerekkel: ezek az *oszlopcsaládok* (column families). Az oszlopalapú adatbázisokhoz képest lényeges különbség, hogy míg azok relációk fizikai tárolását valósítják meg oszlopalapon, addig az oszlopcsaládok egy hierarchikus – nem relációs – adatmodell szerint, *logikailag* oszlopalapon szervezik az adatokat.

Az oszlopcsaládok sorai (rekordjai) kulcs-érték párokból állnak, melyek az oszlop (attribútum) nevét és értékét tartalmazzák. Számos implementáció tartalmaz az egyes kulcs-érték párokhoz tartozó időbélyegeket is (ld. 10.9.4. alszakasz). Egy oszlopcsalád egy kulcs-érték párból áll, ahol a kulcs egy tetszőleges elsődleges kulcs, az érték pedig az előbbiekben leírt oszlopok egy halmaza.

kulcs	oszlopkulcs0	oszlopkulcs1	...	oszlopkulcsN
	érték0	érték1	...	értékN

C.9. ábra. Oszlopcsalád általános felépítése

A C.10. ábra a népszerű közösségi mikroblog szolgáltatás, a Twitter *bejegyzés*einek (postok) tárolásának egy lehetséges módját mutatja be oszlopcsaládokkal. Az oszlopcsaládok – rekordok – kulcsa a bejegyzés egyedi azonosítója, az egyes oszlopok pedig rendre a felhasználónevet (mely a Twitteren egyben egyedi azonosító is), magát a bejegyzést, illetve annak dátumát tartalmazzák.

12100	user_ID	text	datetime
	bmestudent	just decomposed a schema to 3NF #db #exam	2011-01-03 07:30:11

12187	user_ID	text	datetime
	bmestudent	just decomposed a schema to BCNF #db #exam	2011-01-03 07:41:36

C.10. ábra. Twitter bejegyzések tárolása oszlopcsaláddal

Az oszlopcsaládokat egy további kulcs-érték szinttel bővítve *szuperoszlopcsaládokat* (super column family) kapunk. Az így létrejövő szinteket *szuperoszlopoknak* (super column) nevezzük. A C.11. ábra szemlélteti a szuperoszlopcsaládok általános felépítését.

kulcs	szuperoszlop kulcs0				...	szuperoszlop kulcsN			
	oszlop kulcs0	oszlop kulcs1	...	oszlop kulcsN		oszlop kulcs0	oszlop kulcs1	...	oszlop kulcsN
	érték0	érték1	...	értékN		érték0	érték1	...	értékN

C.11. ábra. Szuperoszlopcsalád általános felépítése

Az előbbi példához kapcsolódóan a C.12. ábrán az egy felhasználó által leggyakrabban hivatkozott weboldalakat összegyűjtő szuperoszlopcsaládot láthatunk. Ebben az esetben egy szuperoszlopcsalád kulcsa a felhasználó neve, a szuperoszlopoké pedig a hivatkozott weboldalak URL-je. A szuperoszlopokat alkotó oszlopok kulcsai a szuperoszlopban található URL által kijelölt oldalnak azon aloldalai, melyekre a felhasználó leggyakrabban hivatkozott. Az oszlopok érték mezői az adott aloldalra való legutóbbi hivatkozás idejét mutatják.

	tmit.bme.hu		en.wikipedia.org	
bmestudent	/History	/InfoBScMedia	/wiki/ Relational_schema	/wiki/ Tuple_calculus
	2011-01-06 11:21:14	2011-01-06 11:25:01	2010-12-14 03:22:31	2010-12-14 04:00:56

C.12. ábra. Hivatkozott oldalak tárolása szuperoszlopcsaláddal

Példák (szuper)oszlopcsaládokra: Amazon SimpleDB, BigTable, HBase, Cassandra, HyperTable.

C.5.3. Dokumentumtárolók

Napjainkban a szemistrukturált adatok (ld. B. függelék) egyre nagyobb jelentőséggel bírnak, így például tartalomkezelő, kereső-, ajánlórendszerekben. A szemistrukturált adatok különleges tulajdonságai tették szükségessé az ún. *dokumentumtároló* (document store) adatbázis-kezelők megalkotását. A dokumentumtárolókban szemistrukturált adatok tárolhatók, melyeket jellemzően JSON vagy XML nyelveken írnak le [35] [36]. A dokumentumtárolók tehát nem szövegfájlokat kezelnek, hanem adatok olyan halmazát, melyek valamilyen laza módon strukturáltak.

Az alábbiakban egy JSON (JavaScript Object Notation) dokumentumra láthatunk példát, mely személyek jellemzőit és járművek egy listáját tartalmazza. Ha egy relációban szeretnénk tárolni a példa dokumentum tartalmát, igencsak bajban lennénk, ugyanis számos, ritkán használt attribútumot kéne felvennünk a relációs sémába. Az adathalmazt dokumentumként kezelve azonban nem okoz problémát, hogy Klemensről és Colinról más-más információkkal rendelkezünk (pl. age, company).

Vegyük észre, hogy a dokumentumok is kulcs-érték párokat tartalmaznak – jelen esetben egyes entitások attribútum típusait és attribútum értékeit (pl. city–Stuttgart).

```
{"document": [
       {
              "firstname": "Klemens",
              "city": "Stuttgart",
              "age": "42"
       },
       {
              "firstname": "Rajesh",
              "city": "Delhi",
              "age": "29"
       },
       {
              "firstname": "Colin",
              "company": "Oracle"
```

```
        },
        {
                "cars": ["BMW 320d", "Jaguar XF"]
        }
]}
```

Példák dokumentumtárolókra: CouchDB, MongoDB, Terrastore.

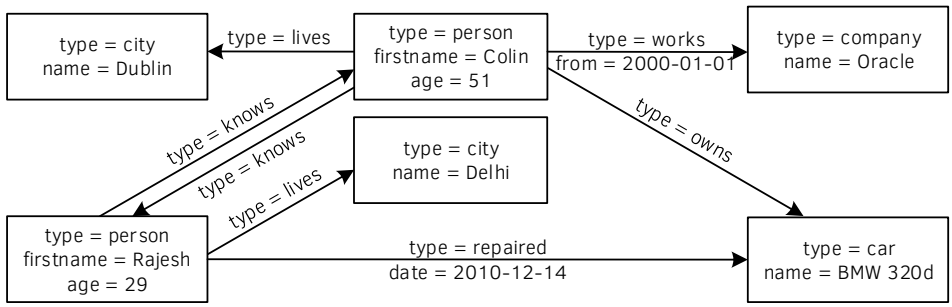
C.5.4. Gráfadatbázisok

Komplex, sok összefüggést tartalmazó adathalmazokat gyakran célszerű gráffal reprezentálni. Sajnos a gráfok relációs adatbázis-kezelőkben való tárolása esetén a gráfokon végzett műveletek rendkívül költségesek lehetnek: egy gráf bejárásához például több természetes illesztésre lehet szükség, ami köztudottan költséges művelet.

A *gráfadatbázisok* (graph database) gráfok hatékony tárolását és ezáltal gráfműveletek gyors végrehajtását teszik lehetővé [35].

A gráfadatbázisok jellemzően *tulajdonsággráfokat* (property graph) tárolnak, melyek csomópontjaihoz és éleihez tulajdonságok köthetők – rendszerint kulcs-érték párok formájában. Ezen tulajdonságok között jellemzően megtalálható az adott csomópontok és élek típusa is.

A C.13. ábrán látható tulajdonsággráfról például a következő állítást olvashatjuk le: az 51 éves Colin Dublinban lakik.



C.13. ábra. Tulajdonsággráf

A gazdag adatmodell miatt a gráfadatbázisok jellemzően kevésbé skálázódnak a többi NoSQL rendszernél, a legtöbb gráfadatbázis csak replikációt támogat.

Példák gráfadatbázisokra: Neo4j, AllegroGraph, HypergraphDB, InfiniteGraph, FlockDB.

C.5.5. További NoSQL típusok

A részletesebben tárgyalt négy nagy csoport (core NoSQL rendszerek) mellett találhatók kisebb, kevésbé elterjedt típusok (soft NoSQL):

– objektum adatbázisok, pl. DB4o, Versant, ZODB,
– XML adatbázisok (bár külön csoportként tartják számon, tulajdonképpen dokumentumtárolók), pl. EMC xDB, eXist,
– grid adatbázisok, pl. GigaSpaces, Hazelcast.

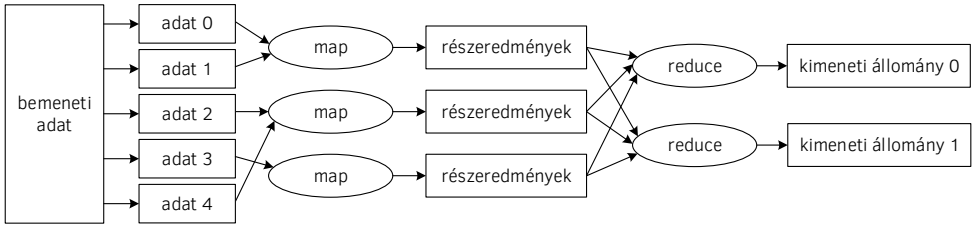
C.6. MapReduce

A MapReduce a NoSQL korszak jellegzetes, elosztott adatfeldolgozó algoritmusa, melyet a Google kifejezetten nagy adathalmazok párhuzamos feldolgozására fejlesztett ki [4] [34]. Valójában nem tartozik szorosan az adatbázis-kezeléshez, azonban sikeressége/hatékonysága miatt több NoSQL rendszerben, ill. hozzájuk kapcsolódva is megtalálható.

A MapReduce paradigma a funkcionális nyelvek (pl. Erlang, Haskell, LISP) `map()` és `reduce()` (egyes nyelvekben `fold()`) eljárásaiban gyökerezik. A map a bemeneti lista minden elemére végrehajt egy meghatározott műveletsort, és visszatér a módosított listával, amit a reduce eljárás aggregál egy végeredménnyé.

A MapReduce keretrendszerek a map és a reduce eljárásokat párhuzamosan hajtják végre. Az eljárásokat a fejlesztőnek kell implementálnia annak megfelelően, hogy milyen feladatokat kíván egyidejűleg végrehajtani. A MapReduce keretrendszerekben az adatok feldolgozása a következő fázisokra tagolódik (ld. C.14. ábra):

1. Bemeneti adatok felosztása az egyes map processzek részére.
2. Párhuzamos adatfeldolgozás a map processzekben.
3. Részeredmények tárolása és kulcs szerinti rendezése – az azonos köztes kulccsal rendelkező részeredmények kerülnek ugyanahhoz a reduce processzhez.
4. Részeredmények párhuzamos feldolgozása a reduce processzekben.
5. Kimeneti adat(ok) írása reduce folyamatonként.



C.14. ábra. A MapReduce fázisai

A map metódus implementációja kötelező, míg a reduce-é jellemzően opcionális. Lehetséges továbbá a bemeneti adatok felosztásának, a köztes eredmények csoportosításának, illetve a végeredmények összesítésének egyedi implementációja is, azonban alapesetben ezeket a feladatokat a keretrendszerek végzik. Lényeges, hogy a bemeneti, a köztes és a kimeneti adatoknak kulcs-érték formátumban kell

lenniük. Megjegyzendő továbbá, hogy egy bemeneti kulcs-érték párból nem feltétlenül egy köztes értékpár lesz a map és/vagy a reduce folyamatok során (pl. egy szöveges értéket karakterenként is feldolgozhatunk).

A reduce függvény implementálásakor eleget kell tennünk az alábbi szempontoknak:

- A függvény bemeneti típusa egyezzen meg a map függvény kimeneti típusával. Így ha a map függvény kimenetén csak egyetlen kulcs-érték pár jelenik meg, a reduce függvényt nem kell futtatni.
- A művelet legyen idempotens, azaz ha függvény kimenetét egy (egyetlen elemet tartalmazó) tömbbe helyezzük és újra meghívjuk rá a reduce függvényt, az eredmény maradjon változatlan. A reduce függvényt ugyanis a feldolgozás során többször is lefuthat, például a feldolgozás végén a master szerver egy reduce művelettel összesíti a különböző szerverektől kapott adatokat [34].

Az alábbiakban a MapReduce alkalmazására láthatunk egy példát, mely az egyes URL-ekre hivatkozó külső URL-eket gyűjti össze.

A map függvény URL-eket és az alattuk található weboldalak forráskódját kapja bemenetként, eredményül pedig `<cél, forrás>` párokat ad, ahol cél egy olyan weboldal URL-je, amelyre a feldolgozott – forrás – oldal hivatkozik.

A reduce függvény az egyes cél URL-ekhez tartozó forrás URL-eket fűzi össze egy-egy listába.

Az ismertetett példa map és reduce függvényeinek pszeudokódja a következő:

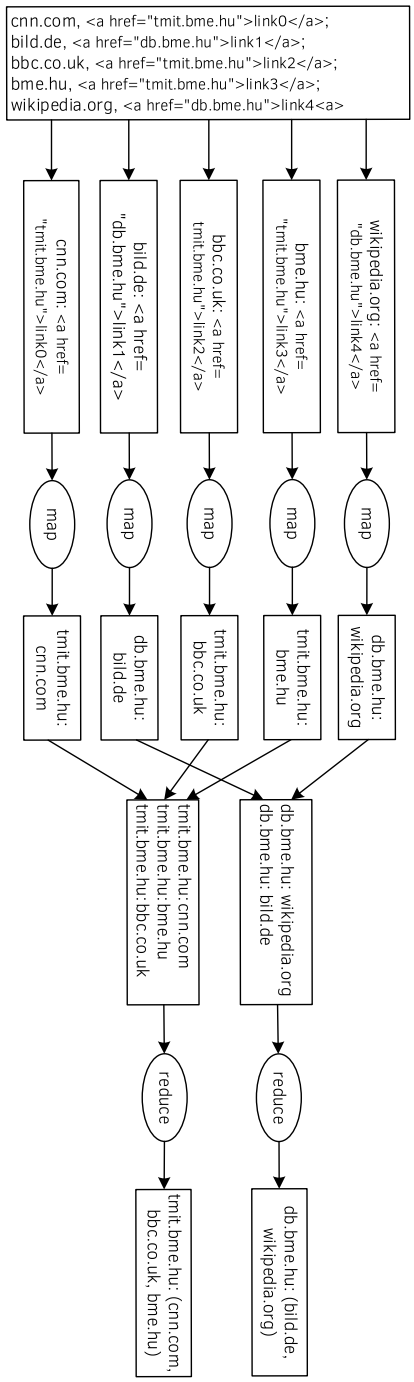
```
map(key, value) {
       // key = URL
       // value = page content
       For each url, linking to target
              Generate(output target, source);
}

reduce(key, values) {
       // key = target URL
       // values = all URLs that point to the target URL
       For each values
              Generate(key, values[i]);
}
```

A függvények működése a C.15. ábrán látható.

C.7. Összegzés

Láthattuk, hogy a NoSQL adatbázis-kezelők a korábbiakban megismert rendszerekhez képest újszerű igényeket elégítenek ki évtizedes és friss ötletek kombinálásával. Az ilyen rendszerek száma az utóbbi években folyamatosan növekedett – ez a tendencia a jövőben is igaznak bizonyulhat.

A NoSQL rendszerek térnyerése mellett a relációs adatbázis-kezelők szerepe továbbra is jelentős marad, lévén a klasszikus – és tömeges – igényekre még mindig ezek tekinthetők a leggazdaságosabb megoldásnak.



C.15. ábra. A MapReduce működése

C.8. Irodalomjegyzék a C függelékhez

[1] IBM, *What is Big Data? – Bringing Big Data to the Enterprise,* http://www-01.ibm.com/software/data/bigdata/

[2] Louis Columbus, *Roundup of Big Data Forecasts and Market Estimates,* Forbes, 2012, http://www.forbes.com/sites/louiscolumbus/2012/08/16/roundup-of-big-data-forecasts-and-market-estimates-2012/

[3] Edd Dumbill, *Volume, Velocity, Variety: What You Need to Know About Big Data,* Forbes, 2012, http://www.forbes.com/sites/oreillymedia/2012/01/19/volume-velocity-variety-what-you-need-to-know-about-big-data/

[4] Jeffrey Dean, Sanjay Ghemawat, *MapReduce: Simplied Data Processing on Large Clusters,* Google Inc., OSDI, 2004, http://research.google.com/archive/mapreduce.html

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data,* Google Inc., OSDI, 2006, http://research.google.com/archive/bigtable.html

[6] Mike Burrows, *The Chubby lock service for loosely-coupled distributed systems,* Google Inc., OSDI, 2006, http://research.google.com/archive/chubby.html

[7] *no:sql(east),* https://nosqleast.com/2009/

[8] Couchbase, *NoSQL Database Technology – Post-relational data management for interactive software systems,* 2011, http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQL-Whitepaper.pdf

[9] Pramod J. Sadalage, Martin Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence,* Addison-Wesley Professional, 2012.

[10] *NoSQL Databases,* http://nosql-database.org/

[11] Mark D. Hill, *What is scalability?,* http://dl.acm.org/citation.cfm?id=121975

[12] Werner Vogels, *Availability & Consistency or how the CAP Theorem ruins it all,* QCon, 2007, http://www.infoq.com/presentations/availability-consistency

[13] Michael Stonebraker, *The Case for Shared Nothing,* HPTS, 1985, http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf

[14] Armando Fox, Eric Brewer, *Harvest, Yield, and Scalable Tolerant Systems,* Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems, 1999, http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=798396&tag=1

[15] Eric Brewer, *Towards Robust Distributed Systems,* PODC Keynote, 2000. július 19, http://www.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[16] Seth Gilbert, Nancy Lynch, *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,* http://portal.acm.

org/citation.cfm?id=564601

[17] Julian Browne, *Brewer's CAP Theorem,* 2009, http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

[18] Nancy Ann Lynch, *Osztott Algoritmusok,* Kiskapu, 2002.

[19] Ron Kohavi, Roger Longbotham, *Online Experiments: Lessons Learned,* IEEE Computer, 2007. szeptember, http://ai.stanford.edu/users/ronnyk/IEEEComputer2007OnlineExperiments.pdf

[20] Daniel Abadi, *Problems with CAP, and Yahoo's little known NoSQL system,* 2010. április 23, http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels, *Dynamo: Amazon's Highly Available Key-value Store,* SIGOPS Oper. Syst. Rev., 2007. december, http://dl.acm.org/citation.cfm?id=1294281

[22] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, Paul Gauthier, *Cluster-Based Scalable Network Services,* SOSP, 1997, http://dl.acm.org/citation.cfm?id=266662

[23] Dan Pritchett, *BASE: An Acid Alternative,* ACM Queue, http://queue.acm.org/detail.cfm?id=1394128

[24] GenieDB, *Beating the CAP Theorem,* http://www.geniedb.com/wp-content/uploads/2011/04/beating-the-cap-theorem-revised.pdf

[25] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, Ramana Yerneni, *PNUTS: Yahoo!'s Hosted Data Serving Platform,* 2008, http://research.yahoo.com/pub/2304

[26] Andrew S. Tanenbaum, Maarten Van Steen, *Elosztott rendszerek,* Panem, 2004.

[27] Werner Vogels, *Eventually Consistent,* ACM Queue, 2008. október, http://dl.acm.org/citation.cfm?id=1466448&bnc=1

[28] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, Brent B. Welch, *Session Guarantees for Weakly Consistent Replicated Data,* 1994, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.71.2269

[29] Jerzy Brzezinski, Cezary Sobaniec, Dariusz Wawrzyniak, *From Session Causality to Causal Consistency,* 2004, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.3608&rep=rep1&type=pdf

[30] Roger Wattenhofer, *Distributed Systems course,* http://www.disco.ethz.ch/lectures/hs12/distsys/

[31] Todd Lipcon, *Design Patterns for Distributed Non-Relational Databases,* cloudera, 2009, http://cloudera-todd.s3.amazonaws.com/nosql.pdf

[32] M. Raynal, G. Thia-kime, M. Ahamad, *From Serializable to Causal Transactions for Collaborative Applications,* 1996, `http://reference.kfupm.edu.sa/content/f/r/from_serializable_to_causal_transactions_126622.pdf`

[33] Tom White, *Hadoop: The Definitive Guide, Second Edition,* O'Reilly Media, 2010.

[34] MongoDB, MapReduce, `http://www.mongodb.org/display/DOCS/MapReduce`

[35] Edlich, Friedland, Hampe, Brauer, *NoSQL,* Hanser, Berlin, 2010.

[36] J. Chris Anderson, Jan Lehnardt, Noah Slater, O'Reilly Media, 2010.

A weblapok elérési ideje: 2012. augusztus

Tárgymutató