A relációs adatbázisoktól az XML adatokig

Tartalomjegyzék

Előszó

Jelen könyv az adatbázis elmélet fogalmait tartalmazza, melyet használhatnak informatikus hallgatók egyetemi jegyzetként, valamint adatbázis alkalmazás programozók. Napjainkban a legelterjedtebbek a relációs adatbázis-kezelő rendszerek, ez a könyv is részletesen a relációs adatmodellt tárgyalja, de az olvasó megismerkedhet az újakkal is, mint az objektumorientált és féligstrukturált adatmodell.

A relációs adatbázis-kezelő rendszerek hatékonyan tudnak kezelni nagyon nagy mennyiségű adatot. Egyidejűleg több felhasználó között megosztják az adatbázist, ellenőrzik a hozzáférési jogokat, rendszerhibák esetén képesek egy helyes adatbázist visszaállítani.

Az adatbázis-kezelő rendszerek különböző indexállományok létrehozásával biztosítják a gyors hozzáférést. A relációs adatmodell segítségével a felhasználó megtervezheti az adatbázist, ugyanakkor segít a felhasználónak abban, hogy az adatot ne csak bitek sorozataként lássa, hanem érthetőbb formában. Magas szintű programozási nyelvekkel rendelkezik az adatok szerkezetének a leírására, adatkezelésre és lekérdezésre.

A szakirodalomban már sok adatbázis könyv jelent meg. Egyik ezek közül a Chris Date könyve már a nyolcadik kiadását is megérte (lásd [Da04]), melynek román nyelvre való fordítása az Editura Plus kiadónál jelent meg. Egy másik adatbázis szakember Jeffrey D. Ullman, az ő könyvei: [Ul88], [Ul89], [UlWi97], [GaUlWi00]. Az utóbbi kettő megjelent magyar fordításban a Panem kiadó kiadásában.

Az adatbázisok elméleti alapjaival kezdődően, az első fejezet bemutatja az adatbázis-kezelő rendszerek (ABKR) előnyeit, tulajdonságait és szerkezetét. Egy kis bevezetőt kapunk az indexállományokról is.

Az „*Adatmodellek*” című fejezet az adatmodellek osztályozásával kezdődik és bővebben tárgyalja az egyed/kapcsolat adatmodellt. Egyed, egyedhalmaz, attribútum és kulcs fogalmakkal ismerkedhetünk meg. Ugyanez a fejezet tartalmazza az egyed/kapcsolat diagram leírását, melyet az adatbázis fogalmi sémájának a tervezésére használhatunk.

A harmadik fejezettel kezdődően az olvasó megismerkedik a relációs adatmodellel, ahol megtalálja a leírását és a tulajdonságait. Egy adatbázis alkalmazás megvalósításában nagyon fontos lépés az adatbázis sémájának a tervezése. Két módszerrel kaphatunk helyes relációs adatbázis sémát: átalakítjuk az egyed/kapcsolat diagramot relációs sémává vagy normalizálással. A fejezet tartalmazza mindkét módszert, az első esetében egy algoritmust ad. A második módszer esetében bemutatja a normál formákat (1NF, 2NF, 3NF és BCNF) és a relációk felbontását, hogy megfelelő normál formájúvá alakítsuk a relációt.

A legelterjedtebb relációs adatbázis nyelv az SQL, melynek elfogadott standardja az SQL2. A könyv általában ezt a standardot mutatja be, egyes esetekben megadja a különbségeket Oracle és MS SQL Server esetén. Az SQL nyelvben lehetőségünk van relációs táblákat létrehozni, helyességi megszorításokat megadni, mint például egyedi kulcs, külső kulcs, attribútum értékre vonatkozó megszorítás, globális megszorítások. Ez az adatleíró nyelv a negyedik fejezetben található.

Minden adatmodell műveleteket is tartalmaz. A relációs adatmodell esetén a relációs algebra egy nyelv, melynek segítségével a relációs adatokon műveleteket tudunk végezni. Az ötödik fejezetben arra találunk példákat, hogyan fogalmazzuk meg a lekérdezéseket relációs algebra segítségével. Szintén ez a fejezet tartalmazza az operátorok algebrai tulajdonságait.

„*Az SQL lekérdező nyelv*” című fejezet sok példa segítségével bemutatja, hogyan fogalmazzuk meg a lekérdezéseket SQL nyelvben. Egyszerűbb lekérdezésekkel kezdődően, eljutunk a komplexebbekig, mint például korrelált alkérdések, csoportosítások és összesítő függvények.

A következő fejezet a nézetek fogalmába ad betekintést. Választ kapunk a következő kérdésekre: mire használhatjuk a nézeteket, mikor módosítható egy nézet.

Az adatbázisok biztonsága az azonos című fejezet témája. Kétféle adathozzáférési módszert mutatunk be, illetve egy kis ízelítőt az Oracle biztonsági renszeréből.

Adatbázisok kliens/szerver architektúráját tárgyalja a következő fejezet, ahol az olvasó részleteket talál az alkalmazások kliens/szerver programozásával kapcsolatban.

Egy erős mechanizmusa az SQL3-nak a triggerek, mely a legtöbb kereskedelmi ABKR-ben már meg is van valósítva. A „*Triggerek*” című fejezet megadja a triggerek leírását, a trigger tervezési elveket, illetve példákat MS SQL Server-ben és Oracle-ban.

Az adatbázisokat egyidejűleg több felhasználó is használja. A különböző felhasználók által végzett módosítások tartósak kell legyenek, még ha hiba lép is fel, az ABKR képes kell legyen egy helyes adatbázis visszaállítására. A tranzakciók mechanizmusa megoldja a helyesség, tartósság és konkurencia problémáit. A „*Tranzakciókezelés*” című fejezet megadja a tranzakció fogalmát. Részletes leírását találjuk a hiba esetén történő helyreállításnak. Háromféle naplózást mutatunk be: semmisségi, helyrehozó és semmisségi/helyrehozó naplózást, illetve mindhárom esetben a helyreállítást ellenőrzőpont képzéssel.

Ami a konkurenciavezérlest illeti, bemutatjuk a három konkurencia problémát: elveszett módosítás, „piszkos” adat olvasása és inkonzisztens analízis. A fejezet végéig megoldódnak a problémák. Megadjuk a sorbaállíthatóság fogalmát, különböző lezárásokat vezetünk be, illetve a szigorú két-fázisú lezárási protokollt. Az ütemező megvalósításáról is elég részletes leírást találunk. A gyakorlatban általában megengednek nagyobb interferenciát a tranzakciók között, mint amit az elmélet kér, a sorbaállíthatóságot. Ezt az interferenciát megadhatjuk az SQL nyelv segítségével, ha beállítjuk a tranzakció izolálási szintjét.

A következő fejezet az osztott adatbázisok fogalmát vezeti be, majd az osztott adatbázisrendszerek problémáit mutatja be. Egy cégnek vagy szervezetnek több helyen is lehet kirendeltsége, minden kirendeltségben találhatók számítógépek lokális hálózatba kötve. Minden kirendeltségnek megvan a maga feladata, adatai és nagy valószínűséggel adatbázis szervere. A kirendeltségek kommunikációs hálózattal össze vannak kapcsolva, együtt egy rendszert alkotnak. Mikor lesz ez osztott adatbázis, ebből a fejezetből azt is megtudhatjuk.

A 90-es években nagy fejlődésnek indultak az objektum-orientált adatbázisok, egy kis ízelítőt kapunk ezek tervezéséről és lekérdezéséről a 13. fejezetben.

Ahogy a világhálón levő adat-mennyiség napról-napra nő, mind több és több adat jelentkezik féligstrukturált formában. Ezalatt azt értjük, hogy, bár az adatnak lehet valamilyen értelemben struktúrája, ez a struktúra nem annyira merev vagy teljes, mint ami szükséges a hagyományos adatbáziskezelő rendszerekhez. A féligstruktúrált adatról, annak XML (Extensible Markup Language) nevű nyelvben való ábrázolásáról és lekérdezéséről a 14. fejezetben talál az olvasó leírást.

A szerző 2005 május 20

1. Bevezető

1.1 Adat és információ

Az ember társas lény, azért kapott társakat, hogy legyen, akikkel együttműködve meg tudja valósítani élete értelmét. Megtalálja azokat, akikre támaszkodhat és megkeresse a támogatandókat. Egész életünk a másokkal való *kommunikáció*, vagyis az ismeretek cseréje határozza meg. A magát tudatos lénynek valló embernek el kell sajátítania a kommunikálás megfelelő módját. Ehhez pedig sokat kellene tudnia magáról az *ismeret*ről.

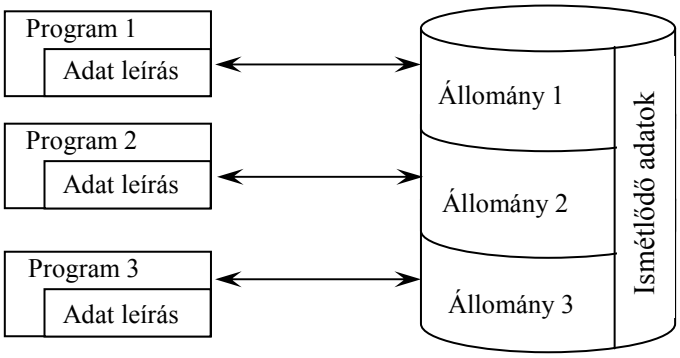
Az *adat* és az *információ* fogalmak az *ismeret* rokonai. Az adatkezelés és adatfeldolgozás szolgáltatás. Ezt a szolgálatot a szakemberek csak úgy tudják mindenki megelégedésére ellátni, ha az ismeretet nem a számítógép, hanem az ember oldaláról nézik.

Az *adat* szó a latin *datum* szóból ered, jelentése „adomány”, „ajándék”, egyébként „adottat” jelent, többes számban „data” vagyis „adottak”, az adott dolgok. Az IBM adatfeldolgozási szótárában: „Az adat tények, fogalmak olyan formalizált reprezentációja (megjelenítése), amely alkalmas az emberi vagy automatikus eszközök által történő kommunikációra, értelmezésre vagy feldolgozásra.”

1.1. Az adatbázis lényege

1.1.1. Állománykezelők

Az ismeretkezelés egyik lehetősége, hogy adatállományokat hozunk létre. Az „állomány” megjelölés azt mutatja, hogy nem egyedi dolgokról, hanem egymással valamilyen szempontból összetartozó ismeretek együtteséről van szó. Az „adat” jelző az ismeretkezelés módjára utal, és megkülönbözteti az ilyen állományokat a program-, rendszer-, szöveg-, képállományoktól.



1.1. ábra: Hagyományos állománykezelés

Léteznek olyan programozási nyelvek, programok, azokból összeállított rendszerek, amelyek az állományok egymástól független kezelésére használatosak. Ezeket az eszközöket *állománykezelők*nek (*file management system*) hívjuk. Az állománykezelők esetén felhasználói program tartalmazza az adatok szerkezetének a leírását. Ha például új mezőt szúrunk be egy állományba, minden felhasználói programot módosítanunk kell, mely az illető állománnyal dolgozik.

1.1. példa: Lássunk egy rosszul szerkesztett állományt:

Cím	*Név*	*Szín*	*Típus*	*Férőhely*
Farkas utca	Mária	fehér	Opel Astra	ötszemélyes
Kertész utca	Tamás	fehér	Opel Astra	ötszemélyes
Farkas utca	Mária	piros	VW Polo	négyszemélyes
Kertész utca	Tamás	zöld	VW Polo	négyszemélyes

Itt nemcsak azt ismételjük, hogy az Opel Astra ötfős, hanem azt is, hogy a Volkswagen Polo négyszemélyes. Ismételten tároljuk Mária címét, illetve Tamásét. A többszörös ismeretmegadás nemcsak a tárolót fogyasztja feleslegesen, hanem adatbeviteli, -módosítási és -törlési erőforrás-pazarlással is jár. Ha például Mária átköltözik a Hajnal utcába, a címét mindenhol meg kell változtatnunk. Ha ezt nem tesszük, nem lehet majd tudni, melyik is Mária valódi címe, azaz *inkonzistencia* lép fel. Ha viszont csak Mária lakóhelyére vagyunk kíváncsiak, miért kellene átböngésszük a kocsijait is?

A kocsi ismeretek ésszerű elrendezése:

Tulajdonos	
Név	***Cím***
Mária	Farkas utca
Tamás	Kertész utca

Kocsi		
Típus	***Szín***	***Név***
Opel Astra	fehér	Mária
Opel Astra	fehér	Tamás
VW Polo	piros	Mária
VW Polo	zöld	Tamás

Kocsitípus	
Típus	***Férőhely***
Opel Astra	5 személy
VW Polo	4 személy

Kapcsolatok: Tulajdonos – Kocsi a Név-en keresztül
Kocsitípus – Kocsi a Típuson át

Első ránézésre bonyolultabb, de nincs redundancia, inkonzisztencia. □

Az állománykezelők alkalmazása esetén is lehetőség van arra, hogy több állományt tervezzünk és az azokban tárolt ismereteket egy programmal kezeljük. Azonban az állományközti összefüggéseket nem tudjuk előre meghatározni. Ezekre a viszonyokra a programjainkban nem tudunk hivatkozni.

Ezzel szemben az adatbázisszerű adatkezelésben az állományok közötti általános logikai összefüggések absztrakt képét előre meghatározhatjuk, sőt meg kell határoznunk. Ezek után a konkrét ismereteket az előbbi képnek megfelelően lehet, sőt kell kezelnünk. Az adatbázis-kezelő rendszer (ABKR) ezt nemcsak lehetővé teszi, hanem ki is kényszeríti. A programozó nem a saját ízlése szerint, hanem az előre megadott (pl: Kocsitípus – Kocsi) viszonynak megfelelően kell kezelje az ismereteket.

1.1.2. Adatbázis értelmezése

Adatbázis: adatok gyűjteménye (összessége), melyek egy szervezet/cég tevékenységére vonatkoznak.

1.2. példa: Legyen egy adatbázis, mely egy egyetem tevékenységét tárolja, tartalmazhatja a következő információkat:
- Egyedek: diákok, karok, előadások, termek, tanárok stb.;
- Kapcsolatok az egyedek között: melyik diák milyen előadást választ; egy karon tartott előadások; melyik tanár milyen előadást tart; termek használata. □

1.3. példa: Legyen egy adatbázis, mely egy kereskedelmi cég adatait tárolja, itt az adatelemek között lehetnek:
- Egyedek: szállítók, árucsoportok, áruk, vevők, rendelések;
- Kapcsolatok egyedek között: mely szállító milyen árukat ajánl, egy áru melyik árucsoporthoz tartozik, egy vevő rendelései, egy rendelés összes áruja stb. □

1.1.3. Adatbázis-kezelő rendszer értelmezése

Adatbázis-kezelő rendszer (ABKR): egy programrendszer (soft), melyet arra terveztek, hogy nagy mennyiségű adatot tudjon tárolni és használni.

Egy ABKR különböző technikákat használ más doméniumokból, mint például programozási nyelvek (objektum-orientáltságot is), operációs rendszerek, konkurens programozás, adatstruktúrák, algoritmusok, párhuzamos és osztott rendszerek, felhasználói felületek, mesterséges intelligencia, statisztika.

E könyv célja válaszolni a következő kérdésekre:

1. Hogyan tud a felhasználó leírni egy létező szervezetet/céget az ABKR-ben tárolt adatok segítségével? Hogyan szervezze az adatokat?
2. Hogyan tud a felhasználó válaszolni a kérdésekre a céggel kapcsolatban, felhasználva az adatbázisban tárolt adatokat?
3. Hogyan tud az ABKR több felhasználóval egyidejűleg dolgozni? Hogyan védi az adatokat esetleges rendszerhibák esetén?
4. Hogyan tárolja az ABKR a nagy mennyiségű adatokat. Hogyan tud hatékonyan válaszolni a lekérdezésekre?

Ahhoz, hogy hatékonyan tudjuk az ABKR–t használni meg kell értsük, hogyan dolgozik.

ABKR versus Állománykezelő rendszer:

Legyen egy cég, melynek adatelemei a részlegek, alkalmazottak, termékek, eladási szerződések, anyagbeszerzés, termelési folyamat, bankügyletek, könyvelés stb. Ezen adatokhoz konkurensen több felhasználó (alkalmazott) is hozzá kell tudjon férni. Nem minden felhasználó láthat minden adatot (pl. fizetés, könyvelés). A lekérdezésekre gyorsan kell válaszolni. A különböző felhasználók által végzett változtatásokat az adatokon mindenhonnan látni kell.

Megpróbálhatjuk ezen feladatokat az operációs rendszer által nyújtott file-kezelés segítségével megoldani. Az állományok kb. 500 GB-ot foglalnak. Ha az állománykezelőt használjuk a következő problémákkal saját magunknak kell elboldogulni:
- Valószínű, hogy háttértárolón kell tároljuk a rengeteg adatot és azt a részét behozni a memóriába, amelyre épp szükség van;
- Operációs rendszer csak jelszó segítségével tud valamennyi védelmet nyújtani, de ez nem megfelelő, mikor különböző felhasználók az adatok különböző részeihez férhetnek hozzá;
- Speciális programokat kell írjunk, hogy minden lehetséges lekérdezésre választ adjunk. Ezek elég összetettek, mert nagy mennyiségű adatot kell végigjárjanak;
- Meg kell védjük az adatokat a helytelen változtatásoktól (többen férnek hozzá, rendszerhiba). Így még komplexebbek lesznek a programjaink;
- Rendszerhiba után az adatbázis egy helyes állapotát kell visszaállítani; ez a mi feladatunk.

Ha adatbázis-kezelő rendszert használunk, ezeknek a feladatoknak nagy részét átveszi az ABKR.

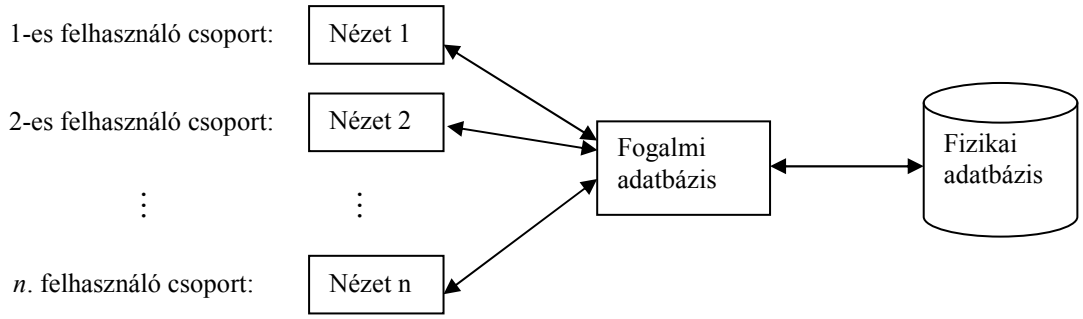
Az állománykezelő rendszerekből kialakuló első adatbázis-kezelő rendszerek különböző adatmodelleket használtak az adatbázisban tárolt információk szerkezetének ábrázolásához. A legfontosabbak ezen adatmodellek közül a hierarchikus és a hálós adatmodell. A hierarchikus

adatmodell, egy fa szerkezettel ábrázolta az adatokat, a hálós pedig egy gráffal. Ezekkel a korai modellekkel és rendszerekkel az volt a gond, hogy nem támogattak semmilyen magas szintű lekérdező nyelvet. A hálós adatmodell lekérdezőnyelvnek olyan utasításai voltak, amelyek csak azt engedték meg a felhasználónak, hogy adatelemről adatelemre mozogjon az elemek között meglévő mutatókból álló gráf mentén (lásd [Ul88]). 1970-ben T. Codd publikált egy cikket (lásd [Co70]), melyben bevezette a relációs adatmodellt, mely a mai napokban is a leghasználatosabb adatmodell. Ezen könyv részletesen a relációs adatmodellel foglalkozik.

1.1.4. Adatbázisok ANSI/SPARC architekturája

Az adatbázisszerű ismeretkezelésben megkülönböztetjük az adatbázis *általános elvi felépítését* (*absztrakt kép*) és az abban őrzött aktuális ismereteket (*konkrét tartalom*). Az adatbázis *általános struktúráját adatmodellnek* nevezzük.

A számítógép, amely bitekkel dolgozik és a felhasználó, aki egyedtípusokkal dolgozik (Személyek, Bankszámlák, Kocsik stb.) között több elvonatkoztatási szint lehetséges. Egy elfogadott nézőpont az ANSI/SPARC architektúra, mely 3 szintet különböztet meg: fogalmi, belső és külső szint.



1.2. ábra: ANSI/SPARC architektúra

Adatbázis tervezés esetén a *fogalmi szintet* (angolul: *conceptual level*) kell először megterveznünk. Ezen a szinten egy alkalmazási környezet valamennyi ismeretét és azok valamennyi összefüggését egyetlen közös adatmodellben kell leírnunk. Ez a reprezentáció független az adatbázis-kezelő rendszer típusától és az a mögött álló „filozófiától". Általában az egyed/kapcsolat adatmodellt használják a fogalmi szint megtervezésére. Ha például egy kereskedelmi vagy termelési cégnek akarjuk az adatfeldolgozását számítógépen megvalósítani, először meg kell tervezzük a cég összes adatának és az azok közötti kapcsolatokat tükröző általános, közös adatmodellt. (Általában sajnos nem így szokott történni. Több programozó dolgozik, mindegyik megtervezi az ő saját részét, ugyanazok az adatok többször is tárolva vannak.) Például a cégnél dolgozó alkalmazottak és a részlegek, amelyeknél az alkalmazottak dolgoznak különböző felhasználó által különbözőképpen tervezhető:

Első programozó az alkalmazottakat tartja nyilván. Ebben az esetben egy dolgozó kerül egy rekordba és itt egy mező a részlegnek van fenntartva.

Második programozó a részlegeket tárolja egy állományban és minden részlegnél az összes ott dolgozó alkalmazottat egymás után egy listában tárolja.

Egy olyan részleg, ahol nincs egy alkalmazott sem, az első megoldás esetén eltűnik. Az illető részleg az utolsó alkalmazott elbocsátásával megszűnik létezni. A második megoldásnál a részleg marad, csak az alkalmazottak száma 0. Egy esetleges lekérdezés esetén, mely a „A létező részlegek listája"-t kéri, a két különböző megoldás, két különböző eredményt ad. Közös megoldást kell találni, amely mindkét programozónak megfelel.

A belső szint a *fizikai adatbázis*ra hivatkozik**.** A fizikai adatázist az állományok összessége és a hozzájuk tartozó indexállományok alkotják. Az adatok a fogalmi szinten megadott szerkezet szerint állományokban vannak tárolva, az indexállományok pedig az adatokhoz való hatékony hozzáférést segítik. A fizikai adatbázis mindig a háttértárolón van. Egy ABKR több fizikai adatbázist is képes kezelni.

A harmadik a *külső szint* (angolul: *external level*). Minden fogalmi szintű adatbázisnak több felhasználója van. Az egyes felhasználók az adatbázis különböző részleteiben érdekeltek. A „külső” jelzőt az indokolja, hogy a felhasználók egyike sem tudja, hogy belül a számítógépen milyen az adatok szerkezete.

Az adatmodellnek a felhasználó által kezelésre kiválasztott részét *nézet*nek (angolul: *view*) nevezzük. A nézetnek levezethetőnek kell lennie a fogalmi szinten megtervezett adatmodellből. A nézetek fontosak az adatvédelem szempontjából, egy bizonyos felhasználócsoportnak csak bizonyos nézetekre van joga.

1.4. példa: Ahhoz, hogy megértsük a különbséget a fogalmi, fizikai és külső szint között, vegyük a tömböket a programozásból:

- a fogalmi szinten a tömböt leírhatjuk:
  integer array A[1..*n*,1..*m*]
- a fizikai szinten a sort úgy látjuk, mint egymás utáni memória lokációkat a következő szabály szerint: A[*i*, *j*] az a_0+4[*m*(*i*-1)+*j*–1];
- legyen egy nézete az A tömbnek az *f*(*i*) függvény, amely az A[*i*, *j*] elemek összege *j*=1-től *m*-ig. Így nem a sorokat látjuk, hanem csak a sorok összegét. □

1.5. példa: Egy repülőtársaság adatbázisa tartalmazza az alkalmazottakra vonatkozó ismereteket (melyik pilóta milyen képesítéssel rendelkezik, mennyi a fizetése stb.), repülőterekre vonatkozó ismereteket, repülőgépeket (minden repülő esetén a típusát, állapotát stb.), járatokat, jegyeket. A repülőjegyek eladása felhasználó csoport csak a járatokra, helyekre, jegyekre vonatkozó ismeretekhez kell hozzáférjnie. Az irányítótoronynak más ismeretekre van szüksége, ez már egy más nézet. Még egy példa nézetre: a repülőket karbantartó felhasználó csoport a repülők technikai jellemvonásaira, repülőkön végzett javításokra kíváncsi, ezekre az információkra a repülőjegyek eladása felhasználó csoportnak nincs joga. □

1.1.5. A szintek megfeleltetése, modellek, sémák

A fogalmi, fizikai és külső szintek között az ABKR teremt kapcsolatot. Amikor a felhasználó megadja az igényét (külső szint), azt az adatbázis-kezelő rendszer leképezi az általános képre (fogalmi szint), hogy megvizsgálja a kérdés teljesíthetőségét és behatárolja az adatbázis érintett részeit. Amikor pedig tényleges kezelésre kerül sor, akkor az adatmodell vonatkozó elemeit mintegy leképezi a tárolási szerkezetre, vagyis meghatározza, hogy hol találhatók a keresett adatok.

Amikor az adatbázist megtervezzük, érdekeltek vagyunk az adatbázis terveinek elkészítésében, amikor használjuk az adatbázist, a benne tárolt adatokkal vagyunk elfoglalva. Az adatbázisban tárolt adatok állandóan változnak, az adatbázis terve viszont változatlan marad, amíg az adatbázis létezik.

Az adatbázis tervében le kell írnunk a használt egyedek típusát, ezen egyedek közötti kapcsolatokat. A tervet az adatbázisok esetében sémának fogjuk nevezni, *fogalmi sémát* használunk a fogalmi adatbázis tervének, valamint a fizikai adatbázis tervét *fizikai sémának* nevezzük. A külső szinten több alsémát is értelmezünk a különböző felhasználó csoportoknak.

1.6. példa:

1) Az A tömb fizikai sémája az, hogy a tömb az a_0 memórialokációtól kezdődően van tárolva és A[*i*, *j*] az a_0+4[*m*(*i*-1)+*j*-1] című memória lokációban van.
2) A fogalmi séma az integer array A[1..*n*,1..*m*], vagyis, hogy A egy egész elemekből álló tömb, melynek *n* sora és *m* oszlopa van.
3) Egy alséma az *f* függvény értelmezése: $f(i) = \sum_{j=1}^{m} A[i, j]$ □

1.1.6. Adatfüggetlenség

Az absztrakció láncolatában a nézettől a fizikai szintig a fogalmin át, beszélhetünk az adatfüggetlenség két szintjéről. Egy jól tervezett adatbázis rendszerben a fizikai séma megváltoztatható anélkül, hogy meg kellene változtatni a fogalmi sémát vagy újraértelmezni az

alsémákat. Ezt nevezzük *fizikai adatfüggetlenség*nek. A fizikai adatbázis megváltoztatása növelheti a felhasználó programok hatékonyságát, de nem várhatja el, hogy a programokban valamilyen változtatásra is szükség legyen, amiért a fogalmi sémának az implementációja a fizikai sémában megváltozott. A példában akkor beszélünk fizikai adatfüggetlenségről, ha a tömb ábrázolását a memóriában megváltoztatjuk, mondjuk a sor szerinti ábrázolásból az oszlop szerinti ábrázolásba, de nincs szükség a program megváltoztatására.

A nézetek és a fogalmi szint között szintén van egyfajta adatfüggetlenség, ezt *logikai adatfüggetlenség*nek nevezzük. Sok esetben előfordul, hogy a fogalmi szintet változtatni kell, például újabb információkkal kell bővíteni. Akkor beszélünk logikai adatfüggetlenségről, ha a fogalmi szinten történő változtatás nem hat ki a kezelőprogramokra. Egyetlen változtatás a fogalmi szinten, bizonyos információk törlése, maga után vonhatja a programok változtatását, ha a törölt információ jelen van valamely nézetben.

1.1.7. ANSI/SPARC architektúra a funkcionalitás szempontjából

Ha az ANSI/SPARC architektúrát a funkciók szempontjából vizsgáljuk és nem az adatok szempontjából, mint az előzőekben, egy sokkal komplikáltabb nézetet kapunk, ahogyan az 1.3. ábrán is látható.

A funkció szerinti ANSI/SPARC architektúra ábrázolása esetén a négyszögek feldolgozási funkciókat jelölnek, míg a hatszögek adminisztrációs szerepet. A nyílak jelölik az adatok és parancsok áramlását, a rajtuk levő „I” alakú vonalak pedig az interfészeket.

A fő komponens, mely a különböző adatszervezési nézetek között a leképezést megvalósítja az adat szótár/katalógus (háromszögként van ábrázolva), amely egy metaadatbázis. Ez kell tartalmazza legalább a séma és a leképezés definícióját. De tartalmazhat még használati statisztikákat, hozzáférést vezérlő információkat.

Az 1.3. ábrán látható több adminisztrátori szerep is, melyek segíthetnek az ANSI/SPARC architektúra funkcionális interpretációjának a meghatározásában. A három szerep az adatbázis, a cég és az applikáció adminisztrátor.

A *cég adminisztrátor* szerepe, hogy előkészítse a fogalmi séma definícióját.

Az *applikáció adminisztrátor* felelős a külső sémáknak az előkészítéséért az alkalmazások, az applikáció programozó számára.

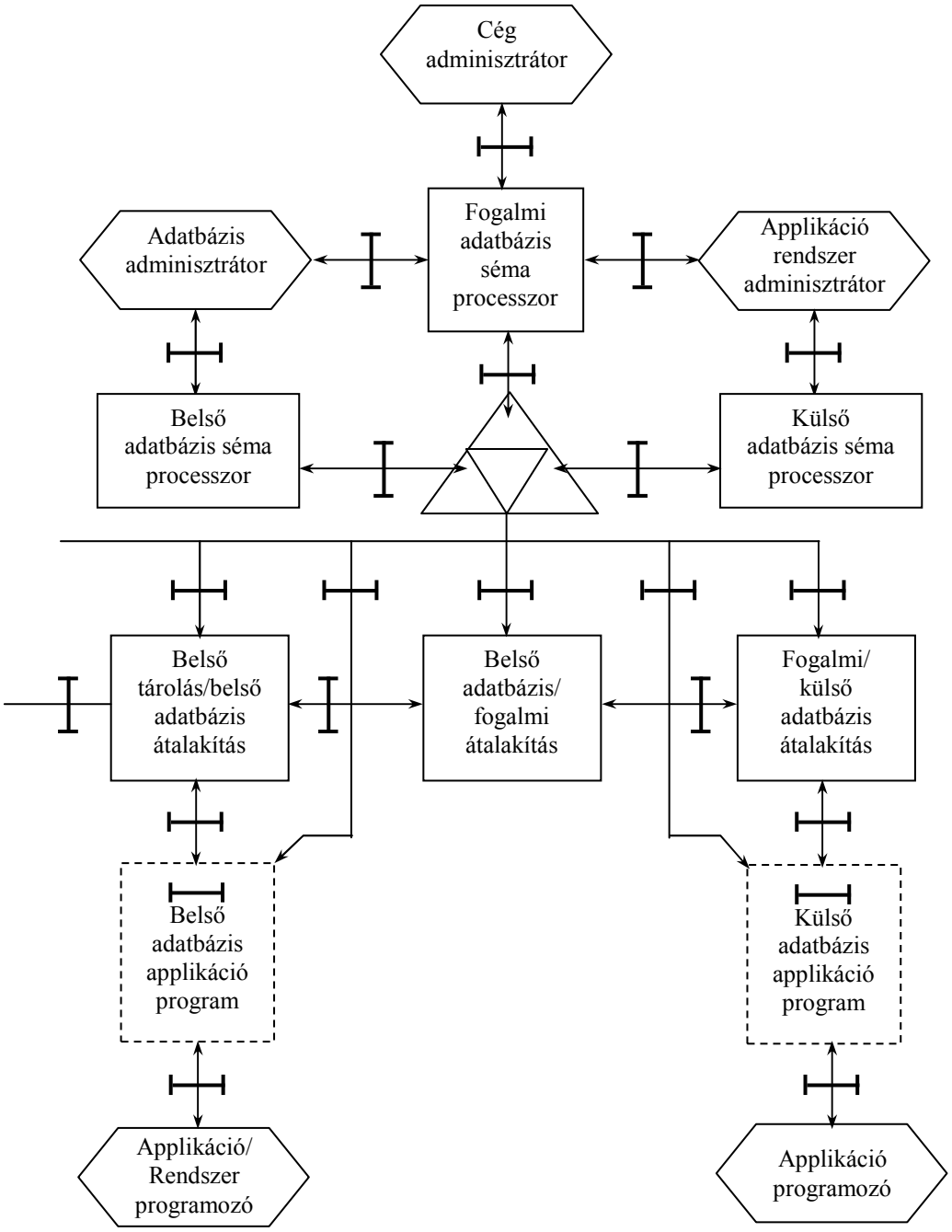
Az *adatbázis adminisztrátor* feladatai:

- Fizikai séma tervezése: táblák szerkezete, táblák közötti kapcsolatok, táblák tárolásának részletei, indexállományok tervezése, azok típusának a meghatározása.
- Biztonság és hozzáférési jogok meghatározása: az adatbázis adminisztrátor felelős azért, hogy ne tudjanak nem megengedett személyek hozzáférni az adatbázis adataihoz. Például a diákok láthatják a különböző előadásokat, mit tartalmaznak, ki tanítja, de nem láthatják a tanárok fizetését. Ezt úgy lehet megvalósítani, hogy a diákok csak a KurzusInfo nevű nézetet olvashatják.
- Adatok elérhetősége, helyreállítás hiba estén: az adatbázis adminisztrátor feladata, hogy rendszerhiba esetén a felhasználók tudjanak hozzáférni legalább azon adatokhoz, melyek nem károsultak, majd állítsa vissza az adatbázisnak egy helyes állapotát. Az ABKR ad erre függvényeket, az adatbázis adminisztrátor felelős, hogy bizonyos időintervallumokban másolatokat készítsen az adatokról.

Az adatbázis alkalmazás programozók követelményei változnak, a cég adminisztrátor feladata, hogy megváltoztassa a fogalmi sémát, az adatbázis adminisztrátor feladata változtatni a fizikai sémát.

E három adminisztratív felhasználó osztály mellett még van kettő, az applikáció (alkalmazás) programozó és a rendszer programozó.

Definiálhatunk másik két felhasználó osztályt is, pontosabban az alkalmi és a vég felhasználók. Az alkalmi felhasználók időnként hozzáférnek az adatbázishoz, hogy visszakeressenek információkat és valószínűleg, adatkezelési műveleket is végeznek. Ezeknek a felhasználóknak segítségül szolgálnak a külső séma definíciók, illetve a könnyen használható lekérdező nyelvek. A vég felhasználóknak általában nincs adatbázis ismeretük és az információkhoz az adatbázis applikációkon keresztül férnek hozzá.



1.3. ábra: ANSI/SPARC architektúra funkcionalitás szempontjából

1.2. Az ABKR-ek előnyei

- Adatfüggetlenség – a felhasználói programok a lehető legfüggetlenebbek az adatok ábrázolásától és tárolásától. Az ABKR egy absztrakt nézetet (képet) ad az adatokról.
- Hatékony hozzáférés az adatokhoz – az ABKR-ek komplex technikákat használnak ennek érdekében.
- Adatok helyességére vonatkozó megszorítások és biztonság – ha az adatokhoz mindig az ABKR-en keresztül férünk hozzá, az ABKR rákényszeríti a helyességre vonatkozó megszorításokat. Például mielőtt egy új alkalmazott fizetését vezeti be, ellenőrzi, hogy befér-e

még a költségvetésbe. Az ABKR minden felhasználó minden hozzáférése esetén ellenőrzi a felhasználó jogait a kért adatokra.
- Adminisztráció – mikor több felhasználó is megosztja az adatokat, szükség van egy központosított adminisztrációra, egy-két emberre, aki profi és átlátja az egész rendszert. Ők felelősek az adat ábrázolásáért, hogy az adatismétlés minimális legyen, megfelelő tárolást és indextechnikát válasszanak ki.
- Konkurens hozzáférés és visszaállítás hiba esetén – a felhasználó ne érezze azt, hogy más is használja ugyanazt az adatot. Hiba esetén az ABKR vissza tud állítani egy helyes adatbázist.
- Megrövidül az alkalmazás elkészítésének ideje – az ABKR-ben sok olyan függvény van, melyet minden alkalmazás használ (report writer, lekérdezés optimalizáló, interface generáló).

Relációs ABKR-ekkel felmerülő problémák:

- A napjainkban legelterjedtebb és ebben a könyvben is részletesen bemutatott relációs ABKR-ek nem tud komplex adatokkal dolgozni, amelyek szükségesek például mérnöki alkalmazások (CAD/CAM rendszerek), földrajzi alkalmazások stb. Az objektumorientált, illetve az objektumrelációs adatmodell a megoldás ezen alkalmazások esetében ([UlWi97], [St96]).
- Az ABKR-ek általában nagyon drágák, kicsi cégek nem tudják megvásárolni.
- Az ABKR-ek programozása nehézkes és a beállításai nagyon komplexek.

1.3. Adatbázis-kezelő rendszerek képességei

Két tulajdonsága van az ABKR-nek, ami megkülönbözteti más programrendszerektől:
1. Azon képessége, hogy állandóan létező adatokat tud kezelni.
2. Azon képessége, hogy hatékonyan tud kezelni nagyon nagy mennyiségű adatot. Ebben különbözik a állománykezelő rendszerektől, amelyek képesek állandóan létező adatokat kezelni, de nem segít az adatok gyors hozzáféréséhez az adatbázis bármely részéből. Erre a tulajdonságra akkor van szükség, ha nagy mennyiségű adatot kell kezelnünk, mert kevés adat esetében egyszerű hozzáférési technikák is (pl.: lineáris keresés) megfelelnek. Az ABKR különböző indexállományok létrehozásával biztosítja a gyors hozzáférést.

Ezen két alapvető tulajdonságon kívül a legtöbb ABKR rendelkezik a következő tulajdonságokkal is :

a) Egy adatmodellre épül, melynek segítségével a felhasználó megtervezheti az adatbázist, ugyanakkor segít a felhasználónak abban, hogy az adatot ne csak bitek sorozataként lássa, hanem érthetőbb formában.
b) Magas szintű programozási nyelvekkel rendelkezik az adatok szerkezetének a leírására, adatkezelésre és lekérdezésre.
c) Egyidejűleg több felhasználó között megosztja az adatbázist, ellenőrzi a hozzáférési jogokat.
d) Rendszerhibák esetén képes egy helyes adatbázist visszaállítani.

1.4. Adatbázisok nyelvei

Általában a programozási nyelvekben az adatok leírása és az utasítások a programon belül ugyanabban a nyelvben történik. Az adatbázisok világában szétválasztják az adatbázis leírását annak különböző funkciójának a programozásától. A különbség a következő: míg egy mindennapi programban az adat is csak addig létezik, amíg a program fut, az adatbázis rendszerekben az adat állandóan létezik, egyszer lehet deklarálni és azontúl mindig létezik. A munka is megvan osztva speciális adatbázis nyelvek és egy „*gazda*” nyelv között.

1.4.1. Adatleíró nyelvek

A fogalmi sémát egy olyan nyelvben tudjuk leírni, amely az ABKR része és ezt *adatleíró nyelv*nek (Data Definition Language) nevezzük. Ez a nyelv nem procedurális, inkább egy jelölés az egyedek és a köztük lévő kapcsolatok leírására az illető adatmodellen belül.

1.7. példa: Relációs adatmodell esetén az SQL nyelv adatleíró nyelve:

```
1) CREATE  TABLE  Csoportok (
2)   CsopKod CHAR(3) PRIMARY KEY,
3)   Evfolyam INT,
4)   SzakKod CHAR(3)
4) );
5) CREATE TABLE Diákok (
6)    BeiktatásiSzám INT PRIMARY KEY,
7)    Név VARCHAR(50),
8)    Cím VARCHAR(100),
9)    SzületésiDatum DATE,
10)   CsopKod CHAR(3) REFERENCES Csoportok (CsopKod),
11)   Átlag REAL
12) );
14) CREATE INDEX CsopIndexDiák ON Diákok (CsopKod);
```

Az első 5 sor leírja a Csoportok relációt az attribútumokkal együtt és az attribútumok típusát. A 6−13 sorban a Diákok reláció szerkezetét láthatjuk. A 11. sorban a két reláció közötti kapcsolatot tudjuk ábrázolni. A 14. sor kéri egy index létrehozását a Diákok relációnak a CsopKod mező szerint, amely a hatékony keresést segíti elő. Ez az index a fizikai séma része lesz. Az indexet a DDL compiler úgy oldja meg, ahogy jónak látja. □

Az adatleíró nyelvet használjuk, amikor megtervezzük az adatbázist vagy ha változtatjuk. Nem használjuk az adatokhoz való hozzáféréshez vagy az adatok változtatásához. A táblák szerkezete, az indexállományok az adatbázisséma részei.

Az alsémák leírásához szükség van alséma adatleíró nyelvre, amely sok esetben hasonló az adatleíró nyelvhez. Az alséma nyelve más adatmodellt is használhat, mint az adatleíró nyelv.

1.8. példa: Nézetek leírása SQL nyelvben:

```
CREATE VIEW MagyarDiákok AS
     (SELECT Nev, CsopKod
       FROM Diákok
       WHERE CsopKod IN (
        SELECT CsopKod
           FROM Csoportok
           WHERE SzakKod = 'IM' OR SzakKod = 'MM' OR SzakKod = 'MIM'
                     )
      ) □
```

A nézetek értelmezése is bekerül az adatbázis sémába.

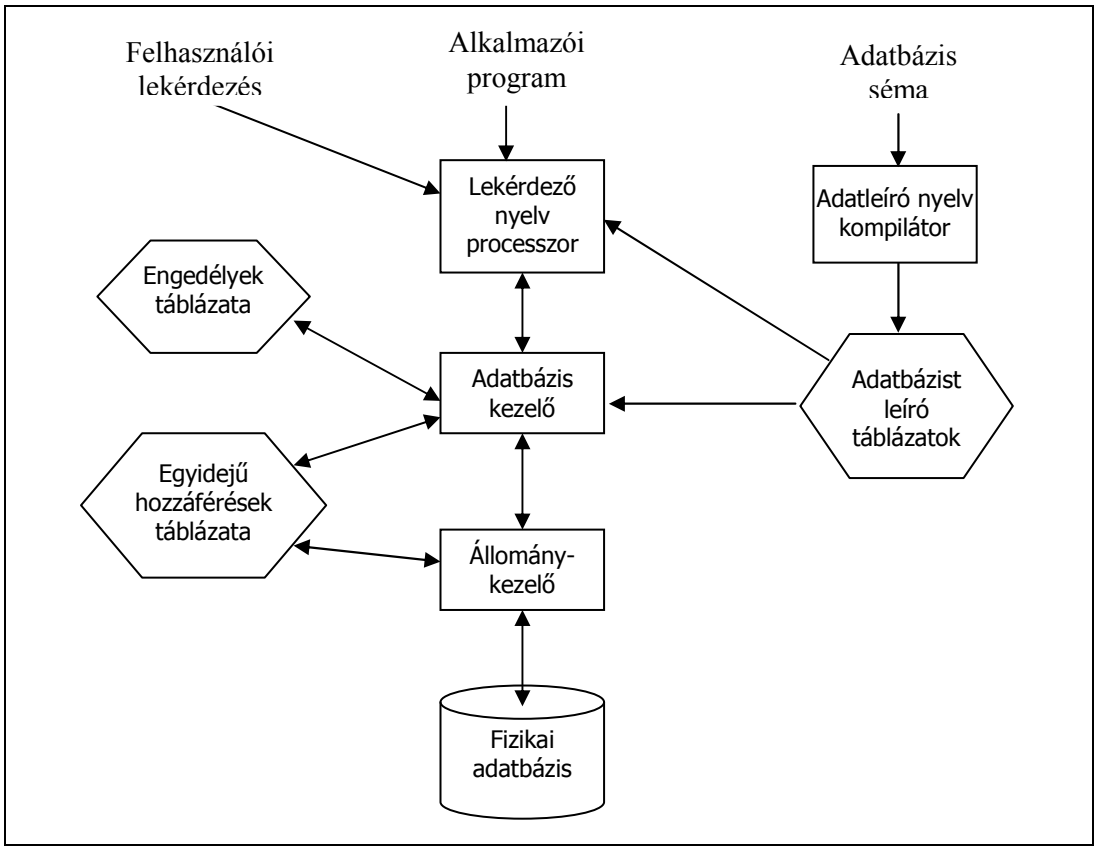
1.4.2. Adatkezelő nyelvek

Ahhoz, hogy műveleteket végezhessünk az adatbázisban tárolt adatokkal, szükségünk van egy ún. *adatkezelő nyelv*re (Data Manipulation Language). Ilyen műveletek például:

1. Olvasd ki az adatbázisból az '531'-es csoport diákjait.
2. Módosítsd 'Szabó János' címét 'Kolozsvár, Bolyai utca 2'-re.
3. Számítsd ki a harmadéves informatikusok átlagát.
4. Vezess fel új diákot az 511-es csoportba, neve: 'Kovács István', címe: 'Marosvásárhely, Petőfi utca 3', beiktatási száma: 56431, születési dátuma: 1982-dec-10.

Az (1) és (3) csupán lekérdezi a létező adatot, a (2) és (4) változtatja az adatot az adatbázisban.

A legtöbb esetben viszont az adatbázisokkal sokkal komplexebb feladatokat is meg kell oldani, ezért szükség van egy programozási nyelvre (lehet C, Delphi, MS Visual Basic, Java, COBOL, …). Ez a „gazda" nyelv. A gazda nyelvből meghívhatók az adatkezelő nyelv műveletei, lásd 9.2.1. Adatbázis szintjén írhatunk tárolt eljárásokat, melyek feladata több adatkezelő művelet végrehajtása, lásd 9.2.



1.4. ábra: Az ABKR felépítése

1.5. Adatbázis-kezelő rendszerek főbb részei

Az adatleíró nyelv kompilátora beolvassa az adatbázis sémáját, elemzi, ha helyes, elkészíti az adatbázist leíró táblázatokat. Az adatbázis sémája sokkal ritkábban változik, mint az adatok az adatbázisban. Az adatbázis sémájának a változtatása egy több felhasználós adatbázis esetén az adminisztrátor feladata, aki ismeri az egész adatbázis szerkezetét, a sémáját, álsémákat és az engedélyeket az adatbázis különböző részeihez való hozzáféréshez.

A lekérdező nyelv processzor elemzi a felhasználói lekérdezést és az alkalmazói programot. Az ő feladata ezen kérések optimalizálása is. Felhasználja az adatbázist leíró táblázatokat, az adatokhoz való hozzáférés érdekében indextáblákat is használhat és átadja az adatbázis-kezelőnek a lekérdezést.

Az adatbázis-kezelő fogalmi szinten kezeli a parancsokat és átalakítja fizikai szintű parancsokká. Az adatbázis-kezelő használja az engedélyek táblázatát, hogy ellenőrizze, a különböző felhasználóknak milyen jogaik vannak az adatbázis különböző részeihez. Ha egyidejűleg több felhasználó is hozzáférhet az adatbázishoz, ennek ellenőrzését az adatbázis-kezelő többféleképpen is megoldhatja. Ha például lakatot (angolul: „lock”) tesz arra a részre, amit egy felhasználó éppen használ, és amíg be nem fejezi ez a felhasználó a munkát, nem enged más felhasználót hozzáférni ehhez a részhez, ezen lakatokat helyezi az „Egyidejű hozzáférések táblázatá”-ba.

Az adatbázis-kezelő átalakítja a kapott parancsokat állományokon való műveletekké, és ezeket átadja az állománykezelőnek. Az állománykezelő lehet az operációs rendszer standard állománykezelője, vagy az ABKR saját állománykezelője.

1.6. Hogyan valósítják meg az indexeket

A leggyakrabban használt adatstruktúra az indexállományok készítésénél a *B*-fa, ahol a B a kiegyensúlyozottságra utal (Balanced − kiegyensúlyozott). A B-fa a kiegyensúlyozott bináris

keresőfa általánosítása. Míg a bináris fa csomópontjának csak legfeljebb 2 gyermeke lehet, a B-fa csomópontjainak sok gyermeke is lehet. A B-fákat a lemezen tárolja a rendszer és nem a memóriában, ezeket úgy tervezik, hogy egy csomópont egy teljes lemezblokkot elfoglaljon. A legtöbb rendszer 4096 bájt méretű blokkot használ, ezért a B-fa egy csomópontjában több száz gyermekre mutató pointer is lehet. Így a B-fában való keresés ritkán mélyebb három szintnél.

A lemezműveletek költsége általában arányos az elért lemezblokkok számával. Ezért a B-fában való keresés, amelyik legtöbbször csak három lemezblokk elérését igényli, sokkal hatékonyabb, mint amilyen a bináris fában való keresés. A bináris keresőfák esetén sok különböző lemezblokkon elhelyezkedő csomopontot kellene bejárjunk. A B-fák és bináris fák közötti különbség csak egy példa a sok közül arra, hogy a lemezen tárolt adatok számára legmegfelelőbb adatszerkezetek mennyire különböznek a memóriában futó algoritmusok által használt adatszerkezetektől, láss részleteket az [Ul89] és [GaUlWi00]-ben.

2. Adatmodellek

2.1. Adatmodellekről általában

Minden adatbázis-kezelő rendszer egy absztrakt adatmodellel dolgozik, azért, hogy az adatokat ne csak bitek sorozataként lássuk. Egy adatmodell egy matematikai formalizmus mely a következő két részből áll:

1. egy jelölés az adat leírása érdekében
2. műveletek halmaza, mely az illető adatok kezelésére használatosak

Az adatmodelleket többféleképpen is csoportosíthatjuk. Egyik csoportosítás:

- értékorientált modellek: relációs adatmodell, logikai adatmodell;
- objektum alapú modellek: hierarchikus, hálós, egyed/kapcsolat és az objektumorientált adatmodell.

Feltevődik a kérdés: melyik a legjobb adatmodell? Mielőtt az adatmodellek közötti különbségeket tárgyalnánk szükségünk van az objektumazonosító fogalmára.

Egy rendszer, mely támogatja az objektumazonosító létezését, képes különbséget tenni két objektum egyenlősége (vagyis előfordulhat, hogy két különböző objektumnak egy adott pillanatban ugyanazon értékei legyenek) és azonossága (két objektum azonos, ha mindig ugyanazok az értékei) között.

Egy objektum az a következő páros: (<OID>,<érték>), ahol <OID> egy objektumazonosító és <érték> lehet egy egyszerű vagy összetett érték. OID az egész rendszerben egyedi kell legyen és nem változhat meg, amíg az objektum létezik, az objektum törlése után, biztosítani kell, hogy egy más objektum se kaphassa ezt az OID-t.

Különbségek az adatmodellek között:

1. Ami a modell célját illeti: a legtöbb adatmodell az adatok egy jelölése, melyen az adatok kezelésére használatos műveletek is alapulnak. Az egyed/kapcsolat adatmodell a fogalmi adatbázis megtervezésére használatos, még soha nem implementálták, műveletek nem léteznek az egyed/kapcsolat adatmodellnél, akkor egyesek megkérdezhetik, hogy egyáltalán adatmodell-e?
2. Érték- vagy objektumorientáltak? Az első ABKR-ek a 60-as években jelentek meg és a hálós vagy a hierarchikus adatmodellre alapultak. Ezen ABKR-ek hatékonyan tudtak nagy mennyiségű adatot kezelni, de nem rendelkeztek deklaratív lekérdező nyelvvel. Objektumalapúak voltak abban az értelemben, hogy támogatták az objektumazonosító létezését, de nem támogatták az absztrakt adattípusokat. Mivel nem deklaratív adatmodellek, nem rendelkeztek deklaratív lekérdező nyelvvel, nem tevődik fel a lekérdezés optimalizálása. Az értékorientált (relációs és logikai) adatmodellek később jelentek meg, támogatják a deklaratívitást és feltehető a lekérdezés optimalizálása. Az egyed/kapcsolat adatmodell megköveteli az objektumazonosító létezését. Például a relációs adatmodell esetén a következő relációban Alkamazottak [Név, Részleg] nem tárolhatunk két ('Kovács','informatika') alkalmazottat. Relációs adatmodell esetén nem lehet egy táblában két teljesen azonos sor, a relációs rendszer értékorientált és nem támogatja az objektumazonosító létezését. Egy objektumorientált rendszer meg tudja különböztetni a két 'Kovács'-ot. A relációsban egy plusz mezőt kell a felhasználónak bevezetnie, hogy tárolni tudja a két 'Kovacs'-ot. Az objektum alapú rendszerek megteszik ezt a felhasználó helyett, de az objektumazonosítót a felhasználó nem látja.
3. Hogyan oldják meg az adatismétlés kiküszöbölését? Minden adatmodell segít valahogy a felhasználónak, hogy ugyanazt az adatot ne tárolja csak egyszer, nem csak a háttértároló jobb kihasználása miatt, hanem az inkonzisztencia megelőzése érdekében is. Az objektumorientált modellek jobban megoldják ezt a problémát. Az objektumot egyszer tárolják, és mikor más helyeken szükség van rá egy pointert használnak az illető objektum fele. A relációs adatmodell esetén egy egész elmélet (normalizálás) létezik a tervezésre, hogy ne jelenjen meg az adatismétlés.

2.2. Az egyed/kapcsolat adatmodell

2.2.1. Egyed és egyedhalmaz értelmezése

Az egyed/kapcsolat adatmodell az adatbázis fogalmi szinten való megtervezésére használatos, anélkül, hogy a fizikai adatbázis tervezését részleteznénk, vagy a hatékonyság érdekelne. Nem lehet egy formális értelmezést adni az egyedeknek úgy, mint a geometriában a pont vagy egyenes fogalmának.

Egyed: egy olyan dolog, mely létezik és megkülönböztethető. (Vagyis egyik egyed megkülönböztethető a másiktól).

2.1. példa: Minden személy egy egyed, minden jármű egy egyed. □

2.2. példa: Minden hangyát tekinthetünk egy egyednek, ha van rá mód, hogy megkülönböztessük őket. (Például pici számokat írunk a hátukra.) □

Az egyedek megkülönböztetése nagyon közel áll az objektum identitás fogalmához, ezért az egyed/kapcsolat modellt objektumalapú adatmodellnek tekintik.

Egyedek halmaza: az összes „hasonló” egyed egyedhalmazt alkot.

2.3. példa: Az összes személy egy egyedhalmazt alkot, az összes vöröshajú személy is, illetve az összes jármű is egy egyedhalmaz. □

A „hasonló” nem valami precízen van fogalmazva. Egy egyedhalmaz minden egyedét ugyanazokkal a jellemvonásokkal kell jellemeznünk, ezen jellemvonásokat attribútumoknak fogjuk nevezni. Tehát a hasonlóság azt jelenti, hogy egy egyedhalmaz minden egyedét ugyanazokkal az attribútumokkal kell jellemeznünk.

2.2.2. Attribútumok és kulcsok

Egy egyedhalmaznak több attribútuma is van. Minden egyednek az egyedhalmazból megfelel egy-egy érték minden attribútum értékeinek halmazából. Általában egy attribútum értékeinek halmaza egész számok, valós számok, karaktersor stb. lehetnek.

Egy egyedhalmaz lényeges attribútumainak kiválasztása egy fontos lépés a fogalmi adatséma megtervezésében.

Egy vagy több attribútum, mely egyértelműen meghatároz egy egyedet az egyedhalmazban *kulcs*nak nevezzük, az illető halmaz egyedkulcsának. Minden egyedhalmaznak van kulcsa, mivel feltételeztük, hogy minden egyed megkülönböztethető. Ha egy egyedhalmazban nem választunk kulcsot nem fogjuk tudni megkülönböztetni az egyedeket egymástól. Gyakran egy plusz attribútumot hozunk be, hogy kulcsa legyen az egyedhalmaznak.

2.4. példa: Minden állampolgárnak a személyi igazolványában van a személyi száma, az lehet kulcs. □

2.5. példa: Minden autónak a rendszáma szolgálhat kulcsként. □

2.2.3. Specializáló „az_egy” (Is_a) hierarchiák

Azt mondjuk, hogy A „az_egy” B (A "Is_a" B), ha a B egyedhalmaz az A egyedhalmaz egy általánosítása, vagy másképp A egy speciális B. Egy elsődleges cél amikor „az_egy” relációkat deklarálunk A és B között, hogy A örökölhesse B attribútumait, és ezek mellett még lehetnek más attribútumai is. B-nek a kulcsa lesz A-nak a kulcsa is.

2.6. példa: Adott egy *Alkalmazottak* egyedhalmaz a következő attribútumokkal: SzemSzám, Név, Fizetés. A cégnek lehet egy futbal csapata, melyben néhány alkalmazott benne van, ezeknek még plusz attribútumaik lesznek, mint LőttGólok, KiállításokSzáma stb., mely a többi alkalmazottnak nem lesz. Tehát egy új *Játékosok* egyedhalmazt tervezünk, melyre a következő reláció áll: *Játékosok* „az_egy” *Alkalmazottak*. A *Játékosok* örökli az *Alkalmazottak* összes attribútumát és

még plussz attribútumai: LőttGólok, KiállításokSzáma. A *Játékosok* kulcsa szintén a SzemSzám lesz. □

2.2.4. Kapcsolatok

Egyedhalmazok közötti kapcsolat az egyedhalmazok egy rendezett listája. Egy adott halmaz nem csak egyszer jelenhet meg. Ha R egy reláció az E_1, E_2,...,E_k egyedhalmazok között, akkor az R egy példánya $(e_1,e_2,...,e_k)$, ahol $e_1 \in E_1$, $e_2 \in E_2$,..., $e_k \in E_k$. k a reláció foka. Leggyakrabban használatos a $k = 2$.

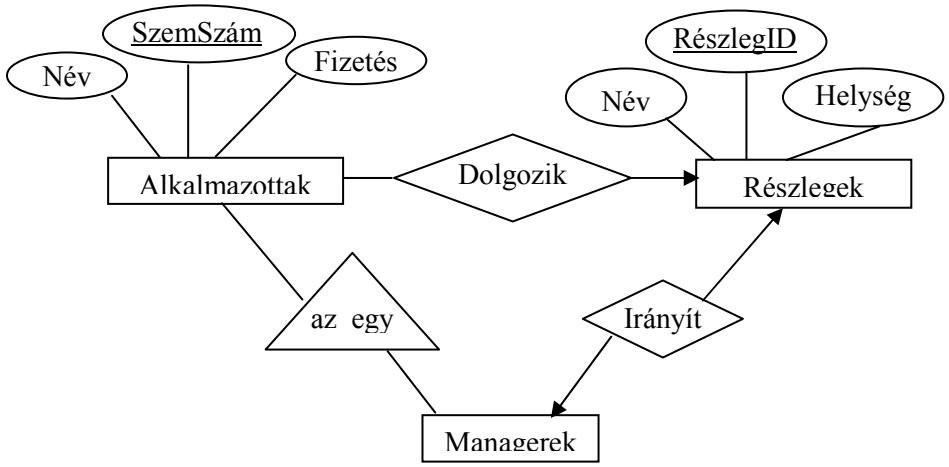
2.7. példa: Legyen egy egyedhalmaz: *Személyek* és egy reláció: ANYJA, mely a *Személyek* és *Személyek* egyedhalmazok között áll fenn. (p_1, p_2) egy megvalósítása az ANYJA relációnak, ha p_2 a p_1 anyja. □
Egy más megközelítésben legyen *Anyák* egy egyedhalmaz úgy, hogy *Anyák* „az_egy” *Személyek*. Az ANYJA reláció pedig *Személyek* és *Anyák* egyedhalmazok rendezett listája.

2.2.5. Egyed/kapcsolat (E/K) diagramok

Az adatmodellezést grafikusan az *egyed/kapcsolat diagram* segítségével tudjuk megoldani. Az E/K diagram három alapeleme a következő:

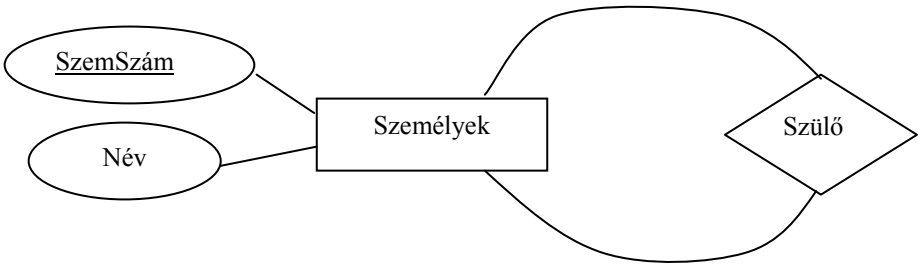
1. *Egyedhalmazok*, melyeket téglalapok segítségével ábrázolunk.
2. *Attribútumok*, az egyedhalmazok attribútumai, melyek jelölésére ellipsziseket használunk. Az ellipsziseket nem irányított élekkel, szakaszokkal az egyedhalmazt jelölő téglalaphoz kötjük. Azon attribútumokat, melyek a kulcs részei, aláhúzzuk.
3. *Kapcsolatok***,** melyek két vagy több egyedhalmazt kapcsolnak össze. Rombuszokat használunk a kapcsolatok ábrázolására. A rombuszokat összekötjük a relációt alkotó egyedhalmazokat jelölő téglalapokkal. Az összekötő szakaszok lehetnek irányítottak vagy nem, attól függően, hogy 1:1, 1: n vagy m : n típusú kapcsolatok (lásd 2.2.6.). A speciális „az_egy” hierarchiák jelölésére háromszöget használunk.

2.8. példa: Legyenek egy cég alkalmazottai az *Alkalmazottak* egyedhalmaz elemei. Az *Alkalmazottak* egyedhalmaz attribútumai: Név, SzemSzám, Fizetés. A különböző alkalmazottak a cég különböző részlegeiben dolgoznak. Legyen *Részlegek* a cég részlegeit tartalmazó egyedhalmaz, attribútumai pedig: Név, RészlegID, Helység. A Dolgozik egy kapcsolat az *Alkalmazottak* és *Részlegek* egyedhalmazok között. Egy alkalmazott egy részlegben dolgozik, de egy részlegben több alkalmazott is dolgozhat. Bizonyos alkalmazottak managerek, ezek alkotják a *Managerek* egyedhalmazt. Az E/K diagramon a *Managerek* kapcsolatát az *Alkalmazottak*hoz egy „az_egy” kapcsolattal ábrázoltuk. Egy manager irányít egy részleget, ezt a kapcsolatot a *Részlegek* és *Managarek* egyedhalmazok között az Irányít kapcsolattal ábrázoltuk.



2.1. ábra: E/K diagram az alkalmazottak és részlegek modellálására

2.9. példa: A *Személyek* között fennálló *Szülő* kapcsolatot E/K diagram segítségével a következőképpen ábrázolhatjuk:



2.2. ábra: Személyek közötti szülői kapcsolat modellálása E/K diagram segítségével

Két élet rajzoltunk a *Szülő*től a *Személyek* felé, az egyik a szülő, a másik a gyerek. A Szülő kapcsolatnak egy aktuális értéke (p_1, p_2), ha p_2 a p_1 szüleje.

2.2.6. E/K kapcsolatok típusai

A valós világ modellezése esetén gyakran szükség van meghatározni, hogy hány egyed az egyik egyedhalmazból, hány egyeddel a másik egyedhalmazból van kapcsolatban. Léteznek 1:1, 1:*n*, *n* : *m* típusú kapcsolatok:

- 1:1 típusú kapcsolat van E_1 és E_2 egyedhalmazok között, ha egy egyed az E_1 egyedhalmazból legfeljebb egy egyeddel hozható kapcsolatba az E_2 egyedhalmazból, és egy egyed az E_2 egyedhalmazból legfeljebb egy egyeddel hozható kapcsolatba az E_1 egyedhalmazból.

2.10. példa: Ahhoz, hogy a *Managerek* és *Részlegek* között azt mondhassuk, hogy a kapcsolat 1:1 típusú nem fordulhat elő soha, hogy egy részlegnek 2 managere is legyen, vagy, hogy egy manager két részleget is irányítson. Az előfordulhat, hogy egy Részleg egy ideig manager nélkül legyen vagy egy manager részleg nélkül. Ez a kapcsolat a legritkább. Az adatbázis tervezőnek el kell döntenie, hogy 1:1-nek tervezi-e a kapcsolatot. □

- 1: *n* típusú kapcsolat van E_1 és E_2 egyedhalmazok között, ha egy egyed az E_1 egyedhalmazból kapcsolatba hozható 0 vagy több egyeddel az E_2 egyedhalmazból és egy egyed az E_2 egyedhalmazból legfeljebb egy egyeddel hozható kapcsolatba az E_1 egyedhalmazból.

2.11. példa: *Részlegek* és *Alkalmazottak* között 1: *n* típusú a kapcsolat, mivel egy részlegben több alkalmazott is dolgozik és minden alkalmazott legfeljebb egy részlegben dolgozik. Lehet olyan alkalmazott, példa az igazgató, aki egy részlegben sem dolgozik. □

- *n*: *m*: típusú kapcsolat van E_1 és E_2 egyedhalmazok között, ha egy egyed az E_1 egyedhalmazból kapcsolatba hozható 0 vagy több egyeddel az E_2 egyedhalmazból és egy egyed az E_2 egyedhalmazból kapcsolatba hozható 0 vagy több egyeddel az E_1 egyedhalmazból.

2.12. példa: Szülő kapcsolat: egy szülőnek lehet több gyereke, egy gyereknek 2 szüleje.□

2.13. példa: *Kurzusok* és *Diákok* között a kapcsolat *n*: *m* típusú. Egy diák több kurzust választhat és egy kurzus több diák által van kiválasztva. □

Gyakran jelenik meg az *n*: *m* típusú kapcsolat. Az egyed/kapcsolat modellben megtervezhetjük, de a legtöbb modell nem engedi meg az *n*: *m* típusú reláció direkt deklarálását. Ez szétválasztható több 1: *n* típusú relációvá, mivel nincs megfelelő adatstruktúra, mely az *n*: *m* típusú relációt implementálja.

A diagramon irányított éleket használunk a különböző típusú relációk ábrázolására.

- Ha a kapcsolat 1:1 típusú, mind a két egyedhalmaz felé nyíl mutat, lásd *Részlegek* és *Managerek* között.
- Ha *A* és *B* egyedhalmazok között 1: *n* típusú a reláció, a nyíl az *A*-t ábrázoló téglalap felé fog mutatni. Ha feltételezzük, hogy egy alkalmazott csak egy részlegben dolgozhat, akkor a Dolgozik kapcsolat az *Alkalmazottak*tól mutat a *Részlegek* felé.
- Ha a kapcsolat *n*: *m* típusú, akkor az egyedhalmazokat összekötő él nem irányított.

A speciális „az_egy" kapcsolat esetén a háromszög alapját a speciális egyedhalmazzal, a háromszög csúcsát pedig az általános egyedhalmazzal kötjük össze, lásd *Managerek* és *Alkalmazottak* közötti „az_egy" kapcsolatot a 2.1. ábrán.

2.14. példa: Egy nagykereskedő cég egyszerűsített adatbázisának a megtervezése érdekében tekintsük a 2.8. példát, amit kibővítünk *Áruk* egyedhalmazzal. A cég különböző részlegei különböző típusú árukat árulnak, például az Építőanyag nevű részleg forgalmaz faárut, fűtéshez szükséges árukat, fürdőszoba-felszerelést stb. A Kozmetikumok nevű részleg forgalmaz mosószereket, szappanokat stb. Láthatjuk, hogy az árukat csoportosíthatjuk, ezért bevezetjük az *ÁruCsoportok* egyedhalmazt. Egy áru egy árucsoporthoz tartozik, de egy árucsoport több árut is tartalmaz, például Fürdőszoba-felszerelések árucsoportban szerepel több típusú fürdőkád is, mosdókagyló stb. Különböző részlegek különböző árucsoportokkal foglalkoznak, láttuk az Építőanyag nevű részleget, és lehet a cégnek egy autóalkatrészeket, vagy papírárut, vagy bútorokat forgalmazó részlege.

Az árukat különböző szállítók, különböző árban ajánlhatják, mindig az aktuális ajánlat érdekel. A *Szállítók* és *Áruk* egyedhalmazok között *n*: *m* típusú kapcsolat van, egy szállító több árut is ajánl, de ugyanazt az árut több cég is ajánlhatja. Amint az E/K diagramon láthatjuk, egy attribútumot rendeltünk a kapcsolathoz, az árat amiben a szállító az árut ajánlja. Modellezhettünk volna egy plussz egyedhalmazt, *Ajánlatok*, aminek egy attribútuma van: Ár.

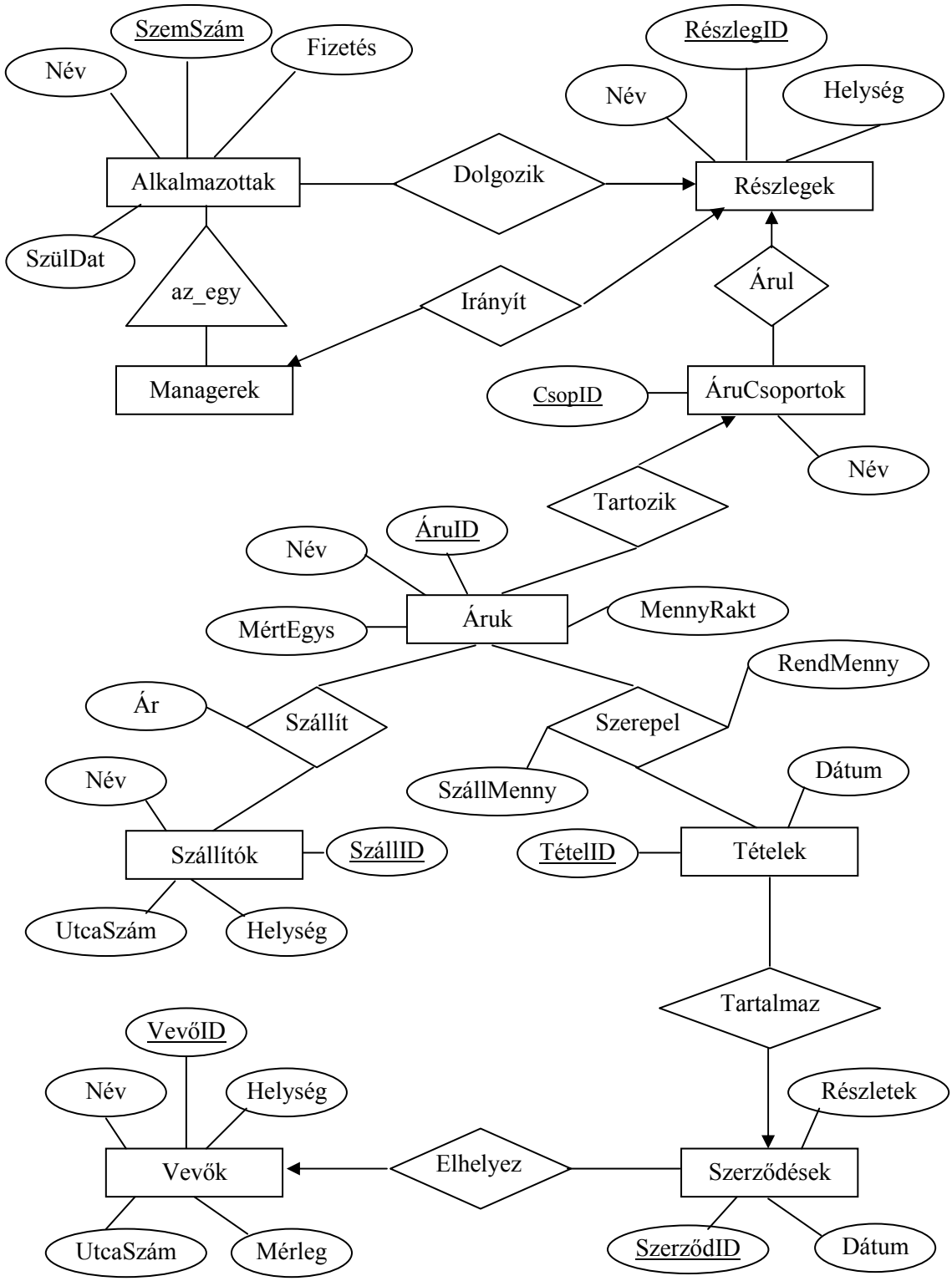
A vevőkkel a cégünk szerződéseket köt a szállítandó árukra és a szállítási feltételekre vonatkozóan. A szerződések általánosságokat tartalmaznak. Egy szerződéshez több tétel is tartozik, amiben rögzítik, hogy melyik áruból mennyit rendelnek meg egy adott dátumig. Ugyanazt az árut többször is kell szállítani egy éven belül, különböző mennyiségekben ugyanazon a szerződés keretében. Ebben az esetben is rendeltünk attribútumokat a Szerepel kapcsolathoz, a rendelt mennyiséget, illetve a szállított mennyiséget. Egy tételen belül több különböző árut is megrendelhet a vevő egy adott dátumra. Ugyanaz az áru több különböző tételen is szerepelhet, tehát a Szerepel kapcsolat *n*: *m* típusú.

Az adatbázis E/K diagramát a 2.3. ábrán láthatjuk.

2.3. Gyakorlatok

2.1. Tervezzük meg egyed-kapcsolat diagram segítségével egy ingatlan ügynökség adatbázisát, mely a következő ingatlanok: lakások, házak és telkek eladását irányítja. Az ingatlanoknak vannak tulajdonosai, akiknek neve, címe, telefonszámai az adatbázisba kell kerüljenek. Minden ingatlan esetén el akarjuk tárolni, hogy milyen helységben van, a város melyik negyedében, a negyednek lehet egy jegye, hogy mennyire jó, keresett (pl. Györgyfalvi negyed 9), az ingatlan címe, ingatlan ára. A telkeknek a területére vagyunk kíváncsiak. A lakásoknak is a területét, viszont a ház esetén telek terület és lakható terület is tárolandó adatok. A lakások és házak esetén is fontos, hogy új-e, szobák száma, fürdőszobák száma, fűtés típusa, van-e telefon és extrák, amivel esetleg rendelkezik a lakás, illetve ház. Extrák lehetnek: termopán ablakok, felszerelt konyha, csempe típusa, stb. A vásárlók is bekerülnek az adatbázisba, nevük, címük, telefonszámaik.

2.2. Tervezzük meg egyed-kapcsolat diagram segítségével a zenés CD-k adatbázisát. Egy CD típusa lehet: mp3-as; audio; videoklipp. Egy CD-n lehet több zeneszám, kíváncsiak vagyunk a zeneszám előadójára, időtartamára, mennyi helyet foglal, szerzőire (1 számnak több szerzője is lehet), az album nevére, a zene stílusára, megjelenési évére. Egy előadónak több albuma is lehet, egy albumon több zeneszám, de feltételezzük, hogy egy zeneszám csak egy albumon jelenik meg, egy albumon azonos stílusú zeneszámok vannak. Ugyanabban a stílusban több zeneszám is található a CD-ken.



2.3. ábra: A Nagyker adatbázis egyed/kapcsolat diagramja

2.3 Tervezzük meg egyed-kapcsolat diagram segítégével egy család költségvetését. Egy családban egy tagnak egy szerepe van, pl. apa vagy anya vagy gyerek vagy nagyapa, stb. Egy családtagnak több féle jövedelme is lehet, pl: havi fizetés, nyereségreszesedés, öröklés, stb. A jövedelem esetén érdekeltek vagyunk az összegben és annak időpontjában. A család költségeit két szinten csoportosítjuk, például a lakásfenntartáson belül lehet villany, gáz, stb. költség, telefonköltségen

belül a fix, illetve a családtagok mobil készülékének költségei, szórakozási költségekhez tartozik a kirándulás, disco, cigaretta, stb. Fontos, hogy melyik családtagra vonatkozik a költség, mennyi az értéke és az időpont. Példa egy költségre: 2004-07-05-én a Csaba nevű gyerek kirándulni megy a tengerpartra, ez 4300000-ba kerül és a kirándulás nevű költség alcsoporthoz tartozik. A közös költségeknek bevezethetünk egy plussz sort a családtagok közé, „Közös" névvel.

2.4 Tervezzük meg egyed-kapcsolat diagram segítségével a kolozsvári Filharmónia adatbázisát, mely tartalmazza az előadásra kerülhető zeneműveket és azok előadási programját. A zeneműről tároljuk a szerzőjét, a zene stílusát (zongoraverseny, rapszódia, stb.), időtartamát. Egy szerzőnek több zeneműve is lehet, a szerzőnek a nemzetiségére vagyunk kíváncsiak. A zeneműveket előadók adják elő, lehetnek szóló előadók és/vagy zenekar. A szóló előadókról tároljuk a hangszert (hegedű, zongora, kántó, stb.), melyen játszik. Egy zeneműt több dátumon is előadásra kerülhet különböző előadó által.

2.5 Tervezzük meg egyed-kapcsolat diagram segítségével a rendőrség adatbázisát. A rend megszegése különböző kategóriákba sorolható: lopás, csalás, gyilkosság, kábítószer, stb. A kategóriákat alkategóriákra oszthatjuk, például: a lopáson belül lehet: autólopás, pénzlopás, stb., kábítószer kategórián belül lehet: kábítószer csempészet, kábítószer forgalmazás, kábítószer fogyasztás. A rendőrség alkalmazottai (személyi szám, név, cím, telefonszám) különböző csoportokba vannak szervezve, egy csoport egy alkategóriával foglalkozik, minden csoportnak van egy főnöke. Ugyanazzal az alkategóriával több csoport is foglalkozhat, a város különböző kerületeiben. A rendőrség alkalmazottainak különböző kiképzéseik lehetnek, egy adott pillanatban egy rangjuk van (őrmester, hadnagy, százados, stb.). A bűnözőket is nyilvántartjuk (személyi szám, név, cím, telefonszám, fénykép, újlenyomat) és az elkövetett bűneiket is. Az elkövetett bűn egy alkategóriába tartozik, az elkövetésnek van dátuma, büntetés (ami lehet: pénzbüntetés, börtön, stb.)

2.6 Tervezzük meg egyed-kapcsolat diagram segítségével egy személy ismerőseinek az adatbázisát. (egy ismerősről tároljuk a nevét, címét, telefonszámjait, stb.) Az ismerősöket csoportosítjuk, lehetséges értékek a csoportban: barátok, szerelmek, orvosa, tanára, stb. Egy ismerős több csoportba is tartozhat, lehet orvosa és barátja is egyidőben. Egy ismerősnek van egy alapképzése (pl. orvos, tanár, iskolás, stb.) és érthet több dologhoz is (pl. számítógép javítás, vízszerelés, autójavítás, stb.). Egy ismerősnek lehetnek kedvenc szórakozásai (pl. színház, mozi, disco, hegymászás, stb.), melynek több helyszíne is lehet, (pl. 2 discoba szokott járni). Az ismerőstől kölcsön kérhettem bizonyos dolgokat, (pl. könyvet, fúrógépet, pénz, stb.), amit egy adott határidőre vissza kell vinni.

2.7 Tervezzük meg egyed-kapcsolat diagram segítségével egy város mozijaiban vetítésre kerülő filmek adatbázisát. A városban több mozi is van. Hosszabb idő leforgása alatt egy film több moziban is vetítésre kerül. A felhasználó érdekelt olyan filmekben, amelyekben adott filmszínész szerepel vagy adott rendező rendezte, a filmet milyen évben gyártották. Információk a filmekről: címe, rendezője, szereplői, filmtipusa (bűnügyi, romatikus, rajzfilm stb.), studió, mely gyártotta, milyen díjakat kapott.

Feltételezzük, hogy egy moziban egy nap ugyanazt a filmet vetítik, több kezdeti időpontban is. Egy moziban az összhelyek száma változatlan. Egy adott pillanatban egy nap, egy kezdeti órában egy moziban tudni akarjuk mennyi az eladott jegyek száma.

2.8 Tervezzük meg egyed-kapcsolat diagram segítségével egy autó alkatrészeket áruló üzlet adatbázisát. Az alkatrészeket csoportokba sorolhatjuk, például: fényszórók, ablaktörlők, gumiabroncsok, szűrők, stb. Az alkatrésznek van gyártója, akiknek van neve, címe és fontos, hogy milyen nemzetiségű. Minden alkatrésznek több jellemvonása is lehet, érdekeltek vagyunk a jellemvonás értékében is, pl: a gumiabroncsnak van mérete (jellemvonás), annak értéke a különböző gumiabroncsoknál különböző lehet, pl: 175x15, 145x13, stb. Az ablaktörlőnek is lehet mérete: 20 cm, 23 cm, stb. A gumiabroncsnak olyan jellemvonása is van, hogy milyen időben lehet használni, a jellemvonás értéke lehet: téli, nyári, eső. Az alkatrész használható különböző autó

típusoknál. Az üzletben létező alkatrészről tudni akarjuk, hogy mennyi az ára és milyen mennyiségben található az üzletbe.

2.9 Tervezzük meg egyed-kapcsolat diagram segítségével egy város szórakozóhelyeit. Szórakozóhely típusa lehet disco, vendéglő, bár stb. Egy szórakozóhelyről a felhasználót érdekli, hogy a város melyik negyedében van, mi a pontos címe, telefonszámok és mit ajánl az illető szórakozóhely: mit lehet fogyasztani, mit lehet játszani, milyen előadás van. Fogyasztható több féle ital, desszert, előétel, főétel, mindenik esetében érdekli a felhasználót a tömeg és az ár. Ha lehet játszani akkor például billiárdozni, rulletezni, stb. Esetleg előadás is van, ahol a felhasználót érdekli a rendező, az előadók, előadás címe, típusa. A fogyasztható dolgok, mit lehet játszani időben nem vátozik, az előadásoknak viszont van időpontja.

2.10 Tervezzük meg egyed-kapcsolat diagram segítségével egy könyvtár könyveinek a kezelésére szükséges adatbázist. Tároljuk a könyvekről a következő információkat: ISBN, könyv címe, nyelv, szerzők, doméniumok, (pl.: lineáris algebra, projektív geometria, grafika, adatbázis stb.) mely megadja, a könyv milyen doméniumokból tartalmaz anyagot. A könyvtárban egy könyvből több példány is lehet, a raktári szám azonosít egy példányt. A kölcsönzők a könyvek példányait különböző időpontokban kikölcsönzik. Ugyanazt a példányt többször is elviheti ugyanaz a személy. Kölcsönzőről tárolandó információk: név, cím, telefonszám, könyvtári igazolvány száma.

3. A relációs adatmodell

Az első ABKR-ek hálós vagy hierarchikus adatmodellt használták. Manapság a relációs adatmodell a legelterjedtebb. A népszerűséget annak köszönheti, hogy nagyon egyszerű deklaratív nyelvvel rendelkezik az adatok kezelésére, illetve lekérdezésére. A relációs adatmodell értékorientált, ez ahhoz vezet, hogy a relációkon értelmezett műveletek eredményei szintén relációk.

A relációs adatmodellt E. Codd vezette be, lásd [Co70] és [Co82].

3.1. A relációs adatmodell értelmezése

Ha adottak a $D_1, D_2, \ldots, D_n$ nem szükségszerűen egymást kizáró halmazok, akkor R egy reláció a $D_1, D_2, \ldots, D_n$ halmazokon, ha $R \subseteq D_1 \times D_2 \times \ldots \times D_n$ (Descartes-féle szorzat).

A relációs adatmodell szempontjából D_i az A_i attribútum értékeinek tartománya (doméniuma). D_i lehet egész számok halmaza, karaktersorok halmaza, valós számok halmaza stb., n a reláció foka.

Egy ilyen relációt táblázatban ábrázolhatunk:

R	A_1	...	A_j	...	A_n
r_1	a_{11}	...	a_{1j}	...	a_{1n}
⋮					
r_i	a_{i1}	...	a_{ij}	...	a_{in}
⋮					
r_m	a_{m1}	...	a_{mj}	...	a_{mn}

ahol $a_{ij} \in D_j$.

A táblázat sorai a reláció elemei. Nagyon sok esetben a tábla megnevezést használják a reláció helyett. A relációt a következőképpen jelöljük: R (A_1, A_2,..., A_n). A reláció nevét és a reláció attribútumainak a halmazát együtt *relációsémának* nevezzük.

3.1. példa: Diákok reláció:

Név	***SzületésiDátum***	***CsopKod***
Nagy Ödön	1975-DEC-13	512
Kiss Csaba	1971-APR-20	541
Papp József	1973-JAN-6	521

3.2. példa: Könyvek reláció:

Szerző	***Cím***	***Kiadó***	***KiadÉv***
C. J. Date	An Introduction to Database Systems	Addison-Wesley	1995
Paul Helman	The Science of Database	IRWIN	1994

3.2. A relációs adatmodell tulajdonságai

A relációs adatbázis relációi vagy táblái a következő tulajdonságokkal rendelkeznek:

1. A tábla nem tartalmazhat két teljesen azonos sort, azaz két egyed előfordulás (sor) legalább egy tulajdonság (attribútum) konkrét értékében el kell hogy térjen egymástól.

2. Kulcs értelmezése: egy *S* attribútumhalmaz az *R* reláció kulcsa, ha:
   - *R* relációnak nem lehet két sora, melynek értékei megegyeznek az *S* halmaz minden attribútumára.
   - *S* egyetlen valódi részhalmaza sem rendelkezik a) tulajdonsággal.

Ha a konkrét egyedek több olyan tulajdonsággal is rendelkeznek, amelyek értéke egyedi minden egyes előfordulásra nézve, akkor több *kulcsjelölt*ről beszélhetünk. Ezek közül egyet *elsődleges kulcs*nak kell kijelölni. Az is megtörténhet, hogy nincs olyan tulajdonság, amelynek értéke egyedi lenne az egyed-előfordulásokra nézve. Ekkor több tulajdonság értéke együtt fogja jelenteni az elsődleges (*összetett*) kulcsot. Az 1. tulajdonságból következik, hogy mindig kell legyen elsődleges kulcs, ha más nem, a teljes sor mindig egyedi. Elsődleges kulcs értéke soha nem lehet null vagy üres.

3. A táblázat sorainak vagyis az egyedelőfordulásoknak a sorrendje lényegtelen.
4. A táblázat oszlopaira vagyis a tulajdonságtípusokra, attribútumokra nevükkel hivatkozunk, tehát két attribútumnak nem lehet ugyanaz a neve.
5. A táblázat oszlopainak a sorrendje lényegtelen.

3.3. Relációs adatbázis séma meghatározása

A relációs adatbázis sémáján az adatbázist alkotó relációk sémájának az összességét értjük. A relációkban tárolt konkrét értékek pedig alkotják a relációs adatbázist. Egy helyes relációs adatbázis sémát két módszerrel kaphatunk: az *egyed/kapcsolat diagramot átírjuk* relációsémákká, illetve egy létező relációs adatbázis sémát *normalizálás* segítségével normál formára hozzuk.

3.3.1. Az egyed/kapcsolat diagramok átírása relációs modellé

Három lépésben valósíthatjuk meg az E/K diagram átírását relációs modellé:

1. Az egyed/kapcsolat diagramban található egyedhalmazokat a relációs modellben relációként ábrázoljuk. Egy *E* egyedhalmaz az egyed/kapcsolat diagramból egy relációval ábrázolható, melynek attribútumai az *E* egyedhalmaz attribútumai lesznek.
2. Egy *R* kapcsolatot az E_1, E_2,..., E_m egyedhalmazok között az egyed/kapcsolat diagramból szintén egy relációval ábrázoljuk a relációs modellben, melynek attribútumai az E_1, E_2,..., E_m egyedhalmazok kulcsai. Ha ugyanazon attribútum nevek fordulnak elő, át kell őket nevezni. Legyenek K_1, K_2,..., K_m az E_1, E_2,..., E_m egyedhalmazok kulcs attribútumainak a halmazai, ahol $K_i \in E_i$, akkor a relációs modellben kapott relációnak attribútumai a K_1, K_2,..., K_m. Ha a kapcsolatokhoz attribútumokat rendeltünk az E/K diagramon, a relációs modell relációjában a kulcsok mellett a kapcsolat attribútumai is szerepelnek.
3. Közös kulcsú relációk összevonása: Ha két relációnak van egy közös kulcsjelöltje, a két relációt összevonjuk és helyettesítjük egy relációval. Előnyök: kevesebb helyet foglal a reláció, a lekérdezések hamarabb megválaszolhatók, nincsenek annyira szétdarabolva az összetartozó adatok.

3.3. példa: Írjuk át a 2.14. példából az egyedhalmazokat. Az egyedhalmazok kulcsai a relációk kulcsai lesznek, és ezt aláhúzással jelöljük a relációs modell relációi esetében is.

```
(1) Alkalmazottak (SzemSzám, Név, Fizetés)
(2) Managerek (SzemSzám)
(3) Részlegek (RészlegID, Név, Helység)
(4) Szállítók (SzállID, Név, Helység, UtcaSzám)
(5) ÁruCsoportok (CsopID, Név)
(6) Áruk (ÁruID, Név, MértEgys, MennyRakt)
(7) Vevők (VevőID, Név, Helység, UtcaSzám, Mérleg)
(8) Szerződések (SzerződID, Dátum, Részletek)
(9) Tételek (TételID, Dátum)
```

A kapcsolatok átírása a következő relációkat eredményezi:

(10) Dolgozik(SzemSzám, RészlegID)
(11) Irányít (SzemSzám, RészlegID)
(12) Árul (CsopID, RészlegID)
(13) Tartozik (CsopID, ÁruID)
(14) Szállít (SzállID, ÁruID, Ár)
(15) Elhelyez (VevőID, SzerződID)
(16) Tartalmaz (SzerződID, TételID)
(17) Szerepel (TételID, ÁruID, RendMenny, SzállMenny)

Nagyon fontos, hogy a kapcsolatokból kapott relációk kulcs attribútumait helyesen határozzuk meg, mert a következő lépésben szükségünk lesz rájuk. Egy relációnak több kulcsjelöltje is lehet, a tervező dönti el, hogy melyiket választja.

Ha a kapcsolat **bináris**, E_1 és E_2 egyedhalmazok között áll fenn, legyen E_1 egyedhalmaz kulcs attribútumainak a halmaza K_1, E_2 egyedhalmaz kulcs attribútumainak a halmaza K_2, irányelvként elfogadhatjuk a következőket:

– 1:1 típusú kapcsolatok esetén lehet a K_1 vagy a K_2 is kulcsjelölt.

Például a *Managerek* és *Részlegek* egyedhalmazok közötti Irányít kapcsolat esetén a SzemSzám és a RészlegID is lehet kulcs.

– ha 1: *n* típusú kapcsolat áll fenn E_1 és E_2 egyedhalmazok között, akkor a kapcsolatnak megfelelő reláció kulcsjelöltje a K_2.

Például a *Részlegek* és *Alkalmazottak* egyedhalmazok között a Dolgozik kapcsolat 1: *n* típusú, a Dolgozik relációnak a relációs adatmodellben a kulcsa a SzemSzám.

– ha *n*: *m* típusú kapcsolat áll fenn E_1 és E_2 egyedhalmazok között, akkor a kapcsolatnak megfelelő relációnak kulcsjelöltje összetett kulcs lesz, a K_1 és a K_2 egyesítése.

Például a *Szállítók* és *Áruk* közötti *Szállít* kapcsolat *n*: *m* típusú, így a kulcs összetett, SzállID és ÁruID együtt.

Az E/K modellben a specializáló („az_egy”) hierarchiák mögött az a filozófia áll, hogy a hierarchiát olyan egyedek népesítik be, amelyek amelyek specializáló kapcsolatokon keresztül kapcsolódnak egymáshoz. Emiatt természetes, hogy az általános egyedhalmazhoz egy olyan relációt készítünk, amelynek a sémája tartalmazza az egyedhalmaz attribútumait. Ami az alosztályokat illeti, ha egy általános osztálynak több alosztálya is van, minden alosztályban megismételjük az általános egyedhalmaz kulcsát. A specializáló kapcsolatokhoz nem készítünk relációkat. Ehelyett a specializáló kapcsolatokat burkolt formában az a tény fejezi ki, hogy a kapcsolódó egyedeknek ugyanazok a kulcsértékei. A fenti példa esetén, a Managerek reláció fogja azon alkalmazottak személyi számait tárolni, akik managerek. Ha a managereket keressük, megtaláljuk őket a Managerek relációban, és az a tény, hogy a személyi számukat tároljuk, biztosít minket arról, hogy a managerek is alkalmazottak, és ha a többi attribútum értékére vagyunk kíváncsiak egy manager esetében, ezeket megtaláljuk az Alkalmazottak relációban a személyi szám segítségével.

Közös kulcsú relációk összevonása:

Alkalmazottak és Dolgozik közös kulcsa SzemSzám, az Alkalmazottak relációt kiegészítjuk a RészlegID-al, így minden alkalmazott esetében tudjuk, melyik részlegnél dolgozik. Tudjuk, hogy egy alkalmazott csak egy részlegnél dolgozhat. Eredményként kapjuk:

Alkalmazottak (SzemSzám, Név, Fizetés, RészlegID)

Részlegek és Irányít közös kulcsa a RészlegID, a Részlegek relációt kiegészítjük a managere személyi számával, át is kereszteljük ManSzemSzám-á, hogy világosabb legyen mit tartalmaz az attribútum. Eredményként kapjuk a:

Részlegek (RészlegID, Név, Helység, ManSzemSzám)

Kérdés: - ha a Részlegekben van egy olyan sor, mely az Irányítban hiányzik, vagyis épp nincs managere az egyik részlegnek a ManSzemSzám attribútumnak használhatunk egy speciális *nullértéket*, amit NULL-lal fogunk jelölni. Amikor NULL szerepel egy sor valamely attribútumhoz tartozó komponensében, az azt jelenti, hogy a sornak ehhez az attribútumához nincsen megfelelő érték megadva. A NULL érték nem 0 (zéró), nem üres karaktersor, hanem egy speciális érték.

ÁruCsoportok és Árul közös kulcsa a CsopID, tehát kiegészíthetjük az ÁruCsoportok relációt a RészlegID-al, amelyik az illető árucsoportot árulja. Egy árucsoportot egy részleg árul, az Árul kapcsolat 1: *n* típusú a Részlegek és ÁruCsoportok között. Eredményként kapjuk a:

```
ÁruCsoportok (CsopID, Név, RészlegID)
```

Hasonlóan Áruk és Tartozik összevonása után:

```
Áruk (ÁruID, Név, MértEgys, MennyRakt, CsopID)
```

Szerződések és Elhelyez összevonása eredményeként:

```
Szerződések (SzerződID, Dátum, Részletek, VevőID)
```

Tételek és Tartalmaz összevonása után:

```
Tételek (TételID, Dátum, SzerződID)
```

Az összetett kulcsú Szállít és Szerepel nem vonható össze más relációkkal, mivel ezek *m*: *n* típusú kapcsolatokat modellálnak.

Az összevont relációkon kívül még megmaradt relációk ahhoz, hogy az adatbázisséma teljes legyen a közös kulcsú relációk összevonása után:

```
Managerek (SzemSzám)
Szállítók (SzállID, Név, Helység, UtcaSzám)
Vevők (VevőID, Név, Helység, UtcaSzám, Mérleg)
Szállít (SzállID, ÁruID, Ár)
Szerepel (TételID, ÁruID, RendMenny, SzállMenny)
```

A Managerek relációt esetleg elhagyhatnánk, ha nincs olyan managerünk, aki nem irányít részleget. Ha minden manager irányít egy részleget, akkor megtaláljuk őket a Részlegek relációban. Ha megengedtük volna, hogy egy részleget több manager is irányítson, akkor az Irányít kapcsolat a Részlegek és Managerek között 1: *n* típusú kapcsolat lett volna. Ebben az esetben a kapcsolat relációvá való átírása után kapott Irányít relációban csak a SzemSzám lett volna kulcs, nem vonhattuk volna össze a Részlegekkel, hanem a Managerek relációval vonhattuk volna össze, minden manager személyi száma mellett szerepelt volna annak a részlegnek az ID-je, amelyiket irányítja. Ugyanaz a RészlegID több managernél is szerepelhet. □

3.3.2. Normalizálás

3.3.2.1. Funkcionális függőségek

Legyen egy reláció

$R(A_1, A_2, ..., A_n)$, ahol A_i attribútumok.

Jelöljük az attribútumok halmazát

$A = \{A_1, A_2, ..., A_n\}$.

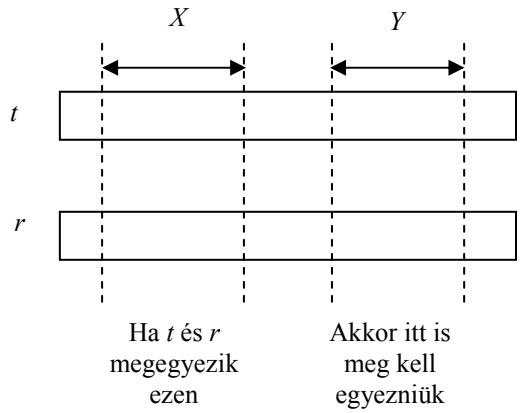
Legyenek X és Y az R reláció attribútumhalmazának részhalmazai, vagyis $X, Y \subset A$. Ezeket a jelöléseket használjuk a továbbiakban, ha esetleg nem ismételjük meg.

X attribútumhalmaz *funkcionálisan meghatározza* Y attribútumhalmazt (vagy Y *funkcionálisan függ* X-től), ha R minden előfordulásában ugyanazt az értéket veszi fel Y, amikor az X értéke ugyanaz.

Másképp: X funkcionálisan meghatározza Y-t, ha R két sora megegyezik az X attribútumain (azaz ezen attribútumok mindegyikéhez megfeleltetett komponensnek ugyanaz az értéke a két sorban), akkor meg kell egyezniük az Y attribútumain is. Ezt a függőséget fomálisan $X \to Y$-nal jelöljük.

Relációs algebrai műveletek (lásd 5.1.) segítségével a következőképpen értelmezhetjük a funkcionális függőséget:

$X \to Y$, ha $\forall t, r \in R$ sor esetén, melyre $\pi_X(t) = \pi_X(r)$, akkor $\pi_Y(t) = \pi_y(r)$.



3.1. ábra: A funkcionális függőség két soron vett hatása

3.4. példa: SzállításiInformációk reláció:

SzállID	***SzállNév***	***SzállCím***	***ÁruID***	***ÁruNév***	***MértEgys***	***Ár***
111	Rolicom	A. Iancu 15	45	Milka csoki	tábla	25000
222	Sorex	22 dec. 6	45	Milka csoki	tábla	26500
111	Rolicom	A. Iancu 15	67	Heidi csoki	tábla	17000
111	Rolicom	A. Iancu 15	56	Milky way	rúd	20000
222	Sorex	22 dec. 6	67	Heidi csoki	tábla	18000
222	Sorex	22 dec. 6	56	Milky way	rúd	22500

Funkcionális függőségek:

SzállID $\rightarrow$ SzállNév

SzállID $\rightarrow$ SzállCím.

Mivel mindkét függőségnek ugyanaz a bal oldala, SzállID, ezért egy sorban összegezhetjük:

SzállID $\rightarrow$ {SzállNév, SzállCím}

Szavakban, ha két sorban ugyanaz a SzállID értéke, akkor a SzállNév értéke is ugyanaz kell legyen, illetve a SzállCím értéke is.

Ezenkívül:

ÁruID $\rightarrow$ ÁruNév

ÁruID $\rightarrow$ MértEgys (azzal a feltevéssel, ha más mértékegységben árulják az árut, más ID-t is kap).

Hasonlóan egy sorban:

ÁruID $\rightarrow$ {ÁruNév, MértEgys}

A funkcionális függőséget felhasználva adhatunk még egy értelmezést a reláció kulcsának. Egy vagy több attribútumból álló $\{C_1, C_2, \ldots, C_k\}$ halmaz a *reláció kulcsa*, ha:

- Ezek az attribútumok funkcionálisan meghatározzák a reláció minden más attribútumát, azaz nincs az *R*-ben két különböző sor, amely mindegyik $C_1, C_2, \ldots, C_k$ -n megegyezne.
- Nincs olyan valódi részhalmaza $\{C_1, C_2, \ldots, C_k\}$-nak, amely funkcionálisan meghatározná az *R* összes többi attribútumát, azaz a kulcsnak minimálisnak kell lennie.

3.5. példa: a SzállításiInformációk reláció kulcsa a {SzállID, ÁruID}, egy szállító egy árut egy árban szállít egy adott pillanatban. Nincs a táblában 2 sor, ahol ugyanaz legyen a SzállID és az ÁruID is. Csak a SzállID nem elég kulcsnak, mert egy szállító több árut is szállíthat, az ÁruID sem elég, mert egy árut több szállító is ajánlhat. □

*Szuperkulcs*oknak nevezzük azon attribútumhalmazokat, melyek tartalmaznak kulcsot. A szuperkulcsok eleget tesznek a kulcs definíció első feltételének, de nem feltétlenül tesznek eleget a minimalitásnak. Tehát minden kulcs szuperkulcs.

Az *R* (A_1, A_2,..., A_n) reláció esetén A_i attribútum *prim*, ha létezik egy *C* kulcsa az *R*-nek, úgy hogy $A_i \in C$. Ha egy attribútum nem része egy kulcsnak, akkor *nem prim* attribútumnak nevezzük.

*Triviális funkcionális függőség*ről beszélünk, ha az *Y* attribútum halmaz részhalmaza az *X* attribútum halmaznak $(Y \subset X)$, akkor *Y* attribútum halmaz funkcionálisan függ *X* attribútum halmaztól ($X \to Y$).

3.6. példa: Triviális funkcionális függőség**:** {SzállID, ÁruID} $\to$ SzállID. □

Minden triviális függőség érvényes minden relációban, mivel amikor azt mondjuk, hogy „két sor megegyezik *X* minden attribútumán, akkor megegyezik ezek bármelyikén is".
Nem triviális egy $X_1X_2\ldots X_p \to Y_1Y_2\ldots Y_s$ funkcionális függőség, ha az *Y*-ok közül legalább egy különbözik az *X*-ektől, vagyis

$\exists Y_j, j \in [1,s]\ j \in \{1,2,...,s\}$ úgy, hogy $Y_j \neq X_k, \forall\, k \in \{1,2,...,p\}$.

Teljesen nem triviális egy $X_1X_2\ldots X_p \to Y_1Y_2\ldots Y_s$ funkcionális függőség, ha az *Y*-ok közül egy sem egyezik meg az *X*-ek valamelyikével, vagyis
$\forall Y_j, j \in [1,s]\ j \in \{1,2,...,s\}$ -re $Y_j \neq X_k, \forall\ k \in \{1,2,...,p\}$.

Parciális függőség: Ha *C* egy kulcsa az *R* relációnak, az *Y* attribútumhalmaz valódi részhalmaza a *C*-nek ($Y \subset C$) és *B* egy attribútum, mely nem része az *Y*-nak ($B \notin Y$), akkor az $Y \to B$-t egy parciális függőség. (B függ a kulcs egy részétől.)

3.7. példa: parciális függőségre: SzállID $\to$ SzállNév. □
A SzállításiInformációk relációban {SzállID, ÁruID} a kulcs, tehát
{SzállID, ÁruID} $\to$ SzállNév,
mivel a kulcs funkcionálisan meghatároz minden más attribútumot, de a SzállNév függ a kulcs egy részétől is.
Tranzitív függőség: Legyen $Y \subset A$ egy attribútumhalmaz és *B* egy attribútum, mely nem része *Y*-nak ($B \notin Y$). Egy $Y \to B$ funkcionális függőség tranzitív függőség, ha *Y* nem szuperkulcs *R* relációban és nem is valódi részhalmaza *R* egy kulcsának.

Honnan a tranzitív elnevezés? Amint látjuk, *Y* nem kulcs, nem része a kulcsnak, tehát egy nemtriviális funkcionális függőség az, hogy *Y* funkcionálisan függ az *R* kulcsától (*C*-től). Tehát $C \to Y$ és $Y \to B$, és erre mondhatjuk, hogy *B* tranzitív függőséggel függ *C*-től.

3.8. példa: Rendelések (RendelésSzám, Dátum, VevőID, VevőNév, Részletek), egy cég rendeléseit tartalmazó reláció. A különböző vevők rendeléseket helyeznek el a cégnél, a cég más-más számot ad a különböző rendeléseknek, így a RendelésSzám elsődleges kulcs lesz, tehát kulcs révén funkcionálisan meghatározza az összes többi attribútumot:
RendelésSzám $\to$ VevőID.
Ezenkívül fennáll a
VevőID $\to$ VevőNév
funkcionális függőség. Tehát a VevőNév tranzitív függőséggel függ a RendelésSzámtól.

Problémák:

Azokat a problémákat, amelyek akkor jelennek meg, amikor túl sok információt probálunk egyetlen relációba belegyömöszölni, *anomália*nak nevezzük. Az anomáliáknak alapvető fajtái a következők:

- *Redundancia*: Az információk feleslegesen ismétlődnek több sorban, mint például a SzállításiInformációk reláció esetében a szállító címe ismétlődik.
- *Módosítási problémák*: Megváltoztatjuk az egyik sorban tárolt információt, miközben ugyanaz az információ változatlan marad egy másik sorban. Például, ha a szállító címe változik, de csak egy sorban változtatjuk meg, nem tudjuk, melyik a jó cím. Jó tervezéssel elkerülhetjük azt, hogy ilyen hibák felmerüljenek.
- *Törlési problémák*: Ha az értékek halmaza üres halmazzá válik, akkor ennek mellékhatásaként más információt is elveszthetünk. Ha például töröljük a Rolicom által szállított összes árut, az utolsó sor törlésével elveszítjük a cég címét is.

– *Illesztési problémák*: Ha hozzáilleszteni akarunk egy szállítót, amely nem szállít egy árut sem, a szállító címét kitöltjük úgy, hogy az áruhoz „null” értékeket viszünk be, melyet majd utólag ki kell törölni, ha el nem felejtjük.

3.3.2.2. Relációk felbontása

Az anomáliák megszüntetésének elfogadott útja a relációk *felbontása* (*dekompozíció*-ja). *R* felbontása egyrészt azt jelenti, hogy *R* attribútumait szétosztjuk úgy, hogy ezáltal két új reláció sémáját alakítjuk ki belőlük. A felbontás másrészt azt is jelenti, hogyan töltsük fel a kapott két új reláció sorait az *R* soraiból.

Legyen egy *R* reláció $\{A_1, A_2, \ldots, A_n\}$ sémával, *R*-et felbonthatjuk *S* és *T* két relációra, amelyeknek sémái $\{B_1, B_2, \ldots, B_m\}$, illetve $\{C_1, C_2, ..., C_k\}$ úgy, hogy

1. $\{A_1, A_2, \ldots, A_n\} = \{B_1, B_2, \ldots, B_m\} \cup \{C_1, C_2, \ldots, C_k\}$, ahol
   $\{B_1, B_2, \ldots, B_m\} \cap \{C_1, C_2, ..., C_k\} \neq \emptyset$.

2. Az *S* reláció sorai az *R*-ben szereplő összes sornak a $\{B_1, B_2, \ldots, B_m\}$-re vett vetületei, azaz *R* aktuális előfordulásának minden egyes *t* sorára vesszük a *t* azon komponenseit, amelyek a $\{B_1, B_2, \ldots, B_m\}$ attribútumokhoz tartoznak. Mivel a relációk halmazok, az *R* két különböző sorának a projekciója ugyanazt a sort is eredményezheti az *S*-ben. Ha így lenne, akkor az ilyen sorokból csak egyet kell belevennünk az *S* aktuális előfordulásába.

3. Hasonlóan, a *T* reláció sorai az *R* aktuális előfordulásában szereplő sorok $\{C_1, C_2, ..., C_k\}$ attribútumok halmazára vett projekciói.

3.3.2.3. Normálformák

Az adatmodellezés egyik fő célja az optimalizálás, vagyis az adatmodellt alkotó egyedtípusok lehető legjobb szerkezetének a megkeresése. Az optimális adatmodell kialakítására egyéb technikák mellett a normalizálás szolgál. A normalizálás az a folyamat, amellyel kialakítjuk a relációk normálformáját (NF).

A normálformák: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF egymásba skatulyázottak. 2NF matematikailag jobb, mint 1NF, a 4NF jobb, mint a BCNF, az 5NF a legjobb, 3NF alakú reláció szükségszerűen 1NF és 2NF alakú is. Tehát a normálalakok nem függetlenek egymástól, hanem logikusan egymásra épülnek.

3.3.2.4. Első normál forma (1NF)

Értelmezés: Egy *R* reláció 1NF –ben van, ha az attribútumoknak csak elemi (nem összetett vagy ismétlődő) értékei vannak. Ez minimális feltétel, melynek egy reláció eleget kell tegyen, hogy a létező relációs ABKR-ek kezelni tudják.

3.9. Példa: A következő reláció nincs 1NF-ben:

Alkalmazottak:

SzemSzám	Név	Cím			Gyerek1	SzülDát1	…	Gyerek5	SzülDát5
		Helység	Utca	Szám					

Ahol a Cím összetett attribútum, a Helység, Utca és Szám attribútumokból áll. A Gyerek1, SzülDát1, Gyerek2, SzülDát2, Gyerek3, SzülDát3, Gyerek4, SzülDát4, Gyerek5, SzülDát5 ismétlődő attribútum. Egy személynek több gyereke is lehet, érdekeltek vagyunk a gyerekek keresztnevében és születési dátumukban. Jelenleg 5 gyerekről szóló információt tudunk eltárolni. Problémák az ismétlődő attribútumokkal: van olyan alkalmazott, akinek nincs egy gyereke se, nagyon soknak csak egy gyereke van, ezeknél fölöslegesen foglaljuk a háttértárolót. Jelenleg van a cégnek egy alkalmazottja, akinek 5 gyereke van, de akármikor alkalmaznak még egyet, akinek 6 gyereke van, akkor változtathatjuk a szerkezetet. □

3.3.2.5. 1NF-re alakítás

Ha egy reláció nincs 1NF-ben, mivel tartalmaz összetett attribútumokat, első normál formára hozhatjuk, ha az összetett attribútum helyett beírjuk az azt alkotó elemi attribútumokat. A fenti példa esetén a Cím attribútum nem fog szerepelni a reláció attribútumai között, csak a Helység, Utca és Szám attribútumok.

Ha adott egy R (A_1, A_2,..., A_n) reláció, mely nincs első normál formában, mivel ismétlődő attribútumokat tartalmaz, felbontással első normál formába hozható. Jelöljük az attribútumok halmazát

$A = \{A_1, A_2, ..., A_n\}$.

Legyenek C és I az R reláció attribútumhalmazának részhalmazai, vagyis $C, I \subset A$, ahol C kulcs és I ismétlődő attribútumhalmaz, mely tegyük fel, hogy k-szor ismétlődik. Legyen J azon attribútumok halmaza, melyek nem részei a kulcsnak, se nem ismétlődőek, vagyis $J \subset A$, $J \bigcap C = \varnothing$ és $J \bigcap I = \varnothing$. Tehát $A = C \bigcup I_1 \bigcup I_2 \bigcup \ldots I_k \bigcup J$. A felbontás után kapjuk a következő két relációsémát:

$S(C, I)$ és $T(C, J)$.

Vagyis az egyik relációban a kulcs attribútum mellett az ismétlődő attribútumok (csak egyszer) fognak szerepelni, a másikban pedig a kulcs mellett azon attribútumok, melyek nem ismétlődőek.

3.10. példa: A 3.9. példa esetén:

C = {SzemSzám}
I = {GyerekNév, SzülDátum}
J = {Név, Helység, Utca, Szám}.

A két új reláció:

```
Alkalmazott (SzemSzám, Név, Helység, Utca, Szám)
AlkalmGyerekei (SzemSzám, GyerekNév, SzülDátum)
```

Ebben az esetben, ha egy alkalmazottnak csak egy gyereke van az AlkalmGyerekei relációban egy sor lesz neki megfelelő, a SzemSzám attribútumnak ugyanazzal az értékével. Ha egy alkalmazottnak 5 gyereke van, 5 sor, ha ugyannak az alkalmazottnak még születik egy gyereke, akkor 6 sor tartalmazza az AlkalmGyerekei relációban az illető alkalmazott gyerekeit. Ha egy alkalmazottnak nincs egy gyereke se, az AlkalmGyerekei relációban nem lesz egy sor sem, mely hivatkozna rá a SzemSzám segítségével.

3.3.2.6. Második normál forma (2NF)

Értelmezés: Egy reláció 2NF formában van, ha első normál formájú (1NF) és nem tartalmaz $Y \rightarrow B$ alakú parciális függőséget, ahol B nem prim attribútum.

Amint látjuk, csak akkor tevődik fel, hogy egy reláció nincs 2NF-ben, ha a kulcs összetett.

3.11. példa: A 3.4. példa SzállításiInformációk relációja nincs 2NF-ben, mivel a reláció kulcsa a {SzállID, ÁruID} és fennáll a SzállID $\rightarrow$ SzállNév, tehát SzállNév függ a kulcs egy részétől is, tehát létezik parciális függőség.

Megoldás: több relációra kell bontani.

3.3.2.7. 2NF-re alakítás

Legyen R egy reláció, mely attribútumainak a halmaza $A = \{A_1, A_2, ..., A_n\}$ és $C \subset A$ egy kulcs. Ha a reláció nincs második normál formában, azt jelenti létezik egy $B \subset A$ nem kulcs $B \bigcap C = \varnothing$ attribútumhalmaz, mely függ funkcionálisan a kulcs egy részétől, vagyis létezik $D \subset C$, úgy hogy $D \rightarrow B$.

Az R relációt felbontjuk két relációra, melyek sémái:

$T(D, B)$ és $S(A - B)$

3.12. példa: Amint láttuk a 3.4. példa SzállításiInformációk relációjában fennállnak a:

SzállID $\rightarrow$ {SzállNév, SzállCím}

ÁruID $\rightarrow$ {ÁruNév, MértEgys}

funkcionális függőségek, a kulcs pedig a C ={SzállID, ÁruID}.

Első lépésben B = {SzállNév, SzállCím}, D = {SzállID}. Felbontás után kapjuk a

```
Szállítók (SzállID, SzállNév, SzállCím) és
SzállInf (SzállID, ÁruID, ÁruNév, MértEgys, Ár)
```

relációkat.

A Szállítók reláció 2NF-ben van, mivel a kulcs nem összetett, fel sem tevődik, hogy valamely attribútum függjön a kulcs egy részétől.

A SzállInf nincs 2NF-ben, mert fennáll a

ÁruID $\rightarrow$ {ÁruNév, MértEgys}.

Ebben az esetben B = {ÁruNév, MértEgys}, D = {ÁruID}. Tovább bontjuk a következő két relációra:

```
Áruk (ÁruID, ÁruNév, MértEgys),
Szállít (SzállID, ÁruID, Ár).
```

Az Áruk 2NF-ben van, mert a kulcs nem összetett és 1NF-ben van. A Szállít relációban egyetlen nem kulcs attribútum van: az Ár, és az nem függ csak az ÁruID-től, mert különböző szállító különböző árban ajánlhatja ugyanazt az árut, sem a SzállID-től nem függ funkcionálisan, mert egy szállító nem ajánlja ugyanabban az árban az összes árut. A kapott relációk:

Szállítók:

SzállID	***SzállNév***	***SzállCím***
111	Rolicom	A. Iancu 15
222	Sorex	22 dec. 6

Áruk:

ÁruID	***ÁruNév***	***MértEgys***
45	Milka csoki	tábla
67	Heidi csoki	tábla
56	Milky way	rúd

Szállít:

SzállID	***ÁruID***	***Ár***
111	45	25000
222	45	26500
111	67	17000
111	56	20000
222	67	18000
222	56	22500

3.3.2.8. Harmadik normál forma (3NF)

Értelmezés: Egy R reláció *harmadik normál formában* (3NF) van, ha második normál formában van és nem tartalmaz $Y \rightarrow B$ alakú *tranzitív funkcionális* függőséget, ahol B nem prim attribútum.

Értelmezés: Egy R reláció *harmadik normál formában* (3NF) van, ha létezik az R-ben egy $Y \rightarrow B$ alakú nem triviális funkcionális függőség, akkor Y az R reláció szuperkulcsa vagy a B prim attribútum (valamelyik kulcsnak része).

A két értelmezés ekvivalens. A második nem kéri a második normál formát, de mivel bármely létező $Y \rightarrow B$ funkcionális függőség esetén a bal oldal szuperkulcs, nem lehet annak része. Tehát elég, ha az összes létező funkcionális függőség esetén a bal oldal szuperkulcs, akkor a tranzitív függőség nem létezhet, mert a tranzitív függőség esetén a bal oldal nem kulcs és ez nem megengedett.

3.13. példa: A 3.8. példából a Rendelések reláció nincs 3NF-ben, mivel tartalmaz tranzitív funkcionális függőséget.

RendelésSzám $\rightarrow$ VevőID

VevőID $\rightarrow$ VevőNév.

Probléma, ha így ábrázoljuk a rendeléseket, hogy ha egy vevő több rendelést is elhelyez, ami lehetséges, akkor a vevő nevét ismételjük. Megoldás: 2 relációra bontjuk a relációt, mely nincs 3NF-ben. □

3.3.2.9. 3NF-re alakítás

Legyen R egy reláció, mely 2NF-ben van, viszont nincs 3NF-ben, attribútumainak a halmaza $A = \{A_1, A_2,..., A_n\}$ és $C \subset A$ elsődleges kulcs. Ha a reláció nincs harmadik normál formában, azt jelenti, hogy létezik egy $B \subset A$ nem kulcs $B \bigcap C = \varnothing$ attribútumhalmaz, mely tranzitív függőséggel függ a kulcstól, vagyis létezik D, úgy hogy $C \rightarrow D$ és $D \rightarrow B$. Mivel a reláció 2NF-ben van, B nem függ funkcionálisan C-nek egy részétől, tehát D nem kulcs attribútum.

Az R relációt felbontjuk két relációra, melyek sémái:

$T(D, B)$ és $S(A - B)$.

3.14. példa: A 3.8. példa Rendelések relációja esetén: B = {VevőNév}, D = {VevőID}, a felbontás után kapott relációk:

```
Vevők (VevőID, VevőNév)
RendelésInf (RendelésSzám, Dátum, VevőID)
```

Egy adatbázis modell kialakítása szempontjából a legkedvezőbb, ha az adatbázist alkotó relációk 3NF -ben vannak.

3.3.2.10. Boyce−Codd normál forma

Értelmezés: Az R reláció Boyce−Codd normál formában (BCNF) van akkor és csak akkor, ha minden olyan esetben, ha az R-ben érvényes egy $Y \rightarrow B$ nem triviális függőség, akkor az Y attribútumhalmaz az R reláció szuperkulcsa kell hogy legyen.

Amint látjuk a BCNF esetén elmarad a B-re vonatkozó megszorítás, hogy nem prim, tehát B lehet kulcs része.

3.15. példa: A következő reláció 3NF-ben van, viszont nincs BCNF-ban:

```
Postahivatal (Város, UtcaSzám, IrányítóSzám)
```

Egy ország összes postahivatalát tároljuk a fenti relációban, egy városban több postahivatal is lehet. Funkcionális függőségek a relációban:

{Város, UtcaSzám} $\rightarrow$ IrányítóSzám

IrányítóSzám $\rightarrow$ Város.

Tehát ahol ugyanaz az irányítószám, ismételni fogjuk a a város nevét a Postahivatal relációban. Nagyvárosok esetén, kerületenként lehet más az irányítószám, de ugyanabban a kerületben levő postahivatal esetén ugyanaz az irányítószám és a város is.

Két kulcsa van a Postahivatal relációnak a {Város, UtcaSzám} és {IrányítóSzám, UtcaSzám}.

A reláció 2NF-ben van, mindamellett, hogy az IrányítóSzám $\rightarrow$ Város parciális funkcionális függés, mivel IrányítóSzám része az {IrányítóSzám, UtcaSzám} kulcsnak, de a 2NF esetén, csak azon parciális függőségek nem megengedettek, ahol a jobb oldal nem prim, viszont a Város prim attribútum része egy kulcsnak.

A Postahivatal reláció 3NF-ben van, mivel ha az első értelmezést vesszük, 2NF-ben van és nem tartalmaz nem prim attribútumot, ha tartalmaz is tranzitív függőséget, a jobb oldalon a nem prim attribútum feltétel nem állhat fenn. A harmadik normál forma második értelmezése esetén, az IrányítóSzám $\rightarrow$ Város funkcionális függőség olyan, ahol a bal oldal nem szuperkulcs, viszont a jobb oldal prim attribútum és épp az amit a BCNF ki akar küszöbölni.

A Postahivatal relációt felbontjuk két relációra:

```
P1 (IrányítóSzám, Város)
P2 (IrányítóSzám, UtcaSzám).
```

Ha a postahivatalokról szóló információkat a P1 és P2 relációkban tároljuk, nem fogjuk ugyanannak a kis városnak a nevét ismételni minden postahivatala esetében, illetve a nagyvárosok ugyanabban a körzetben levő két postahivatala soraiban nem ismételjük a város nevét.

3.4. Gyakorlatok

3.1. Legyen a következő tábla egy relációs adatbázisból, mely az Athéni olimpia adatbázisának egy része. Egy sportoló egy sportágban indul, például úszás, de több alsportágban is, például 100 m gyors, 400m vegyes, stb. Az elért helyezés az alsportágra vonatkozik.

```
AtheniOlimpia (SportolóID, SportolóNév, OrszágKod, OrszágNév,
  SportAgID, SportAgNév, AlSportAgID, AlSportAgNév, ElértHelyezés)
```

BCNF-ben van-e a tábla? Irjuk fel a funkcionális függőségeket! Ha nincs BCNF-ben, indokoljuk meg miért nincs és alakítsuk át BCNF -ba. A kapott táblák esetén tüntessük fel az elsődleges és külső kulcsokat!

3.2. Legyenek a következő táblák egy gyógyszertár adatbázisából. Egy gyógyszernek van egy alapanyaga, például a Panadol (GyNév) alapanyaga a Paracetamol (AlapanyagNév), származási hely Franciaország. Az alapanyag váltja ki a gyógyszer hatását. Egy alapanyagnak több hatása is lehet. A Paracetamolnak van lázcsökkentő, gyuladáscsökkentő, fájdalomcsillapító. Egyebek csoportjai lehetnek: fogpaszták, szappanok, pelenkák, teák, stb., például a Colgate Herbal egy fogpaszta. A recept esetén a százalék az jelenti, hogy hány százalékos a kedvezmény.

```
Gyógyszerek (GyID, GyNév, AlapAnyagID, AlapanyagNév, Hatás1, Hatás2,
      Hatás3, SzármazásiHelyKod, SzármazásiHelyNév, ErvényességiDátum,
      MennyRakt,EladásiAr)
Egyebek (EgyID, EgyNév, CsopID, CsopNév, MennyRakt, EladásiAr)
Receptek (BetegSzemSzám, BetegNév, BetegCím, Dátum, GyogyszID1, Menny1,
     GyogyszID2, Menny2, …, GyogyszID5, Menny5, Százalék)
```

Harmadik normál formában vannak ezek a táblák? Ha nincsenek, indokoljuk meg, miért nincsenek és alakítsuk át 3NF-ba. Milyen problémák merülnek fel, ha az adott formában tároljuk az adatokat. (módosítás, hozzáillesztés, törlés esetén)? Határozzuk meg az elsődleges és a külső kulcsokat. Írjuk fel a létező funkcionális függőségeket.

3.3. Legyenek a következő táblák egy relációs adatbázisból, mely egy iskola osztályairól, tanárairól, termeiről, diákjairól és azok jegyeiről szóló információkat tárolja egy tanévben. Egy tanár több tantárgyat is taníthat, pl. matematikát és informatikát is, ugyanannak az osztálynak vagy különböző osztálynak.

```
Osztalyok (OsztalyKod, OsztalyfonokID, TeremID, TeremNev, Emelet);
Diakok (DiakID, DiakNev, DiákCím, Ev, OsztalyKod);
Tanárok (TanárID, Nev, Cim, Tel, TgyID, TgyNev, OsztalyKod);
Jegyek (DiákID, TgyID, Dátum, Felev, Jegy, Felevie);
```

A Felevie attribútum értéke 1, ha a jegy a félévi dolgozat jegye és 0 ha nem.

BCNF-ben vannak-e a táblák? Amelyik nincs, indokoljuk meg miért nincs és alakítsuk át BCNF-ba, majd jelöljük az elsődleges és külső kulcsokat.

3.4. Legyen a következő tábla egy relációs adatbázisból, mely háziorvosok betegeinek betegségeit és azoknak felírt orvosságokat tartalmazza. Tudjuk, hogy egy orvoshoz több beteg tartozik, egy rendelőben több orvos is rendelhet, egy beteg egy orvoshoz tartozik és az állapítja meg a betegséget és írja fel az orvosságokat. Egy beteg többször is lehet ugyanabban a betegségben, viszont minden esetben tároljuk a betegség (vizsgálat) dátumát. Ugyanarra a betegségre, különböző dátumon

írhatnak más-más orvosságot. Egy betegnél egy dátumon több betegséget is találhat az orvos. Adott dátumon egy betegnek több orvosságot is felírnak, amik a megállapított betegségekre használnak.

```
Betegek (Betegid, Betegnév, Betegcím, Betegszüldat, Orvosid, Orvosnév,
   RendelőHelységid, RendelőHelységnév, RendelőCím, RendelőMegyekod,
   Betegségdátum, Betegségnév, Orvosság1, Orvosság2, Orvosság3,
   Orvosság4, Orvosság5)
```

Harmadik normál formában van ez a tábla? Ha nincs, indokoljuk meg, miért nincs és alakítsuk át 3NF-ba. Határozzuk meg az elsődleges és a külső kulcsokat. Írjuk fel a létező funkcionális függőségeket!

3.5. Legyen a következő tábla egy relációs adatbázisból, mely egy kar diákjainak a jegyeiről szóló információkat tárolja:

```
Jegyek (DiákBeiktszám, DiákNév, DiákCím, DiákCsopKod, DiákEvfolyam,
   DiákSzakKod, DiákSzakNév,
   TantárgyKod1, TantárgyNev1, TantárgyKreditSzám1, Jegy1, … ,
   TantárgyKod8, TantárgyNev8, TantárgyKreditSzám8, Jegy8)
```

3 NF-ben van-e a tábla? Ha nincs 3NF-ben, indokoljuk meg miért nincs és alakítsuk át 3NF-ba.

3.6. Tervezzük meg számítógép komponenseket forgalmazó cég relációs adatbázis sémáját. Információk, amit az adatbázisnak tartalmaznia kell: Gyártók, KomponensCsoportok (Merevlemezek, Billentyűzetek, Alaplapok, Hangkártyák, stb.), Komponensek (merevlemez, billentyűzet, alaplap, stb.), Tulajdonságok (sebesség, méret, stb.). Egy komponensnek több tulajdonsága is lehet és minden tulajdonságnak van egy mérték egysége és értéke (pl. „Sebesség" tulajdonság mérték egysége a GHz és értéke 2,4. Sebessége van a videokártyának, a merevlemeznek, a processzornak is, de mindenik esetén más az érték). Minden komponensnek van ára.

A relációs séma legyen 3NF-ben, indokoljuk, miért van 3NF-ben.

3.7. Legyen a következő tábla egy relációs adatbázisból, mely egy egyetem tanárairól és az általuk tartott tantárgyakról szóló információkat tárol:

```
Tanít (TanárID, TanárNév, FunkcióID, FunkcióNév, TanszékID, TanszékNév,
   Dr-e, Fizetés, TantgyKod1, TangyNev1, …, TantgyKod5, TangyNev5)
```

3NF-ben van-e a tábla? Irjuk fel a funkcionális függőségeket! Ha nincs 3NF-ben, indokoljuk meg miért nincs és alakítsuk át 3NF-ba. A kapott táblák esetén tüntessük fel az elsődleges és külső kulcsokat!

3.8. Legyen a következő tábla egy relációs adatbázisból, mely egy könyvtár könyveinek és azoknak a kölcsönzését tárolja. Egy könyvet a Kota azonosít, viszont egy könyvnek több példánya is van, melyeket azok raktári száma azonosít. A kölcsönző a példányt viszi el.

```
Könyvek (Kota, Raktáriszám, Szerző1, Szerző2, Könyvcím, Kiadókod,
   Kiadónév, Megjév,ISBN, Pédányszám, Doménium1, Doménium2, Olvasókod,
   Olvasonév, Olvasócím, Dátumki, Dátumbe)
```

Harmadik normál formában van ez a tábla? Ha nincs, indokoljuk meg, miért nincs és alakítsuk át 3NF-ba.

3.9. Legyen a következő tábla egy relációs adatbázisból, mely egy videokölcsönző információit tartalmazza. Egy film több kazettán is meglehet, a kölcsönző a kazettát elviheti többször is:

```
Videokölcsön (Raktáriszám, Rendező, Filmcím, Szereplő1, Szereplő2,
   Szereplő3, Szereplő4, Szereplő5, Filmgyártó, Megjév, Kölcsönzőnév,
   Kölcsönzőid,Kölcsönzőcím, Kölcsönzőtel, Dátumki, Dátumbe)
```

Harmadik normál formában van ez a tábla? Ha nincs, indokoljuk meg, miért nincs és alakítsuk át 3NF-ba. Milyen problémák merülnek fel, ha az adott formában tároljuk az adatokat. (módosítás, hozzáillesztés, törlés esetén)? Határozzuk meg az elsődleges és a külső kulcsokat. Írjuk fel a létező funkcionális függőségeket.

3.10. Legyen egy klinika információit tároló tábla, mely a befektetett betegek és őket kezelő orvosokról szóló információkat tárolja. A szak név lehetséges értékei: sebészet, urológia, stb.

```
Klinika (BetegID, BetegNév, BetegCím, SzakID, SzakNév, OrvosID,
   Orvosnév, OrvosCím, Szakosodás1, Szakosodás2, Szakosodás3,
   BefektKezdDátum, BefektVégsőDátum, Orvosság1, Orvosság2, Orvosság3,
   Orvosság4 )
```

Harmadik normál formában van ez a tábla? Ha nincs, indokoljuk meg, miért nincs, írjuk fel a létező funkcionális függőségeket és alakítsuk át 3NF-ba. Milyen problémák merülnek fel, ha az adott formában tároljuk az adatokat (módosítás, hozzáillesztés, törlés esetén)? A kapott táblák esetén határozzuk meg az elsődleges és a külső kulcsokat.

3.11. A 2. fejezet gyakorlatai esetén minden egyed/kapcsolat diagramot írjunk át relációs adatbázis tábláivá, jelöljük az elsődleges és külső kulcsokat.

4. Relációsémák létrehozása SQL nyelvben

A legtöbb relációs ABKR az SQL-nek (Structured Query Language) nevezett lekérdezőnyelv segítségével kérdezi le és módosítja az adatbázist. Az SQL nyelv tartalmaz relációséma leírására alkalmas utasításokat is. Lehetőségünk van az attribútumnevek, illetve az attribútumok adattípusának a definiálására. Azonkívül megszorításokat is megadhatunk, melyeket az adatoknak ki kell elégíteniük, hogy bekerüljenek az adatbázisba.

Minden relációs ABKR rendelkezik egy SQL interpreterrel, melynek segítségével az SQL nyelv parancsait begépelhetjük és végrehajthatjuk. Az Oracle SQL interpreterje az SQL-PLUS, MS SQL Server-é pedig a Query Analyzer.

4.1. Adattípusok

Egy relációséma definiálása esetén minden attribútumnak meg kell adni az adattípusát. A következőkben a fontosabb adattípusokat mutatjuk be, ezeken kívül a különböző ABKR-ek még nagyon sokféle adattípussal tud dolgozni, így pl. BLOB-bak (Binary Large Objects).

- Karaktersorok, melyek lehetnek rögzített (CHAR(n)) vagy változó hosszúságúak (VARCHAR(n)).
- Egész számok (INT vagy INTEGER, esetleg SHORTINT, melyet kevesebb biten tárol a rendszer).
- Lebegőpontos értékek (REAL, vagy nagyobb pontossággal a DOUBLE PRECISION).
- Dátumok és idők (DATE és TIME).

4.2. Relációk létrehozása

A CREATE TABLE parancs segítségével relációkat (táblákat) tudunk létrehozni. A parancs komplex, a megszorításoknál láthatjuk a további lehetőségeket, legegyszerűbb formája esetén megadjuk a reláció nevét, az attribútumok nevét és típusát. Minden SQL parancs, így a CREATE TABLE is ; jellel végződik, az attribútumokat vesszővel választjuk el egymástól.

4.1. példa: Egy Alkalmazottak relációt a következő paranccsal tudunk létrehozni:

```
CREATE TABLE Alkalmazottak (
        SzemSzám INT,
        Név VARCHAR(30),
        Fizetés REAL,
        SzülDat DATE,
        Nem CHAR(1),
        RészlegID INT
    );
```

Tudva, hogy a nemnek csak 2 értéke lehet, kódolhatjuk F/N-nel és elég 1 karakter a tárolására, így a Nem attribútumot 1 hosszúságúnak deklaráltuk. A név esetében nem tudjuk a hosszúságát, ezért változó hosszúságú karaktersornak deklaráltuk. □

4.3. Relációsémák módosítása

Előfordul, hogy a létező tábla szerkezetét meg kell változtatnunk, újabb attribútumokat kell felvennünk az attribútumok listájába, esetleg a létező attribútumokat törölni szeretnénk az attribútumok listájából. Az ALTER TABLE parancs segítségével módosíthatunk egy létező relációsémát. A megszorítások módosítása is ezzel a paranccsal történik (lásd a 4.7.2.6.). Most egy új attribútum hozzáadását, illetve létező törlését mutatjuk be. A COLUMN kulcsszó bizonyos implementálások esetén hiányzik.

4.2. példa: Az Alkalmazottak reláció attribútumai közé még a Telefon nevű attribútumot vesszük fel.

```
ALTER TABLE Alkalmazottak ADD COLUMN Telefon CHAR(16);
```

A SzülDat nevű attribútumot kitöröljük:

```
ALTER TABLE Alkalmazottak DROP COLUMN SzülDat; □
```

4.4. Relációk törlése

Ha egy relációra már nincs szükségünk van rá lehetőség, hogy töröljük. A legtöbb esetben ideiglenesen létrehozott relációt kell törölnünk, mely valamely bonyolult közbülső lekérdezés eredményeképpen jött létre és többet nincs rá szükségünk. A parancs, mely a törlést megvalósítja:

```
DROP TABLE <tábla_név>;
```

4.5. Alapértelmezés szerinti értékek

Amikor új sorokat hozunk létre egy relációban, nem mindig ismerjük minden oszlop értékét. Ennek a problémának a megoldására az SQL nyelv NULL értéket használ, amit magyarul mondhatnánk úgy, hogy „nem tudom”, kivéve azokat az attribútumokat, ahol nem megengedett a NULL érték (lásd 4.7.2.3.). A fent leírt Alkalmazottak tábla esetén nincs egy attribútumnál sem NOT NULL kikötés, így ha beszúrunk egy sort a táblába és csak a SzemSzám és Név értékét adjuk meg, a többi attribútum értéke NULL értéket kap. Bizonyos esetekben egy más alapértelmezési értéket szeretnénk a NULL helyett, ezt a DEFAULT kulcsszó segítségével adhatjuk meg az attribútum neve és típusa után.

4.3. példa: Az előbbi Alkalmazottak tábla alapértelmezett értékekkel a SzülDat, Nem és Telefon attribútumok esetén:

```
CREATE TABLE Alkalmazottak (
        SzemSzám INT,
        Név VARCHAR(30),
        Fizetés REAL,
        SzülDat DATE DEFAULT '1900-01-01',
        Nem CHAR(1) DEFAULT '?',
        RészlegID INT,
        Telefon CHAR(16) DEFAULT 'ismeretlen'
);
```

Ebben az esetben, ha az új sorban nincs értéke például a Telefon attribútumnak, akkor az értéke 'ismeretlen' lesz az adatbázisban. □

4.6. Értéktartományok

Értelmezhetünk értéktartományokat, ami egy adattípus, esetleg alapértelmezett értékkel és megszorításokkal. Attribútum típusának definiálhatunk egy értéktartományt. Több attribútumnak is lehet ugyanaz az értéktartománya, ami sok esetben hasznos lehet. Ha változik valami, csak egy helyen kell változtatni. Ha két attribútumnak ugyanaz az értéktartománya, akkor az egyik bármikor felveheti a másik értékét. Az értéktartomány definiálásának általános alakja:

```
CREATE DOMAIN <név> AS <típusleírás>;
```

4.4. példa: A címet gyakran használjuk személyek, vevők, szállítók stb. esetén, ezért definiálhatunk neki egy értéktartományt:

```
CREATE DOMAIN CímDom AS VARCHAR(50) DEFAULT 'nem ismert';
```

A tábla definíciójában a Cím attribútum típusa CímDom:

```
Cím CímDom; □
```

Egy létező értéktartományt az ALTER DOMAIN parancs segítségével módosíthatunk, vagy törölhetjük a DROP DOMAIN utasítással.

4.7. Megszorítások

Az adatbázis módosításakor az új információ nagyon sokféleképpen lehet hibás. Ahhoz, hogy az adatbázis adatai helyesek legyenek, különböző feltételeknek kell eleget tenniük.

A *megszorítások* azon követelmények, melyeket az adatbázis adatai ki kell elégítsenek, ahhoz, hogy helyeseknek tekinthessék őket.

4.7.1. Megszorításokat osztályozása

1. *Egyedi kulcs feltétel*: egy relációban nem lehet két sor, melyeknek ugyanaz a kulcsértéke, vagyis ha C egy R reláció kulcsa, $\forall t_1, t_2 \in R$ sorok esetén $\pi_C(t_1) \neq \pi_C(t_2)$ (lásd relációs algebrai műveletek 5.1).
2. *Hivatkozási épség megszorítás*: megkövetelik, hogy egy objektum által hivatkozott érték létezzen az adatbázisban. Ez analóg azzal, hogy a hagyományos programokban tilosak azok a mutatók, amelyek sehova se mutatnak.
3. *Értelmezéstartomány-megszorítások*: azt jelentik, hogy egy attribútum az értékeit a megadott értékhalmazból vagy értéktartományból veheti fel.

Általános megszorítások: tetszőleges követelmények, amelyeket be kell tartani az adatbázisban.

A hivatkozásiépség megszorítást a következőkben részletezzük. A többi megszorítást az SQL nyelv adta lehetőségekkel együtt lásd a 4.7.2. alatt.

Külső kulcs egy *KK* attribútum vagy attribútumhalmaz egy R_1 relációból, mely értékeinek halmaza ugyanaz, mint egy R_2 reláció elsődleges kulcsának az értékhalmaza, és az a feladata, hogy az R_1 és R_2 közötti kapcsolatot modellezze. R_1 az a reláció, mely hivatkozik, az R_2 pedig, amelyre hivatkozik. Más megnevezés: az R_2 az apa és az R_1 a fiú (egy sorhoz az R_2-ből tartozhat több sor az R_1-ből, az R_2-ben elsődleges kulcs az attribútum ami a kapcsolatot megteremti. Fordítva nem állhat fenn a kapcsolat, hogy egy sorhoz az R_1-ből több sor is kapcsolódjon az R_2-ből). A hivatkozási épség megszorítás a következőket jelenti:

- az R_2 relációban azt az attribútumot (esetleg attribútumhalmazt), melyre az R_1 hivatkozik elsődleges kulcsnak kell deklarálni,
- *KK* minden értéke az R_1-ből kell létezzen az R_2 relációban, mint elsődleges kulcs értéke**.**

4.5. példa: Az Alkalmazottak reláció hivatkozik a Részlegek relációra a RészlegID külső kulcs segítségével.

Alkalmazottak (a fiú):

SzemSzám	***Név***	***RészlegID***	***Fizetés (euró)***
111111	Nagy Éva	2	300
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
123444	Vincze Ildikó	1	800
333333	Kovács István	2	500

Részlegek (az apa):

RészlegID	***RNév***	***ManagerSzemSzám***
1	Tervezés	123444
2	Könyvelés	234555
9	Beszerzés	456777

4.6. példa: A 3.12 példa relációi esetén, melyek a következők:

```
Szállítók (SzállID, SzállNév, SzállCím)
Áruk (ÁruID, ÁruNév, MértEgys)
Szállít (SzállID, ÁruID, Ár)
```

A Szállít reláció hivatkozik a Szállítók relációra a SzállID külső kulcs segítségével, illetve az Áruk relációra az ÁruID segítségével. (Szállítók apa, Áruk apa, Szállít fiú.)□

A hivatkozási épség megszorítás értelmében, ha egy R_1 reláció hivatkozik egy R_2 relációra a *KK* külső kulcs segítségével, akkor a *KK* értéke kell létezzen az R_2 relációban. Az adatbázis aktualizálása esetén ez a feltétel három esetben sérülhet.
1) Új sor *hozzáillesztése* az R_1 relációba: a *KK* külső kulcs értékét csak akkor vihetjük be az R_1 (fiú) reláció megfelelő oszlopába, ha az már létezik elsődleges kulcsként az R_2 (apa) relációban.

4.7. példa: A 4.5. példa esetén *nem illeszthetünk* olyan alkalmazottat az Alkalmazottak relációba, amelynek a RészlegID attributúm értéke például 5, mivel ez nem létezik a Részlegek relációban. Lehetséges értékek az aktuális állapotban: 1, 2 és 9.□

2) R_2 (apa) relációból *nem törölhetjük* ki azokat az elsődleges kulcsértékeket, melyekre van hivatkozás az R_1 (fiú) relációból.

4.8. példa: A 4.5. példa esetén nem törölhetjük ki egyik részleget sem, mert mindegyikre van hivatkozás, ha netán kitörölnénk, az Alkalmazottak relációban maradnának ún. „lógó" sorok.□

3) *Módosítás* felmerülhet mind a két relációban. Ha az R_1 (fiú) relációban módosítunk egy *KK* értéket, csak az R_2 (apa) relációban létezőre módosíthatjuk. Ha az R_2 (apa) relációban levő elsődleges kulcsot akarjuk módosítani, csak akkor tehetjük meg, ha nincs rá hivatkozás az R_1 (fiú) relációból.

4.9. példa: Az Alkalmazottak relációban a „Kovács István"-ra vonatkozó bejegyzés esetén a RészlegID-t módosíthatjuk 1 vagy 9-re, de nem módosíthatjuk például 3-ra. A Részlegek egy RészlegID-jét sem módosíthatjuk, mivel mindegyikre van hivatkozás. Esetleg módosíthatnánk egy már létező értékre a Részlegek táblából, de mivel a RészlegID elsődleges kulcs a Részlegek táblában, nem tehetjük meg. □

4.7.2. Megszorítások SQL-ben

Az ABKR-nek gondja kell legyen, hogy az adatbázisba csak helyes adat kerüljön. Az SQL2 számos lehetőséget kínál arra, hogy az épségi megszorításokat az adatbázisséma részeként adjuk meg. Mikor egy adatbázis alkalmazás fut, az ABKR ellenőrzi, hogy a megszorítások teljesülnek-e, és ha nem, nem engedi meg a változtatást.

Többféle megszorítást is láttunk a 4.7.1.-ben, most részletezzük, hogyan tudjuk ezeket SQL parancsok segítségével megadni.

4.7.2.1. Egyedi kulcsok SQL-ben

Az elsődleges kulcsot a CREATE TABLE utasításon belül kétféleképpen adhatjuk meg:

Az attribútumot deklarálhatjuk elsődleges kulcsként, amikor az attribútumot felsoroljuk a relációsémában.

4.10. példa:

```
CREATE  TABLE Áruk (
     ÁruID INT PRIMARY KEY,
     ÁruNév CHAR(30),
     MértEgys CHAR(10)
);□
```

Hozzáadhatunk a sémában deklarált elemek listájához (amelyek eddig csak attribútumokat tartalmaztak) egy további deklarációt, amelyi azt adja meg, hogy egy attribútum vagy egy

attribútumhalmaz kulcsot alkot. Ha a kulcs kettő vagy több attribútumból áll, akkor csak ebben a formában adhatjuk meg.

```
CREATE  TABLE Szállít (
     SzállID INT,
     ÁruID INT,
     Ár REAL,
     PRIMARY KEY (SzállID, ÁruID)
); □
```

Egy másik lehetőség kulcsok deklarálására a UNIQUE kulcsszó, mely segítségével egyedi attribútumokat deklarálhatunk. Amint a 3.2.-ben láthattuk, egy relációnak lehet több kulcsjelöltje, ezeket a kucsjelölteket megadhatjuk a UNIQUE kulcsszó segítségével. A kulcsjelöltek közül egyet kiválasztunk elsődleges kulcsnak és azt a PRIMARY KEY kulcsszó segítségével adjuk meg.

4.11. példa: Az Áruk tábla esetén kérhetjük, hogy az ÁruNév értékei egyediek legyenek:

```
ÁruNév CHAR(30) UNIQUE,  □
```

Hasonlóság az elsődleges kulcs és egyedi attribútumok között, hogy mind a két esetben az ABKR-nek van gondja arra, hogy ne lehessen két sor a táblában, amelyben a kulcsként deklarált attribútumérték ugyanaz. Ha az ÁruNév attribútumot egyedinek deklaráltuk, azt jelenti: nem lehet két azonos nevű áru a táblában.

Különbségek a két deklaráció között:

– Elsődleges kulcs csak egy lehet, egyedi attribútum több is lehet.
– Külső kulcs csak elsődleges kulcsára hivatkozhat egy relációnak.
– ABKR-től függően, az elsődleges kulcsnak megfelelően létrehoz az ABKR indexállományt, egyedi attribútum deklarálása esetén az adatbázis adminisztrátor kell az indexállományt létrehozza, a keresés gyorsítása érdekében.

Abban az esetben, ha az ABKR nem hoz létre indexállományt az elsődleges kulcs vagy egyedi attribútumok deklarálásakor, ajánlott annak a létrehozása, mivel minden hozzáillesztés esetén az ABKR megkeresi, hogy az új sor kulcs attribútumának értéke létezik-e már az adatbázisban. Ha az érték már létezik, nem engedi még egyszer hozzáilleszteni, hiba üzenetet küld a felhasználónak. A keresést meg tudjuk gyorsítani az indexállományok segítségével.

4.7.2.2. Külső kulcsok SQL-ben

Külső kulcsot a CREATE TABLE utasításon belül kétféleképpen deklarálhatunk:

Ha a külső kulcs egyetlen attribútum, akkor a név, típus után megadhatjuk, hogy melyik táblára és abból melyik attribútumra hivatkozik:

```
REFERENCES <táblanév> (<attribútum>)
```

4.12. példa: Egészítsük ki a Szállít tábla deklarálását hivatkozási megszorításokkal:

```
CREATE  TABLE Szállít (
   SzállID INT REFERENCES Szállítók (SzállID),
   ÁruID INT REFERENCES Áruk (ÁruID),
   Ár REAL,
   PRIMARY KEY (SzállID, ÁruID)
   ); □
```

Ha a külső kulcs több attribútumból áll, az attribútumok listája után:

```
FOREIGN KEY <attribútumok> REFERENCES <táblanév> (<attribútumok>)
```

ahol az idegen kulcs attribútumait a FOREIGN KEY kulcsszó után soroljuk fel, a REFERENCES kulcsszó után pedig meg kell adjuk a táblát és az attribútumokat, amire a külső kulcs hivatkozik. (Ezeknek az attribútumoknak elsődleges kulcsnak kell lenniük.)

Az előző példát a következőképpen is megadhatjuk:

```
CREATE  TABLE Szállít (
   SzállID INT,
   ÁruID INT,
   Ár REAL,
   PRIMARY KEY (SzállID, ÁruID),
   FOREIGN KEY SzállID REFERENCES Szállítók (SzállID),
   FOREIGN KEY ÁruID REFERENCES Áruk (ÁruID)
   ); □
```

Hivatkozási épség fenntartása: Ha külső kulcsot deklarálunk, azt jelenti, hogy a külső kulcs bármely nem-NULL értéke elő kell hogy forduljon a hivatkozott reláció megfelelő attribútumában. Az ABKR három lehetséges megoldást ajánl az adatbázis tervezőjének, ahhoz, hogy ezt a megszorítást az adatbázis módosításai közben fenn tudja tartani.

1) *Alapértelmezés szerinti eljárás*: ha a feltétel megsérülne a módosítást az ABKR visszautasítja.

4.13. példa: Ha olyan SzállID-t akarunk bevinni a Szállít táblába, mely nincs a Szállítók táblában, nem engedi meg. Ha törölni akarunk egy szállítót a Szállítók táblából, de az szállít valamit, vagyis a Szállít táblában van olyan sor, melyben az illető szállítónak az ID-je szerepel, nem engedi meg a törlést. Hasonlóan a módosítások esetén. □

2) *Továbbgyűrűző eljárás* (CASCADE): a hivatkozott (apa) táblában történő törlés és módosítás esetén alkalmazható. A törölt sorral együtt a hivatkozott (apa) táblából törli a neki megfelelő értékeket a hivatkozó (fiú) táblából. Ha módosítunk egy értéket a hivatkozott (apa) táblában, az ABKR módosítja a hivatkozó (fiú) táblában is a megfelelő értékeket.

4.14. példa: Ha a Szállítók táblából törlünk egy sort, melyre van hivatkozás a Szállít táblából, törli a Szállít táblából is a megfelelő sorokat, vagyis azokat az ajánlatokat, amit az illető cég ajánl. Ha megváltoztatunk egy SzállID-t például 222-t 333-ra a Szállítók táblában, az ABKR megváltoztatja a Szállít táblában azokat a sorokat, ahol 222 volt 333-ra. □

3) *NULL értékre állítás módszere* (SET NULL): a fenti probléma kezelésére, a törölt vagy módosított szállítóhoz tartozó sorokban a Szállít táblában a SzállID értékét NULL-ra változtatja.

4.15. példa:
```
CREATE  TABLE Szállít (
   SzállID INT REFERENCES Szállítók (SzállID)
   ON DELETE SET NULL
   ON UPDATE CASCADE,
   ÁruID INT REFERENCES Áruk (ÁruID)
   ON DELETE SET NULL
   ON UPDATE CASCADE,
   Ár REAL,
   PRIMARY KEY (SzállID, ÁruID)
);
```

Ebben a példában láthatjuk a CREATE TABLE utasítást ON DELETE és ON UPDATE záradékkal kiegészítve. Tehát, ha az apa táblából (Szállítók, illetve Áruk) törlünk egy sort és van rá hivatkozás a Szállít (fiú) táblából, akkor a Szállít táblában a megfelelő (SzállID vagy ÁruID) értékeket NULL értékre állítja az ABKR. Ha a Szállítók táblában módosítunk egy SzállID értéket, és volt rá hivatkozás a Szállít táblából, a ABKR módosítja a SzállID értékét a Szállít táblában. Hasonlóan, ha az ÁruID egy értéke módosul az Áruk táblában, a megfelelő érték változik a Szállít táblában is. □

A 4.5. példa esetén az Alkalmazottak reláció a Részlegek relációra hivatkozott a RészlegID segítségével. A Részlegek relációban a ManagerSzemSzám is egy létező személyi számnak kell lennie az Alkalmazottak relációban, vagyis a Részlegek hivatkozhatna az Alkalmazottak relációra a ManagerSzemSzám segítségével. Ha tervezéskor beállítjuk ezt az egymásra hivatkozást, nem fogunk tudni egyetlen sort sem beilleszteni sem az Alkalmazottak táblába, mert nincs olyan RészlegID még a Részlegek táblában, amelyre lehetne hivatkozni, sem a Részlegek táblába, mert az Alkalmazottak táblában még nincs egy alkalmazott sem, akire managerként hivatkozhatnánk.

Az SQL2 ad erre lehetőséget, úgy, hogy egy tranzakció (lásd 11. Fejezet a tranzakciókról) keretén belül illesztenénk be egy sort a Részlegek táblába és az adott részleg managerét pedig az Alkalmazottak táblába és csak utána ellenőrzi az ABKR a hivatkozási épséget, miután mind a két sort beillesztette a táblákba. Tranzakció végén ellenőrzött megszorítást a DEFERED kulcsszó segítségével adhatjuk meg, a megszorítás deklarálásában. Nem minden ABKR esetén van erre lehetőség.

Egy más megoldás, hogy az egyik tábla esetében engedjünk NULL értéket a külső kulcsnak. Például a Részlegek relációban a ManagerSzemSzám attribútum értéke lehet NULL. Kezdjük a Részlegek feltöltésével, a ManagerSzemSzám értékét nem töltjük fel, majd miután az Alkalmazottak relációba a managereket beillesztettük, visszatérünk a Részlegek relációhoz és módosítjuk a ManagerSzemSzám attribútum értékeket az Alkalmazottak relációban létező SzemSzám attribútum értékekre.

4.7.2.3. Attribútumértékekre vonatkozó megszorítások

Az attribútumértékekre vonatkozó megszorítások a relációséma definálásakor adhatók meg. Egy lehetséges megszorítás, hogy az adott attribútum értéke nem lehet NULL. Ezt a CREATE TABLE parancs keretében az attribútum megadása után, a **NOT NULL** kulcsszó segítségével, adhatjuk meg.

4.16. példa: Az Áruk tábla definícióját kiegészíthetjük azzal a megszorítással, hogy az ÁruNév nem lehet NULL:

```
CREATE  TABLE Áruk (
    ÁruID INT PRIMARY KEY,
    ÁruNév CHAR(30) NOT NULL,
    MértEgys CHAR(10)
      );
```

A megszorítás következményei: nem illeszthetünk olyan sort az Áruk táblába, amelyiknek az ÁruNév mező értéke NULL, nem módosíthatunk létező áru nevet NULL értékre. □

Egyedi kulcs értéke sohasem lehet NULL, mivel így elveszítené azonosító szerepét. Az ABKR-ek általában nem engedik meg, hogy egyedi kulcsnak deklaráljunk olyan attribútumot, mely értéke lehet NULL.

Bonyolultabb megszorítás rendelhető egy attribútumhoz a **CHECK** kulcsszó segítségével. A CHECK kulcsszó után megadhatjuk a megengedett értékeket, illetve egy feltételt. A feltétel a SELECT SQL parancs WHERE feltételéhez hasonló. Ha a feltételben egy másik relációra is hivatkozni akarunk, akkor azt egy alkérdés FROM záradékában kell megadnunk.

4.17. példa: Az 1.7. példát kiegészítve, az Evfolyam megengedett értékei 1 és 5 közötti egészek.

```
CREATE  TABLE  Csoportok (
      CsopKod CHAR(3) PRIMARY KEY,
      Evfolyam INT CHECK (Evfolyam >= 1 and Evfolyam <= 5),
      SzakKod CHAR(3)
      );
```

vagy, csak a kérdéses sort megismételve:

```
Evfolyam INT CHECK (Evfolyam IN (1, 2, 3, 4, 5)),
```

így az ABKR visszautasítja azokat a hozzáillesztéseket, módosításokat, ahol ez a feltétel nem áll fenn. □

4.7.2.4. Értéktartományokra vonatkozó megszorítások

Egy attribútum értékeit úgy is korlátozhatjuk, hogy deklarálunk egy értéktartományt, és ezt az értéktartományt adjuk meg az attribútum típusaként.

4.18. példa: Az Evfolyam lehetséges értékeit megadhatjuk az EvfolyamErtekek nevű értéktartomány segítségével:

```
CREATE DOMAIN EvfolyamErtekek INT
          CHECK (VALUE >= 1 and VALUE <= 5);
```

A CREATE TABLE-ben a megfelelő sor:

```
Evfolyam EvfolyamErtekek, □
```

4.7.2.5. Globális megszorítások

Egy megszorítás hivatkozhat több attribútumra is, és akár több relációt érintő feltételeket is tartalmazhat. Két csoportra oszthatjuk a globális megszorításokat:
1) Sorra vonatkozó CHECK feltételek, amelyek egyetlen reláció soraira tesznek megszorításokat.

4.19. példa: A 4.5 példa esetén kikötjük, hogy egy manager fizetése legalább 500 euró kell legyen. A Részlegek tábla deklarálásában adhatjuk meg, tehát egy tábla esetén, de a feltételben szerepelhet más tábla is, a mi esetünkben az Alkalmazottak tábla.

```
CREATE  TABLE Alkalmazottak (
     SzemSzám INT PRIMARY KEY,
     Név CHAR(30),
     RészlegID INT REFERENCES Részlegek (RészlegID),
     Fizetés INT
     );
CREATE  TABLE Részlegek (
     RészlegID INT PRIMARY KEY,
     RNév CHAR(30),
     ManagerSzemSzám INT REFERENCES Alkalmazottak (SzemSzám),
     CHECK (ManagerSzemSzám NOT IN
          (SELECT SzemSzám FROM Alkalmazottak
               WHERE Fizetés < 500)
          )
     ); □
```

2) Önálló megszorítások, amelyek a feltételükben teljes relációkat vagy ugyanazt a relációt befutó több sorváltozót is tartalmazhatnak. Az eddigi megszorításokat a tábla deklarálásában adtuk meg, az önálló megszorításokat a CREATE ASSERTION utasítás segítségével adhatjuk meg és különálló sémaelemek lesznek.

4.20. példa: A fenti 4.19. példát megadhatjuk önálló megszorításként is a következőképpen:

```
CREATE ASSERTION ManagerFizetes CHECK
          (NOT EXISTS
               (SELECT * FROM Alkalmazottak, Részlegek
                WHERE SzemSzám = ManagerSzemSzám
                    AND Fizetés < 500)
          ); □
```

A megszorítások nagyon komplexek lehetnek.

4.7.2.6. Megszorítások módosítása

Ahhoz, hogy módosítani, törölni tudjunk egy megszorítást, nevet kell adjunk neki. A CREATE ASSERTION esetén az utasítás részeként adtuk meg a nevet. A többi megszorítás esetén a CONTRAINT kulcsszó segítségével adhatunk neki nevet.

A 4.19. példa Részlegek tábla elsődleges kulcs megszorítását (PK_Reszleg a megszorítás neve) a következőképpen is megadhatjuk:

```
RészlegID INT CONSTRAINT PK_Reszleg PRIMARY KEY,
```

Attribútumra vonatkozó megszorításnak is adhatunk nevet, a 4.17 példa esetén:

```
 Evfolyam INT CONSTRAINT EgyOt
       CHECK (Evfolyam >= 1 and Evfolyam <= 5),
```

Táblához rendelt megszorításokat az ALTER TABLE utasítás segítségével, azon belül a DROP CONSTRAINT kulcsszóval törölhetjük. A fenti példa esetén:

```
ALTER TABLE Csoportok DROP CONSTRAINT EgyOt;
```

Táblához rendelhetünk plusz megszorításokat, a tábla deklaráció után is az ALTER TABLE utasítással, ADD CONSTRAINT kulcsszó segítségével.

```
ALTER TABLE Csoportok ADD CONSTRAINT EgyOt
     CHECK (Evfolyam >= 1 and Evfolyam <= 5);
```

Hasonlóan, ajánlott nevet adni a hivatkozási épség megszorításoknak is, hogy törölhessük őket, esetleg újakat értelmezhessünk, anélkül, hogy a tábladeklarációt megváltoztatnánk.

Az önálló megszorításokat a DROP ASSERTION utasítással törölhetjük.

5. Műveletek a relációs modellben

A felhasználónak szinte állandó jelleggel szüksége van az adatbázisban eltárolt adatok egy részére. Megfogalmaz egy kérést, amelyben leírja, milyen adatokra van szüksége. Egy ilyen kérést az adatbázis nyelvezetében *lekérdezés*nek nevezzük. A relációs adatmodell egy olyan szabványos adatműveletekből álló halmazt tartalmaz, amelyek segítségével kifejezhetjük a lekérdezéseket. Ezen műveletek kifejezésére kétféle jelölés használatos:

– relációs algebra, mely algebrai jelölést használ, a lekérdezéseket algebrai operátorok segítségével adja meg;
– relációs kalkulus, mely matematikai logikán alapul, a lekérdezést logikai formulák segítségével adja meg.

A relációs algebra és a relációs kalkulus ekvivalens: egy relációs algebrai lekérdezés átalakítható egy relációs kalkulusbeli lekérdezéssé és fordítva (lásd [Ul89]).

5.1. Relációs algebra

A relációs algebrai műveletek operandusai a relációk. A relációt a nevével szokták megadni, például *R* vagy Alkalmazottak. A műveletek operátorait a következőkben részletezzük. Az operátorokat alkalmazva a relációkra, eredményként szintén relációkat kapunk, ezekre ismét alkalmazhatunk relációs algebrai operátorokat, így egyre bonyolultabb kifejezésekhez jutunk. Egy lekérdezés tulajdonképpen egy relációs algebrai kifejezés. A relációs algebrai műveletek esetén szükségünk lesz feltételekre. A feltételek a következő típusúak lehetnek:

$$\text{<attribútum\_név>} \left\{ \begin{array}{c} = \\ <> \\ < \\ <= \\ > \\ >= \end{array} \right\} \left\{ \begin{array}{c} \text{<attribútum\_név>} \\ \text{<konstans>} \end{array} \right\}$$

$$\left\{ \begin{array}{c} \text{<attribútum\_név>} \\ \text{<konstans>} \end{array} \right\} \left\{ \begin{array}{c} \text{IS IN} \\ \text{IS NOT IN} \end{array} \right\} \text{<reláció> (melynek egy attribútuma van)}$$

NOT <feltétel>

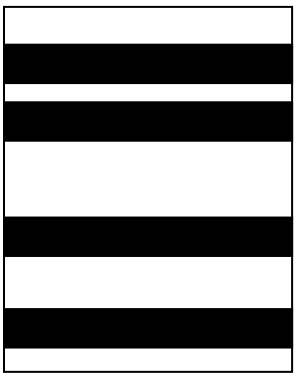
$$\text{<feltétel>} \left\{ \begin{array}{c} \text{OR} \\ \text{AND} \end{array} \right\} \text{<feltétel>}$$

A továbbiaban lássuk a relációs algebra műveleteit. Az első öt az alapvető művelet, a következőket ki tudjuk fejezni az első öt segítségével.

1) *Kiválasztás (Selection)*: Az *R* relációra alkalmazott *kiválasztás* operátor *f* feltétellel olyan új relációt hoz létre, melynek sorai teljesítik az *f* feltételt. Az eredmény reláció attribútumainak a száma megegyezik az *R* reláció attribútumainak a számával. Jelölés:

$$\sigma_f(R).$$

Grafikusan ábrázolva, ha az *R* reláció a nagy téglalap, a kiválasztás eredménye a befestett rész.



5.1. példa: A 4.19 példa esetén keressük a kis keresetű alkalmazottakat (akinek kisebb, vagy egyenlő a fizetése 500 euró-val). A lekérdezés a következő:

$\sigma_{\text{Fizetés <= 500}}$ (Alkalmazottak)

A lekérdezés eredménye:

SzemSzám	***Név***	***RészlegID***	***Fizetés***
111111	Nagy Éva	2	300
222222	Kiss Csaba	9	400
333333	Kovács István	2	500

5.2. példa: Keressük a 9-es részleg nagy fizetésű alkalmazottait (akinek 500 euró-nál nagyobb a fizetése). A lekérdezés:

$\sigma_{\text{Fizetés > 500 AND RészlegID = 9}}$ (Alkalmazottak)

Az eredmény:

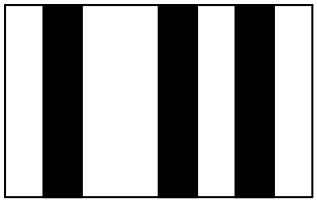
SzemSzám	***Név***	***RészlegID***	***Fizetés***
456777	Szabó János	9	900

2) *Vetítés (Projection)*: Adott *R* egy reláció A_1, A_2,..., A_n attribútumokkal. A vetítés művelet eredményeként olyan relációt kapunk, mely *R*-nek csak bizonyos attribútumait tartalmazza. Ha kiválasztunk *k* attribútumot az *n*-ből: $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ -et, és ha esetleg a sorrendet is megváltoztatjuk, az eredmény reláció a kiválasztott *k* attribútumhoz tartozó oszlopokat fogja tartalmazni, viszont az összes sorból. Mivel az eredmény is egy reláció, nem lehet két azonos sor a vetítés eredményében, az azonos sorokból csak egy marad az eredmény relációban.

Jelölés:

$\pi_{A_{i_1}, A_{i_2}, \ldots, A_{i_k}}(R)$

Grafikusan ábrázolva, ha az *R* reláció a nagy téglalap, a vetítés eredménye a befestett rész.



5.3. példa: Ha az Alkalmazottak relációból csak az alkalmazott neve és fizetése érdekel, akkor a következő művelet eredménye a kért reláció:

$$\pi_{\text{Név, Fizetés}}(\text{Alkalmazottak})$$

Az eredmény:

Név	***Fizetés (euró)***
Nagy Éva	300
Kiss Csaba	400
Szabó János	900
Szilágyi Pál	700
Vincze Ildikó	800
Kovács István	500

5.4. példa: Legyen ismét a Diákok tábla az 1.7 példából:

```
CREATE TABLE Diákok (
       BeiktatásiSzám INT PRIMARY KEY,
       Név VARCHAR(50),
       Cím VARCHAR(100),
       SzületésiDatum DATE,
       CsopKod CHAR(3) REFERENCES Csoportok (CsopKod),
       Átlag REAL
);
```

A következő vetítés:

$$\pi_{\text{CsopKod}}(\text{Diákok})$$

eredménye az összes létező csoportkod a Diákok táblából. Ha egy csoportkod többször is megjelenik a Diákok táblában, a vetítésben csak egyszer fog szerepelni. (Például a Diákok táblában 25 sor esetén a csoportkod ’531’-es, a vetítés eredményében csak egyszer fog az ’531’-es csoportkod szerepelni.)

3) *Descartes szorzat*. Ha adottak az R_1 és R_2 relációk, a két reláció Descartes szorzata ($R_1 \times R_2$) azon párok halmaza, amelyeknek első eleme az R_1 tetszőleges eleme, a második pedig az R_2 egy eleme. Az eredményreláció sémája az R_1 és R_2 sémájának egyesítése.

Legyen R_1 reláció:

A	***B***
12	33
24	46

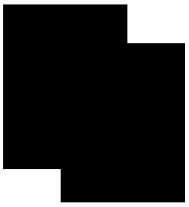
Legyen R_2 reláció:

B	*C*	*D*
20	55	80
30	67	97
40	75	99

Akkor $R_1 \times R_2$ eredménye:

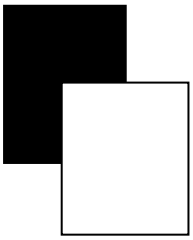
A	$R_1.B$	$R_2.B$	*C*	*D*
12	33	20	55	80
12	33	30	67	97
12	33	40	75	99
24	46	20	55	80
24	46	30	67	97
24	46	40	75	99

4) *Egyesítés*. Ha adottak az R_1 és R_2 relációk, R_1 és R_2 attribútumainak a száma megegyezik, és ugyanabban a pozícióban levő attribútumnak ugyanaz az értékhalmaza, a két reláció egyesítése tartalmazni fogja R_1 és R_2 sorait. Az egyesítésben egy elem csak egyszer szerepel, még akkor is, ha jelen van R_1– és R_2 –ben is (jelölés: R_1 U R_2). Grafikusan ábrázolva az egyesítés eredményét:



5) *Különbség*. Ha adottak az R_1 és R_2 relációk, R_1 és R_2 attribútumainak a száma megegyezik és ugyanabban a pozícióban levő attribútumnak ugyanaz az értékhalmaza, a két reláció különbsége azon sorok halmaza, amelyek R_1-ben szerepelnek és R_2-ben nem (jelölés: $R_1 - R_2$). A különbség eredményét grafikusan ábrázolva:

5.5. példa: Legyen R_1:



SzemSzám	*Név*	*RészlegID*	*Fizetés (euró)*
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500

és legyen R_2:

SzemSzám	*Név*	*RészlegID*	*Fizetés (euró)*
111111	Nagy Éva	2	300
456777	Szabó János	9	900
123444	Vincze Ildikó	1	800

akkor R_1 U R_2:

SzemSzám	*Név*	*RészlegID*	*Fizetés (euró)*
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500
111111	Nagy Éva	2	300
123444	Vincze Ildikó	1	800

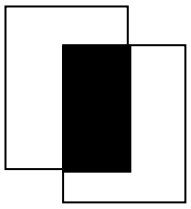
illetve R_1 - R_2:

SzemSzám	*Név*	*RészlegID*	*Fizetés (euró)*
222222	Kiss Csaba	9	400
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500

Ez az öt az alapvető művelet. Még vannak hasznos műveletek: ezek az öt alapvető művelettel kifejezhetőek.

6) *Metszet:* Legyenek az R_1 és R_2 relációk, a két reláció metszete:

$R_1 \cap R_2 = R_1 - (R_1 - R_2)$. Grafikusan jelölve:



7) *Théta-összekapcsolás* (θ-Join): Legyenek az R_1 és R_2 relációk. A Théta-összekapcsolás során az R_1 és R_2 relációk Descartes szorzatából kiválasztjuk azon sorokat, melyek eleget tesznek a θ feltételnek, vagyis: $R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$.

5.6. példa: Legyenek R_1 és R_2 a következő relációk, számítsuk ki: $R_1 \bowtie_{A<D} R_2$

R_1 reláció:

A	*B*	*C*
11	23	32
65	76	82
97	76	82

R_2 reláció:

B	*C*	*D*
23	32	44
23	32	57
76	82	99

$R_1 \bowtie_{A<D} R_2$:

A	$R_1.B$	$R_1.C$	$R_2.B$	$R_2.C$	*D*
11	23	32	23	32	44
11	23	32	23	32	57
11	23	32	76	82	99
65	76	82	76	82	99
97	76	82	76	82	99

8) *Természetes összekapcsolás (Natural join)*: Legyenek az R_1 és R_2 relációk. A természetes összekapcsolás művelete akkor alkalmazható, ha az R_1 és R_2 relációknak egy vagy több közös attribútuma van. Legyen B az R_1, illetve C az R_2 reláció attribútumainak a halmaza, a közös attribútumok pedig: $B \cap C = \{A_1, A_2, \ldots, A_p\}$. A természetes összekapcsolást a következő képlettel fejezhetjük ki:

$$R_1 \bowtie R_2 = \pi_{B \cup C}(R_1 \bowtie_{(R_1.A_1=R_2.A_1)\wedge(R_1.A_2=R_2.A_2)\wedge\ldots\wedge(R_1.A_p=R_2.A_p)} R_2,$$

ahol $R_i.A_j$ jelöli az A_j attribútumot az R_i relációból, $i \in \{1,2\}, j \in \{1,2, \ldots, p\}$.

5.7. példa: Legyenek R_1 és R_2 relációk a Théta-összekapcsolás példából, a természetes összekapcsolás eredménye:

$R_1 \bowtie R_2$ eredménye:

A	*B*	*C*	*D*
11	23	32	44
11	23	32	57
65	76	82	99
97	76	82	99

R_1 és R_2 relációk természetes összekapcsolása esetén azokat a sorokat párosítjuk össze, amelyek értékei az R_1 és R_2 sémájának összes közös attribútumán megegyeznek. Legyen r_1 az R_1 egy sora és r_2 az R_2 egy sora, ekkor az r_1 és r_2 párosítása akkor sikeres, ha az r_1 és r_2 megfelelő értékei megegyeznek az összes $A_1, A_2, \ldots, A_p$ közös attribútumon. Ha az r_1 és r_2 sorok párosítása sikeres, akkor a párosítás eredményét *összekapcsolt sor*nak nevezzük. Az összekapcsolt sor megegyezik az r_1 sorral az R_1 összes attribútumán és r_2 sorral az R_2 összes attribútumán. Az $R_1 \bowtie R_2$ eredményében R_1 és R_2 közös attribútumai csak egyszer szerepelnek.

Egy olyan sort, melyet nem lehet sikeresen párosítani az összekapcsolásban szereplő másik reláció egyetlen sorával sem, *lógó* (dangling) sornak nevezzünk

5.8. példa: Legyenek a Szállítók, Áruk és Szállít relációk a 3.12. példából (2NF-től). Ha az összes szállítási információra van szükségünk, akkor kiszámítjuk a Szállít ⋈ Szállítók ⋈ Áruk természetes összekapcsolást, melynek eredménye:

Szállítók:

SzállID	*SzállNév*	*SzállCím*
111	Rolicom	A.Iancu 15
222	Sorex	22 dec. 6

Áruk:

ÁruID	*ÁruNév*	*MértEgys*
45	Milka csoki	tábla
67	Heidi csoki	tábla
56	Milky way	rúd

Szállít:

SzállID	*ÁruID*	*Ár*
111	45	25000
222	45	26500
111	67	17000
111	56	20000
222	67	18000
222	56	22500

Szállít ⋈ Szállítók ⋈ Áruk eredménye:

SzállID	*SzállNév*	*SzállCím*	*ÁruID*	*ÁruNév*	*MértEgys*	*Ár*
111	Rolicom	A.Iancu 15	45	Milka csoki	Tábla	25000
222	Sorex	22 dec. 6	45	Milka csoki	Tábla	26500
111	Rolicom	A.Iancu 15	67	Heidi csoki	Tábla	17000
111	Rolicom	A.Iancu 15	56	Milky way	Rúd	20000
222	Sorex	22 dec. 6	67	Heidi csoki	Tábla	18000
222	Sorex	22 dec. 6	56	Milky way	Rúd	22500

Relációs algebrai műveletek alkalmazásával újabb relációkat kapunk. Gyakran szükséges egy olyan operátor, amelyik átnevezi a relációkat.

9) *Átnevezés:* Legyen $R(A_1, A_2, \ldots, A_n)$ egy reláció, az átnevezés operátor:

$\rho_{S(B_1,B_2,\ldots,B_n)}(R)$ az R relációt S relációvá nevezi át, az attribútumokat pedig balról jobbra $B_1, B_2, \ldots, B_n$-né. Ha az attribútum neveket nem akarjuk megváltoztatni, akkor $\rho_S(R)$ operátort használunk.

10) *Hányados (Quotient)*: Legyen R_1 reláció sémája: $\{X_1, X_2,\ldots, X_m, Y_1,Y_2,\ldots,Y_n\}$, R_2 reláció sémája pedig: $\{Y_1, Y_2, \ldots, Y_n\}$, tehát $Y_1, Y_2, \ldots,Y_n$ közös attribútumok ugyanazon értékhalmazzal, és R_1-nek még van pluszba m attribútuma: $X_1, X_2,\ldots, X_m$, R_2-nek pedig a közöseken kívül nincs más attribútuma. R_1 az osztandó, R_2 az osztó. Jelöljük X-szel és Y-nal a következő attribútumhalmazokat: $X = \{X_1, X_2,\ldots, X_m\}$, $Y = \{Y_1,Y_2,\ldots,Y_n\}$. Ebben az esetben jelöljük: R_1 (X, Y), R_2 (Y) a két relációt, melynek hányadosát jelöljük:

R_1 DIVIDE BY R_2 (X)-el

Tehát a hányados reláció sémája $\{X_1, X_2,\ldots, X_m\}$. A hányados relációban megjelenik egy x sor, ha minden y sorra az R_2-ből az R_1-ben megjelenik egy r_1 sor, melyet az x és y sorok összeragasztásából kapunk.

Másként fogalmazva, legyen 2 reláció, egy bináris és egy unáris, az osztás eredménye a bináris reláció azon attribútumait tartalmazza, melyek különböznek az unáris reláció attribútumaitól, és a bináris relációból az attribútumok azon értékeit, melyek megegyeznek az unáris reláció összes attribútum értékével.

5.9. példa: Legyen A = $\pi_{\text{ÁruID}}(\text{Áruk})$, S = $\pi_{\text{SzállID, ÁruID}}(\text{Szállít})$ és a következő sorok az S relációban:

SzállID	*ÁruID*
S1	A1
S1	A2
S1	A3
S1	A4
S1	A5
S1	A6
S2	A1
S2	A2
S3	A2
S4	A2
S4	A4
S4	A5

a) Legyen A reláció:

ÁruID
A1

akkor az S DIVIDE A(SzállID) eredménye:

SzállID
S1
S2

b) esetben A reláció:

ÁruID
A2
A4

akkor S DIVIDE A(SzállID):

SzállID
S1
S4

c) esetben A reláció:

ÁruID
A1
A2
A3
A4
A5
A6

akkor S DIVIDE A(SzállID):

SzállID
S1

5.2. Lekérdezések megfogalmazása relációs algebrai műveletek segítségével

A relációs algebra segítségével tetszőleges bonyolultságú kifejezéseket képezhetünk. Az operátorokat alkalmazhatjuk adott relációkra, illetve más operátorok alkalmazásának eredményeként kapott relációkra. A relációs algebrai műveletek megfogalmazásakor zárójeleket használhatunk az operándusok csoportosítása érdekében. A relációs algebrai kifejezéseket megadhatjuk kifejezésfával is (lásd 5.1. ábrát).

5.10. példa: Legyenek a Szállítók, Áruk, Szállít relációk a 5.8. példából és a következő lekérdezés: „Keressük a 'Milka csoki'-t szállító cégek nevét." A választ sokféleképpen is megadhatjuk, egy lehetséges megoldás:

A: lépésekre felbontva:

Megkeressük a 'Milka csoki' áru ÁruID-ját, az eredmény a MCsokiIDk relációban található:

$$\text{MCsokiIDk} = \pi_{\text{ÁruID}}(\sigma_{\text{ÁruNév='Milka csoki'}}(\text{Áruk}))$$

A mi példánk esetében csak egy sort fog ez a reláció tartalmazni, de előfordulhat, hogy több áru esetén a név 'Milka csoki' és akkor a MCsokiIDk reláció több sort is tartalmaz.

Válasszuk ki azon szállítási ajánlatokat, melyek esetén az ÁruID benne van a MCsokiIDk halmazban.

$$\text{MCsokiAjánlatok} = \sigma_{\text{ÁruID IN MCsokiIDk}}(\text{Szállít})$$

A MCsokiAjánlatok relációban csak a szállítók ID-ját kaptuk meg, ahhoz, hogy a nevüket is meg tudjuk adni a következő műveleteket kell elvégezzük:

$$\text{MCsokiSzállítóIDk} = \pi_{\text{SzállID}}(\text{MCsokiAjánlatok})$$

$$\text{MCsokiSzállítóNevek} = \pi_{\text{SzállNév}}(\sigma_{\text{SzállID IN MCsokiSzállítóIDk}}(\text{Szállítók}))$$

A feladatot a következőképpen is megoldhatjuk:

B: $\pi_{\text{SzállNév}}(\sigma_{\text{ÁruNév='Milka csoki'}}(\text{Szállít} \bowtie \text{Szállítók} \bowtie \text{Áruk}))$

5.11. példa: „Keressük azon szállítókat, akik nem szállítják a 67-es ID-jű árut".

Lépesről lépésre oldjuk, hogy érthetőbb legyen:

$$\text{Száll67} = \pi_{\text{SzállID}}(\sigma_{\text{ÁruID }=67}(\text{Szállít}))$$

A Száll67 tartalmazza azon szállítók ID-ját, akik szállítják a 67-es ID-jű árut.

$$\text{NemSzáll67} = \pi_{\text{SzállID}}(\text{Szállítók}) - \text{Száll67}$$

A NemSzáll67 tartalmazza azon szállítók ID-ját, akik szerepelnek a szállítók között és nem szállítják a 67-es ID-jű árut. Ugyanezt a relációt meghatározhattuk volna a következőképpen is:

$$\text{Nem2Száll67} = \pi_{\text{SzállID}}(\sigma_{\text{ÁruID }<>67}(\text{Szállít}))$$

A Nem2Száll67 reláció azon szállítókat tartalmazza, akik szállítanak árukat, de azon áruk között nem szerepel a 67-es. Azon szállítók, akik nem szállítanak semmit, nem fognak megjelenni ebben a relációban, míg a NemSzáll67-ben igen. Kérdés: pont melyikre van szükségünk a konkrét feladat esetén.

Ahhoz, hogy megkapjuk a szállító nevét:

$$\pi_{\text{SzállNév}}(\text{NemSzáll67} \bowtie \text{Szállítók})$$

5.12. példa: „Keressük azon szállítókat, kik szállítják az összes árut."

$$\pi_{\text{SzállNév}}(((\pi_{\text{SzállID, ÁruID}}(\text{Szállít})) \text{ DIVIDE BY } (\pi_{\text{ÁruID}}(\text{Áruk}))) \bowtie \text{Szállítók})$$

Első lépésben szükségünk van egy bináris és egy unáris relációra, hogy tudjuk az osztás műveletét alkalmazni. A bináris reláció:

$$\pi_{\text{SzállID, ÁruID}}(\text{Szállít})$$

az unáris pedig:

$$\pi_{\text{ÁruID}}(\text{Áruk}).$$

Az osztás eredménye tartalmazni fogja az összes szállító ID-jét, vagyis akik az összes árut szállítják. Ahhoz, hogy a szállító nevét megkapjuk, join műveletet alkalmazunk a Szállítók relációval, majd vetítést a szállító nevére.

5.13. példa: „Keressük azon szállítókat, akik legalább azon árukat szállítják, melyeket az 111 ID-jú szállító szállít."

$$\pi_{\text{SzállNév}}(\ ((\pi_{\text{SzállID, ÁruID}}(\text{Szállít}))\ \text{DIVIDE BY}\ (\pi_{\text{ÁruID}}(\sigma_{\text{SzállID=111}}(\text{Szállít}))) \bowtie \text{Szállítók})$$

Először megkeressük azon áru ID-kat, melyeket szállít a 111-es ID-jű szállító:

$$\pi_{\text{ÁruID}}(\sigma_{\text{SzállID=111}}(\text{Szállít}))$$

Az osztás segítségével meghatározzuk azon szállítokat, kik szállítják legalább az előző lekérdezésben megkapott árukat. A join és vetítés a szállító nevének a meghatározására szükséges.

5.14. példa: „Keressük azon szállítókat, akik csak a 67-es ID-jű árut szállítják."

$$\pi_{\text{SzállNév}}(\ ((\pi_{\text{SzállID}}(\sigma_{\text{ÁruID=67}}(\text{Szállít}))) - (\pi_{\text{SzállID}}(\sigma_{\text{ÁruID<>67}}(\text{Szállít}))) \bowtie \text{Szállítók})$$

Először megkeressük azokat a szállítókat, akik a 67-es ID-jű árut szállítják. Ezek között azok is szerepelnek, akik a 67-es ID-jű árun kívül még más árukat is szállítanak. A következő művelet

$$\pi_{\text{SzállID}}(\sigma_{\text{ÁruID<>67}}(\text{Szállít}))$$

megadja azokat a szállítókat, akik a 67-esen kívül akármi más árut szállítanak. Ezek között a szállítók között szerepelnek azok, akik a 67-est és mást is szállítanak és azok, akik nem szállítják a 67-es árut, de szállítanak mást. A különbség segítségével kiküszöböljük azokat a szállítókat, akik a 67-est és mást is szállítanak. Azok, akik nem szállítják a 67-est, de mást igen a különbség eredményében nem fognak szerepelni. Ebben azok a szállítók fognak szerepelni, akik csak a 67-est szállítják. A join és vetítés a szállító nevének a meghatározására szükséges.

Lekérdezések optimalizálása: Minden ABKR-nek van lekérdezés−feldolgozó rendszere, mely a lekérdezést relációs algebrai műveletek sorozatává alakítja. Egy lekérdezést több relációs algebrai műveletek sorozatává is alakíthatjuk, amelyek ugyanazt az eredményt adják, ezeket *ekvivalens* kifejezéseknek nevezzük. A lekérdezés optimalizáló feladata, hogy az ekvivalens kifejezések közül kiválassza a leggyorsabban kiértékelhető kifejezést. A relációs algebrai műveletek tulajdonságait felhasználva a kifejezéseket átalakíthatjuk.

5.3. Relációs algebrai műveletek algebrai tulajdonságai

Az ekvivalenciát ezzel a $\leftrightarrow$-lel jelöljük. Legyenek R, S és T relációk, ahol R reláció sémája $A = \{A_1, A_2, ..., A_n\}$, S-nek $B = \{B_1, B_2, \ldots, B_m\}$ és T-nek $C = \{C_1, C_2, \ldots, C_k\}$, ahol $n, m, k \in N$ az attribútumok száma.

– Join kommutatív:

$$R \bowtie S \leftrightarrow S \bowtie R$$

– Bináris műveletek asszociatívak:

$$(R \times S) \times T \leftrightarrow R \times (S \times T)$$

$$(R \bowtie S) \bowtie T \leftrightarrow R \bowtie (S \bowtie T)$$

– Unáris műveletek idempotensek:

- Ha ugyanarra a relációra több vetítést alkalmazunk, ezeket csoportosíthatjuk, ha $A' \subseteq A, A'' \subseteq A$ şi $A' \subseteq A''$, akkor:

$$\pi_{A'}(\pi_{A''}(R)) \leftrightarrow \pi_{A'}(R)$$

- Ha több kiválasztást ($\sigma_{p_i(A_i)}$) ugyanarra a relációra vonatkozik, ezeket csoportosíthatjuk, ahol p_i az A_i attribútumra alkalmazott feltétel:

$$\sigma_{p_1(A_1)}(\sigma_{p_2(A_2)}(R)) \leftrightarrow \sigma_{p_1(A_1) \wedge p_2(A_2)}(R)$$

– Vetítés és kiválasztás sorrendje felcserélhető:

$$\pi_{A_1,...,A_n}(\sigma_{p(A_p)}(R)) \rightarrow \pi_{A_1,...,A_n}(\sigma_{p(A_p)}(\pi_{A_1,...,A_n,A_p}(R)))$$

Tehát, ha a vetítést a kiválasztás előtt akarjuk végrehajtani, akkor az A_p attribútumnak kell szerepelnie a vetítés attribútumai között. Ha $A_p \in \{A_1,..., A_n\}$, akkor az utolsó vetítés az $\{A_1,..., A_n\}$ attribútumokra jobb oldalon fölösleges.

– Kiválasztás és bináris műveletek sorrendje felcserélhető:

(Emlékeztető: A_i az R reláció attribútuma).

$$\sigma_{p(A_i)}(R \times S) \rightarrow (\sigma_{p(A_i)}(R)) \times S$$

$$\sigma_{p(A_i)}(R \bowtie_{p(A_j,B_k)} S) \rightarrow \sigma_{p(A_i)}(R) \bowtie_{p(A_j,B_k)} S$$

– A kiválasztás és egyesítés sorrendje felcserélhető, ha az R és T relációk sémája ugyanaz:

$$\sigma_{p(A_i)}(R \cup T) \rightarrow \sigma_{p(A_i)}(R) \cup \sigma_{p(A_i)}(T)$$

– Vetítés és bináris műveletek sorrendje felcserélhető:

Legyen $A' \subseteq A$, $B' \subseteq B$, $C = A' \cup B'$, akkor:

$$\pi_C(R \times S) \rightarrow \pi_{A'}(R) \times \pi_{B'}(S)$$

$$\pi_C(R \bowtie_{p(A_i,B_j)} S) \rightarrow \pi_{A'}(R) \bowtie_{p(A_i,B_j)} \pi_{B'}(S)$$

ahol $A_i \in A'$ és $B_j \in B'$.

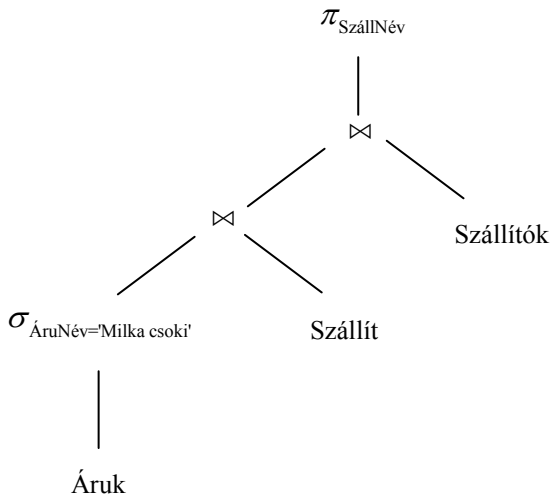
$$\pi_C(R \cup S) \rightarrow \pi_C(R) \cup \pi_C(S)$$

A gyakorlatban a Descartes szorzatot nem célszerű használni, mivel ez a legköltségesebb művelet. A természetes összekapcsolás is drága művelet, a gyakran használatosak közül a legdrágább. Az ABKR lekérdezés optimalizálója a join-t nem úgy végzi, hogy Descartes szorzatot elkészíti és abból kiválogatja az összepárosítható sorokat. Több algoritmus is létezik a join végrehajtására, egyik az ún. *merge-join*, mely a közös attribútum szerint rendezett összekapcsolandó relációt egy új relációba fésüli össze (láss join algoritmusokat az [ÖzVa91]-ben).

5.15. példa: A 5.10 példa egy más megoldása, felhasználva a fenti tulajdonságokat:

C: $\pi_{\text{SzállNév}}(\sigma_{\text{ÁruNév='Milka csoki'}}(\text{Áruk}) \bowtie \text{Szállít} \bowtie \text{Szállítók})$

A relációs algebrai kifejezéseket kifejezésfával is megadhatjuk. Az előbbi relációs algebrai kifejezés kifejezésfája az 5.1 ábrán látható. A lekérdezés végrehajtását, illetve optimalizálását lásd részletesebben a [MoUlWi00]-ben.



5.1. ábra. Relációs algebrai kifejezés kifejezésfája

5.4. Gyakorlatok

5.1. Legyenek a következő táblák egy relációs adatbázisból, mely egy könyvtár részleges információit tárolja:

```
Kiadók (KiadóKod, Knév, Khelység, Kcím, Ktelefon)
Könyvtípusok (TípusID, Név)
(lehetséges értékek: villamosság, informatika, elektronika, ...)
Könyvek (KönyvKod, Cím, Megjelenési-év, KiadóKod, TípusID,
     PéldánySzám, Ár)
Szerzők (SzerzőID, SzerzőNev)
KönyvSzerző(KönyvKod, SzerzőID)
```

Oldjuk meg relációs algebrai műveletek segítségével:

a. Adjuk meg a Teora nevű kiadó telefon számát, címét!
b. Adjuk meg a könyvtárban található könyvek közül azoknak a címét és a megjelenési évét, melyek szerzői között szerepel J. D. Ullman.
c. Adjuk meg a villamossági könyvek címét és szerzőiknek nevét!
d. A Teora nevű kiadó által kiadott informatika könyvek címét és a megjenelési évet.
e. Az „Adatbázisrendszerek megvalósítása" című könyv szerzőinek a nevét.
f. Adjuk meg azon ifjúsági könyvek címét, amelyekből van az üzletben legalább 1 darab!
g. Azon Kiadók neve, akiktől mindenféle könyvtípusból van a könyvtárban könyv (a Könytípusok-ban létező összes fajta könyvből)

5.2. Legyenek a következő táblák egy relációs adatbázisból, mely egy egyetem részleges információit tárolja:

```
Karok (KarKod, KarNév)
Tanszékek (TanszékKod, Nev, KarKod);
Beosztások (BeosztásKod, Nev);
     (lehetséges értékek: PRO – professzor, DOC – docens,
        ADJ – adjunktus, TNS – tanársegéd, GYA – gyakornok)
Tanárok (TanárKod, Nev, SzemSzám, Cím, PhD, TanszékKod,
        BeosztásKod, Fizetés);
Tantárgyak (TantKod, Nev, KreditSzám, TanszékKod);
Tanít (TanárKod, TantKod);
```

(Ha tanárra vonatkozik a lekérdezés, akkor annak a beosztása akármi lehet, mindenki tanár, a professzor is, a gyakornok is.)

Relációs algebrai műveletek segítségével adjuk meg:

a. Adjuk meg a Dávid László nevű tanár címét és fizetését!
b. A Kémia karon tanító docensek nevét.
c. A Villamosság nevű tanszéken tanító professzorok nevét.
d. Adatbázis nevű tantárgyat tanító tanárok nevét.
e. Azon tanárok nevét, akik tanítanak Valószínűség számítást és nem tanítanak Statisztikát!
f. Adjuk meg az Analízis nevű tanszék tanárai által tanított tantárgyak nevét!
g. Adjuk meg azoknak a tanároknak a nevét, akik csak Algebra nevű tantárgyat tanítanak.

5.3. Adott a következő relációs modellben leírt adatbázis:

```
ÁruCsoportok (ÁruCsopKod, Név);
Anyagok (AnyagKod, Név),
Kinek (KinekKod, Név)
Gyártók (GyártóID, Név, Cím);
Üzletek (ÜzletID, Név, Cím, TelefonSzám)
Modellek (ModellID, Név, ÁruCsopKod, KinekKod, FelsőrészAnyagKod,
   BélésAnyagKod, TalpAnyagKod, GyártóID)
Gyárt (ModellID, Szám, Szín, GyártásiÁr)
Árul (ÜzletID, ModellID, Szám, Szín, EladásiÁr, DbRaktáron)
```

Az Anyagok, ÁruCsoportok és Kinek táblák sorait a következő táblázatok adják:

AnyagKod	*Név*
B	Bőr
M	Műanyag
V	Vászon

ÁruCsopKod	*Név*
FC	félcipő
EC	egészcipő
CS	csizma
SC	sportcipő
SZ	szandál

KinekKod	*Név*
F	Férfi
N	Női
G	Gyerek
U	Unisex

Oldjuk meg relációs algebrai műveletek segítségével:

a. Azon üzletek telefonszámát, melyek a Horea úton találhatók.
b. Azon gyártók neve és címe, akik 1.000.000 lejnél olcsóbb sportcipőt készítenek (gyártási árat figyelembe véve)!
c. Azon üzletek neve és címe, amelyek árulnak piros színű női cipőket!
d. Azon üzletek neve és címe, amelyek árulnak 45-ös és annál nagyobb férfi cipőket!
e. Azon üzletek neve és címe, amelyek árulnak 35-ös és annál kisebb női félcipőket!
f. Azon gyártók neve és címe, akik gyártanak tiszta bőr félcipőket (FelsőrészAnyagKod, BélésAnyagKod, TalpAnyagKod is bőr).
g. Azon üzletek neve és címe, amelyek árulnak 2.000.000 lejnél drágább gyerek cipőket!
h. Azon üzletek neve és címe, amelyek árulnak olyan gyerek cipőket, melyek bélésanyaga és felsőrészanyaga is bőr!
i. Azon üzletek neve és címe, ahol minden cipőből, melynek ugyanaz a modellje, a száma és színe, csak 1 db van raktáron.
j. Azon gyártók neve és címe, akik csak sportcipőt készítenek, másfajtát nem!

5.4. Legyenek a következő táblák egy relációs adatbázisból, mely egy iskola osztályairól, tanárairól, termeiről, diákjairól és azok jegyeiről szóló információkat tárolja egy tanévben. Osztály kódok: X B, XI C, stb. az Ev attribútum megfelelő értékei: 10, 11. Egy tanár több tantárgyat is taníthat, pl. matematikát és informatikát is, ugyanannak az osztálynak vagy különböző osztályoknak. A Felevie attribútum értéke 1, ha a jegy a félévi dolgozat jegye és 0 ha nem.

```
Osztályok (OsztályKod, OsztalyfonokID, Ev, TeremID);
Termek (TeremID, TeremNev, Emelet)
Diákok (DiakID, DiakNev, DiákCím, OsztalyKod);
Tanárok (TanárID, Nev, Cim, Tel, TgyID, OsztályKod);
Tantárgyak (TgyID, TgyNev)
Tanít (TanárID, TgyID, OsztályKod)
```

```
Jegyek (DiákID, TgyID, Dátum, Felev, Jegy, Felevie);
```

Oldjuk meg relációs algebrai műveletek segítségével:

a. Adjuk meg az első emeleten tanuló diákok nevét!
b. Adjuk meg azoknak a diákoknak a nevét, akik tanulnak franciát.
c. Adjuk meg azoknak a diákoknak a nevét, akik **nem** tanulnak németet.
d. Adjuk meg a matematikát tanító tanárok nevét!
e. Adjuk meg azon tanárok nevét, akik tanítják.
f. Adjuk meg a „Péter András" nevű diák jegyeit, a tantárgyak nevét is feltüntetve.
g. Adjuk meg a XII B-t tanító tanárok nevét és a tanított tantárgyak nevét.
h. Adjuk meg a tizedikes tanulók jegyeit, a tantárgyak nevét is feltüntetve.

5.5. Adott a következő relációs modellben leírt adatbázis, mely autóalkatrészeket áruló cég részleges adatbázisa. A cégnek több városban is van egy-egy üzlete.

```
Város (VárosID, VNév, ManSzemSzám);
AlkatrészCsoportok (CsopID, CsNév);
     (Név lehetséges értékei: ablaktörlők, gumiabroncsok,
          akkumulátorok, stb.)
Alkatrészek(AlkatrészID, ANév, MértEgys, CsopID);
     1 alkatrész esetén: Erték = MennyiségRaktaron*EladásirAr
Uzletek (VárosID, AlkatrészID, MennyiségRaktáron, EladásiAr)
Szállítók (SzállID, SzNév, Helység, UtcaSzám);
Szállít (SzállID, AlkatreszID, Datum, SzállításiAr);
```

Oldjuk meg relációs algebrai műveletek segítségével:

a. Adjuk meg a kolozsvári üzletben található alkatrészek nevét!
b. Adjuk meg az üzeletekben található akkumulátorokat a következő formában:
   (VNév, ANév, MennyiségRaktáron, EladásiAr)
c. Adjuk meg azon szállítók nevét akik mindenféle alkatrészt szállítanak, az AlkatrészCsoportok táblában létező összes csoportból!
d. Adjuk meg azon szállítók nevét akik csak gumiabroncsokat szállítanak, (CsNév="gumiabroncs")!

6. Az SQL lekérdezőnyelv

A legtöbb relációs ABKR az adatbázist az SQL-nek (Structured Query Language) nevezett lekérdezőnyelv segítségével kérdezi le és módosítja. Az SQL központi magja ekvivalens a relációs algebrával, de sok kiterjesztést dolgoztak ki hozzá, mint például az összesítések.

Az SQL-nek számos verziója ismeretes, szabványokat is dolgoztak ki, ezek közül a legismertebb az SQL-92 vagy SQL2. A napjainkban használt ABKR-ek lekérdezőnyelvei ezt a szabványt tartják be. Az SQL egy új szabványa az SQL3, mely rekurzióval, objektumokkal, triggerekkel stb. terjeszti ki az SQL2-őt. Számos kereskedelmi ABKR már meg is valósította az SQL3 néhány javaslatát.

6.1. Egyszerű lekérdezések SQL-ben

A relációs algebra vízszintes kiválasztás műveletét:

$$\sigma_f(R)$$

az SQL a SELECT, FROM és WHERE kulcsszavak segítségével valósítja meg a következőképpen:

```
SELECT *
FROM R
WHERE f;
```

6.1. példa: Legyen a NagyKer nevű adatbázis a következő relációsémákkal:

```
Részlegek (RészlegID, Név, Helység, ManSzemSzám);
Alkalmazottak (SzemSzám, Név, Fizetés, Cím, RészlegID);
Managerek (SzemSzám);
ÁruCsoportok (CsopID, Név, RészlegID);
Áruk (ÁruID, Név, MértEgys, MennyRakt, CsopID);
Szállítók (SzállID, Név, Helység, UtcaSzám);
Vevők (VevőID, Név, Helység, UtcaSzám, Mérleg, Hihetőség);
Szállít (SzállID, ÁruID, Ár);
Szerződések (SzerződID, Dátum, Részletek, VevőID);
Tételek (TételID, Dátum, SzerződID);
Szerepel (TételID, ÁruID, RendMenny, SzállMenny).
```

Több lekérdezés is fog erre az adatbázisra vonatkozni. Ebben a példában legyen a következő lekérdezés:

„Keressük azon alkalmazottakat, akik a 9-es részlegnél dolgoznak és a fizetésük nagyobb, mint 500 euró”, lásd 5.2 példát, ahol ugyanezt a lekérdezést relációs algebrai műveletek segítségével fogalmaztuk meg.

```
SELECT *
FROM Alkalmazottak
WHERE RészlegID = 9 AND Fizetés > 500; □
```

A FROM kulcsszó után adhatjuk meg azokat a relációkat, jelen esetben csak egyet, melyre a lekérdezés vonatkozik, a fenti példa esetén az Alkalmazottak reláció.

A kiválasztás feltételét a WHERE kulcsszó után tudjuk megadni. A példánk esetében azok a sorok fognak a lekérdezés eredményében megjelenni, melyek eleget tesznek a WHERE után megadott feltételnek, vagyis az alkalmazott RészlegID attribútumának az értéke 9 és a Fizetés attribútum értéke nagyobb, mint 500.

A SELECT kulcsszó utáni * azt jelenti, hogy az eredmény reláció fogja tartalmazni a FROM után megadott reláció összes attribútumát.

Az SQL nyelv nem különbözteti meg a kis és nagy betűket. Nem szükséges új sorba írni a FROM és WHERE kulcsszavakat, általában a fenti módon szokták megadni, de lehet egy sorban kis betűkkel is.

```
select * from alkalmazottak where részlegID = 9 and fizetés > 500;
```

A relációs algebra vetítés művelete

$$\pi_{A_{i_1},A_{i_2},\ldots,A_{i_k}}(R)$$

a SELECT-SQL parancs segítségével a következőképpen adható meg:

```
SELECT  A_i1 , A_i2 ,… , A_ik
FROM R;
```

A SELECT kulcsszó után megadhatjuk az *R* reláció bármely attribútumát és az eredmény sorok ezen attribútumokat fogják csak tartalmazni, ugyanazzal a névvel, amivel az *R* relációban szerepelnek.

6.2. példa: Legyen ismét az 5.3. példa, mely relációs algebra segítségével:

$$\pi_{\text{Név, Fizetés}}(\text{Alkalmazottak})$$

SELECT-SQL parancs segítségével a következőképpen írható fel:

```
SELECT Név, Fizetés
FROM Alkalmazottak; □
```

A lekérdezés feldogozása során a FROM kulcsszó után megadott relációt a feldolgozó végigjárja, minden sor esetén ellenőrzi a WHERE kulcsszó után megadott feltétel teljesül-e. Azon sorokat, melyek esetén a feltétel teljesül, az eredmény relációba helyezzük. A feldogozás hatékonyságát növeli, ha a feltételben szereplő attribútumok szerint létezik indexállomány.

A vetítés során kapott eredmény reláció esetén *megváltoztathatjuk az attribútumok neveit* az AS kulcsszó segítségével, ha a FROM után szereplő reláció attribútum nevei nem felelnek meg. Az AS nem kötelező. A SELECT kulcsszó után kifejezést is használhatunk.

6.3. példa: Ha például a fizetést nem euró-ban, hanem $-ban szeretnénk és az euró/dollár arány mondjuk 1.1, akkor a nagy fizetésű alkalmazottakat a 9-es részlegből a következő paranccsal kapjuk meg:

```
SELECT Név AS Név9, Fizetés * 1.1 AS Fizetes$
FROM Alkalmazottak
WHERE RészlegID = 9 AND Fizetés > 500;
```

Tehát az eredmény reláció két oszlopot fog tartalmazni, melyek nevei: Név9, illetve Fizetés$. □

A WHERE kulcsszó utáni feltétel lehet *összetett*, használhatjuk az AND, OR és NOT logikai műveleteket. A műveletek sorrendjének a meghatározására használhatunk zárójeleket, ha ezek megelőzési sorrendje nem felel meg. Az SQL nyelvben is, mint a legtöbb programozási nyelvben a NOT megelőzi az AND és OR műveletet, az AND pedig az OR-t.

6.4. példa: „Keressük a 3-as és 6-os részleg alkalmazottait akiknek kicsi a fizetése, 200 eurónál kisebb.” A következő paranccsal kapjuk meg:

```
SELECT Név, Fizetés
FROM Alkalmazottak
WHERE (RészlegID = 3 OR RészlegID = 6) AND Fizetés < 200;
```

Ha a zárójelet nem tettük volna ki, akkor csak a 6-os részlegből válogatta volna ki a kis fizetésűeket, és az eredmény relációban a 3-as részlegből az összes alkalmazott szerepelt volna. □

Karakterláncok összehasonlítása esetén használhatjuk a LIKE kulcsszót, hogy a karakterláncokat egy mintával hasonlítsunk össze a következőképpen:

```
k LIKE m
```

ahol k egy karakterlánc és m egy minta. A mintában használhatjuk a % és _ karaktereket. A % jelnek a k-ban megfelel bármilyen karakter 0 vagy nagyobb hosszúságú sorozata. Az _ jelnek megfelel egy akármilyen karakter a k-ból. A LIKE kulcsszó segítségével képezett feltétel igaz, ha a k karakterlánc megfelel az m mintának.

6.5. példa:
```
SELECT *
FROM Alkalmazottak
WHERE Név LIKE 'Kovács%';
```

A lekérdezés eredménye azon alkalmazottakat tartalmazza, kiknek a neve a 'Kovács' karaktersorral kezdődik. Megkapjuk az összes Kovács vezetéknevű alkalmazottat, de a 'Kovácsovics' vezetéknevűt is, ha ilyen létezik az adatbázisban. Ha csak a Kovács vezetéknevűeket akarjuk, akkor a 'Kovács %' mintát használjuk. □
Használhatjuk a

```
k NOT LIKE m
```

szűrő feltételt is.

Más szűrőfeltételek a BETWEEN és IN kulcsszóval képezhetők. A BETWEEN kulcsszó segítségével megadunk egy intervallumot, és azt vizsgáljuk, hogy adott oszlop, mely értéke esik a megadott intervallumba. (Az oszlop itt szintén lehet származtatott oszlop, kifejezés.)

```
WHERE <oszlop> BETWEEN <kifejezés_1> AND <kifejezés_2 >
```

6.6. példa:
```
SELECT Név
FROM Alkalmazottak
WHERE Fizetés BETWEEN 300 AND 500;
```

Ugyanazt az eredményt adja, mint a:

```
SELECT Név
FROM Alkalmazottak
WHERE Fizetés >= 300 AND Fizetés <=500; □
```

Az IN operátor után megadunk egy értéklistát, és azt vizsgáljuk, hogy az adott oszlop mely mezőinek értéke egyezik az adott lista valamelyik elemével. (Az oszlop lehet származtatott oszlop, kifejezés is.)

```
WHERE <oszlop> IN (<kifejezés_1>, <kifejezés_2> [,...])
```

6.7. példa: Legyen az Egyetem nevű adatbázis a következő relációsémákkal:

```
Szakok (SzakKod, SzakNév, Nyelv);
Csoportok (CsopKod, Evfolyam, SzakKod);
Diákok (BeiktatásiSzám, Név, SzemSzám, Cím, SzületésiDatum, CsopKod,
      Átlag);
TanszékCsoportok (TanszékCsopKod, Név);
Tanszékek (TanszékKod, Név, TanszékCsopKod);
Beosztások (BeosztásKod, Név);
Tanárok (TanárKod, Név, SzemSzám, Cím, PhD, TanszékKod, BeosztásKod,
      Fizetés);
Tantárgyak (TantKod, Név);
Tanít (TanárKod, TantKod);
Jegyek (BeiktatásiSzám, TantKod, Datum, Jegy)
```

A diákok összes jegyét eltároljuk a Jegyek relációban, több szemeszterben sok jegye van egy diáknak. A Diákok táblában az utolsó szemeszter vagy utolsó év átlaga szerepel az Átlag oszlopban, ami alapján eldöntik például, hogy kap-e a diák bentlakást, ösztöndíjat stb.

Keressük az '531'-es, '532'-s és '631'-es csoportok diákjait:

```
SELECT Név
FROM Diákok
WHERE CsopKod IN ('531', '532', '631'); □
```

A SELECT SQL parancs lehetőséget ad az eredmény reláció rendezésére az ORDER BY kulcsszavak segítségével. Alapértelmezés szerint növekvő sorrendben történik a rendezés, de ha csökkenő sorrendet szeretnénk, akkor a DESC kulcsszót használhatjuk.

6.8. példa: Ha a fenti lekérdezést kiegészítjük azzal, hogy a diákokat csoporton belül, névsor szerinti sorrendben akarjuk megadni, akkor a SELECT parancsot kiegészítjük az ORDER BY után a megfelelő attribútumokkal a következőképpen:

```
SELECT Név
FROM Diákok
WHERE CsopKod IN ('531', '532', '631')
 ORDER BY CsopKod, Név; □
```

6.9. példa: A diákokat átlag szerint csökkenő sorrendben adja meg:

```
SELECT Név
FROM Diákok
 ORDER BY Átlag DESC; □
```

6.2. Több relációra vonatkozó lekérdezések

A relációs algebra egyik fontos tulajdonsága, hogy a műveletek eredménye szintén reláció, és az eredmény operandus lehet a következő műveletben. Az SELECT-SQL is kihasználja ezt, a relációkat összekapcsolhatjuk, egyesíthetjük, metszetet vagy különbséget is számíthatunk.

A Descartes szorzat

$$R_1 \times R_2$$

műveletét a következő SQL parancs valósítja meg:

```
SELECT *
FROM R1, R2;
```

A Théta-összekapcsolást:

$$R_1 \bowtie_\theta R_2$$

a következő paranccsal adhatjuk meg:

```
SELECT *
FROM R1, R2
WHERE θ;
```

A leggyakrabban használt műveletet, a természetes összekapcsolást (lásd a feltételeket 5.1. alatt)

$$R_1 \bowtie R_2 = \pi_{B \cup C}(R_1 \bowtie_{(R_1.A_1=R_2.A_1)\wedge(R_1.A_2=R_2.A_2)\wedge\ldots\wedge(R_1.A_p=R_2.A_p)} R_2,$$

a következőképpen írhatjuk SQL-ben:

```
SELECT *
FROM R1, R2
WHERE R1.A1 = R2.A1 AND R1.A2 = R2.A2 AND … AND R1.Ap = R2.Ap;
```

Ebben az általános esetben a két összekapcsolandó relációnak p darab közös attribútuma van. A gyakorlatban általában a két relációnak egy közös attribútuma van. Amint látjuk, ha több relációban is szerepel ugyanaz az attribútum név, előtagként a reláció nevét használjuk.

6.10. példa: Legyenek a következő relációk:

```
Csoportok (CsopKod, Evfolyam, SzakKod);
Diákok (BeiktatásiSzám, Név, Cím, SzületésiDatum, CsopKod, Átlag);
```

Ha a diákok esetén szeretnénk kiírni az évfolyamot és szakkódot is, akkor ezt a következő SQL parancs segítségével érjük el:

```
SELECT Név, CsopKod, Evfolyam, SzakKod
FROM Diákok, Csoportok
WHERE Diákok.CsopKod = Csoportok.CsopKod;
```

Tehát a WHERE kulcsszó után megadjuk a join feltételt. Ha elfelejtjük a join feltételt az eredmény Descartes szorzat lesz, melynek méretei nagyon nagyok lehetnek.

Vannak olyan ABKR-ek, melyek az előbbi feladat megoldására a JOIN kulcsszót is elfogadják (pl. MS SQL Server):

```
SELECT Név, CsopKod, Evfolyam, SzakKod
FROM Diákok INNER JOIN Csoportok
ON Diákok.CsopKod = Csoportok.CsopKod;
```

Van OUTER JOIN is (lásd 6.7.).□

Amint az egyszerű lekérdezéseknél láttuk, a WHERE kulcsszó után a kiválasztás feltételét adtuk meg. Ha több reláció összekapcsolása mellett kiválasztás műveletet is meg akarunk adni, a join feltétel után AND logikai művelettel a kiválasztás feltételét is megadhatjuk.

6.11. példa: Az összes harmadéves diák nevét a következő paranccsal is megkaphatjuk:

```
SELECT Név
FROM Diákok, Csoportok
WHERE Diákok.CsopKod = Csoportok.CsopKod
       AND Evfolyam = 3; □
```

Több mint két relációt is összekapcsolhatunk természetes összekapcsolással, fontos, hogy az összes join feltételt megadjuk. Ha az összekapcsolandó relációk száma k, és minden két-két relációnak egy-egy közös attribútuma van, akkor a join feltételek száma k–1. Ha tehát 4 relációt kapcsolunk össze, a join feltételek száma minumum 3.

6.12. példa: A NagyKer nevű adatbázisra vonatkozóan legyen a következő lekérdezés:

„Adjuk meg azon szállítók nevét és címét, kik szállítanak édességet” (ÁruCsoportok.Név = ‘édesség’)

```
SELECT Szállítók.Név, Szállítók.Helység, Szállítók.UtcaSzám
     FROM ÁruCsoportok, Áruk, Szállít, Szállítók
     WHERE ÁruCsoportok.CsopID = Áruk.CsopID
          AND Áruk.ÁruID = Szállít.ÁruID
          AND Szállít.SzállID = Szállítók.SzállID
        AND ÁruCsoportok.Név = 'édesség'; □
```

Az SQL lehetőséget ad arra, hogy a FROM záradékban szereplő R relációhoz hozzárendeljünk egy másodnevet, melyet *sorváltozó*nak nevezünk. Sorváltozót akkor használunk, ha rövidebb vagy más nevet akarunk adni a relációnak, illetve ha a FROM után kétszer is ugyanaz a reláció szerepel. Ha használtunk másodnevet, akkor az adott lekérdezésben azt kell használjuk.

6.13. példa: Keressük azon alkalmazottakat, akik ugyanazon a címen laknak, például férj és feleség, vagy szülő és gyerek.

```
SELECT Alk1.Név AS Név1, Alk2.Név AS Név2
FROM Alkalmazottak AS Alk1, Alkalmazottak AS Alk2
WHERE Alk1.Cím = Alk2.Cím
     AND Alk1.Név < Alk2.Név;
```

A lekérdező feldolgozó ugyanazt a relációt kell kétszer bejárja, hogy a kért párokat megtalálja. Ha az Alk1.Név < Alk2.Név feltételt nem tettük volna, akkor minden alkalmazott bekerülne az eredménybe önmagával is párosítva. Ezt esetleg a <> feltétellel is megoldhattuk volna, de akkor egy férj−feleség páros kétszer is bekerült volna, csak más sorrendben. Például: (‘Kovács István’, ‘Kovács Sára’) és (‘Kovács Sára’, ‘Kovács István’) is. Mivel gyereknek lehet ugyanaz a neve, mint a szülőnek, ezért jobb megoldás a: Alk1.Név < Alk2.Név feltételt kicserélni a következő feltétellel:

```
Alk1.SzemSzám < Alk2. SzemSzám; □
```

Algoritmus egy egyszerű SELECT−SQL lekérdezés kiértékelésére:
Input: R_1, R_2,…, R_n relációk a FROM záradék után
Begin
  Minden t_1 sorra az R_1-ből
    Minden t_2 sorra az R_2-ből
      …
      Minden t_n sorra az R_n-ből
        Ha a WHERE záradék igaz a t_1, t_2, …, t_n attribútumainak az értékeire
          Akkor
          A SELECT záradék attribútumainak értékeiből alkotott sort az eredményhez adjuk
End

A relációs algebra *halmazműveleteit* (egyesítés, metszet és különbség) használhatjuk az SQL nyelvben, azzal a feltétellel, hogy az operandus relációknak ugyanaz legyen az attribútumhalmaza. A megfelelő kulcsszavak: UNION az egyesítésnek, INTERSECT a metszetnek és EXCEPT a különbségnek. Az SQL2 standard adja ezeket a lehetőségeket, de a kereskedelmi rendszerek esetén nem használhatjuk mindegyiket .

6.14. példa: Legyenek a Szállítók és Vevők relációk a NagyKer adatbázisból és a következő lekérdezés: „Keressük a kolozsvári cégeket, akikkel kapcsolatban áll a cégünk.” A megoldást a következő lekérdezés adja:

```
(SELECT Név, UtcaSzám
       FROM Szállítók
       WHERE Helység = 'Kolozsvár')
          UNION
     (SELECT Név, UtcaSzám
       FROM Vevők
       WHERE Helység = 'Kolozsvár'); □
```

6.15. példa: Legyenek az Alkalmazottak és Managerek relációk a NagyKer adatbázisból és a „Keressük azon alkalmazottakat, akik nem managerek” lekérdezés:

```
(SELECT SzemSzám, Név FROM Alkalmazottak)
     EXCEPT
(SELECT SzemSzám, Név FROM Managerek, Alkalmazottak
   WHERE Managerek.SzemSzám = Alkalmazottak.SzemSzám);
```

A fenti parancs esetén a második SELECT parancsban a join műveletre azért volt szükségünk, hogy a managernek keressük meg a nevét is, mert a különbség művelet esetén fontos, hogy az operandus relációknak ugyanaz az attribútumhalmaza legyen.

Ha az alkalmazott névre nem vagyunk kíváncsiak, akkor a következő SQL parancs azon alkalmazottak személyi számát adja meg, akik nem managerek.

```
(SELECT SzemSzám FROM Alkalmazottak)
     EXCEPT
(SELECT SzemSzám FROM Managerek);
```

A feladatot oly módon is megoldhatjuk, ha a kereskedelmi rendszer nem támogatja az EXCEPT műveletet, hogy alkalmazzuk a NOT EXISTS vagy NOT IN záradékot (lásd a 6.5.-ben).

6.16. példa: Legyen a 6.7. példa Egyetem adatbázisa, és tegyük fel, hogy van olyan eset, hogy egy fiatal tanársegéd a matematika szakról, tehát elvégezte a matematika szakot, de még diák az informatika szakon. Legyen a következő lekérdezés: „keressük azon tanárokat, akik még diákok”. A megoldás:

```
(SELECT Név FROM Tanárok)
     INTERSECT
(SELECT Név FROM Diákok);
```

A feladatot a következőképpen is megoldhatjuk, ha a kereskedelmi rendszer nem támogatja az INTERSECT műveletet:

```
SELECT Név FROM Tanárok
WHERE EXISTS
     (SELECT Név FROM Diákok
        WHERE Diákok.SzemSzám = Tanárok.SzemSzám); □
```

6.3. Ismétlődő sorok

Az SQL nyelv relációi az absztrakt módon definiált relációktól abban különböznek, hogy az SQL nem tekinti őket halmaznak, azaz a relációk multihalmazok A SELECT parancs eredményében szerepelhet két vagy több teljesen azonos sor, viszont van lehetőség ezen ismétlődések megszüntetésére.

A SELECT kulcsszó után a DISTINCT szó segítségével kérhetjük az azonos sorok megszüntetését.

6.17. példa: A 6.7. példa Egyetem adatbázisa esetén keressük azon csoportokat, amelyekben vannak olyan diákok, akiknek átlaga kisebb, mint 7.

```
SELECT DISTINCT CsopKod
FROM Diákok
WHERE Átlag < 7;
```

A parancs a Diákok táblából kiválogatja azokat a sorokat, ahol az átlag kisebb, mint 7, ezen sorok diákokról szóló információkat tartalmaznak, többek között a csoportkódot is. Egy csoportban több diák is lehet, akiknek az átlaga kisebb, mint 7, ezért, ha nem használjuk a DISTINCT kulcsszót, akkor előfordulhat, hogy egy csoportkód többször is szerepel az eredményben. □

A SELECT paranccsal ellentétben, a UNION, EXCEPT és INTERSECT halmazelméleti műveletek megszüntetik az ismétlődéseket. Ha nem szeretnénk, hogy az ismétlődő sorok eltűnjenek, a műveletet kifejező kulcsszó után az ALL kulcsszót kell használjuk.

6.18. példa: Az Egyetem adatbázisból keressük a személyeket, akik lehetnek tanárok vagy diákok. A következő parancs nem szünteti meg az ismétlődéseket:

```
(SELECT Név FROM Tanárok)
     UNION ALL
(SELECT Név FROM Diákok);
```

Tehát, ha van olyan tanár, aki közben diák is, akkor az kétszer fog szerepelni az eredményben. □

6.4. Összesítő függvények és csoportosítás

Az SQL nyelv lehetőséget ad egy oszlopban szereplő értékek összegezésére, vagyis hogy meghatározzuk a legkisebb, legnagyobb vagy átlag értéket egy adott oszlopból. Az *összesítés* művelete egy oszlop értékeiből *egy* új értéket hoz létre. Ezenkívül a reláció egyes sorait bizonyos feltétel szerint csoportosíthatjuk, például egy oszlop értéke szerint, és a csoporton belül végezhetünk összesítéseket.

Összesítő függvények a következők:

- SUM, megadja az oszlop értékeinek az összegét;
- AVG, megadja az oszlop értékeinek a átlagértékét;
- MIN, megadja az oszlop értékeinek a minimumát;
- MAX, megadja az oszlop értékeinek a maximumát;
- COUNT, megadja az oszlopban szereplő értékek számát, beleértve az ismétlődéseket is, ha azok nincsenek megszüntetve a DISTINCT kulcsszóval.

Ezeket a függvényeket egy skalár értékre alkalmazhatjuk, általában egy SELECT záradékbeli oszlopra.

6.19. példa: A következő lekérdezés segítségével megkapjuk az alkalmazottak átlagos fizetését:

```
SELECT AVG(Fizetés)
FROM Alkalmazottak; □
```

6.20. példa: Ha az alkalmazottak számára vagyunk kíváncsiak:

```
SELECT COUNT(*)
FROM Alkalmazottak; □
```

Mindkét példa esetén biztosak vagyunk abban, hogy egy alkalmazott csak egyszer szerepel a relációban, mivel a személyi szám elsődleges kulcs.

6.21. példa: Az Egyetem adatbázis esetén keressük azon csoportoknak a számát, amelyekben vannak olyan diákok, akik átlaga kisebb, mint 7:

```
SELECT COUNT(DISTINCT CsopKod)
FROM Diákok
WHERE Átlag < 7;□
```

Az eddigi összesítések az egész relációra vonatkoztak. Sok esetben a reláció sorait csoportosítani szeretnénk egy vagy több oszlop értékei szerint. Például az alkalmazottak átlagfizetését minden részlegen belül akarjuk meghatározni. Az Egyetem adatbázisban minden csoport esetén keressük a legnagyobb átlagot, a diákok számát. A csoportosítást a GROUP BY kulcsszó segítségével érjük el. A parancs általános formája:

```
SELECT < csoportosító oszlopok listája >,
       <összesítő-függvény>(<oszlop>)
FROM <reláció>
[WHERE <feltétel>]
[GROUP BY <csoportosító oszlopok listája>]
[HAVING <csoportosítási-feltétel>]
[ORDER BY <oszlop>];
```

A GROUP BY után megadjuk a csoportosító attribútumok (oszlopok) listáját, melyek azonos értéke szerint történik a csoportosítás. Csak ezeket az oszlopokat válogathatjuk ki a SELECT kulcsszó után és azokat, melyekre valamilyen összesítő függvényt alkalmazunk. Azon oszlopoknak, melyekre összesítő függvényt alkalmaztunk, érdemes más nevet adni, hogy könnyebben tudjunk hivatkozni rá.

6.22. példa: Legyenek az Alkalmazottak reláció sorai:

SzemSzám	***Név***	***RészlegID***	***Fizetés (euró)***
111111	Nagy Éva	2	300
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
123444	Vincze Ildikó	1	800
333333	Kovács István	2	500

A részlegeken belüli átlagfizetést a következő parancs segítségével kapjuk meg:

```
SELECT RészlegID, AVG(Fizetés), MIN(Fizetés), MAX(Fizetés), SUM(Fizetés)
FROM Alkalmazottak
GROUP BY RészlegID;
```

A kapott eredmény:

RészlegID	***AVG(Fizetés)***	***MIN(Fizetés)***	***MAX(Fizetés)***	***SUM(Fizetés)***
1	800	800	800	800
2	500	300	700	1500
9	650	400	900	1300

A lekérdezés processzor először rendezi a reláció sorait a csoportosítandó oszlop értékei szerint, utána azokat a sorokat, ahol ezen oszlopoknak ugyanaz az értéke, az eredmény relációban csak egy sor fogja képviselni, ahol megadhatjuk az oszlop értékét, amely a lekérdezett relációban minden sorban ugyanaz. A többi oszlopra csakis összesítéseket végezhetünk. Ha a SELECT kulcsszó után olyan oszlopot választunk ki, melynek értékei különbözőek a lekérdezett relációban, a lekérdezés processzor nem tudja, hogy a különböző értékekből melyiket válassza az eredménybe. Van olyan implementálása a SELECT−SQL parancsnak, mely megengedi, hogy egy olyan oszlopot is kiválasszunk, mely nincs a csoportosító attribútumok között és a processzor vagy az első, vagy az utolsó értéket választja a különböző értékek közül.

A SELECT parancs megengedi viszont, hogy a csoportosító attribútum hiányozzon a vetített attribútumok listájából.

6.23. példa: A következő lekérdezés helyes:

```
SELECT AVG(Fizetés) AS ÁtlagFizetés
FROM Alkalmazottak
GROUP BY RészlegID;
```

eredménye pedig:

ÁtlagFizetés
800
500
650

6.24. példa: Legyen a Szállít (SzállID, ÁruID, Ár) reláció. Egy árut több szállító is ajánlhatja, különböző árban. Sok esetben szükségünk van az átlagárra, amiben ajánlanak egy árut. A következő lekérdezés minden áru esetén meghatározza az átlagárat, amiben a különböző szállítók ajánlják.

```
SELECT ÁruID, AVG(Ár) AS ÁtlagÁr
FROM Szállít
GROUP BY ÁruID; □
```

A GROUP BY záradékot használhatjuk többrelációs lekérdezésben is. A lekérdezés processzor először az operandus relációkkal a WHERE feltételét figyelembe véve elvégzi a join, esetleg a Descartes szorzat műveletet és ennek az eredmény relációjára alkalmazza a csoportosítást.

6.25. példa: Ha a fenti példa esetén kíváncsiak vagyunk az árunak a nevére:

```
SELECT Áruk.Név, AVG(Ár)
FROM Szállít. ÁruID = Áruk.ÁruID
WHERE Szállít, Áruk
GROUP BY Áruk.Név;
```

Remélhetőleg az áru neve is egyedi kulcs, tehát nem fordul elő egy áru név több ÁruID esetén is, mert a fenti példában a Név attribútum szerint csoportosítunk. Ha nem egyedi a név, akkor a fenti lekérdezés az összes azonos nevű árunak az átlagát adja meg, de sok esetben ez megfelel a felhasználónak. Megoldhatjuk úgy is, hogy először ÁruID szerint, majd áru név szerint csoportosítunk, lásd a csoportosítást több oszlopra. □

Amint a SELECT parancsnak az általános formájánál láttuk, lehetséges több csoportosítási attribútum is.

6.26. példa: Legyenek a következő relációk az Egyetem adatbázisból:

```
Tanszékek (TanszékKod, Név, TanszékCsopKod);
Beosztások (BeosztásKod, Név);
Tanárok (TanárKod, Név, SzemSzám, Cím, PhD, BeosztásKod, TanszékKod,
      Fizetés);
```

és a következő lekérdezés: „Számítsuk ki a tanárok átlagfizetését tanszékeken belül, beosztásokra leosztva.”

```
SELECT TanszékKod, BeosztásKod, AVG(Fizetés)
FROM Tanárok
GROUP BY TanszékKod, BeosztásKod
```

Ha a Tanárok tábla tartalma:

Tanár Kod	*Név*	*Cím*	*PhD*	*Beosztás Kod*	*Tanszék Kod*	*Fizetés*
KB12	Kiss Béla	Petőfi u. 12	Y	ADJ	ALG	150
NL03	Nagy László	Kossuth u. 3	Y	ADJ	REN	160
KG05	Kovács Géza	Ady tér 5	N	ADJ	ALG	160
PI14	Péter István	Dóm tér 14	N	TNS	REN	120
NT55	Németh Tamás	Dózsa u. 55	Y	PRO	ALG	300
VS77	Vígh Sándor	Rózsa u. 77	Y	PRO	REN	310
LL63	Lukács Lóránt	Viola u. 63	Y	ADJ	REN	170
LS07	László Samu	Rákóczi u. 7	N	TNS	REN	110
KP52	Kerekes Péter	Váczi u. 52	Y	PRO	ALG	280

a lekérdezés eredménye:

Tanszék Kod	*Beosztás Kod*	*AVG (Fizetés)*
ALG	ADJ	155
ALG	PRO	290
REN	ADJ	165
REN	PRO	310
REN	TNS	115

6.27. példa: Megismételve a 6.24. példát:

```
SELECT Áruk. ÁruID, Áruk.Név, AVG(Ár)
FROM Szállít. ÁruID = Áruk.ÁruID
WHERE Szállít, Áruk
GROUP BY Áruk.ÁruID, Áruk.Név;
```

Az áru név szerinti csoportosítás nem fog újabb csoportokat behozni, de nem válogathatjuk ki a Név oszlopot, ha nem szerepelt a csoportosítási attribútumok között. A vetítés attribútumai között nem kell feltétlenül szerepeljen az ÁruID, de ha egy név többször is előfordul, akkor az eredmény furcsa lesz. □

A csoportosítás után kapott eredmény reláció soraira a HAVING kulcsszót használva egy feltételt alkalmazhatunk. Ha csoportosítás előtt szeretnénk kiszűrni sorokat, azokra a WHERE feltételt lehet alkalmazni. A HAVING kulcsszó utáni feltételben azon oszlopok szerepelhetnek, melyekre a SELECT parancsban összesítő függvényt alkalmaztunk.

6.28. példa: Keressük azon részlegeket, ahol az alkalmazottak átlagfizetése nagyobb, mint 500 euró, átlagfizetés szerint növekvő sorrendben.

```
SELECT RészlegID, AVG(Fizetés)
FROM Alkalmazottak
GROUP BY RészlegID
HAVING AVG(Fizetés) > 500
ORDER BY AVG(Fizetés);
```

A fenti adatokat figyelembe véve az eredmény reláció a következő lesz:

RészlegID	*AVG(Fizetés)*
9	650
1	800

Ha nem adjuk meg az ORDER BY záradékot, akkor a GROUP BY záradékban megadott oszlopok szerint rendezi az eredményt.

6.29. példa: *Helytelen* a következő parancs:

```
SELECT RészlegID, AVG(Fizetés)
FROM Alkalmazottak
WHERE AVG(Fizetés) > 500
GROUP BY RészlegID;
```

6.30. példa: Keressük azon tanszékeket, ahol a tanársegédeket kivéve a tanárok átlagfizetése nagyobb, mint 240 euró.

```
SELECT TanszékKod, AVG(Fizetés)
FROM Tanárok
WHERE BeosztásKod <> 'TNS'
GROUP BY TanszékKod
HAVING AVG(Fizetés) > 240;
```

Az eredmény a következő lesz:

TanszékKod	*AVG (Fizetés)*
ALG	125

Alkalmazhatunk összesítő függvényt a csoportosítás után.

6.31. példa: Keressük a tanárok tanszékenkénti átlagfizetéseiből a legnagyobb értéket:

```
SELECT MAX(AVG(Fizetés))
FROM Tanárok
GROUP BY TanszékKod;
```

A lekérdezés processzor először elvégzi a csoportosítást TanszékKod szerint, majd az átlagfizetésekből kiválasztja a legnagyobbat.

6.5. Alkérdések

A WHERE záradékban eddig a feltételben skaláris értékeket tudtunk összehasonlítani. Az alkérdések segítségével sorokat vagy relációkat tudunk összehasonlítani. Egy alkérdés egy olyan kifejezés, mely egy relációt eredményez, például egy select-from-where kifejezés.

Alkérdést tartalmazó SELECT SQL parancs általános formája a következő:

```
SELECT <attribútum_lista>
FROM <tábla>
WHERE <kifejezés> <operátor>
          (SELECT <attribútum_lista>
                FROM <tábla>);
```

A rendszer először az alkérdést hajtja végre és annak eredményét használja a „fő” lekérdezés, kivéve a korrelált alkérdéseket (lásd a korrelált alkérdések a 6.6.-ban).

Alkérdéseket annak megfelelően csoportosíthatjuk, hogy az eredménye hány sort és hány oszlopot tartalmaz:

- egy oszlopot, egy sort, vagyis egy *skalár* értéket ad vissza (single-row);
- egy oszlopot, több sort, ún. *több soros alkérdés* (multiple-row subquery);
- több oszlopot, több sort, ún. *több oszlopos alkérdés* (multiple-column);

Ha egy attribútum egyetlen értékére van szükségünk, ebben az esetben a select-from-where kifejezés *skalár* értéket ad vissza, mely konstansként használható. A select-from-where kifejezés eredményeként kapott konstanst egy attribútummal vagy egy másik konstanssal összehasonlíthatjuk. Nagyon fontos, hogy az alkérdés select-from-where kifejezése csak egy attribútumnak egyetlen értékét adja eredményül, különben hibajelzést kapunk.

6.32. példa: Legyenek a Részlegek és Alkalmazottak relációk a NagyKer adatbázisból, és a következő lekérdezés: „Keressük a 'Tervezés' nevű részleg managerének a nevét." A megoldás alkérdés segítségével:

```
1)   SELECT Név
2)   FROM Alkalmazottak
3)   WHERE SzemSzám =
4)        (SELECT ManSzemSzám
5)         FROM Részlegek
6)         WHERE Név = 'Tervezés');
```

Amint látjuk, az alkérdés (4−6 sorok) csak egy oszlopot választ ki a manager személyi számát, de még abban is biztosak kell legyünk, hogy csak egy 'Tervezés' nevű részleg legyen az adatbázisban. Ezt elérhetjük ha egyedi kulcs megszorítást kérünk a Részlegek relációra a CREATE TABLE parancsban a UNIQUE kulcsszó segítségével. Abban az esetben, ha az alkérdés nulla vagy egynél több sort eredményez, a lekérdezés futás közbeni hibát fog jelezni. Az „Összesítések" alfejezet 6.22. példájának az adatait figyelembe véve az alkérdés eredményül az 123444 személyi számot adja, és a lekérdezés a következőképpen hajtódik végre:

```
SELECT Név
FROM Alkalmazottak
WHERE SzemSzám = 123444
```

A lekérdezés eredménye: 'Vincze Ildikó' lesz.□

A *skalár* értéket adó alkérdéssel használható *operátorok* az: =, <, <=, >, >=, <>.

6.33. példa: „Keressük azon alkalmazottakat, kiknek fizetése nagyobb, mint annak az alkalmazottnak, kinek a személyi száma 333333."

```
SELECT Név
FROM Alkalmazottak
WHERE Fizetés >
     (SELECT Fizetés
      FROM Alkalmazottak
      WHERE SzemSzám = 333333); □
```

6.34. példa: „Keressük azon alkalmazottakat, kiknek a fizetése az összes alkalmazott minimális fizetésével egyenlő."

```
SELECT Név
FROM Alkalmazottak
WHERE Fizetés =
          (SELECT MIN(Fizetés)
           FROM Alkalmazottak); □
```

6.35. példa: „Keressük azon részlegeket és az alkalmazottak minimális fizetését a részlegből, ahol a minimális fizetés nagyobb, mint a minimális fizetés a 2-es ID-jű részlegből."

```
SELECT RészlegID, MIN(Fizetés)
FROM Alkalmazottak
GROUP BY RészlegID
HAVING MIN(Fizetés) >
     (SELECT MIN(Fizetés)
      FROM Alkalmazottak
      WHERE RészlegID = 2);
```

A lekérdezés processzor először az alkérdést értékeli ki, ennek eredményeként egy skalár értéket (300) kapunk és a fő lekérdezés ezzel a skalár értékkel fog dolgozni. □

Csínján kell bánjunk a csoportosítással.

6.36. Példa: Egy *helytelen* SELECT parancs:

```
SELECT SzemSzám, Név
FROM Alkalmazottak
WHERE Fizetés =
     (SELECT MIN(Fizetés)
      FROM Alkalmazottak
      GROUP BY RészlegID);
```

Az alkérdés több sort is visszaad, pontosan annyit, ahány különböző RészlegID létezik az Alkalmazottak táblában, minden részleg esetén a minimális fizetést adja vissza. Az egyenlőség az alkérdés előtt csak egy skaláris értéket vár. □

A *több-soros alkérdések* esetén a WHERE záradék feltétele olyan *operátorok*at tartalmazhat, amelyeket egy *R* relációra alkalmazhatunk, ebben az esetben az eredmény logikai érték lesz. Bizonyos operátoroknak egy skaláris *s* értékre is szükségük van. Ilyen operátorok:

- `EXISTS R` – feltétel, mely akkor és csak akkor igaz, ha *R* nem üres.

6.37. példa:
```
SELECT Név
FROM Alkalmazottak, Managerek
WHERE Alkalmazottak. SzemSzám = Managerek.SzemSzám
     AND EXISTS
          (SELECT *
           FROM Alkalmazottak
           WHERE Fizetés > 500);
```

A fenti példa csak abban az esetben adja meg a managerek nevét, ha van legalább egy alkalmazott, kinek a fizetése nagyobb, mint 500 euró.

- `s IN R`, mely akkor igaz, ha *s* egyenlő valamelyik *R*-beli értékkel. Az *s* NOT IN *R* akkor igaz, ha *s* egyetlen *R*-beli értékkel sem egyenlő.

6.38. példa: Legyen a NagyKer adatbázis és a következő lekérdezés: „Adjuk meg azon szállítók nevét és címét, akik valamilyen csokit szállítanak” (Áruk.Név LIKE ‘%csoki%’)

```
1)   SELECT Név, Helység, UtcaSzám
2)   FROM Szállítók
3)   WHERE SzállID IN
4)           (SELECT SzállID
5)            FROM Szállít
6)            WHERE ÁruID IN
7)              (SELECT ÁruID
8)               FROM Áruk
9)               WHERE Név LIKE '%csoki%')
                 );
```

A 7−9 sor alkérdése az összes olyan árut választja ki, melynek nevében szerepel a csoki. Legyen a csoki áruk azonosítóinak a halmaza: CsokiID. A 4−6 sor a Szállít táblából azon SzállID-kat választja ki, ahol az ÁruID benne van a CsokiID halmazban. Nevezzük a csokit szállítók azonosítóinak a halmazát CsokiSzállIDk-nak. Az 1−3 sorok segítségével megkaphatjuk a csokit szállítók nevét és címét. □

A kereskedelmi rendszerek különböző mélységig tudják az alkérdéseket kezelni. Van olyan, amelyik csak 1 alkérdést engedélyez.

- `s > ALL R`, mely akkor igaz, ha *s* nagyobb, mint az *R* reláció minden értéke, ahol az *R* relációnak csak egy oszlopa van. A > operátor helyett bármelyik összehasonlítási operátort használhatjuk. Az `s <> ALL R` eredménye ugyanaz, mint az `s NOT IN R` feltételé.

6.39. példa: Legyen a következő lekérdezés:

```
SELECT SzemSzám, Név
FROM Alkalmazottak
WHERE Fizetés > ALL
     (SELECT MIN(Fizetés)
      FROM Alkalmazottak
GROUP BY RészlegID);
```

Ugyanezt a lekérdezést láttuk egyenlőséggel az alkérdés előtt, helytelen példaként. Mivel az alkérdés több sort is visszaad, a „> ALL” operátort alkalmazva, a Fizetés oszlop értékét összehasonlítja az összes minimális fizetés értékkel az alkérdésből. Tehát a lekérdezés megadja azon alkalmazottakat, kiknek fizetése nagyobb, mint a minimális fizetés minden részlegből. □

- *s* > ANY *R*, mely akkor igaz, ha *s* nagyobb az *R* egyoszlopos reláció legalább egy értékénél. A > operátor helyett akármelyik összehasonlítási operátort használhatjuk.

6.40. példa: „Keressük azokat a tanárokat, akik beosztása nem professzor, és van olyan professzor, akinek a fizetésénél az illető tanárnak nagyobb a fizetése.”

```
SELECT Név, BeosztásKod, Fizetés
FROM Tanárok
WHERE Fizetés > ANY
        (SELECT Fizetés
          FROM Tanárok
          WHERE BeosztásKod = 'PRO')
       AND BeosztásKod <> 'PRO'; □
```

A *több oszlopos alkérdés* esetén, a SELECT kulcsszó után megadhatunk több mint egy oszlopot, és szükségszerűen a fő lekérdezésben is ugyanannyi oszlopot kell megadjunk az összehasonlító operátor bal oldalán is. Az összehasonlítás párokra vonatkozik.

6.41. példa: „Keressük azokat a tanárokat, akiknek a fizetése egyenlő az algebra tanszék beosztásnak megfelelő átlag fizetésével.”

```
SELECT Név, BeosztásKod, Fizetés
FROM Tanárok
WHERE BeosztásKod, Fizetés IN
        (SELECT BeosztásKod, AVG(Fizetés)
         FROM Tanárok
         WHERE TanszékKod = 'ALG'
         GROUP BY BeosztásKod); □
```

Az alkérdés meghatározza az algebra tanszéken belül a beosztásoknak megfelelő átlagfizetéseket. A fő lekérdezés akkor fog egy tanárt kiválasztani, ha az alkérdés eredményhalmazában megtalálja a tanár beosztás kodja mellett a fizetést is, az értékpárt.

6.6. Korrelált alkérdések

Az eddig bemutatott alkérdések esetén az alkérdés csak egyszer kerül kiértékelésre és a kapott eredményt a magasabb rendű lekérdezés hasznosítja. A beágyazott alkérdéseket úgy is lehet használni, hogy az alkérdés többször is kiértékelésre kerül. Az alkérdés többszöri kiértékelését egy, az alkérdésen kívüli sorváltozóval érjük el. Az ilyen típusú alkérdést *korrelált* alkérdésnek nevezzük.

6.42. példa: Az Egyetem adatbázis esetén keressük azon diákokat, akik egyedül vannak a csoportjukban 10-es átlaggal.

```
SELECT Név, CsopKod
FROM Diákok D1
WHERE Átlag = 10 AND NOT EXISTS
          (SELECT D2.BeiktatásiSzám
            FROM Diákok D2
            WHERE D1.CsopKod = D2.CsopKod
               AND D1.BeiktatásiSzám <> D2.BeiktatásiSzám
               AND D2.Átlag = 10);
```

A lekérdezés kiértékelése során a D1 sorváltozó végigjárja a Diákok relációt. Minden sorra a D1-ből a D2 sorváltozó segítségével ismét végigjárjuk a Diákok relációt.

Legyen *d*1 egy sor a Diákok relációból, amelyet a fő lekérdezés az eredménybe helyez, ha megfelel a WHERE utáni feltételnek. Először is a *d*1.Átlag értéke 10 kell legyen és az alkérdés eredménye pedig üres halmaz. Az alkérdés akkor fog sorokat tartalmazni, ha létezik a Diákok relációban egy *d*2 sor, mely esetén ugyanaz a csoport kód, mint a *d*1 sor esetén, az átlag értéke 10 és a beiktatási szám különbözik a *d*1 sor BeiktatásiSzám attribútum értéketől. Ez azt jelenti, hogy az adatbázisban találtunk egy másik diákot, ugyanabból a csoportból, akinek az átlaga 10-es. Mivel az alkérdésben vannak sorok, nem fogja a *d*1 sort kiválasztani. Ha az alkérdés üres halmaz, akkor kiválasztja a *d*1-et, és ekkor találtunk olyan diákot, aki egyedül van a csoportjában 10-es átlaggal. □

6.7. Más típusú összekapcsolási műveletek

A relációs algebra természetes összekapcsolás műveletét eddig a SELECT parancs segítségével láttuk implementálva. Ha a WHERE záradékban adjuk meg a feltételt, vagy INNER JOIN kulcsszót használunk, csak azok a sorok kerülnek be az eredmény relációba, melyek esetében a közös attribútum ugyanaz az értéke mindkét relációban megtalálható. (A lógó sorok nem kerülnek be az eredménybe.) Bizonyos esetekben szükségünk van a lógó sorokra is.

Az OUTER JOIN kulcsszó segítségével azon sorok is megjelennek az eredményben, melyek értéke a közös attribútumra nem található meg a másik táblában, vagyis a lógó sorok, melyekben a másik tábla attribútumai NULL értékeket kapnak. Tehát a *külső összekapcsolás* (outer join) eredménye tartalmazza a belső összekapcsolás (*inner join*) eredménye mellett a lógó sorokat is. A *külső összekapcsolás* 3−féle lehet:

```
R LEFT OUTER JOIN S ON R.X = S.X
```

eredménye tartalmazza a bal oldali *R* reláció összes sorát, azokat is, amelyek esetében az *X* attribútumhalmaz értéke nem létezik az *S* reláció *X* értékei között. Ezt a műveletet külső baloldali összekapcsolásnak nevezzük. Az eredmény az *S* attribútumait is tartalmazza NULL értékekkel.

```
R RIGHT OUTER JOIN S ON R.X = S.X
```

eredménye a jobb oldali *S* reláció összes sorát tartalmazza, azokat is amelyek esetében az *X* attribútumhalmaz értéke nem létezik az *R* reláció *X* értékei között. Ezt a műveletet külső jobboldali összekapcsolásnak nevezzük. Az eredmény az *R* attribútumait is tartalmazza NULL értékekkel.

```
R FULL OUTER JOIN S ON R.X = S.X
```

eredménye azon sorokat tartalmazza, melyek esetében a közös attribútum értéke megegyezik mindkét relációban és mind a bal oldali *R* reláció lógó sorait, mind az *S* reláció lógó sorait magában foglalja.

6.43. példa: Legyenek az Alkalmazottak és Részlegek reláció sorai:

SzemSzám	*Név*	*RészlegID*	*Fizetés*
111111	Nagy Éva	2	300
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
123444	Vincze Ildikó	1	800
567765	Katona József	NULL	600
556789	Lukács Anna	NULL	700
333333	Kovács István	2	500

RészlegID	*RNév*	*ManagerSzemSzám*
1	Tervezés	123444
2	Könyvelés	234555
3	Eladás	NULL
9	Beszerzés	456777

Legyen a következő lekérdezés:

```
SELECT * FROM Alkalmazottak
INNER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

Az eredmény:

SzemSzám	*Név*	*RészlegID*	*Fizetés*	*RNév*	*ManagerSzemSzám*
111111	Nagy Éva	2	300	Könyvelés	234555
222222	Kiss Csaba	9	400	Beszerzés	456777
456777	Szabó János	9	900	Beszerzés	456777
234555	Szilágyi Pál	2	700	Könyvelés	234555
123444	Vincze Ildikó	1	800	Tervezés	123444
333333	Kovács István	2	500	Könyvelés	234555

Tehát azon alkalmazottak esetén, ahol a RészlegID megtalálható a Részlegek táblában megkapjuk a megfelelő részleg nevét és a manager személyi számát. Lógó sorok nem jelennek meg az eredményben. □

6.44. példa: A

```
SELECT * FROM Alkalmazottak
LEFT OUTER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

eredménye:

SzemSzám	*Név*	*RészlegID*	*Fizetés*	*RNév*	*Manager SzemSzám*
111111	Nagy Éva	2	300	Könyvelés	234555
222222	Kiss Csaba	9	400	Beszerzés	456777
456777	Szabó János	9	900	Beszerzés	456777
234555	Szilágyi Pál	2	700	Könyvelés	234555
123444	Vincze Ildikó	1	800	Tervezés	123444
567765	Katona József	NULL	600	NULL	NULL
556789	Lukács Anna	NULL	700	NULL	NULL
333333	Kovács István	2	500	Könyvelés	234555

Ebben az esetben az Alkalmazottak összes sora, és a lógó sorok is megjelennek az eredményben, a Részlegek attribútumai a lógó sorok esetén NULL értéket kapnak. □

6.45. példa: A

```
SELECT * FROM Alkalmazottak
RIGHT OUTER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

eredménye:

SzemSzám	*Név*	*RészlegID*	*Fizetés*	*RNév*	*Manager SzemSzám*
111111	Nagy Éva	2	300	Könyvelés	234555
222222	Kiss Csaba	9	400	Beszerzés	456777
456777	Szabó János	9	900	Beszerzés	456777
234555	Szilágyi Pál	2	700	Könyvelés	234555
123444	Vincze Ildikó	1	800	Tervezés	123444
333333	Kovács István	2	500	Könyvelés	234555
NULL	NULL	3	NULL	Eladás	NULL

Ebben az esetben a Részlegek összes sora jelenik meg, mivel ez a jobb oldali reláció. Az Alkalmazottak reláció attribútumai a lógó részleg esetén NULL értékeket kapnak. □

6.46. példa: A

```
SELECT * FROM Alkalmazottak
FULL OUTER JOIN Részlegek
ON Alkalmazottak.RészlegID = Részlegek. RészlegID;
```

eredménye:

SzemSzám	*Név*	*RészlegID*	*Fizetés*	*RNév*	*ManagerSzemSzám*
111111	Nagy Éva	2	300	Könyvelés	234555
222222	Kiss Csaba	9	400	Beszerzés	456777
456777	Szabó János	9	900	Beszerzés	456777
234555	Szilágyi Pál	2	700	Könyvelés	234555
123444	Vincze Ildikó	1	800	Tervezés	123444
333333	Kovács István	2	500	Könyvelés	234555
567765	Katona József	NULL	600	NULL	NULL
556789	Lukács Anna	NULL	700	NULL	NULL
NULL	NULL	3	NULL	Eladás	NULL

6.8. Változtatások az adatbázisban

Eddig a lekérdezések megfogalmazását láttuk az SQL nyelv segítségével. Szükség van új adatok beszúrására az adatbázisba, a létező adatok törlésére és módosítására is. Ezekre is van lehetőség az SQL nyelv segítségével.

6.8.1. Beszúrás

A beszúrási művelet legegyszerűbb formája:

```
INSERT INTO R(A1, A2, …, An) VALUES (v1, v2, …, vn);
```

A művelet során az R relációba egy új sor kerül, ahol az A_i attribútum értéke v_i, minden i = 1,2, …, n. Ha az attribútumlista nem tartalmazza R összes attribútumát, akkor a hiányzó

attribútumok az alapértelmezés szerinti értéket kapják (alapértelmezés szerinti értéket lásd a 4.5.-ben). A leggyakrabban használt alapértelmezés szerinti érték a nullérték, tehát ha nem adjuk meg valamelyik attribútum értékét, az NULL lesz. Gond akkor lehet, ha megszorítás van egy attribútumra, hogy nem lehet NULL, és ha nincs megadva egy alapértelmezés szerinti érték, akkor a beszúrás hibát jelez és nem történik meg.

6.47. példa: Legyen a Részlegek reláció és a következő parancs:

```
INSERT INTO Részlegek(RészlegID, RNév, ManagerSzemSzám)
      VALUES (4, 'Termelés', 567765);
```

Mivel minden attribútumot felsoroltunk nem lesz szükség alapértelmezett értékekre. A Részlegek relációba egy új sort szúr be az ABKR a megadott értékekkel. □

Minden beszúrási művelet esetén az ABKR ellenőrzi a megszorításokat, ha a beszúrásra kerülő sor nem teljesít egyet is a megszorítások közül, hibát jelez és nem hajtja végre a beszúrási műveletet.

Ha megadjuk az összes attribútum értéket, akkor nem szükséges felsorolni az attribútum neveket, csak arra kell ügyelnünk, hogy pontosan abban a sorrendben adjuk meg őket, ahogy a relációban szerepelnek.

6.48. példa: Az előbbi parancsot megadhatjuk attribútumlista nélkül is:

```
INSERT INTO Részlegek
   VALUES (4, 'Termelés', 567765); □
```

6.49. példa: Legyen az Alkalmazottak reláció és a következő parancs:

```
INSERT INTO Alkalmazottak (SzemSzám, Név, RészlegID)
   VALUES (414141, 'Tamás Erika', 4);
```

Mivel a Fizetés attribútumot nem adtuk meg, ennek értéke NULL értékkel fog feltöltödni, mivel nincs alapértelmezett értéke, viszont lehet null. □

6.50. példa: Az Alkalmazottak reláció esetén a következő parancs hibát fog jelezni, mivel 111111 személyi számmal már létezik alkalmazott és a SzemSzám elsődleges kulcs:

```
INSERT INTO Alkalmazottak (SzemSzám, Név, RészlegID)
      VALUES (111111, 'Tamás Zoltán', 4); □
```

A fenti egyszerű beszúrási művelet egy sort szúr be egy relációba. Az attribútumlista helyett megadhatunk egy alkérdést is, melynek segítségével több sort is beszúrhatunk.

6.51. példa: Legyenek a

```
Részlegek (RészlegID, Név, Helység, ManSzemSzám);
Alkalmazottak (SzemSzám, Név, Fizetés, Cím, RészlegID);
```

relációk és amint a megszorításoknál láttuk a két reláció kölcsönösen hivatkozik egymásra külső kulcsos megszorítással, az Alkalmazottak a RészlegID segítségével a Részlegekre és a Részlegek az Alkalmazottakra a ManSzemSzám segítségével. Tegyük fel az Alkalmazottak táblába először begyűjtjük az összes managert, a külső kulcsos megszorítás még nincs beállítva, a RészlegID-t is feltöltjük a megfelelő értékekkel. Utána át szeretnénk venni az összes létező részleget a Részlegek relációba a manager személyi számával együtt. A következő parancs segítségével ez megvalósítható, a Név és Helység attribútumok NULL értékkel töltődnek fel.

```
INSERT INTO Részlegek (RészlegID, ManSzemSzám)
    SELECT DISTINCT RészlegID, SzemSzám FROM Alkalmazottak; □
```

6.8.2. Törlés

A törlési utasítás általános alakja:

```
DELETE FROM R WHERE <feltétel>;
```

A FROM kulcsszó után meg kell adjuk azon relációt, melyből sorokat szeretnénk törölni. Az utasítás azt eredményezi, hogy a megadott relációból kitörlődik minden olyan sor, mely a WHERE kulcsszó után megadott feltételt teljesíti.

6.52. példa:
```
DELETE FROM Alkalmazottak
WHERE SzemSzám = 111111;
```

Egy adott sor törlése esetén a legjobb, ha a reláció elsődleges kulcsának az értékét, vagy egy egyedi kulcsnak az értékét adjuk meg. A példa esetén a következő sor törlödik az Alkalmazottak relációból.

SzemSzám	*Név*	*RészlegID*	*Fizetés*
111111	Nagy Éva	2	300

Abban az esetben, ha a feltétel nem egyedi kulcs értékére vonatkozik, valószínű, hogy több sor is törlődik.

6.53. példa: A következő parancs törli azon alkalmazottakat, kik fizetése kisebb vagy egyenlő 400-zal.

```
DELETE FROM Alkalmazottak
WHERE Fizetés <= 400; □
```

6.8.3. Módosítás

Módosíthatunk egy vagy több sort egy relációban a következő parancs segítségével:

```
UPDATE R SET <értékadások> WHERE <feltétel>;
```

A parancs az *R* reláció sorait módosítja, az értékadásokat vesszővel választjuk el. Minden értékadás az *R* egy attribútumát, egyenlőségjelet és egy kifejezést tartalmaz.

6.54. példa:
```
UPDATE Alkalmazottak
SET Név = 'Nagy Éva Mária', Fizetés = 450
WHERE SzemSzám = 111111;
```

A sort az elsődleges kulcs segítségével azonosítottuk a WHERE kulcsszó utáni feltételben, majd módosítást kértünk a Név és Fizetés attribútumok értékére. □

Több sort is módosíthatunk egy UPDATE parancs segítségével.

6.55. példa:
```
UPDATE Alkalmazottak
SET Fizetés = Fizetés * 1.2
WHERE Fizetés < 600;
```

A 600 euró-nál kisebb fizetéseket 20%-al megnöveltük. □

6.56. példa: A WHERE feltétel tartalmazhat alkérdéseket is:

```
UPDATE Alkalmazottak
SET Fizetés = Fizetés * 1.5
WHERE SzemSzám IN (SELECT SzemSzám FROM Managerek);
```

A managerek fizetését 50%-kal megnöveltük. □

Problémák a törlés, beszúrás esetén akkor jelenhetnek meg, ha egy relációnak nincs egyedi kulcsa, akkor előfordul, hogy ugyanazt a sort többször is be tudjuk szúrni, törlés esetén pedig egy parancs az összes azonos sort kitörli, nem tudjuk csak az egyiket azonosítani.

6.57. példa: Legyen a Szállít reláció és tegyük fel, hogy nem adtuk meg a (SzállID, ÁruID) párost elsődleges kulcsnak. Az aktuális értékek a relációban:

SzállID	*ÁruID*	*Ár*
111	45	25000
222	45	26500
111	67	17000
111	56	20000
222	67	18000
222	56	22500

Az

```
INSERT INTO Szállít VALUES (111, 45, 25000);
```

parancs az első sort ismét beszúrja a relációba. A

```
DELETE FROM Szállít
WHERE SzállID = 111 AND ÁruID = 45;
```

viszont mind a két sort kitörli. □

6.9. Gyakorlatok

6.1. Legyen az 5.1. gyakorlat relációs modellben leírt adatbázisa:

```
Kiadók (KiadóKod, Knév, Khelység, Kcím, Ktelefon)
Könyvtípusok (TípusID, Név)
(lehetséges értékek: villamosság, informatika, elektronika, ..)
Könyvek (KönyvKod, Cím, Megjelenési-év, KiadóKod, TípusID,
     PéldánySzám, Ár)
Szerzők (SzerzőID, SzerzőNev)
KönyvSzerző(KönyvKod, SzerzőID)
```

Oldjuk meg SQL parancs segítségével a következő feladatokat:

Az 5-ös fejezet 5.1. gyakorlatának a.-g. lekérdezéseit.

h. Adjuk meg hány különböző elektronika könyv található a könytárban!
i. Adjuk meg a könytárban található könyvek összértékét!
   (1 könyv értéke = PéldánySzám * Ár)
j. Adjuk meg a a könytárban található Teora könyvek összértékét!

6.2. Legyenek a következő táblák egy relációs adatbázisból, mely egy egyetem részleges információit tárolja:

```
Karok (KarKod, KarNév)
Tanszékek (TanszékKod, Név, KarKod);
Beosztások (BeosztásKod, Név);
 (lehetséges értékek: PRO – professzor, DOC – docens,
        ADJ – adjunktus, TNS – tanársegéd, GYA – gyakornok)
Tanárok (TanárKod, Név, SzemSzám, Cím, PhD, TanszékKod,
        BeosztásKod, Fizetés);
Tantárgyak (TantKod, Név, KreditSzám, TanszékKod);
Tanít (TanárKod, TantKod);
```

Oldjuk meg SQL parancs segítségével a következő feladatokat:

Az 5-ös fejezet 5.2. gyakorlatának a.-g. lekérdezéseit.

(Ha tanárra vonatkozik a lekérdezés, akkor annak a beosztása akármi lehet, mindenki tanár, a professzor is, a gyakornok is.)

h. Adjuk meg tanszékeken belül a tanárok átlagfizetését!
i. Adjuk meg az egyetem legkisebb fizetésű tanárai a nevét!

j. Adjuk meg azon tanárok nevét, akik olyan tantárgyat tanítanak, mely nem a saját tanszéküknek a tantárgya.
k. Adjuk meg karon belül, tanszékeken, beosztáon belül a tanárok átlagfizetését a követlező formában: (KarNév, TanszékNév, BeosztásKod, AtlagFiz)
l. Adjuk meg karon belül, tanszéken belül a legkisebb fizetésű tanárok nevét a következő formában: (KarNév, TanszékNév, TanárNév, Fizetés)

6.3. Legyen az 5.3. gyakorlat relációs modellben leírt adatbázisa:

```
ÁruCsoportok (ÁruCsopKod, Név);
Anyagok (AnyagKod, Név),
Kinek (KinekKod, Név)
Gyártók (GyártóID, Név, Cím);
Üzletek (ÜzletID, Név, Cím, TelefonSzám)
Modellek (ModellID, Név, ÁruCsopKod, KinekKod, FelsőrészAnyagKod,
              BélésAnyagKod, TalpAnyagKod, GyártóID)
Gyárt (ModellID, Szám, Szín, GyártásiÁr)
Árul (ÜzletID, ModellID, Szám, Szín, EladásiÁr, DbRaktáron)
```

Oldjuk meg SQL parancs segítségével a következő feladatokat:

Az 5-ös fejezet 5.3. gyakorlatának a.-j. lekérdezéseit.

k. Azon gyártók neve és címe, akik a legolcsóbb tiszta bőr félcipőket gyártják (átlag gyártási árat vegyük figyelembe).
l. Azon üzlet neve és címe, amelyben a legnagyobb értékű áru található (egy üzletben található áru értéke = sum (DbRaktáron * EladásiÁr))
m. Azon gyártók neve és címe, melyek legalább 10 különböző modellű gyerekcipőt gyártanak!
n. Azon üzlet neve és címe, mely a legkisebb átlag árréssel dolgozik. (1 termék árrés = EladásiÁr – GyártásiÁr, üzlet átlag árrés = sum (összes termék árrés) / termékek száma)
o. Azon üzlet neve és címe, melyben a legnagyobb a választék a 45-ös és nagyobb lábbeliből.
p. Azon üzletek neve és címe, melyek mindenféle (félcipő, egészcipő, stb) lábbelit árulnak.
q. Azon üzletek neve és címe, melyek csak férficipőt árulnak.

6.4. Legyenek a következő táblák egy relációs adatbázisból, mely egy iskola osztályairól, tanárairól, termeiről, diákjairól és azok jegyeiről szóló információkat tárolja egy tanévben. Osztály kódok: X B, XI C, stb. az Ev attribútum megfelelő értékei: 10, 11. Egy tanár több tantárgyat is taníthat, pl. matematikát és informatikát is, ugyanannak az osztálynak vagy különböző osztályoknak. A Félévie attribútum értéke 1, ha a jegy a félévi dolgozat jegye és 0 ha nem.

```
Osztályok (OsztályKod, OsztalyfőnökID, Ev, TeremID);
Termek (TeremID, TeremNev, Emelet)
Diákok (DiakID, DiakNev, DiákCím, OsztalyKod);
Tanárok (TanárID, Nev, Cim, Tel, TgyID, OsztályKod);
Tantárgyak (TgyID, TgyNev)
Tanít (TanárID, TgyID, OsztályKod)
Jegyek (DiákID, TgyID, Dátum, Félév, Jegy, Félévie);
```

Oldjuk meg SQL parancs segítségével a következő feladatokat:

Az 5-ös fejezet 5.4 gyakorlatának a.-h. lekérdezéseit.

i. Adjuk meg azon tanárok nevét, akik több, mint 1 tanárgyat tanítanak.
j. Számítsuk ki minden diák esetén, minden tantárgynál kapott átlag jegyet, egy paraméter által adott félévben. Az eredményt egy ATLAG nevű táblában tároljuk, melynek szerkezete: (DiakID, TgyID, Felev, Atlag). Vegyük figyelembe, hogy olyan tantágyaknál, ahol van félévi dolgozat, az átlag egy negyedét a félévi teszi ki.
k. Adjuk meg a 12.-es osztályok átlagát a következőképpen: (XII A, 8.5); (XII B, 8.3), (XII C, 8.4) ..

l. Adjuk meg azon osztályok esetén, akiket Szász Róbert nevű tanár tanít matematikára, az osztályban elért legnagyobb és legkisebb matematika átlag jegyet!

6.5. Adott a következő relációs modellben leírt adatbázis, mely autóalkatrészeket áruló cég részleges adatbázisa. A cégnek több városban is van egy-egy üzlete.7

```
Város (VárosID, VNév, ManSzemSzám);
AlkatrészCsoportok (CsopID, CsNév);
     (Név lehetséges értékei: ablaktörlők, gumiabroncsok,
          akkumulátorok, stb.)
Alkatrészek(AlkatrészID, ANév, MértEgys, CsopID);
     1 alkatrész esetén: Erték = MennyiségRaktaron*EladásirAr
Uzletek (VárosID, AlkatrészID, MennyiségRaktáron, EladásiAr)
Szállítók (SzállID, SzNév, Helység, UtcaSzám);
Szállít (SzállID, AlkatreszID, Datum, SzállításiAr);
```

Oldjuk meg SQL parancs segítségével a következő feladatokat:

Az 5-ös fejezet 5.5. gyakorlatának a.-d. lekérdezéseit.

e. Adjuk meg azon alkatrészeket, melyek csak egy üzletben találhatók raktáron a következő formában:
   (ANév, VNév, MennyiségRaktáron formájában)
f. Adjuk meg a kolozsvári üzletben található alkatrészek értékét csoportokon belül a következő formában: CsNév, Erték
g. Adjuk meg azon szállítók nevét, kik pontosan 3 féle alkatrészcsoportot szállítanak!
h. Adjuk meg minden akkumulátor esetén (CsNév = 'akkumulátorok') a különböző szállítók ajánlatainak az átlagát. Minden ajánlat esetén csak a legutolsót vegyük figyelembe!
i. Adjuk meg minden üzlet esetén annak az alkatrészcsoportnak a nevét, melyből a legnagyobb értékben található alkatrész az illető üzletben!
   (VNév, CsNév, Erték formájában)

6.6. Adott a következő relációs modellben leírt adatbázis, mely egy gyógyszertár részleges adatbázisa. Egy gyógyszernek van egy alapanyaga, például a Panadol (GyNév) alapanyaga a Paracetamol (AlapanyagNév), származási hely Franciaország. Az alapanyag váltja ki a gyógyszer hatását. Egy alapanyagnak több hatása is lehet. A Paracetamolnak van lázcsökkentő, gyuladáscsökkentő, fájdalomcsillapító. Egyebek csoportjai lehetnek: fogpaszták, szappanok, pelenkák, teák, stb., például a Colgate Herbal egy fogpaszta. A recept esetén a százalék az jelenti, hogy hány százalékos a kedvezmény.

```
AlapAnyagok (AlapAnyagID, AlapanyagNév)
Hatások (HatásID, HatásNév)
AnyagHatások (AlapAnyagID, HatásID)
Országok (OrszagKod, OrszagNév)
Gyógyszerek (GyID, GyNév, AlapAnyagID, SzármazásiOrszágKod,
      ErvényességiDátum, MértEgység, MennyRakt, EladásiAr)
EgyebekCsoportok (CsopID, CsopNév)
Egyebek (EgyID, EgyNév, CsopID, MennyRakt, EladásiAr)
Betegek (BetegSzemSzám, BetegNév, BetegCím)
Receptek (ReceptID, BetegSzemSzám, Dátum, Százalék)
Receptek (ReceptID, GyogyszID, Menny)
```

Oldjuk meg SQL nyelv segítségével a következő feladatokat:

a. Adjuk meg a gyúladáscsökkentő gyógyszerek nevét!
b. Adjuk meg azon gyógyszerek nevét, melyek hatása csak fájdalomcsillapítás!
c. Adjuk meg a legolcsóbb vérnyomáscsökkentő gyógyszer nevét, melyből van raktáron!
d. Adjuk meg az egyebek értékét csoportokon belül a következő formában:
   CsNev, Ertek (1 termék értéke = MennyRakt * EladásiAr)
e. Adjuk meg azon gyógyszerek nevét, melyeknek csak egy hatásuk van!
f. Adjuk meg a receptek értékét egy adott napon!

g. Adjuk meg adott beteg, receptjeinek összértékét!
h. Adjuk meg a 100%-ál kisebb receptek értékét egy adott napon!
i. Adjuk meg azon beteg nevét, akié az utóbbi 2 hét legnagyobb értékű receptje!
j. Töröljük az adatbázisból azon gyógyszereket, melyek 1 héten belül lejárnak!
k. Adjuk meg az egyebek esetén annak a csoportnak a nevét, melyből a legnagyobb értékben található áru a gyógyszertárban! (CsNev, Ertek formájában)

6.7. Adott a következő relációs modellben leírt adatbázis, mely készruhát áruló cég részleges adatbázisa. A cégnek több üzlete is van.

```
Uzletek (UzletID, UNev, ManSzemSzám);
RuhaCsop (CsopID, CsNev);
   (Nev lehetséges értékei: nadrágok, szoknyák, télikabát, stb.)
Ruhak (RuhaID, RNev, CsopID);
Raktar (UzletID, RuhaID, MennyiségRaktaron, EladasiAr)
          1 ruha esetén Ertek = MennyiségRaktaron*EladasiAr
Szallitok (SzallID, SzNev, Helyseg, UtcaSzam);
Szallit (SzallID, RuhaID, Datum, SzallitasiAr);
```

Oldjuk meg SQL nyelv segítségével a következő feladatokat:
a. Adjuk meg az Alexis nevű üzletben található ruhák nevét!
b. Adjuk meg azon szállítók nevét, akik csak nadrágot szállítanak!
c. Adjuk meg a legolcsóbb esőkabátot, melyből van raktáron!
   (UNev, RNev, MennyiségRaktaron formájában)
d. Adjuk meg a Stefanis nevű üzletben található ruhák értékét csoportokon belül a következő formában: CsNev, Ertek
e. Adjuk meg azon szállítók nevét, kik pontosan 5 féle ruha csoportot szállítanak!
f. Adjuk meg minden télikabát esetén (CsNev = 'télikabát') a különböző szállítók ajánlatainak az átlagát. Egy szállító egy télikabát ajánlatai esetén csak a legutolsót (időben) vegyük figyelembe!
g. Adjuk meg minden üzlet esetén annak a ruha csoportnak a nevét, melyből a legnagyobb értékben található ruha az illető üzletben! (UNev, CsNev, Ertek formájában)
h. Irjunk tárolt eljárást, melynek bemenő paramétere egy *n* egész szám és egy *csID* árúcsoportID. Az eljárás szolgáltassa a megadott árucsoportból a frissebb *n* áruajánlatokat a következő formában:
   SzNév, RNév, Datum, SzállitásiAr.
i. Irjunk tárolt eljárást, melynek bemenő paraméterei:
   - egy *csIDk* karaktersor, melyben árúcsoportID-k pontosvesszővel elválasztva találhatóak,
   - *l* egész szám, a karaktersor hosszát adja meg;
   - *KezdDat* egy kezdeti dátum;
   - *VegsDat* egy végső dátum;

   Az eljárás szolgáltassa a megadott árucsoportok *KezdDat* és *VegsDat* közötti áruajánlatait a következő formában:
   CsNév, SzNév, RNév, Datum, SzállitásiAr.

7. Nézettáblák

Az ANSI-SPARC architektúra külső szintje nézetekből áll, különböző felhasználók különböző nézetét látják az adatbázisnak. A nézettábla egy ablaka az adatoknak. Ha a nézettábla módosítható (lásd később 7.3.1.-ben) és az ablakban változtatunk, az ABKR elvégzi a változtatást a megfelelő alaptáblában.

7.1. Nézettábla értelmezése

Nézettábla értelmezése a következő parancs segítségével történik:

```
CREATE VIEW <nézet_név>
  AS < SELECT _SQL_parancs> [WITH CHECK OPTION];
```

A SELECT parancs lehet komplex, de nem tartalmazhat ORDER BY záradékot.

7.1. példa: Legyenek a következő relációk:

```
Alkalmazottak (SzemSzám, ANév, Cím, Fizetés, RészlegID)
Részlegek (RészlegID, RNév, Helység, ManSzemSzám);
```

értelmezzük a következő nézetet:

```
CREATE VIEW Részleg10 AS
          SELECT SzemSzám, ANév, Fizetés
          FROM Alkalmazottak
          WHERE RészlegID = 10;
```

A Részleg10 nézet, csak a 10-es részleg alkalmazottairól szóló információt tartalmazza. □

A nézettábla eredménye is egy reláció, melyet származtatott (derived) relációnak nevezünk. A rendszer csak a nézettábla értelmezését tárolja, ahányszor szüksége van rá, végrehajtja a Select SQL parancsot a nézettábla értelmezéséből, a nézettábla eredményét nem tárolja, ezért a nézettábla csak egy virtuális reláció.

7.2. Mire használhatjuk a nézeteket

- Korlátozzuk a hozzáférést az adatbázishoz;
- A felhasználóknak jogokat a nézetekre adunk és nem az alaptáblákra;
- Komplex lekérdezéseket egyszerűbbé tenni (makro lehetõség);
- Biztosíthatjuk az adatfüggetlenséget;
- Különböző nézetét mutassuk ugyanannak az adatnak.

A felhasználó tekintheti úgy a nézetet, mint egy reláció. Egy lekérdezés vonatkozhat nézetre.

7.2. példa: Legyen a következő reláció:

```
Szállítók (SzállID, Név, Helység, UtcaSzám, Hihetőség);
```

és a következő nézettábla:

```
CREATE VIEW Jo_Száll AS
     SELECT * FROM Szállítók
     WHERE Hihetőség > 7;
```

A Jo_Szall nézet csak azokat a szállítókat tartalmazza, akiknek a hihetősége nagyobb, mint 7. □

7.3. példa: Legyen a következő lekérdezés:

```
SELECT Név FROM Jo_Száll
WHERE Helység = 'Kolozsvar'
```

A rendszer, a lekérdezés végrehajtása esetén, a nézettábla nevét helyettesíti az értelmezésével. A lekérdezés relációs algebrai műveletek segítségével kifejezve:

$$\pi_{\text{Név}}(\sigma_{\text{Helység = 'Kolozsvar'}}(\text{Jo\_Száll}))$$

a következő lépésben:

$$\pi_{\text{Név}}(\sigma_{\text{Helység = 'Kolozsvar'}}(\sigma_{\text{Hihetoseg}>7}(\text{Szállítók}))).$$

Egy nézet értelmezését a `CREATE OR REPLACE VIEW` segítségével módosíthatjuk. Például a Részleg10 nézet értelmezése esetén új nevet adunk minden oszlopnak.

```
CREATE OR REPLACE VIEW Reszleg10 AS
          (AlkalmSzemelyiSzam, AlkalmazottNev, AlkalmazottFizetes)
          SELECT SzemSzam, ANev, Fizetes
          FROM Alkalmazottak
          WHERE ReszlegID = 10;
```

Az oszlopok nevei a `CREATE VIEW`-ban ugyanabban a sorrendben szerepelnek, mint az oszlopok a megfelelő alkérdésben .

7.4. példa: Tekintsük a 7.1. példa Alkalmazottak és Részlegek tábláját. Ez egy olyan komplex nézet létrehozására szolgáló példa, mely csoportosítást használ és a két táblára vonatkozik:

```
CREATE VIEW ReszlOszz
          (RészlegNév, minfiz, maxfiz, avgfiz)
     AS SELECT r.RNév, MIN(a.Fizetes), MAX(a.Fizetes), AVG(a.Fizetes)
          FROM Alkalmazottak a, Részlegek r
          WHERE a. RészlegID = r. RészlegID
          GROUP BY r.RNev;
```

A ReszlOssz nézet minden részleg esetén tartalmazza a részleg nevét, a minimális fizetést a részlegből, a maximális és az átlagfizetést. □

A WITH CHECK OPTION szerepe: ha a nézet módosítható és INSERT vagy UPDATE parancsot hajtunk végre a nézeten, a rendszer ellenőrzi, hogy csak olyan sor kerüljön a nézettáblába, mely eleget tesz a nézettáblát értelmező feltételnek. Ha ezt az opciót nem írjuk oda, az alaprelációba bekerülhet a sor, de a nézettáblába nem fog látszani. Így például a Jo_Száll nézetbe, mivel nem írtuk oda, hogy WITH CHECK OPTION, a felhasználó be tud gyűjteni olyan Szállítót, melynek a hihetősége 5. Logikailag ez nem helyes, ha a nézet módosítható, ajánlott odaírni a WITH CHECK OPTION-t.

A nézetek egyik feladata a *logikai adatfüggetlenséget biztosítani*, vagyis a felhasználó ne érezze, ha a fogalmi adatbázis szerkezete változik, vagyis ha bővül vagy átszerveződik.

- A fogalmi adatbázis bővül, ha egy adott reláció új attribútumokkal bővül. Például a

  Vevők (VevőID, Név, Helység, UtcaSzám, Mérleg)

  tábla esetén telefonszám, bankszámla, faxszám lesznek az új attribútumok. Azon felhasználóknak, kiket nem érdekelnek az új attribútumok, értelmeznek egy Vevők nézetet, a felhasználó alkalmazása marad a régi. A Vevők táblát helyettesítjük egy másikkal, például Vevő névvel, azok a felhasználók, kiket érdekelnek az új információk amúgy is meg kell változtassák az alkalmazásukat.
- Ha a fogalmi adatbázis új táblákkal bővül, nem befolyásolja a létező programokat, nem érinti a nézeteket.
- A fogalmi adatbázis átszerveződhet, egy relációt kettővel helyettesítünk valamilyen oknál fogva, esetleg különböző csomopontokban tároljuk őket osztott adatbázis esetén. Például a Vevők táblát felbontjuk:

  ```
  Vevő [VevőID, V_Nev, V_Helyseg, V_UtcaSzám]
  Vevőh [VevőID, Hihetoseg]
  ```

  Az eredeti táblát visszakaphatjuk a két táblából join művelet segítségével.

  ```
  CREATE VIEW Vevők AS
       SELECT * FROM Vevő, Vevőh
       WHERE Vevő.VevőID = Vevőh.VevőID
  ```

Minden művelet, mely eddig a Vevők alaprelációra hivatkozott, most a Vevők nézetre fog hivatkozni. Ha a rendszer tud helyesen dolgozni a nézetekkel, a felhasználó nem érzi meg az adatok átszervezését.

A nézetek más előnyei, hogy megengedik, hogy ugyanazt *az adatot*, különböző felhasználó egy időben *különbözőképpen lássa*. Így a felhasználó összpontosíthat az adatbázisnak csak arra a részére, mely érdekli, a többit nem kell figyelembe vegye.

A nézet egy *macro lehetőség*.

7.5. példa: Legyenek a következő relációk:

```
Személyek (SzemSzám, VNév, KNév, Város, UtcaSzám, TelefonSzám)
Alkalmazottak (AlkID, SzemSzám, RészlegID, KezdésIdeje, MunkaIdo)
SzállKapcsolatEmberek (SzKapcsID, SzemSzám, SzállID)
VevőKapcsolatEmberek (VKapcsID, SzemSzám, VevőID)
```

(A gyakorlatban ezeknek a relációknak még sokkal több attribútumuk van.)

Az alkalmazottak, szállítók, illetve vevők kapcsolat emberei mind személyek, vannak közös attribútumaik, azokat a Személyek táblában tároljuk, ami pedig specifikus a megfelelő táblában, és ezekből a SzemSzám segítségével a Személyek táblára hivatkozunk. A lekérdezések esetében az alkalmazott neve a legtöbb esetben szükséges. Értelmezzük a következő nézetet:

```
CREATE VIEW AlkSzem AS
       SELECT Alkalmazottak.AlkID, Személyek.VNév, Személyek.KNév,
              Személyek.SzemSzám, Személyek.Város,
              Személyek.UtcaSzám, Személyek.TelefonSzám,
              Alkalmazottak.RészlegID, Alkalmazottak.KezdésIdeje,
              Alkalmazottak.MunkaIdo
         FROM Alkalmazottak, Személyek
         WHERE Alkalmazottak.SzemSzám = Személyek.SzemSzám
```

Ha azon alkalmazottakat keressük, akik 9 óra előtt kezdenek, akkor használhatjuk a fenti nézetet a következő lekérdezésben:

```
SELECT VNév, KNév FROM AlkSzem WHERE KezdésIdeje < 9.
```

További nézetek:

```
CREATE VIEW SzállKapcsEmbSzem AS
     SELECT SzállKapcsolatEmberek.SzKapcsID,
          Személyek.VNév,Személyek.KNév, Személyek.SzemSzám,
          Személyek.Város, Személyek.UtcaSzám,
          Személyek.TelefonSzám, SzállKapcsolatEmberek.SzállID,
          Szállítók.Név, Szállítók.Helység, Szállítók.UtcaSzám,
          Szállítók.Hihetoseg
     FROM SzállKapcsolatEmberek, Személyek, Szállítók
     WHERE SzállKapcsolatEmberek.SzemSzám = Személyek.SzemSzám
          AND SzállKapcsolatEmberek.SzállID = Szállítók.SzállID
```

A SzállKapcsEmbSzem nézet a szállító kapcsolat emberének a nevét és a szállító nevét is tartalmazza, mivel ez a lekérdezésekben a legtöbb esetben szükséges.

```
CREATE VIEW VevőKapcsEmbSzem AS
          SELECT VevőKapcsolatEmberek.VKapcsID, Személyek.VNév,
              Személyek.KNév, Személyek.SzemSzám, Személyek.Város,
              Személyek.UtcaSzám, Személyek.TelefonSzám,
              VevőKapcsolatEmberek.VevőID, Vevők.Név, Vevők.Helység,
              Vevők.UtcaSzám, Vevők.Hihetoseg
          FROM VevőKapcsolatEmberek, Személyek, Vevők
          WHERE VevőKapcsolatEmberek.SzemSzám = Személyek.SzemSzám
               AND VevőKapcsolatEmberek.VevőID = Vevők.VevőID
```

A VevőKapcsEmbSzem nézet tartalmazza a vevő kapcsolat emberének a nevét és a vevő nevét is, mivel ez a legtöbb esetben szükséges a lekérdezésekben.

Keressük a kolozsvári szállítók kapcsolat embereit:

```
SELECT Név AS SzállNév, VNév, KNév
FROM SzállKapcsolatEmberek
WHERE Helység = 'Kolozsvar'
```

A nézet értelmezését egyszer kell megírjuk és nagyon sok lekérdezésben használhatjuk, így a lekérdezések egyszerűbbek lesznek. □

A nézetek *automatikus biztonságot* jelentenek a rejtett adatoknak. Rejtett az az attribútum, mely egyetlen nézetből sem elérhető. Ha a rendszer csak nézeteken keresztül engedi a felhasználót, hogy hozzáférjen az adatbázishoz, akkor ez automatikus biztonságot ad. A nézet összekapcsolja a külső sémát a fogalmi sémával, olyan értelemben, hogy megadja, a külső szinti objektum (nézet), miként vetítődik le a fogalmi sémára.

Egy nézetet el lehet törölni a

```
DROP VIEW <nézet_név> [RESTRICTED ! CASCADE]
```

parancs segítségével. Ahogy a DROP TABLE hibát jelez, ha egy nézet hivatkozik a táblára, amit el akarunk törölni, így a DROP VIEW is, ha egy másik nézet a törlendő nézetre hivatkozik és a RESTRICTED opció van jelen. Ha a CASCADE opciót írjuk, törli az összes nézetet, amely a törlendő nézetre hivatkozik.

7.6. példa: A Részleg10 értelmezését a következő parancs segítségével törölhetjük:

```
DROP VIEW Részleg10;
```

7.3. Adatkezelési műveletek végrehajtása egy nézeten

Adatkezelési műveletek (INSERT, UPDATE, DELETE) nem végezhetőek akármilyen nézeten.

7.3.1. SQL2 feltételei a módosítható nézetekre

Az SQL2 olyan nézeten fogad el adatkezelési műveletet, mely egy reláción van értelmezve, melyre esetleg szelekciót vagy vetítést alkalmazhatunk. Egy nézet módosítható (updatable view), ha eleget tesz a következő feltételeknek:

1. A SELECT SQL parancs, amely segítségével a nézettáblát értelmezzük, FROM záradékában csak egy reláció szerepelhet, mely alapreláció vagy más módosítható nézet lehet.
2. A SELECT SQL parancs, mely segítségével a nézettáblát értelmezzük, nem tartalmazhatja a JOIN, UNION, INTERSECT, EXCEPT kulcsszavakat.
3. A SELECT SQL parancs nem tartalmazhat DISTINCT kulcsszót.
4. A SELECT SQL parancs -ban csak a tábla attribútumai szerepelhetnek, esetleg más névvel (AS segítségével), de SUM, MAX, MIN, COUNT, AVG nem.
5. A SELECT SQL parancs nem tartalmazhat csoportosítást (GROUP BY és HAVING-et).

7.3.2. Az Oracle feltételei a módosítható nézetekre

Az Oracle is hasonló feltételek mellett tud adatkezelési műveleteket végezni, de vannak bizonyos különbségek is. A következő táblázat az egyszerű és komplex nézetek közötti különbségeket szemlélteti.

Jellemvonás	*Egyszerű nézetek*	*Komplex nézetek*
Táblák száma	Egy	Egy vagy több
Tartalmaz függvényt	Nem	Igen
Tartalmaz csoportosítást	Nem	Igen
DML nézeten keresztül	Igen	Nem minden esetben

Egyszerű nézeteken (csak egy táblára vonatkozik, nem tartalmaz függvényt és csoportosítást sem) végezhetünk adatkezelési műveleteket.

– Nem törölhetünk sorokat egy nézetből, ha tartalmaz:
  - összesítő függvényt
  - GROUP BY záradékot
  - DISTINCT kulcsszót.
– Nem módosíthatjuk a nézet adatait, ha a nézet tartalmaz:
  - a törlésnél felsorolt feltételeket;
  - oszlopokat, melyek kifejezésként vannak értelmezve;
  - ROWNUM pseudooszlopot.
– Nem tudunk adatokat beszúrni, ha a nézet tartalmaz:
  - a módosításnál felsorolt feltételeket;
  - vannak olyan NOT NULL oszlopok az alaptáblákban (DEFAULT nélkül), melyek nincsenek kiválasztva a nézetben.

Ha egy nézet két tábla join-jának az eredménye, (join nézet), a nézet módosíthatósága kérdéses. Bármely UPDATE, INSERT, DELETE *művelet egy join nézeten, csak egy alaptáblát tud módosítani*.

Egy reláció *kulcsőrző* (*key-preseved*), ha minden kulcsa a táblának a join eredményének is kulcsa lesz. Tehát egy kulcsőrző tábla kulcsai megőrződnek a join után. A nézeten végzett adatkezelési műveletek a kulcsőrző táblán hajtódnak végre.

7.7. példa: Ha adottak a következő relációk:

```
Alkalmazottak (SzemSzám, ANév, Cím, Fizetés, RészlegID)
Részlegek (RészlegID, RNév, Helység, ManSzemSzám);
```

legyen a következő nézet:

```
CREATE VIEW Alk_Reszl_Nezet AS
     SELECT a.SzemSzem, a.ANév, a.RészlegID, a.Fizetés, r.RNév, r.Helység
          FROM Alkalmazottak a, Részlegek r
          WHERE a. RészlegID = r. RészlegID
               AND r.Helység IN ('Kolozsvar', 'Varad', 'Temesvar')
```

Az Alkalmazottak tábla kulcsőrző, a Részlegek nem. Legyen a következő módosítási művelet:

```
UPDATE Alk_Reszl_Nezet
   SET ANév = 'Kovacs'
   WHERE ANév = 'Szabo'
```

A művelet módosítja az Alkalmazottak táblában az összes 'Szabo' nevű alkalmazott nevét 'Kovacs'-ra.

Elfogadott parancsok:

```
UPDATE Alk_Reszl_Nezet
    SET Fizetés = Fizetés * 1.10
    WHERE RészlegID = 10;

 DELETE FROM Alk_Reszl_Nezet
    WHERE ANév = 'Szabo';

 INSERT INTO Alk_Reszl_Nezet (ANév, SzemSzám, RészlegID)
    VALUES ('Szentandrasi', 1600202125167, 20);
```

Mindhárom parancs hatása csak az Alkalmazottak táblát érinti és csak azt fogja módosítani.

Nem megengedett parancsok:

```
UPDATE Alk_Reszl_Nezet
  SET Helység = 'Varad'
   WHERE ANév = 'Szabo';
```

A Helység attribútum a Részlegek tábla attribútuma. Lehet több Szabo nevű alkalmazott különböző részlegeknél, amelyek különböző helységekben vannak. Az ABKR nem változtathatja meg az érintett részlegek Helység attribútumát.

```
INSERT INTO Alk_Reszl_Nezet (ANév, SzemSzám, RNév)
   VALUES ('Szentandrasi', 1600202125167, 'Tervezes');
```

Az RNév attribútum a Részlegek tábla attribútuma, mely nem kulcsőrző, így az illesztés nem megengedett. □

Ha egy kulcsőrző táblát önmagával join-olunk WITH CHECK OPTION záradékkal, nem törölhetünk sorokat belőle.

7.8. példa: Legyen a következő nézet, mely az azonos című alkalmazott párt tartalmazza:

```
CREATE VIEW AlkPárAzonosCimen
   AS SELECT Alk1.ANév AS Név1, Alk2.ANév AS Név2
     FROM Alkalmazottak AS Alk1, Alkalmazottak AS Alk2
     WHERE Alk1.Cím = Alk2.Cím
               AND Alk1.ANév < Alk2.Anév
                  WITH CHECK OPTION;
```

Nem törölhetünk sorokat belőle, mivel két sorra is vonatkozik és az adatkezelési műveletek csak egy táblára vonatkozhatnak. □

A rendszer *visszautasítja az adatkezelési műveleteket* WITH READ ONLY záradék segítségével. Így biztosíthatjuk, hogy a nézeten semmiféle adatkezelési műveletet se lehessen végezni.

7.9. példa: A Részleg10 nézetet lehet csak olvasásra értelmezni:

```
CREATE VIEW Részleg10 AS
        SELECT SzemSzám, ANév, Fizetés
            FROM Alkalmazottak
          WHERE RészlegID = 10
          WITH READ ONLY; □
```

Ha a külcsőrző táblák adta lehetőség a nézet módosítására nem felel meg, az Oracle a triggerek mechanizmusát is felhasználja erre (lásd 10. fejezetet a triggerekről).

- Oracle INSTEAD OF Triggert csak nézetekre engedélyez.
- Oracle nézetekre csak INSTEAD OF Triggert engedélyez, (AFTER, BEFORE nem)

7.10. példa: Legyenek a fenti példából az Alkalmazottak és Részlegek relációk és a következő nézet, mely minden részleg esetén a manager nevét is tartalmazza:

```
CREATE VIEW manager_info AS
       SELECT r. RészlegID, r. RNév, a. SzemSzám, a.ANév
         FROM Alkalmazottak a, Részlegek r
         WHERE  a. SzemSzám =  r. ManSzemSzám;
```

Ez a nézet esetén a Részlegek a kulcsőrző reláció, az ABKR csak olyan adatkezelési műveletet fogad el, mely a Részlegek táblán elvégezhető. Beszúrás esetén megadhatjuk a RészlegID-t, RNév-et és a manager SzemSzám-át, a nevét viszont nem, mert az az Alkalmazottak táblában van tárolva. Viszont használhatunk egy triggert és ennek a segítségével a manager nevét is beszúrjuk az Alkalmazottak táblába.

```
CREATE TRIGGER manager_insert
      INSTEAD OF INSERT ON manager_info
      REFERENCING NEW AS n  -- új manager információi
      FOR EACH ROW
    DECLARE AlkSzam NUMBER;
         ReszlegSzam NUMBER;
    BEGIN
         /* Ellenőrizzük, hogy az Alkalmazottak táblában létezik-e az új
```

```
           manager */
     SELECT COUNT(*) INTO AlkSzam
          FROM Alkalmazottak a
           WHERE SzemSzam = :n. SzemSzam;   /* (new.SzemSzám) */
     IF AlkSzam = 0 THEN/* a manager nem létezett az Alkalmazottak
               táblában  ezért beszúrjuk az Alkalmazottak táblába, azzal a
               feltevéssel, hogy a ki nem töltött mezők (Cím, Fizetés)
               értéke lehet NULL vagy van DEFAULT értékük */
          INSERT INTO Alkalmazottak (SzemSzam, ANev, ReszlegID)
               VALUES (:n.SzemSzam, :n.ANev, :n.ReszlegID);
     END IF;
     SELECT COUNT(*) INTO ReszlegSzam
          FROM Reszlegek
          WHERE ReszlegID = :n.ReszlegID;
     IF ReszlegSzam = 0 THEN
               /* A beszúrt új sor részlege nem létezik a Reszlegek
                táblába, ezért beszúrjuk, azzal a feltevéssel, hogy a
                Helység mező lehet NULL */
          INSERT INTO Reszlegek (ReszlegID, RNev, ManSzemSzam)
               VALUES (:n.ReszlegID, :n.RNev, :n.SzemSzam);
     END IF;
     END;
     /* ha létezett az új RészlegID a Részlegek táblában, a manager
SzemSzam-a is az Alkalmazottak között nem történik semmi a beszúrás
eredményeként. Előfordulhat, hogy az új sor esetén más név tartozzon egy
már létező személyi számhoz, ebben az esetben sem történik semmi. Esetleg
ezt is le lehet kezelni. */
```

Egy más lehetőség ugyanarra az INSERT triggerre:

```
CREATE TRIGGER manager_info_insert
  INSTEAD OF INSERT ON manager_info
  REFERENCING NEW AS n  -- új manager információi
  FOR EACH ROW
DECLARE AlkSzam NUMBER;
BEGIN     /* Ellenőrizzük, hogy az alkalmazottak száma nagyobb, mint
          1 a  részlegben, ahova a nézeten keresztül egy új managert
          akarunk beszúrni */
  SELECT COUNT(*) INTO AlkSzam
     FROM Alkalmazottak e
     WHERE  e.  RészlegID  =  :n.  RészlegID;      /*  ha  van  elég
         alkalmazott a részlegben,  akkor megtesszük managernek az
         újonan beszúrt managert (new.SzemSzám) */
  IF AlkSzam>= 1 THEN
     UPDATE Részlegek r
          SET ManSzemSzám = :n.SzemSzám
          WHERE r. RészlegID = :n. RészlegID;
 END IF;
END;
```

8. Adatbázisok biztonsága

Egy alkalmazás fejlesztése esetén a biztonság szempontjából a következő célkitűzéseket kell figyelembe venni:
1. *Titoktartás* (Secrecy). Olyan felhasználó, akinek nincs joga, ne férjen hozzá az információkhoz, például egy diák ne láthassa más diák kreditjeit.
2. *Sértetlenség* (Integrity). Csak olyan felhasználó módosíthassa az adatokat, akinek joga van hozzá, például a diák láthatja a saját kreditjeit, de ne változtathassa meg.
3. *Hozzáférhetőség* (Availability). Feljogosított felhasználó ne kapjon olyan üzenetet, hogy nem férhet hozzá az adatbázishoz, például a titkarnő, aki felelős a kreditekért meg tudja változtatni azokat, ne kapjon olyan üzenetet, hogy nincs joga a módosításra.

Ki kell dolgozni egy *biztonsági politikát* (security policy): az adatok, mely részét kell levédeni, ki férhet hozzá olvasás, módosítás céljából. A következő lépésben egy *biztonsági mechanizmust* kell kidolgozni. Emberi tényezők is közrejátszanak, egyesek túl egyszerű parolát választanak, mások elárulják a parolájukat. Az ABKR az operációs rendszerre támaszkodik, ha az nem elég biztonságos (például valaki megszerzi a rendszergazda paroláját, azt tehet az adatbázisban amit akar), az ABKR sem az.

8.1. Hozzáférés ellenőrzése (access control)

Két típusú hozzáférés ellenőrzést dolgoztak ki:
a) *a tetszés szerinti hozzáférés ellenőrzést* (discretionary access control);
b) *a meghatalmozott hozzáférés ellenőrzést* (mandatory access control).

8.1.1. Tetszés szerinti hozzáférés ellenőrzés

Ez a típusú hozzáférés ellenőrzés hozzáférési jogosultságokon alapszik és egy mechanizmuson, mely ezen jogosultságokat a felhasználóknak (*user*) vagy a szerepköröknek (*role*) osztogatja.

Ha több felhasználónak ugyanazokra a jogosultságokra van szüksége, szerepköröket hozhatunk létre és a jogosultságot csak a csoport szintjén kell kiosztani. A továbbiakban a felhasználóra hivatkozunk, de hasonlóan járunk el a szerepkörök esetén is. A jogosultság megengedi, hogy a felhasználó hozzáférhessen bizonyos adatokhoz egy bizonyos módon (olvassa, változtassa).

A felhasználó által létrehozott adatbázis objektumok (például táblák, nézetek, tárolt eljárások stb.) a felhasználó *sémá*jába kerülnek. Az a felhasználó, amely egy adatbázis objektumot hoz létre, automatikusan megvan minden joga neki felette. Az ABKR rendszer nyilvántartja a különböző felhasználók jogait az adatbázis különböző része felett és biztosítja, hogy ezeket be is tartsák. Adatbázis objektumot csak az hozhat létre, aki a megfelelő joggal rendelkezik. Különböző objektum létrehozásában különböző rendszerjogosultságra van szükség. Minden ABKR másképp valósítja meg a rendszerjogosultságokat (lásd 8.4-ben Oracle esetén).

SQL2-ben a következő paranccsal adhatunk jogosultságokat:

```
GRANT <jogosultság> ON <objektum> TO <fehasználó>
    [WITH GRANT OPTION];
```

– <objektum> − egy alapreláció vagy egy nézettábla
– <jogosultság> − lehetséges értékei:
  - SELECT − olvashatja az összes attribútumot az adott relációból, azokat is amelyeket utólag ALTER TABLE segítségével illesztettünk hozzá.
  - INSERT (<oszlop_név>) − az a jogosultság, hogy új sorokat vihet be a táblába a megadott oszlopnév esetén. Ha az összes oszlopra akarunk hozzáillesztési jogot adni, csak INSERT, oszlopnév nélkül.
  - UPDATE (<oszlop_név>) − az a jogosultság, hogy módosíthat egy oszlopot vagy az összeset, az INSERT-hez hasonlóan
  - DELETE − a megadott táblából sorokat törölhet ki.

- REFERENCES (<oszlop_név>) − azon jogosultság, hogy hivatkozhat más táblára egy épségi megszorítási feltételben. Egy megszorítás lehet önálló, attribútum alapú vagy hivatkozási épséget ellenőrző megszorítás. Egy megszorítás csak akkor ellenőrizhető, ha az ellenőrzéshez szükséges összes adatbáziselemre megvan a REFERENCES jogosultság.

– WITH GRANT OPTION: ha egy felhasználónak olyan jogosultsága van, melyet ezzel az opcióval kapott, tovább adhatja más felhasználónak a GRANT parancs segítségével.

Egy felhasználó, mely létrehoz egy alaprelációt, az összes jogosultsága megvan felette és az illető relációra más felhasználónak is adhat jogosultságokat.

Ahhoz, hogy egy felhasználó létrehozzon egy nézetet, SELECT jogosultsága kell legyen azon táblákon, melyek a nézetben szerepelnek, és így neki is lesz SELECT jogosultság a nézeten. Tovább adhatja a jogosultságokat a nézetre más felhasználónak, ha az alaprelációkon, amire a nézet alapszik volt neki ilyen jogosultsága (vagy ő hozta létre, vagy WITH GRANT OPTION-nal kapta meg a megfelelő jogosultságokat). Ha a nézet módosítható, a felhasználónak van INSERT, DELETE vagy UPDATE jogosultsága az alapreláción, akkor ugyanazon jogosultságai meglesznek a nézeten is.

A CREATE, ALTER és DROP parancsot csak a séma tulajdonosa adhatja ki, ezeket nem lehet GRANT paranccsal megadni vagy REVOKE paranccsal visszavonni.

A nézet egy fontos komponens a biztonsági mechanizmusban. Ha az alaprelációkra nézeteket értelmezünk, ezekbe belefoglaljuk azon oszlopokat, amelyre a felhasználónak szüksége van. A többi elrejtve marad a felhasználó elől.

8.1. példa: Legyen „Zoli” egy felhasználó (vagy egy felhasználó csoport). „Zoli” hozta létre az összes alaprelációt, mondjuk a NagyKer adatbázis esetén (lásd 6.1. példa). „Zoli” adja ki a következő parancsokat.

```
CREATE VIEW Kol_Száll AS
          SELECT SzállID, Név, Hihetőség
          FROM Szállítók
          WHERE Szállítók.Helység = 'Kolozsvar';

GRANT SELECT, INSERT, UPDATE (Név, Hihetőség), DELETE
          ON Kol_Száll TO Arpi, Csaba WITH GRANT OPTION;

GRANT SELECT ON Szállítók, Szállít TO Mihály WITH GRANT OPTION;

GRANT UPDATE (Hihetőség) ON Szállítók TO Eva;
```

Mihály kiadhatja:

```
GRANT SELECT ON Szállítók TO Ildikó;
```

Ha Eva a következő parancsot adja ki:

```
UPDATE  Szállítók S
          SET S.SzállID = 8899
          WHERE S.Név = 'Word Trade';
```

hibaüzenetet kap, mert nincs joga csak a Hihetőség mezőt módosítani. Mihály és Ildikó nem adhat ki UPDATE parancsot. □

8.2. példa.(REFERENCES): Ha Zoli hozta létre az Áruk táblát, Balázs hozta létre a Szállít táblát. Zoli ad REFERENCES jogot:

```
GRANT REFERENCES (ÁruID) ON Áruk To Balázs;
```

Ezek után Balázs az ÁruID-t külső kulcsnak deklarálhatja, mely hivatkozik az Áruk-ra. Ha Balázsnak az Áruk táblából csak az ÁruID attribútumra lenne SELECT jogosultsága, de REFERENCES jogosultsága nincs, nem adhatja ki a FOREIGN KEY hivatkozást. Ha Balázs elveszti a REFERENCES jogát, törlődik a külső kulcs deklaráció. □

A REVOKE parancs visszavonja a kiadott jogosultságokat, vagy azt, hogy a jogosultság átadható:

```
REVOKE [GRANT OPTION FOR] <jogosultság> ON <objektum>
     FROM <felhasználó> {RESTRICT | CASCADE};
```

A tábla vagy a nézet létrehozója, akinek minden jogosultsága megvan felette és adott másoknak is jogosultságot rá, visszavonhatja a kiadott jogosultságokat. A probléma akkor lép fel, ha egy felhasználó több más felhasználótól is ugyanolyan jogosultságot többször is kap.

Ha a REVOKE parancsot a CASCADE opcióval adjuk ki, ez visszavonja az összes felhasználónak kiadott jogot, melyet láncban átadott egyik felhasználó a másiknak.

8.3. példa: Zoli a következő parancsot adja ki:

```
GRANT SELECT ON Szállítók TO Mihály WITH GRANT OPTION;
```

Mihály a következő parancsot adja ki:

```
GRANT SELECT ON Szállítók TO Ildikó WITH GRANT OPTION;
```

Zoli a következő parancsot adja ki:

```
REVOKE SELECT ON Szállítók FROM Mihály CASCADE;
```

Mihály és Ildikó is elvesztik a jogosultságaikat. □

8.4. példa: Zoli a következő parancsot adja ki:

```
GRANT SELECT ON Szállítók TO Mihaly WITH GRANT OPTION;
```

Mihály a következő parancsot adja ki:

```
GRANT SELECT ON Szállítók TO Ildikó;
```

Zoli is ad ugyanolyan jogosultságot Ildikónak:

```
GRANT SELECT ON Szállítók TO Ildikó;
```

Zoli a következő parancsot adja ki:

```
REVOKE SELECT ON Szállítók FROM Mihály CASCADE;
```

Ebben az esetben Ildikónak megmarad a jogosultsága, mert Zolitól is megkapta. □

Ha ugyanazt a jogosultságot szórakozottságból kétszer is kiadjuk, elég egyszer visszavonni. Vissza lehet vonni csak a GRANT OPTION-t.

8.5. példa: Zoli a következő parancsot adja ki:

```
GRANT SELECT ON Szállítók TO Mihály WITH GRANT OPTION;
```

Zoli a következő parancsot adja ki:

```
REVOKE GRANT OPTION FOR SELECT ON Szállítók FROM Mihály CASCADE;
```

Mihálynak megmarad a SELECT joga a Szállítók táblán, de nem tudja továbbadni másnak. □

A RESTRICT opció esetén, ha a jogosultságok tovább voltak adva, a rendszer a REVOKE parancsot visszautasítja.
A 8.3. példa esetén, ha a REVOKE parancsot Zoli RESTRICT opcióval adná ki, azt a rendszer visszautasítaná.

8.6. példa: Ha Mihálynak van SELECT jogosultsága a Szállítók táblán, Zolitól kapta, és létrehozza a következő nézetet:

```
CREATE VIEW Jo_Szall AS
     SELECT * FROM Szállítók
     WHERE Szállítók.Hihetőség > 7;

GRANT SELECT ON Jo_Szall TO Lehel;
```

Lehel létrehozza a következő nézetet:

```
CREATE VIEW Jo_Kol_Sz AS
     SELECT * FROM Jo_Szall S
     WHERE S.Helység = 'Kolozsvar';
```

Ha Zoli visszavonja a jogot Mihálytól, hogy olvashassa a Jo_Szall nézetet, többet nem tudja használni azt, a rendszer eltörli a Jo_Szall értelmezését (DROP), hasonlóan a Jo_Kol_Sz-t is. Ha Zoli meggondolja magát és visszaadja Mihálynak a jogot, Mihály ismét létre kell hozza a nézetet.

Más helyzet: minden marad (Mihály létrehozza a Jo_Szall nézetet, ad rá jogot Lehelnek, Lehel létrehozza a Jo_Kol_Sz nézetet) kivéve mikor Zoli visszavonja a jogot Mihálytól, helyette INSERT jogot ad neki is a Szállítók táblára. Mivel a Jo_Szall nézet módosítható, Mihály a Jo_Szall nézetbe INSERT-tel sorokat tud beírni. Lehel már hiába kapja meg utólag ezt a jogot, mert a nézet létrehozása pillanatában még nem volt meg neki, esetleg újra létre kell hozza a nézetet.

8.1.2. Meghatalmozott hozzáférés ellenőrzés

Az előző módszer egyik gyenge pontja: Pl. Kása felhasználónak van joga a diákok kreditjeit módosítani. Balázs felhasználó kíváncsi a Kredit táblára, mit tehet: létrehoz Titkos névvel egy új táblát, Kása felhasználónak INSERT jogot ad a Titkos táblára. Megváltoztatja a Kása adatbázis alkalmazásának kódját, melyben megnyitja a Kredit táblát, olvassa, majd a Titkos táblába soronként átmásolja.

Ezt egy olyan alkalmazásba írja be, melyet Kása gyakran futtat. Balázs vár, míg a Titkos táblában meglesznek az adatai, majd az alkalmazásból kitörli a változtatásokat, hogy Kása észre sem veszi.

Noha az ABKR rendszer szempontjából minden szabály be volt tartva, és a Kása alkalmazása tudott hozzáférni a Kredit táblához, Balázsnak mégis sikerült azt lemásolni.

Egy megoldást ad a Bell-La Padula módszer, amely az alábbiakat használja:

- objektumokat (pl. táblák, nézet-k, oszlopok, sorok);
- alanyokat (subject) (felhasználók, programok);
- biztonsági osztályokat (security classes);
- igazolvány osztályokat (clearance classes).

A módszer minden adatbázis objektumhoz egy biztonsági osztályt rendel, minden alanyhoz pedig egy igazolványt. Legyen *O* egy objektum, a hozzárendelt biztonsági osztályt jelöljük *class*(*O*)-val, hasonlóan, ha *A* egy alany, a hozzárendelt igazolvány osztályt jelöljük *class*(*A*)-val.

4 osztály van:

- *Top secret* (*TS*) (nagyon titkos);
- *Secret* (*S*) (titkos);
- *Confidential* (*C*) (bizalmas);
- *Unclassified* (*U*) (osztályozatlan).

Ha *A* és *B* titkossági osztály között $A > B$ reláció áll fenn, azt mondjuk, hogy *A* kényesebb, titkosabb (sensitive), mint *B*. A módszer osztályai között a következő relációk állnak fenn:

$TS > S > C > U$.

A Bell-La Padula modell a következő megkötéseket teszi az adatbázis objektumaira:

1. Egyszerű biztonsági tulajdonság (simple security property): Egy *A* alanynak megengedett, hogy az *O* objektumot olvassa, ha *class*(*A*) >= *class*(*O*).

Például egy felhasználó, melynek *TS* igazolványa van, olvashat egy táblát, melynek *C* a biztonsági osztálya. Egy *C* igazolvánnyal rendelkező felhasználó nem olvashat egy olyan táblát, melynek *TS* a biztonsági osztálya.

2. * tulajdonság: Az *A* alany csak akkor írhatja az *O* objektumot, ha *class*(*A*) <= *class*(*O*).

Például egy *S* igazolvánnyal rendelkező felhasználó írhat *S* vagy *TS* biztonsági osztállyal rendelkező táblát.

Ebben az esetben is a felhasználónak meg kell legyenek a szükséges jogosultságai (amit GRANT paranccsal kaphat) és plusszba, a felhasználó és a táblák biztonsági osztályai között a szükséges megkötések fenn kell álljanak.

Az előző példa esetén, mondjuk a Kredit táblának *S* a biztonsági osztálya, Kása felhasználónak *S* igazolványa van, Balázsnak *C* vagy kisebb a biztonsági osztálya. Balázs csak *C* vagy kisebb biztonsági osztályú táblát tud létrehozni, tehát a Titkos táblának legfeljebb *C* biztonsági osztálya lehet. Így Kása programja nem tud írni a Titkos táblába, mert a 2-es tulajdonság nincs betartva.

8.2. Adat kriptálás, titkosítás (Data encryption)

Eddig olyan felhasználóról volt szó, aki használhatta a rendszert, hogy hozzáférjen az adatokhoz, azonban nem az egész adatbázishoz, hanem csak annak egy részéhez. Ebben az esetben, olyan felhasználó elől védjük meg az adatokat, aki fizikailag el akarja lopni azokat. Az adatbázisban általában nem kriptálva tároljuk az adatokat, csak ha telefon vonalon vagy számítógépes hálozaton át küldeni kell azokat, akkor titkosítjuk. Az adatok titkosítására kriptálási algoritmusokat használhatunk.

Az adat eredeti formában a *nyílt szöveg* (plain text) és a kriptálási kulcs, a titkosítási (kriptálási) algoritmus bemeneti adata. Az algoritmus adja a *titkosított szöveg*et (ciphertext).
A kriptálási algoritmus általában nem titkos, csak a kulcs. Két módszer ismeretes:

- helyettesítés
- permutáció

A *helyettesítés* módszer esetén a kriptálási kulcsot felhasználva, a nyílt szöveg minden karakterét átalakítjuk, majd helyettesítjük a kiszámítottal. Például amilyen hosszú a kriptálási kulcs, olyan hosszú darabokra osztjuk a nyílt szöveget, összeadjuk karakterenként a kulcs karaktereivel (természetesen a megfelelő kódokat).

A *permutáció* esetén a nyílt szöveg karaktereit más sorrendben tároljuk, összekeverjük.
Mindkettő feltörhető, de ha a kettőt kombináljuk, már nehezebb.

Kidolgoztak egy standardot is: Data Encryption Standard (DES). E standard esetén a nyílt szöveget 64 bitet tartalmazó blokkokra osztják, a kulcs is 64 bit (56 a kulcs, 8 parity bit). Van 2^{56} lehetséges kulcs. A blokkot először permutáljuk, majd a kriptálási kulcsot felhasználva helyettesítjük. Ezekből többféle kombinációt is hasznáhatunk.

Publikus kulcsú kriptálás (Public-key encription)

A DES nem elég biztonságos. Publikus kulcsú kriptálás esetén, a kriptálási algoritmus mellett a kriptálási kulcs is ismert. A nyílt szövegből mindenki létrehozhatja a titkosított szöveget. A dekriptálási kulcs titkos, nem lehet a kriptálási kulcsból könnyen kiszámolni, csak az kaphatja meg, akinek joga van hozzá.

Az egész adatbázist nem szokták kriptálva tartani, csak a jelszavakat és a nagyon titkos információkat, amiket még az adatbázis adminisztrátor sem láthat, mivel neki az egész adatbázisra van joga.

8.3. Kereskedelmi rendszerek és a biztonság

Manapság nagyon sok relációs adatbázis-kezelő rendszer létezik a piacon. Nem könnyű eldöntenünk melyiket használjuk. Egy alkalmazás tervezésére, fejlesztésére és tesztelésére szükséges eszközök esetén a licencet kevés számítógépre kell megvásárolnunk, mert csak a fejlesztők használják, viszont az ABKR licencet annyi számítógépre (kliens részét) kell megvásárolnunk, ahányan az alkalmazást fogják használni. Általában egy alkalmazást több évig is használnak és így megtérül a befektetés. A továbbiakban a biztonság szempontjából elemezzük az ABKR-eket.

A Microsoft Foxpro ABKR egy-felhasználós környezetben használatos. A rendszer semmiféle biztonsági mechanizmusról nem gondoskodik. XBase fájl formátumot használ, minden egyes tábla külön állományban van tárolva, az indexállományok is külön állományokban. Egy kevés SQL parancsot tud használni.

Microsoft Access relációs ABKR több SQL-t implementál. Mindamellett, hogy az Access-t PC-n való használatra tervezték egy kezdetleges biztonsági mechamizmust tartalmaz. Lekérdezéseket tud tárolni adatbázis szinten. Az egész adatbázis és összes objektuma egy állományban van tárolva.

Az Oracle relációs ABKR szinte az egész SQL standardot támogatja. Ezenkívül a PL/SQL az SQL-nek harmadik generációs nyelvek jellemvonásaival való kiterjesztése, melynek segítségével tárolt eljárásokat stb. tudunk fejleszteni. Komplex biztonsági mechanizmussal rendelkezik, mint amilyen többek között szerepkörök (roles) létrehozása és jogosultságok osztogatása az adatbázis különböző objektumaira.

Ami a képességeit illeti, a Sybase SQL Server hasonló az Oracle-hoz. Saját biztonsági mechanizmussal rendelkezik, mely a biztonsági jellegzetességek széles körét implementálja. Az SQL standardot kiterjeszti a Transact-SQL nyelv segítségével. Ugyanitt említjük a Microsoft SQL Server-t is.

Ha XBase alapú rendszereket használunk, előfordulhat, hogy más felhasználó egyik-másik állományt módosítja, eltörli stb. Hasonlóan egy másik felhasználó egy egész Access adatbázist eltörölhet. Oracle vagy Sybase típusú rendszerek esetén ez nem fordulhat elő.

8.4. Personal Oracle és a biztonság

Ami a biztonságot illeti, a Personal Oracle esetén mutatunk be bizonyos részleteket. Az itt bemutatott jellemvonásokat felhasználva más rendszer esetén is boldogulunk. Mikor a biztonsági politikánkat dolgozzuk ki, a következő kérdéseket kell észben tartanunk:

- Kinek jut az adatbázis adminisztrátor szerepe? (az adatbázis adminisztrátor feladatait lásd az 1.1.7-nél).
- Hány felhasználó fogja az adatbázist használni?
- Melyik felhasználónak milyen adatbázis objektumon milyen jogosultságra van szüksége?
- Hogyan fogunk egy felhasználót eltávolítani az adatbázis használatától, ha nem használhatja többet az adatbázist?

A Personal Oracle a biztonságot a következő elemek segítségével implementálja:

- Felhasználók (Users)
- Szerepkörök (Roles)
- Jogosultságok (Privileges)

Felhasználó létrehozása: A felhasználóknak joguk van az adatbázishoz kapcsolódni (belépni). Új felhasználót a következő parancs segítségével tudunk létrehozni:

```
CREATE USER <felhasználó_név>
     IDENTIFIED {BY <password> | EXTERNALLY}
     [DEFAULT TABLESPACE <tablespace_név>]
     [TEMPORARY TABLESPACE <tablespace_név>]
     [QUOTA {integer [K|M] | UNLIMITED} ON <tablespace_név>]
     [PROFILE <profile_név>]
```

8.7. példa: Zoli nevű felhasználót hozunk létre Micimacko jelszóval:

```
SQL> CREATE USER Zoli IDENTIFIED BY Micimacko;
User created.□
```

Abban az esetben, ha a felhasználó létrehozása esetén a jelszót az IDENTIFIED BY kulcsszó segítségével adjuk meg, akkor ahányszor a felhasználó be akar lépni, a rendszer fogja kérni a jelszavát. Az EXTERNALLY kulcsszóval létrehozott felhasználót egy külső rendszer, általában az operációs rendszer azonosít. Az Oracle nem ajánlja ezt az azonosítási módot. A parancs segítségével a felhasználónak alapértelmezett `tablespace`t lehet adni, lásd az Oracle dokumentációt a tablespace-szel kapcsolatban.

Mint minden CREATE parancsnak, ennek is van ALTER parancs párja. Egy létező felhasználó jelszavát stb. a következő parancs segítségével változtathatjuk meg:

```
ALTER USER <felhasználó_név>
     [IDENTIFIED {BY password | EXTERNALLY}]
     [DEFAULT TABLESPACE <tablespace_név>]
     [TEMPORARY TABLESPACE <tablespace_név>]
     [QUOTA {integer [K|M] | UNLIMITED} ON <tablespace_név>]
     [PROFILE profile]
     [DEFAULT ROLE { <role_név> [,<role_név>] ...
       | ALL [EXCEPT <role_név>  [,<role_név>] ...] | NONE}]
```

8.8. példa: A Zoli felhasználó jelszavát kicseréljük:

```
SQL> ALTER USER Zoli
  2  IDENTIFIED BY Malacka;
```

```
User altered.
```

Egy felhasználót a következő parancs segítségével törölhetünk:

```
DROP USER <felhasználó_név> [CASCADE];
```

Ha a CASCADE záradékot használjuk, a törölt felhasználó tulajdonában levő összes objektumot (tábla, index, tárolt eljárás stb.) a rendszer törli, mielőtt a felhasználót törölné. Ha egy felhasználónak vannak a tulajdonában objektumok és nem használjuk a CASCADE záradékot a rendszer nem törli ki a felhasználót sem. (Miért? mi legyen a tulajdonában levő objektumokkal? ki lesz a tulajdonosuk? fogja-e valaki még használni vagy marad szemétnek?)

Szerepkör létrehozása: A szerepkör egy jogosultság vagy jogosultságok halmaza, melynek segítségével az adatbázison bizonyos műveleteket lehet végezni. Ha több felhasználónak is ugyanazokat a jogosultságokat akarjuk megadni, hozzunk létre egy szerepkört és a szerepkörnek osszuk ki a jogosultságokat, majd a felhasználóknak adjuk meg szerepkört. Egy új szerepkör kezdetben üres, GRANT parancs segítségével jogosultságokat osztunk neki. Szerepkört CREATE ROLE (lásd Oracle documentáció) parancs segítségével hozunk létre. Ajánlatos jelszót is rendelni a szerepkörhöz. Egy felhasználónak egy szerepkörre a GRANT parancs segítségével adhatunk engedélyt:

```
GRANT <role_név> TO <user_név> [WITH ADMIN OPTION];
```

Ha a WITH ADMIN OPTION-t használjuk, akkor a felhasználó más felhasználónak is engedélyt adhat a szerepkörre.

Egy szerepkört egy felhasználótól a REVOKE paranccsal vonhatunk vissza:

```
REVOKE <role_név> FROM <user_név>;
```

Ha létrehozunk egy felhasználót és semmi más jogot nem adunk neki, akkor csak be tud lépni az adatbázis-kezelő rendszerbe és kilépni, és ez minden amit a felhasználó az adatbázissal végezni tud.

Oracle a következő szerepköröket adja, mely segítségével egy felhasználónak jogosultságokat adhatunk.

```
CONNECT
RESOURCE
DBA
```

A CONNECT szerepkört úgy lehet felfogni, mint egy belépés szintű szerepkör. Egy felhasználó, mely egy ilyen szerepkörre van jogosítva, miután még plusz jogosultságokat kap, tehet valamit az adatbázisban. A szerepkör megengedi egy felhasználónak, hogy SELECT, INSERT, UPDATE, DELETE műveleteket végezzen más felhasználó tábláin, miután ezeket a jogosultságokat is megkapta. Ezenkívül a felhasználó táblákat, nézeteket, sequence-t és szinonimákat tud létrehozni.

A RESOURCE szerepkör mindent magába foglal amit a CONNECT szerepkör, és megengedi, hogy a felhasználó tárolt eljárásokat, triggereket és indexeket is létrehozzon.

A DBA szerepkör minden létező jogosultságot magába foglal. Azon felhasználók, kik ezzel a szerepkörrel rendelkeznek lényegében akármit tehetnek az adatbázisban. Nagyon kevés felhasználónak szabad ezt a szerepkört megadni, ha a rendszer biztonságát meg akarjuk tartani.

8.9. példa: `GRANT RESOURCE TO Zoli;`

Jogosultságok (privileges) Oracle-ban: Egy jogosultság egy jog, hogy egy adott SQL parancsot végrehajtsunk vagy más felhasználó objektumához hozzáférhessünk. Példák jogosultságokra, melyek jogot adnak:

- egy adatbázishoz való kapcsolódásra;
- egy táblát létrehozni;
- sorokat lekérdezni más felhasználó tábláiból;
- végrehajtani más felhasználó tárolt eljárásait.

Két kategóriába sorolhatjuk a jogosultságokat:

- rendszerjogosultságok (System privileges);
- séma objektumjogosultságok (Schema object privileges).

8.4.2. Rendszerjogosultságok

A rendszerjogosultság az jog, hogy egy sajátos műveletet végrehajtsunk vagy az a jog, hogy egy műveletet bármely séma objektumon végrehajtsunk, melynek sajátos típusa van. Például CREATE ANY INDEX jogosultság megengedi, hogy bármely sémában indexet hozzunk létre. A CREATE PROCEDURE jogot ad a saját sémában eljárást, függvényt és csomagot (package) létrehozni. A DROP ANY TRIGGER megengedi, hogy bármely sémából triggert töröljünk. Több mint 60 rendszerjogosultság van, lásd az Oracle dokumentációt.

Adhatunk rendszerjogosultságokat a felhasználóknak vagy szerepköröknek, de csak az adminisztratív szereppel rendelkező felhasználóknak adjunk, ne az egyszerű felhasználónak. Ha szerepköröknek adunk rendszerjogosultságokat, akkor a szerepköröket arra használhatjuk, hogy rendszerjogosultságokat kezeljünk. (Például az MS SQL Server 7 szerver szerepkört értelmez.)

A GRANT parancs Oracle esetén a következő szintaxissal rendelkezik:

```
GRANT <rendszer_jogosultság> TO {<user_név> | <role_név> | PUBLIC}
     [WITH ADMIN OPTION];
```

A PUBLIC kulcsszó segítségével az összes felhasználónak adhatunk jogot.

8.10. példa: A következő parancs minden felhasználónak megengedi, hogy nézetet hozzon létre a saját sémájában:

```
SQL> GRANT CREATE VIEW
       TO PUBLIC;
```

Ha az összes sémában jogot akarunk adni nézet létrehozására, akkor az ANY kulcsszót is használnunk kell (CREATE ANY VIEW). □

8.4.3. Séma objektum-jogosultságok

A tetszés szerinti hozzáférés esetén felsorolt jogosultságok (SELECT, INSERT, UPDATE, DELETE, REFERENCES) a séma objektum-jogosultság lehetséges értékei.

8.11. példa: Legyen Zoli az a felhasználó, aki létrehoz még két felhasználót Arpi és Csaba névvel.

```
SQL> create user Arpi identified by Emi;
User created.

SQL> create user Csaba identified by Zsuzsa;
User created.

SQL> grant connect to Arpi;
Grant succeeded.

SQL> grant resource to Csaba;
Grant succeeded.
```

Zoli létrehoz egy SZALLITOK táblát és SELECT jogosultságot ad Arpi-nak rá, illetve SELECT és UPDATE jogosultságot Csabának. Megteheti, mivel ő a sématulajdonos.

```
SQL> GRANT SELECT ON SZALLITOK TO Arpi;
Grant succeeded.

SQL> GRANT SELECT, UPDATE ON SZALLITOK TO Csaba;
Grant succeeded.
```

Most akár Csaba, akár Arpi kiadhatja a következő parancsot:

```
SQL> SELECT *
 2 FROM Zoli.SZALLITOK;
```

Nagyon fontos, hogy a séma tulajdonosának a nevét is megadjuk, e nélkül ugyanis olyan hibát kaptunk volna, hogy a SZALLITOK tábla nem létezik. Csaba vagy Arpi nem a saját sémájában létező táblát kérdezi le, ezért szükséges megadni a séma nevét.

Ha Arpi vagy Csaba INSERT parancsot próbál kiadni, hibajelzést kapnak, mivel ilyen joguk nincs.

```
SQL> INSERT INTO Zoli.SZALLITOK
 2  VALUES(12345,'FLAMINGO', 'Kolozsvar', 'Babes utca 7');
INSERT INTO Zoli.SZALLITOK
                    *
ERROR at line 1:
ORA-01031: insufficient privileges
```

Módosítani Csaba tud, Arpi nem. □

A tetszés szerinti hozzáférés esetén részletezett példák Oracle-ban hasonlóak, csak a séma nevet nem szabad elfelejteni, ha nem a tulajdonos fér a táblákhoz.

8.4.4. Oracle által javasolt biztonság politika

Egy adatbázishoz általában több alkalmazás is hozzáfér. Oracle-ban és MS SQL Server esetén is lehet alkalmazás szerepkört (application role) létrehozni. Egy alkalmazást sok felhasználó is használhat. Elképzelhető egy olyan biztonsági politika, hogy minden adatbázis objektumhoz (tábla, nézet, tárolt eljárás stb.) csak az alkalmazás szerepkör férhet hozzá. Viszont a felhasználók hozzáférését az adatbázis különböző részéhez nyilván kell tartani (auditing), ez az információ biztonságának egyik alapelve. Ha csak egy nagy alkalmazás szerepkör fér hozzá az adatbázishoz, nem lehet nyomon követni a különböző felhasználók által végzett műveleteket. Többek között ezért javasolja az Oracle, hogy minden alkalmazás felhasználója legyen az adatbázisnak is felhasználója és bízzuk az Oracle-ra a felhasználók azonosítását, jelszavának titkosítását, nyilvántartását. Ezen feladatok az adatbázis-kezelő rendszerek esetén implementálva vannak, fölösleges, hogy minden alkalmazás ismét implementálja. Az Oracle-nak egy nagyon komplex biztonsági rendszere van, és csak akkor használhatjuk ki az összes lehetőséget, ha az alkalmazás felhasználója az adatbázisnak is felhasználója.

Több felhasználónak is lehet ugyanaz a szerepköre, hozzunk létre szerepköröket és adjunk jogosultságokat a szerepköröknek, hogy ne különálló felhasználóknak ismételjük ugyanazoknak a jogosultságoknak a kiosztását. Ugyanazokkal a jogosultságokkal rendelkező felhasználóknak a megfelelő szerepkört osztjuk ki (lásd a GRANT <role_név> TO <user_név> parancsot). Több szerepkör is lehet egy alkalmazáson belül, tehát az alkalmazás szerepkört mindegyiknek kiosztjuk. Az alkalmazás szerepkörnek pedig kiosztjuk az adatbázis objektumokra szükséges jogosultságokat.

Az adatbázisok biztonságáról a szerző saját eredményei a [VaKa95]-ben találhatóak.

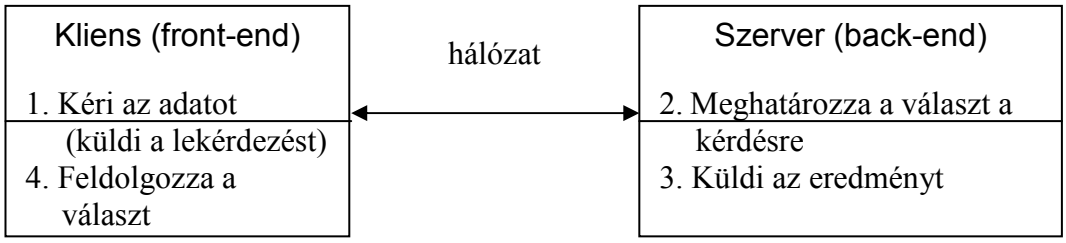
9. Adatbázisok kliens-szerver architektúrája

Az osztott processzálást azért vezették be, hogy minikomputerekből álló rendszer elvégezhesse ugyanazt a munkát, amit egy központi számítógép végez. Az adatokat a szerver (adatbázis-szerver) tárolja, az alkalmazások a kliens számítógépeken futnak. A szerver egy erős számítógép kell legyen, mely több kliens számítógéptől kap adatokat tárolásra, illetve lekérdezéseket, azért, hogy feldolgozza (processzálja) őket, majd a kért adatokat a szerverről visszatérítse a kliens számítógépnek. A kliens számítógép lehet gyengébb is.

Az adatok helyességére vonatkozó feltételeket a szerveren megszorítások formájában ajánlott megadni és ezek ellenőrzését az adatbázis szerverre bízni. Ha az adatok helyességének az ellenőrzését a kliensen végeznénk, és egy felhasználó ODBC-n vagy OLE DB Provider-en keresztül férne az adatbázishoz, akkor helytelen adatokat tudna az adatbázisba beszúrni, de mivel a szerverre bíztuk az adatok helyességének az ellenőrzését nem kerülhetnek semmiképp helytelen adatok az adatbázisba.

Eddig csak a direkt SQL-lel foglalkoztunk és feltételeztük, hogy az SQL parancsok bekérésére és végrehajtására alkalmas SQL interpreterrel rendelkezünk. A gyakorlati életben előforduló alkalmazások esetében az SQL parancsok valamilyen szoftverfejlesztő eszközzel elkészített eljárások vagy függvények részeként lesznek felhasználva. Ezeket az alkalmazásokat általában valamilyen ún. *befogadó* nyelven (például C, Java, Cobol, Visual Basic, Delphi, Visual C++ stb.) készítik és egyes függvények vagy eljárások törzsében SQL utasításokat helyeznek el.

Kliens /szerver munkamegosztás



Kliens oldal tipikus feladatai:

- Össze kell gyűjtse az összes szükséges információt, mielőtt kérelmet küldene a szervernek;
- A kliens dolgozzon csak a lekérdezések eredményeivel, ne az alaptáblákkal. A lekérdezést hagyja a szerverre és csak az eredmény relációval dolgozzon. A szerver gyorsabb a lekérdezésekben, így kisebb táblákat kell szállítani a hálózaton;
- A kliens kell hogy elvégezze az összes adatkezelési műveletet;
- A kliens kell hogy az információ és az adat formázását elvégezze a raportokban;
- A kliens felelős az összes adat kiíratásáért a felhasználó fele;

A szerver oldal tipikus feladatai:

- Task orientált, képes kell legyen a műveletek gyors elvégzésére;
- Nagy mennyiségű információ tárolása, módosítása, gyors keresés;
- Adatok helyességének az ellenőrzése;
- A szerver létre kell hozza az eremények halmazát (result sets), amit a kliens alkalmazás kér;
- Előfordul, hogy az adatbázis szervernek nincs felhasználói felülete, scriptet lehet csak futtatni;
- Nagyon fontos a biztonság, melyik felhasználó az adatbázis mely részéhez férhet hozzá.

9.1. Kliens-szerver standardok

SQL2–ben egy SQL alkalmazásnak egyszer egy CONNECT operációt kell végeznie, hogy bekötődjön a szerverhez, mielőtt akármilyen adatbázis műveletet végezne. Ha létrejött a kapcsolat, az alkalmazás, vagyis a kliens kiadhatja az SQL parancsait és a szerver végrehajthatja.

A standard megengedi, hogy egy SQL kliens, mely már be van kötődve egy szerverhez, bekötődjön egy másik szerverhez is. Ha létrejön a második kapcsolat, az első kapcsolat „alvó" (dormant) lesz. A kiadott SQL parancsokat a második szerver hajtja végre, mindaddig, míg a kliens:

- visszaállítja az első kapcsolatot (SET CONNECTION paranccsal);
- egy más szerverhez kapcsolódik, amikor a második is „alvó" lesz.

Egy adott pillanatban egy SQL kliensnek egy „aktív" kapcsolata van és akárhány „alvó" kapcsolata és minden parancs a klienstől az aktív szerverhez irányul. A standard megengedi, de nem követeli a „multi-szerver transactions"-t. Vagyis egy SQL kliens képes kell legyen egy tranzakció közepén egyik szerverről egy másikba váltani (switch), úgy, hogy a tranzakció egy része az egyik szerveren, a másik része a másikon hajtódjon végre. Minden létrehozott kapcsolatot (legyen az aktív vagy alvó) meg kell szakítani a DISCONNECT művelettel.

Más standardok:

- ISO –tól Remote Data Acces (RDA);
- az IBM – től a Distributed Relational Database Arhitecture (DRDA).

9.2. Kliens/szerver alkalmazások programozása

A kliens/szerver alkalmazás egy speciális, összetett rendszer. A kliens/szerver megközelítés megváltoztatja az alkalmazás programozását. Egy fontos része, hogy a relációs adatmodellen alapuló adatbázis szerver halmaz szintű (set-level) rendszer. Az alkalmazás programozó nemcsak úgy használja a szervert, mint egy hozzáférési utat és rekord szintű kódot írjon, hanem, amennyire az alkalmazás megengedi a lekérdezéseket halmazokra bontja, különben túl sok üzenet lesz.

Üzenetek száma csökkenthető a tárolt eljárások (stored-procedure) mechanizmus segítségével. Egy tárolt eljárás egy prekompilált program, mely a szerver oldalon van tárolva. A kliens meghívja egy REMOTE PROCEDURE CALL (RPC) segítségével. Ha többször kell ugyanazt az SQL parancsot végrehajtani, érdemes az SQL parancsot az adatbázis mellett tárolt eljárás formájában tárolni, melyet az ABKR lefordít, meghatározza a végrehajtási tervet és optimalizálja. Ha az ABKR az SQL parancsot karaktersorként kapja meg, minden alkalommal le kell fordítsa, a végrehajtási tervet ki kell dolgozza és optimalizálja. Sokkal hatékonyabb meghívni a tárolt lefordított eljárást.

A tárolt eljárások előnyei:

- egy eljárás megosztható több kliens által;
- jobb biztonság: bizonyos felhasználóknak megadjuk a jogot, hogy meghívja ezeket, de az adatokon direkt ne változtathasson;
- az optimalizálás végrehajtható ezen eljárások megírásakor, nem csak futtatáskor;
- elrejthetők a felhasználó elől a rendszer, illetve adatbázis specifikus részletek, ami nagyobb adatfüggetlenséget biztosít.

Tárolt eljárások hátrányaként említhetjük, hogy még nincs ezekre vonatkozó szabvány, így minden rendszer sajátosan oldja meg:

- az Oracle a PL/SQL nyelvet adja, mely egy procedurális kiterjesztése az SQL-nek harmadik generációs nyelvek jellemvonásaival (lásd az Oracle dokumentációt);
- az MS SQL Server pedig a TRANSACT-SQL-t (lásd az MS SQL Server dokumentációt).

Ha a lekérdezéseket tárolt eljárásokkal oldjuk meg nagy a hatékonyságuk, viszont nem portábilisak (hordozhatók). Ha kicseréljük az ABKR-t, a tárolt eljárásokat át kell írjuk. Ha az SQL parancsokat a kliens alkalmazásban írjuk meg, akkor az ABKR-t kisebb erőfeszítéssel lecserélhetjük. Kérdés, hogy a hatékonyság a célunk, vagy a hordozhatóság?

A három szintű programozási modell esetén, mely az adat hozzáférési réteg (Data Access Layer – DAL), az üzleti logika réteg (Business Logic Layer – BLL) és a grafikus felhasználói felület (Graphical User Interface – GUI) réteget tartalmazza, a tárolt eljárások átvehetik a BLL

feladatát és a DAL csak tárolt eljárások meghívásából áll. Ha nem használunk tárolt eljárásokat, akkor a DAL az összes SQL parancsot tartalmazza és az adatokat a BLL-nek már objektumok halmazaként vagy tömbbök segítségével adjuk át, mely az alkalmazás logikáját programozza meg. Ha egy alkalmazásnak egyrészt Internetes felülete van, másrészt Windows-os felülete is, ugyanazt a BLL-t vagy ugyanazokat a tárolt eljárásokat használhatják.

Befogadó nyelvből az SQL-t kétféleképpen lehet használni:
- Beágyazott SQL, programba épített SQL parancsok által;
- SQL API (Application Programmming Interface) vagy CLI (Call Level Interface) által, mindkettő speciális függvényeket hív meg, hogy hozzáférjen az adatbázishoz.

9.2.1. Beágyazott SQL

Beágyazott SQL esetén a programozó feladata az adatfeldolgozó algoritmusnak a befogadó nyelven való megfogalmazása és a beágyazott SQL utasítások programba szerkesztése. Ezután a programot egy ún. előfeldolgozó (prekompilátor) program a programba beágyazott SQL utasításokat a befogadó nyelv utasításaira alakítja. A folyamat során létrehozott befogadó nyelvi utasítások például olyan függvényeket hívhatnak meg, amelyek paraméterükben egy SQL utasítást várnak karakterlánc formában és képesek a paraméterben kapott SQL utasításokat végrehajtani. Nem szükséges az előfeldolgozó, ha a programozó közvetlenül az SQL utasítások végrehajtására képes függvényeknek a hívását ágyazza be a programba.

Ezután a tisztán befogadó nyelven írt forrásprogramot a befogadó nyelvi komplilátor lefordítja. Az elkészült modulokat össze kell szerkeszteni az adatbázis-kezelő rendszer gyártója által forgalmazott programkönytárakkal, melyek az előbb említett függvények implementációját tartalmazzák, és karakterlánc formájában leírt SQL utasítások végrehajtására képesek.

Oracle adja a PRO*C/C++, PRO*CoBoL, PRO*Fortran pre-komplikátorokat. Az MS SQL Server is ad C prekompilátort.

Az SQL nyelv adatmodellje a hagyományos programozási nyelv adatmodelljétől lényegesen különbözik. Az SQL magját a halmazorientált relációs adatmodell képezi, míg a programozási nyelvek az elemi adattípusokat és a típuskonstruktorokat támogatják csak, a halmaz fogalmát nem. Az SQL viszont nem támogatja a tömb, mutató és más programozási elem alkalmazását.

Ahhoz, hogy a befogadó nyelv és SQL egymás között kommunikálni tudjanak egy interfészre van szükség. Az adatbázis eléréséért felelős SQL utasítások és a befogadó nyelven megírt alkalmazói program utasításai közti adatcsere a befogadó nyelven deklarált változókon keresztül történhet, ezeket *megosztott elérésű változóknak* nevezzük. A befogadó programnyelven írt programból a megosztott elérésű változóra csak a nevével kell hivatkozni, míg a beágyazott SQL utasításokban nevük elé egy kettőspont karaktert kell írni (az SQL utasítást átalakító előfeldolgozó ez alapján különbözteti meg őket az SQL nyelvű utasítások által elérhető adatbáziselemek neveitől).

A beágyazott SQL utasításokat a legtöbb rendszerben az utasítás elejére kiírt EXEC SQL kulcsszavakkal adjuk meg, ezzel jelölve az előfordító programnak, hogy egy SQL utasítás következik, amit egy megfelelő könyvtári függvény meghívására valahogyan át kell alakítania.

Az SQL2 szabvány egy SQLSTATE nevű változót definiál az SQL és a befogadó nyelvi környezetek összekapcsolására. Az adatbázis elérését támogató könyvtárak úgy vannak elkészítve, hogy visszatéréskor ebben a változóban helyezik el a végrehajtott SQL művelet során fellépett esetleges problémákat leíró kódokat.

A változók deklarációját két beágyazott SQL utasítás közé kell tenni, lásd a következő példát.

9.1. példa: A továbbiakban a példák az Egyetem adatbázisra vonatkoznak. Legyenek a következő relációk:

```
Diákok (BeiktSzám, Név, SzemSzám, Cím, SzülDatum, CsopKod, Átlag);
Csoportok (CsopKod, Evfolyam, SzakKod);
```

A deklarációs rész a következő:

```
EXEC SQL BEGIN DECLARE SECTION;
     char CsopK[3],  int Ev, char SzakK[2];
     char SQLSTATE[6];
EXEC SQL END DECLARE SECTION; □
```

Az SQL utasításokban bármely konkrét érték helyére írhatunk változót, neve előtt kettősponttal. A következő példában a definiált változók értékét a Csoportok táblába szúrjuk be.

9.2. példa: A `beolvasCsoport` nevű függvény a felhasználótól egy csoport kódot, egy évfolyam értéket és egy szak kódot kér be, majd a beadott adatokat a Csoportok táblába szúrja be.

```
void beolvasCsoport() {
     EXEC SQL BEGIN DECLARE SECTION;
          char CsopK[3],  int Ev, char SzakK[2];
          char SQLSTATE[6];
     EXEC SQL END DECLARE SECTION;
     /* bekér a felhasználótól egy csoport kódot,
        egy évfolyamot és egy szak kódot és azokat
        eltárolja a CsopK, Ev, illetve SzakK
         változókban */
     EXEC SQL INSERT INTO Csoportok(CsopKod, Evfolyam, SzakKod)
                VALUES (:CsopK, :Ev, :SzakK);
     }
```

Amint látjuk egy INSERT parancsot használunk, hogy az adatbázisba beszúrjunk egy sort. A parancsot EXEC SQL kulcsszavak vezetik be, hogy az előfordító program végezze el a szükséges átalakításokat. A beszúrandó értékek nem szövegkonstansok, mint ahogy az SQL elvárná, hanem megosztott elérésű változók, ezek aktuális értéke lesz behelyettesítve az SQL utasítás kiértékelésekor. A program a változók aktuális értékéből állít össze egy sort, amit beszúr a táblába.□

Az összes SQL parancs, amelynek nincs visszatérési értéke (vagyis nem lekérdezés, beágyazható beszúrást, törlést, módosítást leíró utasítás, illetve adatbázissémát manipuláló utasítások) megosztott elérésű változókat használva is beágyazható programokba. Minden beágyazott SQL utasításban hivatkozhatunk változókra konstans értékek helyett.

A SELECT SQL lekérdezések általában nem ágyazhatóak be, mivel eredményül egy halmazt adnak vissza és a legtöbb programozási nyelv ezt nem tudja kezelni. A SELECT utasítás alkalmazására két megoldást ajánlanak:

1. Az egyetlen sort eredményező lekérdezések esetén, a rendszer az eredménysor egyes komponenseit külön-külön változókba helyezi el.
2. Egy olyan lekérdezésnek, amely eredményként egynél több sort (sorhalmazt) ad vissza egy *sormutatót* definiálunk. A sormutató befutja az eredményreláció összes sorát, és az eredménysorok egyes komponenseit külön-külön változókba elhelyezve, az adatokat a befogadó programnak juttatjuk el.

Egyetlen sort eredményező lekérdezések:

Az alkalmazásokba beágyazható egyetlen sort eredményező lekérdezés formája abban különbözik az ismert SELECT parancstól, hogy a SELECT záradék után INTO kulcsszó segítségével azokat a változókat soroljuk fel, amelyekbe a lekérdezés eredményeként visszakapott adatokat el akarjuk tárolni. Ezek a változók megosztott elérésű változók, tehát egy-egy kettőspontot kell írni a változó neve elé.

Ha a lekérdezés egyetlen sort eredményez, a megosztott elérésű változók rendre felveszik az eredménysor komponenseiben képződött értékeket, az SQLSTATE változó pedig a sikeres végrehajtást jelző értéket. Amennyiben a lekérdezés egyetlen sort sem eredményez, vagy egynél több sort, a megosztott elérésű változók semmilyen értéket sem kapnak, az adatbázis-kezelő rendszer az SQLSTATE változóba a megfelelő hibakódot írja be.

9.3. példa: Az Egyetem adatbázis esetén keressük egy adott csoportban lévő diákok átlag médiáját. A CsoportAtlag függvény bekéri a csoport kódját és kiírja annak átlagát.

```
void CsoportAtlag () {
     EXEC SQL BEGIN DECLARE SECTION;
          char CsopK[3], real CsopAtlag;
          char SQLSTATE[6];
     EXEC SQL END DECLARE SECTION;
     /* bekér a felhasználótól egy csoport kódot,
            és eltárolja a CsopK változóban */
     EXEC SQL SELECT AVG(Atlag) INTO :CsopAtlag
                  FROM Csoportok
                  WHERE CsopKod = :CsopK;
     /* kiíratjuk a CsopAtlag értékét, ha az SQLSTATE
          nem jelez hibát  */
     }
```

Amint látjuk, a CsopK változóban levő értéket a WHERE záradékban használtuk, egy szöveges konstans helyett, a CsopAtlag változót pedig az INTO záradék után, ebbe kapjuk meg az eredményt. □

Több sort eredményező lekérdezések

A több sort eredményező lekérdezések eredményhalmazán egy ún. sormutatóval lehet végigmenni soronként. Sormutató használata a következőképpen történik:

Deklarálni kell a sormutatót a következőképpen:

```
EXEC SQL DECLARE <sormutató> [INSENSITIVE] [SCROLL]
CURSOR FOR <lekérdezés>
[ORDER BY <oszlop-lista>]
[FOR READ ONLY | FOR UPDATE]
```

1. Inicializálni kell a sormutatót a következőképpen:

   ```
   EXEC SQL OPEN <sormutató>
   ```

2. Ezután soronként hozzáférhetünk a sormutató mögött levő lekérdezés eredményadataihoz, a `FETCH` utasítás segítségével, mely a sormutatóval elérhető sorokból veszi a következőt és tartalmát áthelyezi az utasításban megadott változókba. Ha egy sormutatóval már nincs több elérhető sor, akkor a fetch utasítás nem tesz semmit a benne megadott változókba, az SQLSTATE változóba pedig a '02000' konstans értéket teszi, ami azt jelenti, hogy nincs több beolvasható sor. Egy sormutató olvasó utasításának általános alakja a következő:

   ```
   EXEC SQL FETCH FROM <sormutató> INTO <változók listája>
   ```

Végül a sormutatót a következő paranccsal kell lezárni, hogy a rendszer a sormutató számára lefoglalt erőforrásokat felszabadítsa:

```
EXEC SQL CLOSE <sormutató>.
```

A sormutatót akkor érdemes használni, ha egy komplex lekérdezést nem tudunk adatbázis szinten megoldani. Az adatbázis szervernek számítható rendszerek esetén, pl. Oracle, MS SQL Server, tárolt eljárással mindenképpen meg lehet oldani a feladatot, mert ezek is ajánlanak sormutatót (cursor), mellyel hasonlóan lehet dolgozni, mint amilyent bemutattunk fentebb a kliens alkalmazás esetén. (lásd TRANSACT-SQL nyelv az MS SQL esetén, illetve PL/SQL az Oracle esetében).

Ha a használt adatbázis-kezelő rendszer nem tud tárolt eljárásokkal dolgozni (pl. MySQL, MS Access stb.), kénytelenek vagyunk azon lekérdezéseket, melyeket nem tudunk egy SELECT paranccsal megválaszolni kliens alkalmazás szintjén használt sormutatóval megoldani. Ez soha nem lesz olyan gyors, mintha adatbázis szintjén oldottuk volna meg a feladatot, mivel gyakrabban kell használja a hálózatot, mert valószínű, hogy a sormutató minden sora esetén ismételten kell adatokat kérjen az adatbázisból.

9.4. példa: Egy egyszerű példát mutatunk be, ki akarjuk írni az informatika szak harmadéves hallgatóit a képernyőre, ezért a sormutatót feltöltjük a feltételt kielégítő diákok adataival.

```
# define NINCS_TOBB-SOR ! (strcmp(SQLSTATE, "02000"))

void KiirInfo3() {
EXEC SQL BEGIN DECLARE SECTION;
     char nev[50]; char CsopK[3]; char cim [60];
      char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE Info3Sorm CURSOR FOR
  SELECT CsopKod, Név, Cím from Diákok, Csoportok
       WHERE Diakok.CsopKod = Csoportok.CsopKod
       and Evfolyam = 3 and (SzakKod=İ SzakKod=İM);
EXEC SQL OPEN Info3Sorm;
while(1) {
       EXEC SQL FETCH FROM Info3Sorm
             INTO :CsopK, :nev, :cim;
       If (NINCS_TOBB_SOR) break;
       printf("%s %s %s\n", CsopK, nev, cim);
       }
EXEC SQL CLOSE Info3Sorm;
} □
```

Abban az esetben, mikor az adatbázist több felhasználó is használja (lásd tranzakciók a 11. fejezetben), a mi alkalmazásunk a sorokat egy sormutatóból olvassa, előfordulhat, hogy egy másik alkalmazás módosítja a feldolgozás alatt álló táblákat. Ha minket nem érdekel, hogy közben más felhasználó módosította vagy törölte valamelyik sort, melyet mi épp a sormutatóban feldolgozunk, akkor az INSENSITIVE kulcsszó segítségével a sormutatónkat *érzéketlennek* nyilváníthatjuk az egyidejű módosításokkal szemben.

Ha egy sormutatóról biztosan tudjuk, hogy nem módosítja azt a relációt, melyből adatokat olvas, READ ONLY-nak deklarálhatjuk. Ebben az esetben az ABKR egyidejűleg futtathatja más olyan sormutatókkal, melyek szintén csak olvasók vagy érzéketlenek a módosításra, megkönnyítve így az ABKR munkáját.

A sormutatót végigjárhatjuk többször is, hátulról előre is, ha a SCROLL kulcsszó segítségével deklaráljuk, különben csak elejétől tudjuk egyszer végigjárni. Ha használtuk a SCROLL kulcsszót, akkor a FETCH utasításban megadhatjuk, hogy melyik a beolvasandó sor a FIRST (első), LAST (utolsó), NEXT (következő), PRIOR (előző) kulcsszavak segítségével.

Eddig olyan SQL parancsokkal foglalkoztunk, melyek alakja a program lefordításakor már pontosan ismert. Ekkor az előfeldolgozó megtalálja benne az SQL parancsokat. Miután az SQL kompilálva már nem lehet megváltoztani, ekkor SQL *statikus*, azt jelenti, hogy a kompilálás ideje alatt építi fel a hozzáférési utat, hajtja végre az optimalizálást.

Vannak azonban olyan esetek, amikor a kiértékelendő SQL utasítás csak a program futása közben állítható össze, mivel futás közben a felhasználótól még kér bizonyos adatokat. Ebben az esetben azt mondjuk, hogy az SQL dinamikus, mivel karaktersorként küldjük a szervernek, végrehajtás céljából, és minden alkalommal amikor futtatni kell, lefordítja, kidolgozza a hozzáférési tervet, optimizál, ismerve az adatok méreteit. Ezeket az ún. *dinamikus SQL utasítás* segítségével végezhetjük el:

Az első ilyen utasítás az EXEC SQL PREPARE, ennek révén az SQL parancsot karakterlánc formájában adhatjuk meg. A másik az EXEC SQL EXECUTE, mely végrehajtja azt. Két változót használunk, egyik a befogadó nyelvi változó, ebben a végrehajtandó SQL parancs karakterlánc formájában szerepel, illetve egy SQL változó, melybe a PREPARE parancs hatására az ABKR az SQL utasítás elemzése után kapott formát tárolja, s ez a későbbiekben könnyen végrehajtható.

9.5. példa: Az SQL parancsot beolvassuk a billentyűzetről és végrehajtjuk a DinamSQL() nevű program segítségével:

```
void DinamSQL() {
     EXEC SQL BEGIN DECLARE SECTION;
          char *lekerdez;
     EXEC SQL END DECLARE SECTION;
     /* beolvassuk a billentyűzetről az SQL utasítást,
```

```
        eltároljuk a lekerdez címtől indulva;  */
     EXEC SQL PREPARE SQLquery FROM :lekerdez;
     EXEC SQL EXECUTE SQLquery;
}
```

E fenti két lépés egy utasításban is elvégezhető a következőképpen:

```
EXEC SQL IMMEDIATE :lekerdez
```

Akkor érdemes a két lépést különválasztani, ha az SQL utasítást többször is végre akarjuk hajtani, mert ilyenkor a többszöri elemzéshez szükséges időt megtakaríthatjuk.

9.2.2. SQL API

Az SQL API-t a nem homogén adatbázisok lekérdezésére is lehet használni. Egy alkalmazásból különböző operációs rendszerek alatt futó különböző ABKR-ek által kezelt adatbázisokat tudunk használni.

Példák SQL API-kra:

- ODBC (Open DataBase Conectivity) – Microsoft írta a Windows alkalmazásokhoz.
- Microsoft OLE DB provider, illetve ADO (ActiveX Data Objects);
- SAG : SQL Acces Group írta, ismert X/Open CLI néven is;
- IDAPI (Integrated Database Application Programing Interface) Borland, IBM, Novell, Wordperfect írta együtt;
- JDBC (Java Database Conectivity) – SUN.

Egy ilyen univerzális környezet (mint az ODBC vagy OLE DB) úgy jelenik meg, mint egy függvény könyvtár, mely a következőket kell tartalmazza:

- Connect/Deconect – kapcsolatot létesít /megszakít egy adatforráshoz;
- SQL parancs küldése és végrehajtása;
- annak a helynek a meghatározása, ahová az SQL eredményét vagy valami üzenetet küldeni kell;
- beszámolás arról, hogyan fejeződött be egy SQL parancs végrehajtása – hiba kódok;
- ahhoz, hogy adatcserét tudjanak végezni szükség van standard adattípusokra.

Az SQL parancsot a szerver karaktersorként kapja meg, analizálja (ellenőrzi a helyességét), meghatároz egy végrehajtási tervet, optimalizálja azt (felhasználva információkat az adatbázisból, esetleg indexeket is használ), meghatározza a választ és visszaküldi azt.

Több típusú driver is van:

- nem adatbázis szervernek megfelelő driver (Access, Foxpro, Excel, Btrieve, Paradox, DBase) – az SQL parancsot maga a driver végzi el (keres az adatbázisban küldi az eredményt);
- adatbázis szervernek (Oracle, MS SQL Server) megfelelő driver. A driver ellenőrzi az SQL parancsot, de végrehajtásra az adatbázis szerverhez küldi.

10. Triggerek

A 4.7. alfejezetben láttuk a megszorításokat, ezek az SQL2 szabványt követték. A rendszer akkor hajtja végre a megszorítás által kért ellenőrzést, ha az adat, melyre a megszorítás vonatkozik megváltozik. Az SQL3 további lehetőségeket ad az adatbázisba tárolásra kerülő adatok helyességének az ellenőrzésére. Ezek közül a triggerek már sok kereskedelmi rendszerben meg is vannak valósítva, pl. Oracle, MS SQL Server.

A trigger szó jelentése: elsüt, kivált. A triggerek tárolt eljárások, melyek elsütő események hatására indulnak. Ilyen elsütő esemény lehet egy INSERT, DELETE vagy UPDATE parancs. Az ABKR felismeri az adott helyzeteket és meghívja a triggert. Azokat az adatbázisokat, melyekben triggerek is implementálva vannak *aktív* adatbázisoknak nevezik.

10.1. Triggerek leírása

A trigger leírása 3 részből áll:
- *esemény* (event) – a triggereket a rendszer akkor ellenőrzi, mikor bizonyos esemény bekövetkezik. Ilyen esemény egy relációra vonatkozó beszúrás, törlés vagy módosítás.
- *feltétel* (condition) – a triggert a rendszer végrehajtja, ha a feltétel igaz. Ha a megadott feltétel nem igaz, nem történik semmi a triggerrel összefüggésben.
- *művelet* (action) – egy eljárás, melyet a rendszer akkor hajt végre, amikor a trigger aktiválva van és a feltétel igaz. Ez a művelet akár meg is akadályozhatja a kiváltó esemény megtörténtét, vagy meg nem történtté teheti azt (pl. kitörölheti az épp felvitt sorokat). A trigger művelet része hivatkozhat a triggert kiváltó parancs által módosított sorok régi és új értékeire, végrehajthat új lekérdezéseket, változtathatja az adatbázist. Ezen kívül végrehajthat adatdefiníciós parancsokat (új táblákat hozhat létre, megváltoztathatja a hozzáférési jogokat).

A triggerek segítségével megoldható feladatok:
- származtatott oszlopértékek automatikusan generálhatók;
- komplex biztonsági mechanizmusok;
- hivatkozási épség megszorítások osztott adatbázis csomópontjai között;
- komplex logikai összefüggések programozhatóak;
- átlátszó módon az események naplózása (loggolása);
- másolt táblák napratevése osztott adatbázisok esetén;
- statisztikákat készíthetünk a táblához való hozzáférésről;
- a biztonság ellenőrzésében segíthetnek, bejegyezni egy táblába ki milyen táblákon végzett valamilyen műveletet.

A trigger számos lehetőséget kínál a programozónak:
- a művelet végrehajtható a kiváltó esemény előtt, után vagy helyette;
- a művelet hivatkozhat a műveletet kiváltó esemény által törölt, beszúrt vagy módosított sorok régi és új értékeire;
- ha az esemény módosítás, akkor megadhatók egy bizonyos oszlop vagy oszlopok, amelyre az esemény vonatkozik;
- a WHEN záradékban megadható egy feltétel és a műveletet a rendszer csak akkor hajtja végre, ha a trigger actívvá válik, és a kiváltó esemény bekövetkezésekor a feltétel igaz;
- a programozó megadhatja, hogy a művelet végrehajtása a következő két lehetőség közül milyen módon történjen meg:
  - minden módosított sorra egyszer;
  - egy adatbázis-művelet által módosított összes sorra vonatkozóan egyszer.

10.1. példa: Egy SQL3 szintaxissal megírt triggerre:

```
1) CREATE TRIGGER Részl9Trigger
2)   AFTER UPDATE OF Fizetés ON Alkalmazottak
3)   REFERENCING
4)   OLD AS Régi,
5)   NEW AS Uj
6)   WHEN (Régi.Fizetés > Uj.Fizetés and Uj.RészlegID = 9)
7)     UPDATE Alkalmazottak
8)       SET Fizetés = Régi.Fizetés
9)       WHERE SzemSzám = Uj.SzemSzám
10)    FOR EACH ROW
```

A Részl9Trigger trigger az Alkalmazottak tábla, Fizetés oszlopának módosítása esetén kerül végrehajtásra minden módosított sorra külön. Az első sor tartalmazza a trigger nevét, a második adja meg a kiváltó eseményt, ami az Alkalmazottak tábla Fizetés attribútumának a módosítása. A 3−5 sor segítségével hivatkozni tudunk a trigger művelet részében és a feltételében a módosítás előtti régi és módosítás utáni új sorokra. A módosítás előtti sorokra `Régi`, a módosítás utáni sorokra `Uj` néven hivatkozhatunk, mint az SQL lekérdezés esetén a FROM záradékbeli sorváltozókra.

A 6. sor a trigger feltétel része, mely azt mondja ki, hogy a trigger művelet részét akkor fogjuk végrehajtani, ha a módosított alkalmazott a 9-es részleg alkalmazottja (ezek − mondjuk − valamilyen speciális csoport) és az alkalmazott módosítás előtti fizetése nagyobb, mint amire az aktuális UPDATE parancs módosítja. A 7−9 sorok a trigger művelet részét tartalmazzák, amelynek révén az alkalmazott fizetését visszaállítja a módosítás előtti értékre. Tehát a trigger nem engedi csökkenteni a 9-es részleg alkalmazottainak a fizetését. □

A trigger *művelet*ét végrehajthatjuk a kiváltó *esemény* előtt (BEFORE), után (AFTER) vagy helyette (INSTEAD OF). A példa esetén a kiváltó esemény után fogja a rendszer a triggert végrehajtani. Az INSTEAD OF beállítás esetén, ha a WHEN után megadott feltétel igaz, a trigger művelet része végrehajtásra kerül, magát a kiváltó eseményt a rendszer azonban semmiképp nem hajtja végre.

A trigger lehet sor szintű (FOR EACH ROW) vagy utasítás szintű (ha elmarad a FOR EACH ROW). Egy utasítás szintű trigger egyszer hajtódik végre a kiváltó esemény hatására, viszont a sor szintű trigger minden sorra végrehajtásra kerül. A triggerek közötti különbség akkor látszik, ha egy kiváltó esemény több sorra is vonatkozik, például az UPDATE művelet egy tábla több sorát is módosítja, nem csak egyet. Ebben az esetben az utasítás szintű trigger csak egyszer kerül végrehajtásra, és hivatkozni tudunk a módosítás előtti értékekre a régi sorok halmazának (OLD_TABLE kulcsszó) a segítségével, illetve a módosítás utáni értékekre, az új sorok halmazával (NEW_TABLE).

Ha az UPDATE művelet több sort módosít és a trigger sor szintű, akkor a trigger művelet része minden sorra végrehajtódik. Abban az esetben, ha a trigger sor szintű és a kiváltó esemény beszúrás, akkor a beszúrt sornak a NEW AS záradék segítségével adhatunk nevet. Törlés esetén az OLD AS záradékkal nevezhetjük el a törölt sort. Ha a kiváltó esemény a módosítás, a rendszer a módosítás előtti sort régi sornak, a módosítás utánit pedig újnak tekinti.

Az utasítás szintű trigger akkor hasznos, ha a trigger művelet része nem függ a kiváltó esemény által érintett sorok értékeitől, például biztonsági ellenőrzés esetén az aktuális felhasználóra vonatkozóan vagy amikor generál egy sort a kiváltó eseményre vonatkozóan.

10.2. példa Utasítás szintű trigger SQL3-ban:

```
CREATE TRIGGER set_count AFTER INSERT ON Diakok /*event*/
REFERENCING NEW TABLE AS InsertedTuples
FOR EACH STATEMENT
     INSERT
          INTO StatisticTable(ModifiedTable, ModificationType, Count)
          SELECT 'Diakok', 'Insert', count (*)
          FROM InsertedTuples I
          WHERE I.age <18
```

NEW TABLE – tábla nevet adhatunk (Inserted_Tuples) az újonnan beillesztett rekordoknak.
FOR EACH STATEMENT – utasítás szintű trigger, elhagyható, mert ez az implicite adott.

A trigger minden INSERT parancs esetén egyszer értékeli ki a trigger feltétel részét és ha igaz végrehajtja a művelet részt ki, mely a Diákok táblába illeszt rekordokat. Hatása, hogy beilleszt egy sort a statisztikai táblába, ahol az első oszlop a megváltoztatott tábla neve, a második a változtatás típusa, az utolsó azon rekordok száma, melyek a beillesztettek közül az age>18 feltételt teljesítik.

Egy trigger lehet:

- Sor szintű, és kiváltó esemény előtt végrehajtva, amikor is mielőtt módosítaná a kiváltó esemény által érintett sorokat és mielőtt ellenőrizné az összes helyességi megszorítást, végrehajtja a trigger művelet részét, ha a trigger feltétel része igaz.
- Utasítás szintű, és kiváltó esemény előtt végrehajtva. Ebben az esetben mielőtt végrehajtaná a triggert kiváltó eseményt, végrehajtja a trigger műveletét.
- Sor szintű, és kiváltó esemény után végrehajtva. Miután módosítja a kiváltó esemény által érintett sorokat és ellenőrzi a helyességi megszorításokat, végrehajtja a trigger művelet részét az aktuális sorra, ha a feltétel igaz. Ellentétben a BEFORE triggerekkel, az AFTER trigger esetén a rendszer lezárja a sorokat. (Lásd a lezárás a 11. fejezetben)
- Utasítás szintű, és kiváltó esemény után végrehajtva. Ebben az esetben a kiváltó esemény után ellenőrzi a helyességi megszorításokat, majd végrehajtja a trigger műveleti részét.

10.2. Triggerek tervezése

A trigger egy erős mechanizmus, hogy az adatbázisban végzett módosításokat könnyebben tudjuk feldolgozni. A triggereket elővigyázatosan kell használni. A triggerek halmazának a hatása lehet nagyon komplex, ezért egy aktív adatbázist karbantartása nehézzé válhat. Előfordulhat, hogy a felhasználó nincs tisztában azzal, hogy a trigger mely program mellékhatásaként lett végrehajtva.

Ha egy parancs egynél több triggert aktivál, az ABKR mindegyiket végrehajtja, tetszőleges sorrendben. Fontos tudni, hogy egy trigger művelet része egy másik triggert aktiválhat. Sajátos esetben előfordulhat, hogy ismét aktiválja az előző triggert – ezeket rekurzív triggereknek nevezik. Az ilyen lánc aktiválásakor, mivel nem tudjuk, hogy az ABKR milyen sorrendben hajtja őket végre, nehezen lehet eldönteni, hogy mi lesz a hatásuk.

A triggerek tervezésekor vegyük figyelembe a következő irányelveket:

- Ne írjunk triggereket olyan tevékenységre, melyre a megszorítások adnak lehetőséget (például hivatkozási épség megszorítás).
- Korlátozzuk a triggert 60 sorra. Ha több műveletre is szükségünk van, helyezzük azokat egy tárolt eljárásba és hívjuk meg a triggerből a tárolt eljárást
- Ne tervezzünk rekurzív triggereket.
- Használjuk mértékkel a triggereket, gondoljunk arra, hogy minden egyes adatkezelési művelet esetén meghívódik és nagyon sok munkát igényelhet.
- Csak olyan műveletek elvégzésére használjuk, melyek minden felhasználó esetében szükségesek.

Megszorítások triggerekkel szemben:

- A triggereket is azért találták ki, hogy az adatbázis konzisztens maradjon. A helyességi megszorításokat is, viszont ezek nem műveletként vannak értelmezve, ellentétben a triggerekkel, így könnyebb megérteni a helyességi megszorításokat.

– A helyességi megszorítások megvédik az adatot mindenféle parancs esetén, hogy ne kerüljenek inkonzisztens állapotba, de triggereket csak speciális parancsok (INSERT, DELETE, UPDATE) aktiválnak.
– A triggerek az adatbázis integritását rugalmasabb úton tartják meg.

10.3. Példák triggerekre MS SQL Server-ben

Az MS SQL Server esetén az új sorra az `inserted` kulcsszóval hivatkozhatunk (nem kell a a trigger elején deklarálni, hanem a művelet részben lehet hivatkozni az újonan beszúrt sorokra, vagy csak egy sorra, ha az INSERT csak egy sort szúrt be), mint egy táblára. A törölt sorokra a `deleted` kulcsszóval lehet hivatkozni (vagy csak egy sorra, ha a DELETE csak egy sort törölt ki). Az UPDATE úgy történik, hogy kitörli a módosított sorokat (vagy csak egy sort) és beszúrja az új sort, ezért hivatkozni lehet az `inserted` és `deleted` sorokra is. Ezen kívül TRANSACT-SQL parancsokat használhatunk.

Egy software cég projektjeinek a tárolása esetén érdekeltek vagyunk a megrendelő cég (tárolva a Kliensek táblában), illetve a rendelt projekt (tárolva a Projektek táblában) adataiban. Egy projekt különböző állapotban lehet: tervezett, aktív, megszakítva, befejezett stb. állapotban (tárolva a ProjAllapotok relációban). Egy projektet több kiszabott feladatra (Feladatok) osztunk fel, melyet egy alkalmazott (Alkalmazottak) kell elvégezzen. Egy feladat több napra is kiterjedhet. Egy feladatot apró tevékenységekre osztunk, melyeket ugyanaz az alkalmazott végzi, aki a feladatot, és ez nem terjedhet több napra, csak egyre. Tehát a Projektek és Feladatok között a kapcsolat 1:*n* típusú, akárcsak a Feladatok és Alfeladatok között (1:*n*).

A projektnek, feladat és tevékenységnek is vannak dátumai: tervezett kezdési dátum (TervezettKezdetiIdoPont), effektív kezdési dátum (EffektivKezdetiIdoPont), tervezett befejezési dátum (TervezettBefejezIdoPont), effektív befejezési dátum (EffektivBefejezIdoPont). A projekt, feladat és tevékenység lehet befejezett vagy nem. A Feladatok és Alfeladatok relációk esetén a Befejezette (1 ha befejezett, 0 ha nem befejezett) mező adja ezt meg, a Projektek esetén pedig a ProjAlID (4-es ID a befejezett terv).

A következőkben megadjuk a relációk szerkezetét, mely nem teljes, csak amennyire a triggerek bemutatására szükséges.

```
Alkalmazottak (AlkID, ANev, Cím, Orabér, ...)
Kliensek (KliensID, KNev, KCim, Hihetoseg, ...)
ProjAllapotok (ProjAlID, PANev)
Projektek (ProjektID, Megnevezés, KliensID, Leírás, ProjAlID,
  TervezettKezdetiIdoPont, TervezettBefejezIdoPont,
  EffektivKezdetiIdoPont, EffektivBefejezIdoPont, ...)
Feladatok (FeladatID, Megnevezés, ProjektID, Leírás, AlkID,
  TervezettKezdetiIdoPont, TervezettBefejezIdoPont,
  EffektivKezdetiIdoPont, EffektivBefejezIdoPont, Befejezette)
AlFeladatok (AlFeladatID, Megnevezés, FeladatID, Leírás,
  TervezettKezdetiIdoPont, TervezettBefejezIdoPont,
   EffektivKezdetiIdoPont, EffektivBefejezIdoPont, Befejezette,
   Idotartam, Ertek, …)
```

A feladatok, amelyeket a triggerek segítségével szeretnénk megoldani a következők:

– helyességi feltételek ellenőrzése:
  - nem lehet egy projekt befejezett állapotban, ha létezik a projekthez tartozó olyan feladat, mely nincs befejezve (lásd Proj_upd trigger).

```
CREATE TRIGGER Proj_upd ON Projectek
     FOR UPDATE
AS

declare @vProjID int, @vProjAlID as int
```

```
/*  Teszteli, hogy ha a ProjAllID befejezettre módosul, vannak-e a
projekthez tartozo feladatok, melyek nincsenek befejezve */
set nocount on

select @vProjAlID = i. ProjAllID, @vProjID =i.ProjektID from inserted i
if @vProjAlID = 4
  begin
    if exists (select FeladatID from Feladatok
                   where ProjektID = @vProjID and Finished = 0)
       begin
           raiserror ('Mivel vannak a projekthez tartozo feladatok,
            melyek nincsenek befejezve, a projektet nem lehet
            befejezni',16,1)
           ROLLBACK TRANSACTION
       end
    end
set nocount off
```

Amint látjuk, a nem helyes adatok esetén a trigger hibát fog jelezni a `raiserror` segítségével, amit az alkalmazás (pl. Visual Basic, Visual C++ stb.) át tud venni és nem engedi meg a módosítást a `ROLLBACK TRANSACTION` segítségével. (lásd részletesen a tranzakciókat a 11. fejezetben)

- nem lehet egy feladat befejezett állapotban, ha létezik a feladathoz tartozó olyan tevékenység, mely nincs befejezve. Ha egy feladat Befejezette mezőjét 0-ra módosítjuk, azt jelenti, hogy azelőtt befejezett volt és most be nem fejezetté módosítjuk. Előfordulhat, hogy már a projekt is befejezettnek lett nyilvánítva, és ha megengedjük, hogy a feladatot a felhasználó be nem fejezetté módosítsa, akkor olyan állapotba jut az adatbázis, hogy a projekt be van fejezve, de van egy feladat, mely hozzá tartozik és nincs befejezve (lásd Feladat_upd)

```
CREATE TRIGGER Feladat_upd ON Feladatok
      FOR INSERT, UPDATE
 AS
declare @task_finished tinyint, @vFelID int, @vfinished tinyint
declare @proj_state int, @vProjID int

/*  Teszteli, hogy ha a feladat befejezettre módosul, vannak-e a
feladathoz tartozo alfeladatok, melyek nincsenek befejezve */

set nocount on
select @task_finished = i.Befejezette, @vFelID=i.FeladatID,
      @vProjID = i.ProjektID from inserted i
if @task_finished = 1
  begin
    if exists (select AlfeladatID from Alfeladatok where
                FeladatID = @vFelID and Befejezette = 0)
       begin
            raiserror ('Mivel vannak a feladathoz tartozo alfeladatok,
                melyek nincsenek befejezve, a feladatot nem lehet
                    befejezni',16,1)
           ROLLBACK TRANSACTION
       end
   end

/*  Teszteli, hogy ha Befejezette mező 0-ra változik, a projekt amihez a
feladat tartozik befejezettre van-e allitva. (ProjStateID= 4) */
```

```
if @task_finished = 0
  begin
    select @proj_state = ProjAlID from Projektek
          where ProjektID = @vProjID
    if @proj_state = 4
       begin
           raiserror ('A projekt amihez a feladat tartozik be van
                fejezve, tehát nem lehet a feladatot nem befejezettre
                állítani.',16,1)
           ROLLBACK TRANSACTION
       end
    end
set nocount off
```

A Projektek esetén csak UPDATE triggert írtunk, míg a Feladatok esetén UPDATE és INSERT triggert. A Feladatok esetén a ProjektID-ra hivatkozási épség megszorítás van, ami azt jelenti, hogy nem tudunk feladatot begyűjteni projekt előtt, először léteznie kell egy projektnek és ahhoz gyűjthetünk be feladatokat. Tehát INSERT esetén a projekthez még nem tartozhat egyetlen feladat sem, nem áll fenn a helytelen állapot veszélye, mely módosításkor fennállhat. A feladatnál, előfordulhat, hogy egy befejezett projekthez akarunk beszúrni egy új, be nem fejezett feladatot.

10.4. Példák triggerekre Oracle-ban

Egy INSERT művelet beszúr egy rekordot a DIAK táblába, mely egy triggert aktivál, arra használjuk, hogy napra tegyen egy statisztikát, hogy hány 18 évnél fiatalabb diákot gyűjtöttünk be egy adott esetben. Ez a trigger, kezdeti értéket ad egy olyan változónak, melyet arra használunk, hogy megszámoljuk az ezen az úton bevitt új rekordokat. A trigger műveletét az INSERT előtt kell végrehajtani. Ha szükségünk van a begyűjtött rekordban szerepelő az információra, ami a begyűjtött rekordban, akkor a triggert az INSERT után kell végrehajtani.

```
 CREATE TRIGGER init_count BEFORE INSERT ON Diakok   /*event*/
     DECLARE
          count INTEGER;
     BEGIN  /*action*/
          count :=0;
     END

CREATE TRIGGER inti_count AFTER INSERT ON Diakok   /*event*/
WHEN (new.age<18)  /* condition : „new” éppen beillesztett rekord*/
FOR EACH ROW
BEGIN /*action*/
     count :=count + 1;
END
```

A példa bemutat egy más szempontot is. A triggert minden rekord esetén végre kell hajtani, vagy csak egyszer, az aktiváló parancs szintjén. Ha a folyamat a változó rekordtól függ, például a begyűjtött diák esetén kell elemezni a diák korát, hogy növeljük vagy nem a count változó értékét, ahhoz a trigger eseményt minden megváltozott rekord esetén értelmezni kell, tehát sor szintű triggert alkalmazunk.

10.5. Gyakorlatok

10.1 Legyen a 10.3-ban leírt adatbázis, mely egy software cég projektjeiről szóló információkat tárolja.

a) Irjunk INSERT, UPDATE Triggert az AlFeladat táblára, mely kiszámolja az Idotartam és Ertek attribútum értékét a következőképpen:

- ha a Befejezette attribútum értéke 1, akkor
  Idotartam = (EffektivBefejezIdoPont – EffektivKezdetiIdoPont)/60.
  Ertek = Idotartam * Oraber (egy feladatot és a hozzá tartozó összes alfeladatot egy alkalmazott (AlkID) végez és annak az Oraber-et kell itt figyelembe venni)
- ha a Befejezette attribútum értéke 0, akkor
  Idotortam = 0; Ertek = 0

b) Egészítsük ki a INSERT, UPDATE triggert a Feladat táblára, mely az EffektivKezdetiIdoPont és EffektivBefejezIdoPont attribútum értékét a következőképpen módosítja:

- ha a Befejezette attribútum értéke 1, akkor
  A feladathoz tartozó összes alfeladat közül
- ha a Befejezette attribútum értéke 0, akkor
  Idotortam = 0; Ertek = 0

11. Tranzakciókezelés

A legtöbb adatbázist több felhasználó is használja. Banki vagy repülőgép-helyfoglalási alkalmazások esetén az adatbázist egyidejűleg akár több száz művelet is módosíthatja. A banki alkalmazások esetén ezen műveleteket kezdeményezhetik például banki alkalmazottak, bankautomaták, vagy olyan üzletek, ahonnan valaki bankkártya segítségével vásárolt stb. Repülőgép-helyfoglalási alkalmazások esetén több repülőjegy- iroda is árulja ugyanazokat a jegyeket. Ilyen esetekben konfliktushelyzetek léphetnek fel, például egyik felhasználó pénz utal ugyanarra a bankszámlára, amelyikről egy másik fehasználó kivesz pénz. Ugyanazt a repülőjegyet kétszer is eladják. A mai adatbázis alkalmazások esetén nem fordulnak elő hasonló problémák, mert az adatbázis-kezelő rendszernek van rá gondja, hogy az adatbázis a tranzakciók során konzisztens maradjon.

Több felhasználós adatbázis esetén nagyon fontos, hogy mindig legyen egy helyes adatbázis, amit a rengeteg felhasználó használhasson. Ha egyfelhasználós alkalmazásunkban hiba lép fel, csak az egy felhasználó nem tudja az alkalmazást használni, ez sem elfogadott, de a többfelhasználós alkalmazásoknál megengedhetetlen. Mindig kell legyen az adatbázisnak egy helyes állapota, amit hiba esetén vissza lehet állítani. A *tranzakció* az, ami ebben segít.

11.1. A tranzakció fogalma

Tranzakció: az adatbázison végzett műveletek (parancsok) sorozata, mely az ABKR szempontjából egy egységet alkot, olyan értelemben, hogy vagy az egész tranzakciót végrehajtja, ha ez lehetséges, ha nem lehetséges, akkor egyáltalán semmi változtatást ne végezzen az adatbázison. Lehessen parancsra vagy automatikusan a tranzakciókból elvégzett operációkat visszagörgetni, az adatbázis egy helyes állapotát visszaállítani.

A tranzakciókra vonatkozó feltételezésünk: ha a tranzakciót minden más tranzakciótól függetlenül és rendszerhiba nélkül végrehajtjuk, és ha induláskor az adatbázis konzisztens (helyes) állapotban volt, akkor a tranzakció végrehajtása után ismét konzisztens állapotban lesz.

A tranzakció a *konkurencia*, a *helyesség* és a *visszaállítás* egysége. Részletesen is tárgyaljuk majd mindegyiket. Másképp fogalmazva, szokták a tranzakciók ACID tulajdonságait említeni:

A (atomicity) atomi, a tranzakció atomi, vagyis a tranzakciót alkotó műveletek egy egységet alkotnak.

C (consistency), a tranzakció a helyesség (konzisztencia) egysége, az adatbázist egy helyes állapotból egy másik helyes állapotba alakít.

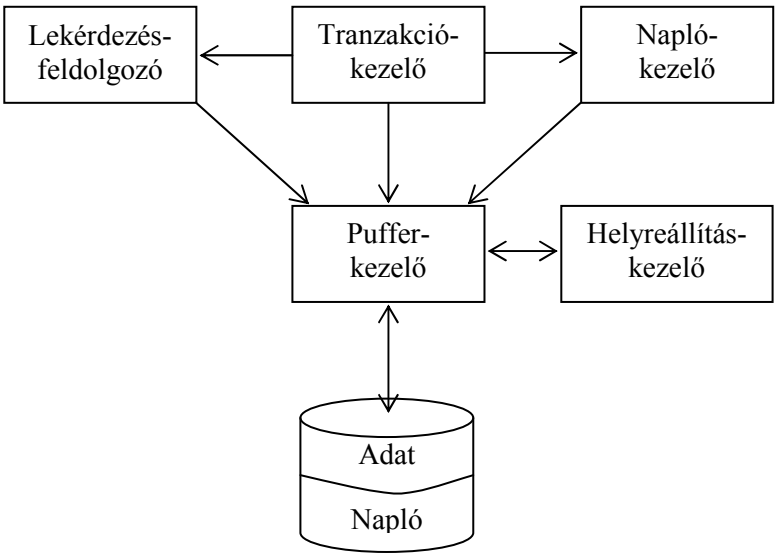
I (isolation), különböző tranzakciók egymástól elszigetelten futnak, mintha egymás után hajtódnának végre, de valójában egyidejűleg versengenek az adatbázis elemekért.

D (durability) tartósság, ha a tranzakció elért a végpontjához (COMMIT), az általa végzett adatbázis módosítások véglegesek, még ha közben esetleg hiba lép fel.

A tranzakciók helyes végrehajtásának biztosítása a *tranzakciókezelő* feladata. A tranzakciókezelő többek között jelzéseket ad a naplókezelőnek, hogy a szükséges információkat a naplóállományban tárolja, s biztosítsa, hogy a párhuzamosan végrehajtott tranzakciók egymás működését ne zavarják.

Az ABKR naplóállományokat (log, journal) tárol a háttértárolón, ahol minden tranzakció esetén a műveletekről információkat tárol, például a módosított objektumok változtatás előtti és változtatás utáni értékeit. Ha valamely változtatást szükség esetén vissza kell görgetni, a rendszer az eredeti érték visszaállítására a megfelelő értéket a naplóállományból használja fel.

A tranzakciókezelő a tranzakció tevékenységéről üzeneteket küld a naplókezelőnek, üzen a pufferkezelőnek, hogy másolhatja-e vagy sem a pufferek tartalmát a háttértárolóra, üzen a lekérdezés-feldolgozónak a tranzakcióban szereplő lekérdezések végrehajtásával kapcsolatban, lásd 11.1 ábrát.



11.1. ábra: A Tranzakciókezelő kapcsolata az ABKR más komponenseivel

A naplókezelő a naplóállományt tartja karban, együttműködik a pufferkezelővel, mivel a naplózandó információkat a pufferekből bizonyos időnként a háttértárolóra kell másolni. A naplóállomány a háttértárolón található.

A helyreállításkezelő akkor dolgozik, ha valami hiba történt. Felhasználva a naplóállományt helyreállítja az adatokat, de a háttértárlót csak a pufferkezelőn át tudja elérni.

11.2. A tranzakciók alaptevékenységei

Lássuk milyen az adatbázis és a tranzakció kölcsönhatása. Ennek a kölcsönhatásnak három fontos színhelye van:

- az adatbázis elemeit tartalmazó lemezblokkok területe;
- a pufferkezelő által használt memóriaterület
- a tranzakció memóriaterülete.

Mielőtt a tranzakció egy adatbáziselemet használhatna, azt egyszer a lemezről a pufferbe kell hozni, ha még nincs ott, azután a tranzakció a saját memóriaterületébe olvashatja.

A módosított vagy új értéket a tranzakció memóriaterületéből a megfelelő pufferbe kell másolni. A pufferek tartalmának a lemezre írása a pufferkezelő feladata. Előfordul, hogy nem írja ki rögtön minden módosítás után a puffer tartalmát a lemezre. Az ABKR egy bizonyos ideig megengedi a módosításnak csak a pufferben való végrehajtását, egyfelől az I/O műveletek számának a csökkentése érdekében, másfelől előfordulhat, hogy a tranzakciót vissza kell görgetni és akkor az ABKR nem végzett fölösleges I/O műveleteket. Ha ugyanazt az adatbáziselemet több tranzakció is használja, mindegyik először a pufferben keresi és csak ha nincs a pufferben, akkor olvassa be a lemezről, így egyik tranzakció módosítását a másik tranzakció látja.

A tranzakciókezelő algoritmusainak részletes tanulmányozása érdekében szükségünk lesz egy pár alapműveletre, használjuk a továbbiakban a következő jelöléseket:

- INPUT (*X*): Az *X* adatbázis elemet tartalmazó lemezblokk másolása a pufferbe.
- READ (*X*, *v*): Az *X* adatbáziselem bemásolása a tranzakció *v* lokális változójába. Ha az *X* adatbáziselemet tartalmazó blokk nincs a pufferben, akkor előbb az INPUT(*X*) hajtódik végre, a *v* változó *X* értékét azután kapja meg.
- WRITE (*X*, *v*): A *v* lokális változó tartalma az *X* adatbáziselem pufferbeli tartalmába másolódik. Ha az *X* adatbáziselemet tartalmazó blokk nincs a pufferben, akkor előbb az INPUT (*X*) hajtódik végre, azután másolódik át a *v* lokális változó értéke a puffebeli *X*-be.
- OUTPUT (*X*): Az *X* adatbáziselemet tartalmazó puffer kimásolása lemezre.

Ezen műveleteknek akkor van értelme, ha egy adatbázis elem elfér egy lemezblokkban, illetve egy pufferben. Adatbáziselem lehet egy sor az adatbázisból. Ha az adatbázis-kezelő rendszer nem enged meg hosszabb sorokat, mint egy lemezblokk mérete, addig nincs semmi gond a fenti műveletekkel. Ha lehetnek hosszabb sorok, mint egy blokk, abban az esetben egy blokkot tekinthetünk adatbáziselemnek. A naplózási mechanizmus, melyet arra használunk, hogy a tranzakció ne fejeződhessen be az *X* kiírása nélkül, atomi, azaz *X* összes blokkját lemezre írja, vagy semmit sem ír ki. A gyakorlatban, a legtöbb esetben több sor is befér egy blokkba. A továbbiakban úgy tekintjük, hogy egy adatbáziselem nem nagyobb egy blokknál.

A tranzakciókat megadhatja explicit a felhasználó, illetve vannak implicit tranzakciók.

Minden SQL parancs egy implicit tranzakció: a CREATE TABLE például vagy létrehozza a táblát vagy nem, ha netán nagyon sok az oszlopok száma és mondjuk felét az oszlopoknak létrehozza, a többit valamilyen hiba miatt nem, ilyen nem fordulhat elő, akkor egyáltalán nem hozza létre a táblát. Az INSERT, UPDATE és DELETE parancsok is implicit tranzakciók. Például az:

```
INSERT  INTO Alkalmazottak VALUES (123454, 'Kovács Lehel', 2, 500)
```

parancsot csak akkor hajtja végre, ha sikerül az összes megadott értéket az adatbázisba beírnia és a megfelelő indexet is napra tenni (SzemSzám elsődleges kulcs), nem hagyja az adatbázist inkonzisztens állapotban.

Más példa, az alkalmazottak fizetését megemelik 20%-kal a következő parancs segítségével:

```
UPDATE Alkalmazottak SET Fizetés = Fizetés * 1.2
```

A parancs akkor hajtódik végre, ha minden alkalmazott esetén végre tudja hajtani a módosítást, ha esetleg van valami megszorítás bizonyos alkalmazottaknál, hogy nem lehet nagyobb a fizetésük, mint egy adott összeg, akkor nem tudja a módosítást elvégezni és a tranzakciót visszagörgeti, azoknál, akiknél már módosította, visszaállítja a kezdeti értékre, a naplóállomány segítségével.

Explicit megadott tranzakció esetén a tranzakciónak van egy kezdő-, illetve egy végpontja. Speciális parancsokkal megadjuk a tranzakció kezdetét, illetve végét.

A tranzakció kezdetét a

```
BEGIN TRANSACTION
```

parancs segítségével adhatjuk meg.

```
COMMIT TRANSACTION
```

utasítással egy tranzakció sikeres befejeződését jelezzük. Egy sikeresen befejeződött tranzakció kezdete óta végrehajtott utasítások által az adatbázison végzett módosítások *véglegesíthetőek*. A COMMIT utasítás végrehajtása előtt az adatmódosítások még nem véglegesítődnek.

```
ROLLBACK TRANSACTION
```

utasítás segítségével egy tranzakció sikertelen befejezését jelezzük Ha egy tranzakció ROLLBACK-kel fejeződik be, a tranzakció utasításai által végzett módosításokat a rendszer meg nem történtekké teszi, *visszagörgeti* őket.

11.3. Helyrellítás hiba esetén

A naplóállomány sorai naplóbejegyzések (log records), melyek a tranzakció tevékenységeit tárolják, ezek segítségével rekonstruálható az amit a tranzakció tett a rendszerhiba előtt. Ha rendszerhiba fordul elő, a naplóbejegyzéseket felhasználva a helyreállításkezelő egy helyes adatbázist kell visszaállítson.

Több típusú naplózás is van: semmisségi (undo), helyrehozó (redo), illetve semmisségi/helyrehozó.

11.3.1. Semmisségi (undo) naplózás

A semmisségi naplózás abban az esetben használható helyreállításra, ha nem lehet tudni, hogy a tranzakció helyesen végrehajtódott-e vagy sem, módosításai tárolódtak-e a lemezen, és ezért semmissé kell tenni minden változtatást, amit a tranzakció esetleg végzett és az adatbázist a tranzakció indulása előtti állapotba kell visszaállítani.

A naplóblokkok is először a memóriában jönnek létre és a pufferkezelő bizonyos időközönként a lemezre kell írja őket. A naplóbejegyzések különböző formájúak lehetnek, lássunk egy párat:

< START T > − a T tranzakció kezdetét jelzi.

<COMMIT T> − a T tranzakció sikeresen befejeződött, nem akar több módosítást végezni az adatbázison. Mivel nem tudjuk pontosan a pufferkezelő mikor másolja a memória tartalmát a lemezre, nem biztos, hogy ha <COMMIT T> bejegyzést találunk a naplóállományban a tranzakció összes művelete már a lemezre is ki van mentve. A naplókezelő kikényszerítheti a mentést, a különböző naplózási módszerek különbözőképpen oldják meg ezt.

<ABORT T> − a T tranzakció nem tudott sikeresen befejeződni, ezért a módosításokat nem kell lemezre másolni. Ha esetleg a módosítások egy része már ki volt mentve a lemezre, akkor a tranzakció indulása előtti állapotot vissza kell állítani.

< T, X, v > − a T tranzakció módosította az X adatbáziselemet, melynek módosítás előtti értéke v volt. Ez a módosítás csak a pufferbeli értékre vonatkozik, tehát a WRITE műveletre és nem az OUTPUT-ra. Amint látjuk, a semmisségi naplózás a tranzakció által módosított adatbáziselemnek csak a módosítás előtti értékét tárolja, az új (módosított) értéket nem, mivel ezt a fajta naplózást használó rendszerekben a helyreállításkezelő feladata a tranzakció hatásának a semmissé tétele.

Ahhoz, hogy a semmisségi naplózást helyreállításra tudjuk használni, a következő szabályokat kell betartanunk:

$U1$: A < T, X, v > típusú naplóbejegyzést kell először a naplóállományba kiírni és utána lehet az X adatbáziselem új értékét is lemezre írni.

$U2$: Ha a tranzakció sikeresen ért véget, először a tranzakció által módosított összes adatbáziselemet a lemezre kell írni és utána a COMMIT naplóbejegyzés is minél hamarabb a naplóállományba kell menteni.

Figyelembe véve ezeket a szabályokat, egy tranzakció végrehajtása során a következő sorrendet kell betartanunk:

A START `naplóbejegyzés kiírása` a naplóállományba.
Ismételd minden módosított adatbáziselemre:
    A naplóállományba azon `naplóbejegyzés kiírása`, amely az adatbáziselem módosítására vonatkozik.
    `Adatbáziselem` módosított értékének a `lemezre írása`
A COMMIT `naplóbejegyzés kiírása` a naplóállományba

A naplókezelő egy FLUSH LOG parancsot használ, melynek segítségével kikényszeríti, hogy a pufferkezelő a lemezre írja a még ki nem mentett naplóblokkokat. A tranzakciókezelő az OUTPUT parancs segítségével szólítja fel a pufferkezelőt, hogy az adatbáziselem módosított értékét a lemezre írja.

11.1. példa: Tekintsük a következő műveleteket: eladunk a raktárból egy bizonyos m mennyiséget egy adott áruból egy adott vevőnek. A vevővel szerződése van a cégnek. Követjük, hogy mennyi az elszállított mennyiség, tehát az eladott mennyiséget hozzá kell adjuk az elszállított mennyiséghez. Legyen X azon adatbáziselem, mely egy adott áru raktáron levő mennyiségét tárolja, Y pedig, ugyanazon áru egy adott vevőnek egy szerződésen belül elszállított mennyisége. Egy tranzakció segítségével oldjuk meg a feladatot, mert nem szeretnénk, hogy a raktáron levő mennyiségből levonjuk, de a szállított mennyiséghez ne adjuk hozzá, mert ebben az esetben az adatbázis inkonzisztens állapotba jut és az áru el nem szállítottnak jelenik meg. Ez egy példa arra, hogy a tranzakció a *helyesség egysége*. A megoldás röviden:

```
BEGIN TRANSACTION
X:= X − m;
Y:= Y + m;
COMMIT TRANSACTION
```

A 11.2 ábrán láthajuk a tranzakció által végzett tevékenységeket, ahol az $m = 10$, a v változó megfelelő értékeit, az X, illetve Y adatbázis elemek memóriabeli (*Mem*), illetve háttértárolón (*D*) levő értékeit és a naplóbejegyzéseket (*Napló*).

A 8-as lépéstől betartottuk a semmisségi naplózás szabályait. □

	Tevékenység	***v***	***Mem–X***	***Mem–Y***	***D–X***	***D–Y***	***Napló***
1)							< START *T* >
2)	READ(*X*, *v*)	50	50		50	20	
3)	*v*: = *v* – 10	40	50		50	20	
4)	WRITE(*X*,*v*)	40	40		50	20	< *T*, *X*, 50 >
5)	READ(*Y*,*v*)	20	40	20	50	20	
6)	*v*:= *v* + 10	30	40	20	50	20	
7)	WRITE (*Y*, *v*)	30	40	30	50	20	< *T*, *Y*, 20 >
8)	FLUSH LOG						
9)	OUTPUT (*X*)	30	40	30	40	20	
10)	OUTPUT (*Y*)	30	40	30	40	30	
11)							< COMMIT *T*>
12)	FLUSH LOG						

11.2. ábra: Tevékenységek és naplóbejegyzések a 11.1 példa esetén

11.3.2. Helyreállítás semmisségi naplózással

A helyreállítás-kezelő első feladata a tranzakciók elemzése, melyek azon tranzakciók, melyek sikeresen befejeződtek és melyek nem.

Ha van < COMMIT *T*s> naplóbejegyzés, akkor a semmisségi naplózás második szabálya alapján a *T*s tranzakció által végzett módosítások már lemezre íródtak. Tehát a *T* tranzakció nem hagyta az adatbázist inkonzisztens állapotban.

Ha a naplóállományban találunk < START T_h > bejegyzést, de nem találunk <COMMIT T_h> bejegyzést, azt jelenti, hogy T_h nem komplett tranzakció és hatását semmissé kell tenni, vagyis a T_h által módosított összes adatbáziselemet vissza kell állítani a T_h indulása előtti értékre. A semmisségi naplózás első szabálya biztosítja, hogy adatbáziselem módosítása előtt a naplóállományba kerül a < T_h, *X*, *v*> bejegyzés, innen hiba esetén vissza tudjuk állítani a *v* értéket.

A naplóállományban sok tranzakció tevékenysége is naplózva van, ugyanazt az adatbázis-elemet több tranzakció is módosíthatja. A helyreállítás-kezelő a naplóállományt a végéről kezdi átvizsgálni. Megjegyzi azokat a tranzakciókat, amelyekre vonatkozóan talál < COMMIT *T* >, vagy < ABORT *T* > bejegyzést.

Ha < T_s, *X*, *v*> bejegyzést talál és a T_s-re már találkozott (hátulról jövet) <COMMIT T_s> bejegyzéssel, azt jelenti, hogy T_s sikeresen végetért, a módosítások ki is lettek mentve a lemezre. T_s tranzakciót nem kell semmissé tenni.

Ha a helyreállítás-kezelő <T_h, *X*, *v*> bejegyzést talál és a T_h-ra nem találkozott <COMMIT T_h> bejegyzéssel, a T_h hatását semmissé kell tegye, vagyis *X* adatbáziselem értékét *v*-re kell állítsa. Ha a tranzakció abortált, akkor is a *v* értéket kell visszaállítani.

Miután a helyreállítás-kezelő befejezte a munkáját, a semmissé tett tranzakciókra vonatkozóan <ABORT> bejegyzést helyez el a naplóállományban, illetve FLUSH LOG-ot, és utána mehet tovább a munka, jöhetnek az új tranzakciók.

Ha a fenti példában a hiba a COMMIT után állt be, nincs semmi munkája a helyreállítás-kezelőnek.

Ha a hiba a COMMIT naplóállományba való beírása körül történt, minden attól függ, hogy a COMMIT bekerült-e vagy sem a naplóállományba. Ha igen, akkor a tranzakció sikeresnek tekinthető, nem kell semmissé tenni. Ha viszont a COMMIT nincs kiírva a naplóállományba, akkor

a helyreállítás-kezelő a naplóállományban visszafele haladva találkozik a <T, Y, 20> bejegyzéssel és az Y adatbáziselem értékét 20-ra állítja vissza, hasonlóan a <T, X, 50> bejegyzés esetén az X értékét 50-re állítja. Mikor befejezte a munkát, <ABORT T> bejegyzést ír a naplóállományba.

Ha a hiba az OUTPUT műveletek körül történt, a COMMIT még nem került a naplóállományba, így a helyreállítás-kezelő mindkét módosítást semmissé fog tenni.

Ha az első FLUSH LOG előtt áll be a hiba, akkor nincs mit helyreállítani, mivel még nem történt lemezre írás a semmisségi naplózás első szabálya alapján.

Abban az esetben, ha a helyreállítás közben újra hiba áll be, ismét indítjuk a helyreállítást, mivel az többször is végrehajtható. Ugyanazt a régi v értéket többször is visszaállíthatjuk.

11.3.3. Ellenőrzőpont-képzés

Annak érdekében, hogy a helyreállítás-kezelő ne kelljen az egész naplóállományt átvizsgálja, ún. ellenőrzőpontokat helyeznek el a naplóállományba. A rendszerben nagyon sok tranzakció fut egyidőben, ezért az ellenőrzőpont elhelyezése a következő lépések segítségével történik:

1. <START CKPT (T_1, T_2, ..., T_m) > naplóbejegyzés lemezre írása (FLUSH), ahol T_1, T_2, ..., T_m az éppen aktív tranzakciók.
2. A T_1, T_2, ..., T_m tranzakciók sikeres vagy sikertelen befejezését megvárni, miközben újabb tranzakciók indulhatnak. Ez azt jelenti, hogy a tranzakciók módosításainak a lemezre mentése már megtörtént, az összes T_1, T_2, ..., T_m tranzakcióra a <COMMIT T_i> i=1,2,..., m bejegyzés a naplóállományba került.
3. Ha T_1, T_2, ..., T_m tranzakciók befejeződtek (ez azt jelenti, hogy azon tranzakciók, melyek sikeresek voltak, módosításainak a lemezre mentése már megtörtént, illetve a <COMMIT> bejegyzések naplóállományba kerültek, amelyek abortáltak, azok esetében az <ABORT> bejegyzés került a naplóállományba, a módosításokat nem kellett lemezre írni), <END CKPT> naplóbejegyzés lemezre írása (FLUSH).

11.3.4. Helyreállítás ellenőrzőponttal kiegészített semmisségi naplózás segítségével

Rendszerhiba esetén a visszaállítás a következőképpen történik: A naplót a rendszer visszafele olvassa.

- Ha először egy <END CKPT> bejegyzést talál, azt jelenti, hogy a legközelebbi (visszafele haladva) <START CKPT (T_1, T_2, ..., T_m) > -ig az összes be nem fejezett tranzakciót megtaláljuk. Ennél a <START CKPT (T_1, T_2, ..., T_m) >-nél megállhatunk a naplóállomány olvasásában, a korábbiak nem érdekesek, mivel az ellenőrzőpont kezdetének képzésekor, az összes aktív tranzakció bekerül a listába. Az <END CKPT> azt mutatja, hogy a T_1, T_2, ..., T_m tranzakciók esetén a mentés megtörtént. Közben indulhattak más tranzakciók, azok közül a be nem fejezett tranzakciókat semmissé kell tennünk, helyreállítsuk a naplóállományban elmentett régi értékre (<T,X, v> típusú bejegyzésekből).
- Ha a helyreállítás-kezelő először <START CKPT (T_1, T_2, ..., T_m) > naplóbejegyzést talál, azt jelenti, hogy a hiba az ellenőrzőpont képzése közben történt, tehát a T_1, T_2, ..., T_m tranzakciók nem fejeződtek be. Ebben az esetben, ezen tranzakciók közül meg kell keresnünk azt, mely legkorábban indult (<START> bejegyzés). Nem elég az előző <END CKPT>-ig visszamenni, mert indulhattak a tranzakciók az előző ellenőrzőpont képzése közben is. A T_1, T_2, ..., T_m tranzakciókat semmisségi naplózás segítségével helyre kell állítanunk.

11.2. példa: Legyenek a következő naplóbejegyzések:

<START T_1>
<T_1, X, 10>
<START T_2>
< T_2, Y, 20>
<START CKPT (T_1, T_2)>
< T_2, Z, 30>
<START T_3>
< T_1,P, 40>
<COMMIT T_1>

< T_3, Q, 50>
<COMMIT T_2>
<END CKPT>
< T_3, R, 60>

Tegyük fel, hogy ebben a pontban hiba lép fel. Hátulról elemezve a naplóállományt, először <END CKPT> bejegyzést találunk, ez arról biztosít, hogy a megfelelő <START CKPT (T_1, T_2)> bejegyzésben található T_1, T_2 tranzakciók sikeres véget értek és a módosításaik lemezre mentése is megtörtént. Viszont a T_3 tranzakció nem befejezett, tehát semmissé kell tenni hatásait, R-et 60-ra, Q-t 50-re kell visszaállítanunk. T_3-on kívül más tranzakció nem indul az <END CKPT> és <START CKPT (T_1, T_2)> között.

Abban az esetben, ha az ellenőrzőpont képzése közben lép fel hiba, mondjuk a <COMMIT T_1> után. Visszafele elemezve a naplóállományt, először a <START CKPT (T_1, T_2)> bejegyzést találjuk, tehát a T_1, T_2 tranzakciók valószínű nem befejezett tranzakció. Mivel a T_1 esetén a COMMIT bejegyzést megtaláljuk, csak a T_2 által végzett módosításokat kell semmissé tegyük, Z-t 30-ra, Y-t 20-ra állítjuk. T_3 sem befejezett, T_3 módosításait is semmissé kell tennünk, de COMMIT T_1 előtt nem volt módosítása.□

11.3.5. Helyrehozó (redo) naplózás

A semmisségi naplózás problémája, hogy csak akkor tudjuk a tranzakciót befejezni, ha az adatbázison végzett összes módosítás már lemezre íródott. Takarékoskodhatnánk a lemezre írással, ha az adatbázis módosítások csak a memóriában történnének. Előfordul, hogy több felhasználó is módosítja ugyanazt az adatbáziselemet, így csak az utolsó értéket kellene kiírnunk. A helyrehozó naplózás elkerüli az adatbáziselem azonnali kiírását a lemezre, viszont a naplóállomány minden módosítást rögzít. Amíg a semmisségi naplózás a be nem fejezett tranzakciók hatásait semmissé teszi, a befejezettekkel pedig nem tesz semmit, addig a helyrehozó naplózás a be nem fejezett tranzakciókat figyelmen kívül hagyja és megismétli a normálisan befejezett tranzakciók által végzett változtatásokat.

A helyrehozó naplózás esetében a <T, X, v> alakú naplóbejegyzések esetén az X adatbáziselem a T tranzakció által módosított (új) értékét tartalmazza. A helyrehozó naplózás szabálya:

R1: A <T, X, v> és a <COMMIT T> naplóbejegyzéseket kell először a naplóállományba kiírni és utána lehet az X adatbáziselem új értékét is lemezre írni.

Ezt a szabályt figyelembe véve, egy tranzakcióra vonatkozó műveleteket a következő sorrendben kell végrehajtani:

A START `naplóbejegyzés kiírása` a naplóállományba.
Ismételd minden módosított adatbáziselemre:
    A naplóállományba azon `naplóbejegyzés kiírása`, melyek az adatbáziselem módosítására vonatkozik.
A COMMIT `naplóbejegyzés kiírása` a naplóállományba.
Ismételd minden módosított adatbáziselemre:
    `Adatbáziselem` módosított értékének a `lemezre írása.`

11.3. példa: A 11.1. példa átírva helyrehozó naplózással:

	Tevékenység	*v*	*Mem*-X	*Mem*-Y	*D*-X	*D* -Y	*Napló*
1)							<START *T*>
2)	READ(*X*, *v*)	50	50		50	20	
3)	*v*: = *v* – 10	40	50		50	20	
4)	WRITE(*X*,*v*)	40	40		50	20	<*T*, *X*, 40>
5)	READ(*Y*,*v*)	20	40	20	50	20	
6)	*v*:= *v* + 10	30	40	20	50	20	
7)	WRITE (*Y*, *v*)	30	40	30	50	20	<*T*, *Y*, 30>
8)							<COMMIT *T*>
9)	FLUSH LOG						
10)	OUTPUT (*X*)	30	40	30	40	20	
11)	OUTPUT (*Y*)	30	40	30	40	30	

11.3. ábra: Tevékenységek és naplóbejegyzések a 11.3 példa esetén

11.3.6. Helyreállítás helyrehozó naplózás esetén

Az R1 szabály fontos következménye, ha a naplóban nincs <COMMIT *T*> bejegyzés, akkor a módosítások az adatbázisban nem történtek meg, tehát nem kell őket visszaállítani. Tehát a be nem fejezett tranzakciókkal nincs gondja a helyreállítás-kezelőnek. Probléma a befejezett tranzakciókkal van, mivel nem tudjuk, hogy lemezre íródtak-e vagy sem, mivel először a naplóállományba a COMMIT kerül. De a helyrehozó naplózás a módosított (új) értéket a naplóbejegyzésekben tárolja, így a helyreállítás-kezelő ezeket az értékeket lemezre írja, ha esetleg már ki voltak írva, akkor ismét kíirja. Hiba esetén a következőket kell tennünk:

1. Megkeresni a befejezett tranzakciókat.
2. A naplóállományt az elejétől (ellentétben a semmisségi naplózás helyreállításával) kezdve végigjárni. (Mivel több befejezett tranzakció is adhatott új értéket ugyanazon *X* adatbáziselemnek, a végső érték kell az érvényes legyen.) Minden <*T*, *X*, *v*> típusú bejegyzés esetén
   a) ha *T* nem befejezett, nincs semmi dolgunk
   b) ha *T* befejezett tranzakció, akkor a v értéket az *X* adatbáziselembe kiírni.
3. Minden *T* be nem fejezett tranzakcióra vonatkozóan <ABORT *T*> bejegyzést írni a naplóállmányba és a naplót lemezre írni.

11.4. példa: A 11.3. példa esetén a következő esetek állhatnak fenn:

i) A hiba a 9-es lépés után következik, a <COMMIT *T*> már lemezre íródott, a helyreállítás *T*-t befejezettnek tekinti és minden módosítást (esetleg ismét) kiír a lemezre (*X* értéke 40, *Y* értéke 30 lesz).

ii) A hiba a 8-as és 9-es lépések között jelentkezik, attól függően, hogy a <COMMIT *T*> bejegyzés ki lett-e mentve a naplóállományba, előbbi i) eset, különben iii) eset.

iii) A hiba a 8-as lépés előtt jelenik meg, *T* be nem fejezett tranzakciónak számít, az *X*, illetve *Y* értéke nem változott meg, <ABORT *T*> bejegyzés kerül a naplóállományba. □

11.3.7. Helyrehozó naplózás ellenőrzőpont-képzés segítségével

Ellenőrzőpontok elhelyezése esetén a probléma az, hogy a befejezett tranzakciók módosításainak a lemezre írása sokkal később is történhet, ezért az ellenőrző pontok kezdete és vége közötti időben lemezre kell írnunk az összes olyan adatbáziselemet, melyet befejezett tranzakciók módosítottak. Ennek érdekében a pufferkezelőnek nyilván kell tartania, az ún. piszkos puffereket, melyek a már végrehajtott, de lemezre még ki nem írt módosításokat tárol. Az ellenőrzőpont végét beállíthatjuk, anélkül, hogy megvárjuk az aktív tranzakciók (normális vagy abnormális) befejezését, mivel a lemezre írás még akkor sem történik meg.
Ellenőrzőpont képzése a helyrehozó naplózás esetén:

1. <START CKPT (T_1, T_2, ..., T_m) > naplóbejegyzés lemezre írása, ahol T_1, T_2, ..., T_m az összes éppen aktív (még nem befejezett) tranzakció.
2. Azon adatbáziselemek lemezre írása, melyeket olyan tranzakciók írtak a pufferekbe, melyek a START CKPT naplóba írásakor a COMMIT-jukhoz már eljutottak, de a puffereik még nem voltak kimentve.
3. <END CKPT> naplóbejegyzés lemezre írása.

11.5. példa: Legyenek a következő naplóbejegyzések:

<START T_1>
<T_1, X, 10>
<START T_2>
<COMMIT T_1>
< T_2, Y, 20>
<START T_3>
<START CKPT (T_2, T_3)>
< T_2, Z, 30>
< T_3, P, 40>
<END CKPT>
<COMMIT T_2>
< T_3, R, 60>
<COMMIT T_3>

Az ellenőrzőpont-képzés pillanatában a T_1 tranzakció elért a COMMIT-hoz, de nem biztos, hogy az általa módosított X adatbáziselem lemezre íródott. Az ellenőrzőpont- képzés pillanatában T_2 és T_3 tranzakciók az aktív tranzakciók. Mielőtt az ellenőrzőpont végét jelző <END CKPT> bejegyzést a naplóállományba írnánk, a T_1 tranzakció által módosított X adatbáziselem értékét lemezre kell mentenünk. □

11.3.8. Helyreállítás ellenőrzőponttal kiegészített helyrehozó naplózás segítségével

Helyreállítás esetén először hátulról keresünk a naplóállományban az utolsó ellenőrzőpontig. Két eset fordulhat elő, attól függően, hogy az utolsó ellenőrzőpont-bejegyzés a START vagy az END.

- Ha a hiba előtt a naplóállományba bejegyzett utolsó ellenőrzőpont-bejegyzés <END CKPT>, akkor tudjuk, hogy azokkal a tranzakciókkal, melyek elérték a COMMIT pontjukat a <START CKPT (T_1, T_2, ..., T_m)> naplóbejegyzés előtt nincs többet gondunk, mert az END CKPT előtt ezen tranzakciók által végzett módosítások lemezre íródtak. Helyreállítani maradnak a T_1, T_2, ..., T_m tranzakciók és azok, melyek indultak az ellenőrzőpont-képzés közben, legyenek ezek T_{m+1}, T_{m+2}, ..., T_p. Ennek érdekében meg kell keresnünk a T_1, T_2, ..., T_p tranzakciók közül, melyik indult legkorábban (<START T_e> bejegyzést, melyre jellemző, hogy az összes <START T_i>-nél, $i = \overline{1, p}, i \neq e$ korábban található a naplóban). Nem elég az első ellenőrzőpontig visszamennünk, mivel egy tranzakció több ideig is lehet aktív, több ellenőrzőpontban is szerepel, mint aktív tranzakció. Viszont elég visszamennünk a <START T_e>-ig, mivel az utolsó ellenőrzőpontban aktív összes tranzakció közül így azonosítottuk a legelsőt. A <START T_e> ponttól indulva a naplóban, most már előre haladunk és használjuk a helyrehozó naplózás helyreállítási technikáját a T_1, T_2, ..., T_p tranzakciókra (lásd 11.3.6.). Tehát ezek közül a be nem fejezett tranzakciók nem érdekesek, mivel nem írtak még semmit lemezre és nem érték el a COMMIT pontjukat. Azon tranzakciók esetében, melyek már elérték a COMMIT pontjukat a naplóban megtaláljuk az általuk módosított új értékeket és ezeket felhasználva az adatbáziselemek új értékét a lemezre is kiírjuk.
- Abban az esetben, ha a hiba előtt utoljára a naplóállományban <START CKPT (T_1, T_2, ..., T_m)> típusú bejegyzést találunk, nem biztos, hogy ezek előtt COMMIT pontig elért tranzakciók által végzett módosítások ki lettek mentve a pufferből. Ezen kiírás az <END CKPT>-ig kellett volna megtörténjen, de közben hiba lépett fel. Ezért visszafele kell keresnünk egy előbbi <START CKPT (U_1, U_2, ..., U_m)> bejegyzésig, és a helyrehozó naplózás helyreállítási technikáját

használva helyre kell állítanunk az összes tranzakciót, mely ezen pont után a COMMIT pontját elérte.

11.6. példa: Ha a fenti 11.5. példa naplóbejegyzései esetén a legutolsó naplóbejegyzés után hiba lép fel, visszafele haladva először <END CKPT> ellenőrzőpont bejegyzést találunk. A megfelelő START a T_2, T_3 tranzakciókra vonatkozik, ez azt jelenti, hogy a T_1 tranzakció által végzett módosítások az ellenőrzőpont végére már ki lettek mentve, marad a T_2, T_3, amivel foglalkoznunk kell. Mivel mindkettőre van COMMIT, mind a kettő módosításait helyre kell állítanunk. Megkeressük a <START T_2>-t, onnan indulva a következő adatbáziselemek értékét kell lemezre írnunk: $Y \leftarrow 20$; $Z \leftarrow 30$; $P \leftarrow 40$; $R \leftarrow 60$.

Ha a hiba a <COMMIT T_2> után lép fel, a helyzet az előbbihez hasonló, először END CKPT pontot kapunk, de mivel a T_3 nem ért el a COMMIT-ig, ezért csak a T_2 által végzett módosításokat kell lemezre írnunk ($Y \leftarrow 20$; $Z \leftarrow 30$), míg T_3-ra vonatkozóan egy <ABORT T_3> bejegyzést a naplóba.

Ha a hiba az <END CKPT> előtt jelent meg, akkor egy <START CKPT>-al találkozunk először és visszafele kell haladjunk a naplóállományban egy előző <START CKPT> pontig. A mi esetünkben nincs, akkor egészen a napló elejéig kell visszalépjünk. A T_1 az egyedüli tranzakció, mely elérte a COMMIT pontját, csak ezen tranzakció módosításait kell lemezre írnunk ($X \leftarrow 10$), illetve <ABORT T_2>, <ABORT T_3> bejegyzést a naplóállományba. □

11.3.9. A semmisségi/helyrehozó (undo/redo) naplózás

Az eddig bemutatott semmisségi, illetve helyrehozó naplózási módszereknek bizonyos hátrányai vannak:

- a semmisségi naplózás esetén nagy a lemezműveletek száma;
- a helyrehozó naplózás esetén nagy a pufferigény, mivel a módosított adatbáziselemeket több ideig is a pufferben tartja;
- ellenőrzőpontképzésnél a két módszer ellentétes igényeket támaszt. Például ha a puffer egy X és egy Y adatbáziselemet tartalmaz, az X adatbáziselemet pedig egy olyan tranzakció módosította, mely sikeres véget ért és a semmisségi naplózást használva feltétlenül lemezre kell menteni, viszont az Y adatbáziselemet egy olyan tranzakció módosította, mely esetében helyrehozó naplózást használ a rendszer és a COMMIT bejegyzés még nem került a lemezre, akkor az Y értékét nem írhatjuk lemezre. Ezek a problémák nem merülnek fel, ha adatbáziselem egy egész blokk.

A semmisségi/helyrehozó naplózás növeli a tevékenységek elvégzési sorrendjének ruggalmasságát, azáltal, hogy bővíti a naplózott információk körét. Az adatbáziselem módosítását leíró naplóbejegyzés alakja: <T, X, r, u> és ez azt jelenti, hogy a T tranzakció az X adatbáziselemet módosította, melynek módosítás előtti értéke r volt, a módosítás utáni pedig u lett. A semmisségi/helyrehozó naplózást alkalmazó rendszer a következő szabályt kell betartsa:
*UR*1: Mielőtt az X adatbáziselem értékét a lemezen módosítanánk (egy T tranzakció által végzett módosítás következtében), a <T, X, r, u> bejegyzést a naplóállományba kell írnunk.
*UR*2: A <COMMIT T> naplóbejegyzést, amint megjelenik a naplóban, nyomban lemezre kell írni.

A COMMIT bejegyzés megelőzheti vagy követheti az adatbáziselemek lemezre írását. A második szabály azért szükséges, mivel a hiba közvetlen a COMMIT bejegyzés lemezre kerülése előtt felléphet és a felhasználó már megkapta a COMMIT-ról szóló beszámolót, de valószínű, hogy az adatbáziselemek módosítása is megtörtént a lemezen, de mivel a COMMIT bejegyzés nem lett kimentve, helyreállítás esetén a tranzakciót semmissé teszi a rendszer.

11.7. példa: Lássuk az ismert 11.1. példa esetében, hogy a semmisségi/helyrehozó naplózás, milyen naplóbejegyzéseket és milyen sorrendet igényel. A <COMMIT T> bejegyzés kerülhetett volna a 9-es lépés elé, vagy a 12-es lépés után is.

	Tevékenység	*v*	*Mem−X*	*Mem−Y*	*D−X*	*D−Y*	*Napló*
1)							<START *T*>
2)	READ(*X*, *v*)	50	50		50	20	
3)	*v*: = *v* – 10	40	50		50	20	
4)	WRITE(*X*,*v*)	40	40		50	20	<*T*, *X*, 50, 40>
5)	READ(*Y*,*v*)	20	40	20	50	20	
6)	*v*:= *v* + 10	30	40	20	50	20	
7)	WRITE (*Y*, *v*)	30	40	30	50	20	<*T*, *Y*, 20, 30>
8)	FLUSH LOG						
9)	OUTPUT (*X*)	30	40	30	40	20	
10)							<COMMIT *T*>
11)	FLUSH LOG						
12)	OUTPUT (*Y*)	30	40	30	40	30	

11.4. ábra: Tevékenységek és naplóbejegyzések a 11.7 példa esetén

11.3.10. Helyreállítás semmisségi/helyrehozó naplózás esetén

Helyreállításkor a semmisségi/helyrehozó naplózás esetén a következő két alapelvet kell betartani:

– minden befejezett tranzakciót állítsunk helyre a legkorábbitól kezdve (helyrehozó naplózás segítségével)
– minden be nem fejezett tranzakciót tegyünk semmissé a legutolsótól kezdve (semmisségi naplózás segítségével)

Mivel a COMMIT bejegyzés és az adatbáziselemek lemezre mentésének a sorrendje nincs lerögzítve, előfordulhat, hogy a befejezett (COMMIT bejegyzés a naplóban) tranzakciók által végzett módosítások még nem lettek lemezre mentve, illetve a be nem fejezett tranzakciók által végzett módosítások már lemezre lettek írva. A fenti két szabályt betartva, mind a két esetet helyesen kezeli a rendszer.

11.8. példa: Abban az esetben, amikor a hiba a <COMMIT *T*> naplóbejegyzés lemezre írása után jelentkezik, a *T* tranzakciót befejezettnek tekintjük és helyreállítjuk abban a sorrendben, ahogy az események megjelentek. A naplóban megtaláljuk az *X* és *Y* adatbáziselemek új értékét ($X \leftarrow 40$, $Y \leftarrow 30$) és ezeket írjuk (esetleg ismételten) lemezre.

Ha a hiba a <COMMIT *T*> naplóbejegyzés lemezre írása előtt jelentkezik, akkor a *T* tranzakció befejezetlen tranzakciónak számít és semmissé kell tenni. Tehát, ha a hiba a 11-edik lépés előtt következik be az *X* értéke 50, és az *Y* 20-ra lesz visszaállítva. Bizonyos esetekben már nem lenne szükséges a lemezre írás, de a biztonság kedvéért a visszaállítást mindig végre kell hajtani. □

11.3.11. Semmisségi/helyrehozó naplózás ellenőrzőpont-képzéssel

A semmisségi/helyrehozó naplózás esetén az ellenőrzőpont-képzés a következőképpen történik:

1. Képezzük a <START CKPT (T_1, T_2, ..., T_m)> naplóbejegyzést az összes aktív tranzakcióval és írjuk ki a lemezre.
2. Írjuk lemezre az összes piszkos puffert, azokat, melyek módosított adatbáziselemet tartalmaznak. Ha a helyrehozó naplózás esetén csak a már befejezett tranzakciók által módosított puffereket mentettük lemezre, itt az összest lemezre írjuk.
3. Az <END CKPT> naplóbejegyzést a naplóba írjuk, majd lemezre mentjük.

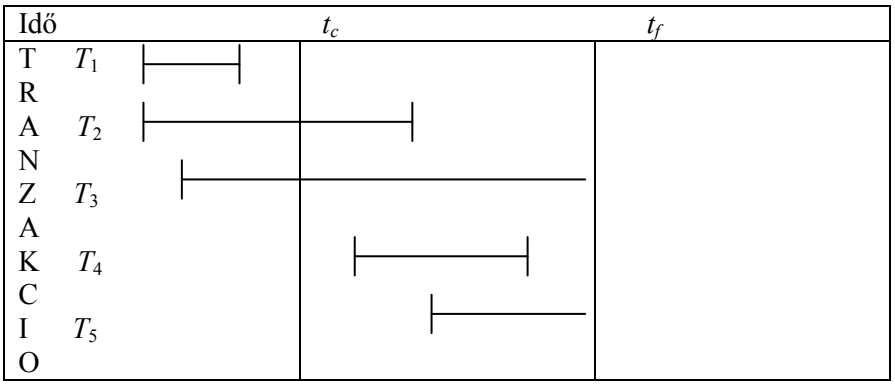
A 2-es pont megengedi a be nem fejezett tranzakciók által végzett módosítások lemezre írását is. Egy előírást viszont be kell tartani, mely a konkurencia esetén is nagyon fontos, hogy a tranzakciók közötti inkonzisztens kölcsönhatást megelőzzük.

– A tranzakció semmilyen értéket nem írhat (még a memóriapufferbe sem), amíg biztosak nem vagyunk abban, hogy nem abortál.

11.9. példa: Legyenek a következő naplóbejegyzések:

< START T_1 >
< T_1, X, 10, 15 >
< START T_2 >
< T_2, Y, 20, 25 >
< T_1, Z, 30, 35 >
< START T_3 >
< COMMIT T_1 >
< START CKPT (T_2, T_3) >
< T_3, P, 40, 45 >
< T_2, Q, 50, 55 >
< END CKPT >
< COMMIT T_2 >
< START T_4 >
< T_3, R, 60, 65 >
< T_4, U, 70, 75 >
< START T_5 >
< COMMIT T_4 >
< T_5, V, 80, 85 >

A 11.5. ábra a fenti tranzakciókat időben ábrázolja. A t_c időpontban a rendszer ellenőrzőpontot képezett, a t_f időpontban hiba állt be. A T_1 tranzakció sikeresen véget ért t_c előtt, T_2 a t_c előtt indul és t_f előtt sikeresen véget ért, T_3 t_c előtt indul, de nem fejeződik be a hiba fellépése előtt. T_4 t_c után indul és t_f előtt sikeresen véget ér, T_5 t_c után indul és nem fejeződik be t_f előtt.



11.5. ábra: A 11.9 példa tranzakcióinak időbeli ábrázolása

Tegyük fel, hogy a hiba ebben a pontban lép fel. Helyreállítás céljából visszafele haladva a naplóállományban azonosítjuk a befejezett tranzakciókat: $\{T_4, T_2\}$, illetve a be nem fejezetteket: $\{T_3, T_5\}$. A T_1 tranzakció a COMMIT pontját elérte az ellenőrzőpont előtt, az ellenőrzőpont-képzés közben a T_1 által végzett módosítások lemezre íródtak, tehát a T_1 tranzakcióval nincs semmi dolgunk. Mivel a COMMIT T_2 az ellenőrzőpont után jelenik meg, nem biztos, hogy a hiba fellépésekor lemezre íródtak a módosítások, ezért megkeressük a < START T_2>-t, ami a < START T_4> előtt található és a T_2 és T_4 által végzett összes módosítást lemezre írjuk ($Y \leftarrow 25$; Q $\leftarrow$ 55; $U \leftarrow$ 75). A $\{T_3, T_5\}$ nem érték el a COMMIT pontjukat, ezért semmissé kell tennünk az általuk végzett módosításokat, ami előfordulhat, hogy lemezre íródtak ($P \leftarrow$ 40, $R \leftarrow$60, $V \leftarrow$ 80).

Ha a hiba a COMMIT T_2 után lép fel, a T_4 és T_5 még el sem indult, az ellenőrzőpont-képzés viszont sikeres volt, ez azt jelenti, hogy a T_1 módosításai lemezre íródtak, a T_2 befejezett, a T_3 pedig nem befejezett tranzakció. Tehát a T_2-t helyrehozó naplózással kell helyreállítanunk, a T_3-at pedig semmissé kell tennünk.

Ha az ellenőrzőpont-képzés közben vagy még hamarabb (ha az <END CKPT> még nem került a naplóállományba), lép fel a hiba, egy előző ellenőrzőpontot kell keresnünk. A mi példánk esetében a naplóállomány elejére kell mennünk, azon tranzakciókat, melyek elérték a COMMIT

pontot a hiba fellépése előtt (a mi esetünkben esetleg a T_1 tranzakció), és helyrehozó naplózással kell helyreállítanunk a be nem fejezetteket, a $\{T_2, T_3\}$-at pedig semmissé kell tennünk. □

A semmisségi/helyrehozó naplózás alkalmazásakor a helyreállítás során nem részleteztük, hogy először a semmisségi (undo) vagy helyrehozó (redo) lépéseket tesszük meg. Előfordulhat, hogy ugyanazt az *X* adatbáziselemet két tranzakció is módosítja, egyik sikeres, másik sikertelen véget ér, a helyreállítást akármilyen sorrendben végeznénk a várt eredményt nem érnénk el. Az ABKR-ek a módosítások naplózása mellett a konkurencia problémáját is meg kell oldják. Ez a következő alfejezet témája.

11.4. Konkurenciavezérlés

Amint láttuk, a tranzakció részben a konkurencia egysége. Ha több felhasználó egyidőben fér hozzá ugyanazon adatbázishoz, még ha tranzakciók segítségével is, melyek ha külön-külön meg is őrzik az adatbázis konzisztens állapotát, ha egyidőben futnak inkonzisztenciához vezethetnek. A következőkben bemutatjuk azt a három konkurencia problémát, mikor a dolgok rosszul mehetnek.

– az elveszett módosítás (lost update) problémája
– tranzakció, mely nem véglegesített adatokat olvas (uncommitted dependency)
– helytelen analízis (inconsistent analysis)

Amikor a konkurenciát tanulmányozzuk, az olvasási és írási műveletek a központi memória puffereiben történnek, nem lemezen. Egy *X* adatbáziselemet, melyet valamely *T* tranzakció hozott be a pufferbe, ebben a pufferben nemcsak a *T* tudja olvasni vagy írni, hanem más tranzakciók is hozzáférhetnek az *X*-hez. A 11.2.-beli READ és WRITE műveletek először egy INPUT utasítást hívnak meg, hogy az adatbáziselemet a lemezről betöltsék, ha még nincs a pufferben, különben a READ és WRITE műveletek közvetlenül a pufferben férnek hozzá az elemhez, ezért csak a READ és WRITE műveletek sorrendje számít, amikor konkurenciával foglalkozunk. Használunk egy *f*, illetve *g* függvényt, melyek segítségével a változó értéke módosul.

11.4.1. Az elveszett módosítás problémája

Legyen két tranzakció, mely ugyanazt a *P* adatbáziselemet akarja módosítani. Mindkettő először olvassa, utána módosítja, de az első módosítását a második felülírja, lásd a 11.6. ábrát.

A probléma jobb megértése érdekében lássunk két konkrét példát, amikor az *A* tranzakció elvesztette a módosítást t_4 időpontban.

11.10. példa: Egy adott bankszámla értékét két különböző fióktól (bankautomatától) módosítanak. Tegyük fel, hogy a bankszámla értéke 5000 euró. Az egyik fióknál 500 eurót akarnak a bankszámlára utalni, a másik fióknál 1000 eurót akarnak a számláról kivenni. Semmi gond nem lép fel, ha a két műveletet az ABKR egymás után hajtja végre.

A tranzakció	*Idő*	*B tranzakció*
READ (P, *v*)	t_1	-
-	|	-
-	t_2	READ (P, *s*)
-	|	-
v: = *f* (*v*)	t_3	-
WRITE (P, *v*)	|	-
-	|	-
-	t_4	*s*: = *g* (*s*)
-	|	WRITE (P, *s*)

11.6. ábra: Az elveszett módosítás problémája

Legyen *X* az az adatbázis objektum, mely a számla értékét tárolja. Egy lehetséges párhuzamos futási terv, mely inkonzisztens állapotba hozza az adatbázist a 11.7. ábrán látható.

Ha egymás után hajtanánk végre a két tranzakciót (*A* után *B*-t, vagy *B* után *A*-t), akkor helyes eredményt kapnánk, a bankszámla értéke 4500 euró lenne. A fenti végrehajtás esetén a számla értéke 4000 euró lesz, tehát az 500 euró elveszett, amit hozzá kellett volna adni a számla értékéhez.□

11.11. példa: Két utas két különböző jegyirodából akarja megvenni egy adott járatra egy adott időpontban az utolsó jegyet. Legyen *Y* az adatbáziselem, mely tartalmazza az adott járatra az adott időpontban az üres helyek számát. Mindegyik utas épp egy jegyet akar vásárolni. A 11.8. ábrán látható párhuzamos futási terv esetén mind a két jegyirodában 1-et ad az üres helyek száma lekérdezés, tehát még van hely, 2-szer le is vonja, mégis az üres helyek száma az adatbázisban 0 lesz, mivel a t_3 időpontbeli módosítást (a pufferbeli értéket) a B tranzakció a saját *s* változójából felülírja. Ha egymás után hajtanánk végre a két tranzakciót (A után B-t, vagy B után A-t), mikor a második olvassa az üres helyek számát 0-t kap, s így már nem tudja eladni. □

A tranzakciók párhuzamos végrehajtását, akkor tekintjük helyesnek, ha ugyanazt a hatást érjük el, mintha egymás után hajtanánk végre a tranzakciókat.

A tranzakció	*Idő*	*B tranzakció*
READ (*X*, *v*)	t_1	-
(*v* = 5000)	|	-
-	t_2	READ (*X*, *s*)
-	|	(*s* = 5000)
v: = *v* + 500	t_3	-
(*v* = 5500)	|	-
WRITE (*X*, *v*)	t_4	*s*: = *s* – 1000
-	|	(*s* = 4000)
-	|	WRITE (*X*, *s*)

11.7. ábra: Futási terv a 11.10. példa esetén

A tranzakció	*Idő*	*B tranzakció*
READ (Y, *v*)	t_1	-
(*v* = 1)	|	-
-	t_2	READ (Y, *s*)
-	|	(*s* = 1)
v: = *v* - 1	t_3	-
(*v* = 0)	|	-
WRITE (Y, *v*)	|	-
-	t_4	*s*: = *s* – 1
-	|	(*s* = 0)
	|	WRITE (Y, *s*)

11.8. ábra: Utolsó jegy eladása kétszer

11.4.2. Tranzakció, mely nem véglegesített adatokat olvas („piszkos adat olvasása”)

A 11.9. ábrán két esetet láthatunk, mikor az *A* tranzakció egy nem véglegesített módosítást (uncommitted update) lát t_2 időben, mely t_3 időpontban vissza lesz pörgetve, így *A* hamis értékkel dolgozik, (első esetben csak olvassa, második esetben módosítja is) mely visszaváltozik arra, mely t_1 előtt volt. Az *A* tranzakció egy nem végérvényesített módosítástól függ, és az ő módosítása elvész, a ROLLBACK után *P* a t_1 időpont előtti értéket veszi fel. Ezt a problémát szokták „piszkos adat olvasásának” is emlegetni (dirty read).

11.12. példa: Vehetjük a bankszámlás példát, amikor egyik fióknál 500 eurót utalnak a számlára (A tranzakció), a másiknál 1000 eurót vesznek ki ugyanarról a számláról (B tranzakció). Ebben az esetben a B indul előbb és ki is vonja az 1000 eurót a számla értékéből, az A tranzakció a t_2 időpontban olvassa a számla értékét, amikor már 4000 euró. Az A tranzakció már a módosított

értékkel dolgozik, nem mint az elveszett módosítás esetén, a módosított értékhez adja hozzá az 500 eurót, be is írja a módosított (4500 euró, helyes) értéket a pufferbe. A probléma akkor jelentkezik, mikor a következő pillanatban a B tranzakció visszagörgeti a tranzakciót és a számla értéke az eredeti 5000 euró értéket kapja vissza. A B tranzakció esetén a felhasználó vagy a rendszer, kap üzenetet, hogy nem sikerült a tranzakciót végrehajtani és újra fogja indítani a B-t, de az A módosítása elvesz és üzenetet sem kap, hogy gondok voltak a végrehajtásával. Tehát ez az eset különbözik az elveszett módosítás problémájától. □

A tranzakció	*Idő*	*B tranzakció*
-	t_1	READ (P, *s*)
-	|	*s*: = *g* (*s*)
-	|	WRITE (P, *s*)
-	|	-
READ (P, *v*)	t_2	-
-	|	-
-	t_3	ROLLBACK

A tranzakció	*Idő*	*B tranzakció*
-	t_1	READ (P, *s*)
-	|	*s*: = *g* (*s*)
-	|	WRITE (P, *s*)
-	|	-
-	t_2	-
READ (P, *v*)	|	-
v: = *f* (*v*)	|	-
WRITE (P, *v*)	|	-
	t_3	ROLLBACK

11.9. ábra: „Piszkos adat olvasása”

11.4.3. Az inkonzisztens analízis problémája

Egyik tranzakció módosításokat, miközben a másik valamilyen analízist végez. Például az *A* tranzakció összegzi 3 különböző bankszámlán levő összegeket, *B* tranzakció áttesz 10 egységet a 3. számláról az első bankszámlára, amint a 11.10. ábrán látható. Jelöljük BankSz1-gyel, BankSz2-vel, illetve BankSz3-mal a három bankszámla értékét tartalmazó adatbáziselemet.

A tranzakció	*Idő*	*B tranzakció*
-	|	-
sum: = 0	t_1	-
READ (BankSz1, *v*)	|	-
sum: = *sum* + *v*	|	-
(*sum* = 40)	|	-
-	|	-
READ (BankSz2, *v*)	t_2	-
sum: = *sum* + *v*	|	-
(*sum* = 90)	|	-
-	|	-
-	t_3	READ (BankSz3, *s*)
-	|	*s*: = *s* –10
-	|	(s = 20)
-	|	WRITE (BankSz3, *s*)
-	|	-
-	t_4	READ (BankSz1, *s*)
-	|	*s*: = *s* +10
-	|	(s = 50)
-	|	WRITE (BankSz1, *s*)
-	|	-
-	t_5	COMMIT
-	|	-
READ (BankSz3, *v*)	t_6	-
sum: = *sum* + *v*	|	-
(*sum* = 110)	|	-

11.10. ábra: Az inkonzisztens analízis problémája

A két tranzakció módosításai előtt az értékek:

BankSz1 = 40
BankSz2 = 50
BankSz3 = 30

Az *A* tranzakció által kapott összeg helytelen. Ha ezt *A* beírja az adatbázisba, az adatbázis inkonzisztens lesz, úgy mondjuk, hogy *A* egy inkonzisztens állapotát látja az adatbázisnak és így egy inkonzisztens analízist végzett. Itt nem az a probléma, hogy *A* egy nem véglegesített változtatástól függ, mivel *B* végérvényesítette az összes változtatását, nem volt visszagörgetés. Ha a két tranzakciót egymás után végeztük volna el, akár *A*-t előbb, akár *B*-t, az *A* tranzakció helyes eredményt kapott volna: 120-at.

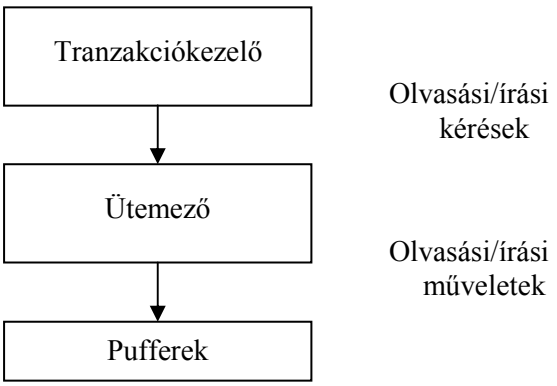
11.5. Soros és sorba rendezhető ütemezések

11.5.1. Ütemező

A konkurencia problémák megoldására az ABKR-ek az ütemezőt (scheduler) használják. A tranzakciók olvasási/írási kérelmei először az ütemezőhöz kerülnek. Bizonyos esetekben az ütemező késleltetheti az olvasási/írási műveleteket, a konkurencia-vezérlési technikától függően, utólag pedig átadja a pufferkezelőnek a műveleteket.

Az ütemezés (schedule) több tranzakció által végrehajtott műveletek időrendben vett sorozata. Az olvasási/írási műveletek a központi memória puffereiben történnek, nem lemezen. Ha egy *X* adatbáziselemet valamely *T* tranzakció behozott a pufferbe, nemcsak a *T* tranzakció tud hozzáférni, hanem más tranzakció is.

A konkurencia tanulmányozása esetén csak a READ és WRITE műveletek sorrendje a fontos, ezek meghívják az INPUT műveletet, ha a megfelelő adatbáziselem nincs a pufferben, de ha az adatbáziselem már a pufferben van, a READ és WRITE műveletek közvetlenül a pufferben férnek hozzá.



11.11. ábra: Az ütemező és kapcsolatai

11.5.2. Soros ütemezés

Egy ütemezés *soros* (serial schedule), ha bármely két T_1 és T_2 tranzakcióra, ha T_1-nek van olyan művelete, mely megelőzi a T_2 valamelyik műveletét, akkor a T_1 összes művelete megelőzi a T_2 valamennyi műveletét.

11.13. példa: Legyen T_1 a következő tranzakció:

BEGIN TRANSACTION
$X := X - 10$;
$Y := Y + 10$;
COMMIT TRANSACTION

T_2 pedig:

```
BEGIN TRANSACTION
X:= X * 2;
Y:= Y * 2;
COMMIT TRANSACTION
```

A 11.12. ábrán a T_1 tranzakció megelőzi T_2-t, a 11.13. ábrán pedig T_2 tranzakció előzi meg T_1-et.

T_1	T_2	v	$Mem-X$	$Mem-Y$
READ(X, v)		50	50	
v: = v – 10		40	50	
WRITE(X, v)		40	40	
READ(Y, v)		20	40	20
v:= v + 10		30	40	20
WRITE(Y, v)		30	40	30
	READ(X, v)	40	40	30
	v: = v * 2	80	40	30
	WRITE(X, v)	80	80	30
	READ(Y, v)	30	80	30
	v:= v * 2	60	80	30
	WRITE (Y, v)	60	80	60

11.12. ábra: T_1, T_2 tranzakciók soros ütemezése, T_1 tranzakció megelőzi T_2-t

T_1	T_2	v	$Mem-X$	$Mem-Y$
	READ(X, v)	50	50	
	v: = v * 2	100	50	
	WRITE(X, v)	100	100	
	READ(Y, v)	20	100	20
	v:= v * 2	40	100	20
	WRITE (Y, v)	40	100	40
READ(X, v)		100	100	40
v: = v – 10		90	100	40
WRITE(X, v)		90	90	40
READ(Y, v)		40	90	40
v:= v + 10		50	90	40
WRITE (Y, v)		50	90	50

11.13. ábra: T_1, T_2 tranzakciók soros ütemezése, T_2 tranzakció megelőzi T_1-et

A soros ütemezésben a műveletek sorrendje csak a tranzakciók sorrendjétől függ, ezért az első ütemezést jelölhetjük (T_1, T_2)-vel, a másodikat pedig (T_2, T_1)-gyel jelölhetjük. A kapott végeredmény a két ütemezés esetén különböző. Általában nem várjuk el, hogy az adatbázis végső állapota a tranzakciók sorrendjétől független legyen. A példa esetén az X és Y adatbáziselem összege 140 mind a két ütemezés esetén, mivel az egyik tranzakció a 10-et, amit kivon az egyik elemből, hozzáadja a másikhoz, illetve a 2-el való szorzás mind a két elem esetén megtörténik. □

11.5.3. Sorba rendezhető ütemezések

Egy tranzakció megőrzi az adatbázis helyességét. Egymás után − tehát soros ütemezéssel − végrehajtott tranzakciók is megőrzik az adatbázis helyességét. A soros ütemezésen kívül, a sorba rendezhető (serializable schedule) ütemezés biztosítja még az adatbázis konzisztenciájának a megmaradását.

Egy ütemezés *sorba rendezhető*, ha ugyanolyan hatással van az adatbázis állapotára, mint valamelyik soros ütemezés.

T_1	T_2	v	*Mem−X*	*Mem−Y*
READ(X, v)		50	50	
v: = v – 10		40	50	
WRITE(X, v)		40	40	
	READ(X, v)	40	40	
	v: = v * 2	80	40	
	WRITE(X, v)	80	80	
READ(Y, v)		20	80	20
v:= v + 10		30	80	20
WRITE (Y, v)		30	80	30
	READ(Y, v)	30	80	30
	v:= v * 2	60	80	30
	WRITE (Y, v)	60	80	60

11.14. ábra: T_1, T_2 soros ütemezésnek egy sorba rendezhető ütemezése.

11.14. példa: A 11.14 ábra a T_1, T_2 soros ütemezésnek egy sorba rendezhető ütemezését ábrázolja. Amint látjuk, a kapott eredmény megegyezik a T_1, T_2 soros ütemezés esetén kapott eredménnyel.

11.15. példa: A 11.15. ábrán a T_1, T_2 soros ütemezésnek egy nem sorba rendezhető ütemezését láthatjuk, ahol az eredmény nem egyezik meg a T_1, T_2 soros ütemezés esetén kapott eredménnyel. Ez az ütemezés nem kívánt a konkurenciavézérlés esetén.

11.5.4. Konfliktusok

Konkfliktus: olyan egymást követő műveletpár az ütemezésben, amelynek ha a sorrendjét felcseréljük, akkor legalább az egyik tranzakció viselkedése megváltozhat.
Nem cserélhetjük fel a műveletek sorrendjét a következő esetekben:

– Ugyanannak a tranzakciónak két művelete konfliktus, pl.: $r_i(X)$; $w_i(Y)$. Egy tranzación belül a műveletek sorrendje rögzített, az ABKR ezt a sorrendet nem rendezheti át;

T_1	T_2	v	*Mem−X*	*Mem−Y*
READ(X, v)		50	50	
v: = v – 10		40	50	
WRITE(X, v)		40	40	
	READ(X, v)	40	40	
	v: = v * 2	80	40	
	WRITE(X, v)	80	80	
	READ(Y, v)	20	80	20
	v:= v * 2	40	80	20
	WRITE (Y, v)	40	80	40
READ(Y, v)		40	80	40
v:= v + 10		50	80	40
WRITE (Y, v)		50	80	50

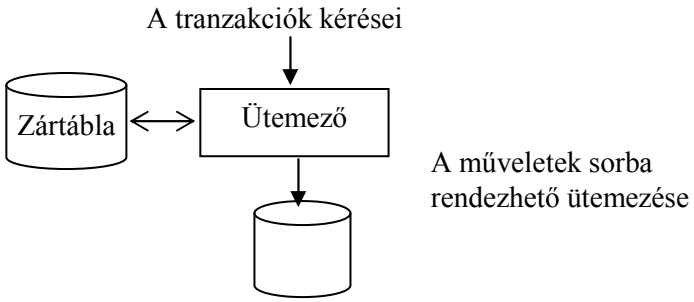
11.15. ábra: A T_1, T_2 soros ütemezésnek egy nem sorba rendezhető ütemezése

– Különböző tranzaciók ugyanarra az adatbáziselemre vonatkozó írása konfliktus. Pl. $w_i(X)$; $w_j(X)$. Ebben a sorrendben X értéke az marad, melyet T_j ír, fordított sorrendben pedig az marad, melyet T_i ír. Ezek az értékek pedig nagy valószínüséggel különbözőek.
– Különböző tranzakcióknak ugyanabból az adatbáziselemből való olvasása és írása konfliktus. Tehát $r_i(X)$; $w_j(X)$ konfliktus, mivel ha felcseréljük a sorrendet, akkor a T_i által olvasott X-beli érték az lesz, melyet a T_j ír, és az nagy valószínűséggel nem egyezik meg az X korábbi értékével. Hasonlóan a $w_i(X)$; $r_j(X)$ is konfliktus.

11.5.5. A sorbarendezhetőség biztosítása zárakkal

Ha egy tranzakcióhalmaz megszorítások nélkül hajtaná végre a műveleteit sok esetben nem kívánt eredményt érnénk el. Az ütemező feladata, hogy megakadályozza az olyan műveleti sorrendeket, amelyek nem sorba rendezhető ütemezésekhez vezetnek. Ezt a feladatot az ütemező több módszerrel is elérheti. A leggyakrabban használt módszer, hogy a tranzakciók zárolják azokat az adatbáziselemeket, amelyekhez hozzáférnek, azért, hogy megakadályozzák azt, hogy ezekhez az elemekhez ugyanakkor más tranzakciók is hozzáférjenek.

A 11.16. ábrán olyan ütemezőt látunk, mely zártábla segítségével végzi a feladatát.



11.16. ábra: Az ütemező és zártábla együttműködése

11.5.6. Osztott és kizárólagos zárak

Egy adatbáziselemet több tranzakció is olvashat egyidőben, de ebben az esetben sem kerülhetjük el a zárolást, hogy közben más tranzakció ne tudja írni ugyanazt az adatbáziselemet. Ezért egyelőre kétféle zárat vezetünk be: osztott zárat (shared locks) és kizárólagos zárat (exclusive locks).

Használjuk a következő jelöléseket:
$r_i(X)$ − a T_i tranzakció olvassa az X adatbáziselemet (r − read)
$w_i(X)$ − a T_i tranzakció írja az X adatbáziselemet (w − write)
$sl_i(X)$ − a T_i tranzakció osztott zárat kér az X-re
$xl_i(X)$ – a T_i tranzakció kizárólagos zárat kér az X-re
$u_i(X)$ − a T_i tranzakció feloldja X adatbáziselemen tartott zárát.

Ha egy tranzakció osztott zárat helyezett egy adatbáziselemre, más tranzakció is használhatja, de csak olvasás céljából, vagyis osztott zárat kérhet rá. Ha egy adatbáziselemet egy tranzakció írni akar, akkor kizárólágos zárat kell kérnie rá, amit csak egy tranzakció kaphat meg egyidőben, vagyis senki más nem használhatja, nem kap rá sem osztott, sem kizárolagos zárat. Ha egy tranzakciónak már van osztott zára egy adatbáziselemen, melyet módosítani akar, felminősítheti a zárat kizárólagossá (upgrade lock to exclusive). A zárakat fel kell szabadítanunk, ha befejeztük a munkát az adott adatbáziselemmel.

Amint láttuk, ha egy ütemezés sorba rendezhető, az adatbázis konzisztenciája megmarad.

Kétfázisú lezárási tétel: Ha minden tranzakció betartja a kétfázisú lezárási protokollt, az összes lehetséges ütemezés sorba rendezhető.

Kétfázisú lezárási protokoll: A zárolásoknak meg kell előzniük a zárak feloldását:
– mielőtt egy tranzakció egy objektummal dolgozna, sikeresen le kell azt zárja;
– miután a tranzakció felszabadított egy zárat nem kérhet más zárat.

A tranzakciók kérhetnek különböző zárakat, ha az ütemező nem tudja megadni a kért zárat, a tranzakciók egy *első-beérkezett-első-kiszolgálása* (first-come-first-served) várási listába kerülnek, míg a szükséges adat felszabadul.

Azért, hogy a tranzakciók konzisztenciája megmaradjon, minden T_i tranzakció esetén bevezetjük a következő követelményeket:

- az $r_i(X)$ olvasási műveletet meg kell előzze egy $sl_i(X)$ vagy $xl_i(X)$ úgy, hogy közben nincs $u_i(X)$
- a $w_i(X)$ olvasási műveletet meg kell előzze egy $xl_i(X)$ úgy, hogy közben nincs $u_i(X)$
- ha $xl_i(X)$ szerepel egy ütemezésben, akkor ezután nem következhet $xl_j(X)$ vagy $sl_j(X)$ valamely i-től különböző j-re, anélkül, hogy közben $u_i(X)$ ne szerepelne
- ha $sl_i(X)$ szerepel egy ütemezésben, akkor ezután nem következhet $xl_j(X)$ $j \neq i$ -re, anélkül, hogy közben $u_i(X)$ ne szerepelne.

Lezárható az egész adatbázis. Annak, akinek sikerült lezárnia a programozás szempontjából könnyű dolga van, de senki más nem férhet hozzá, lezárható egy rekord, általában ez a legkézenfekvőbb megoldás vagy egy rekordcsoport (egy lemezblokkban lévő rekordok, vagy fa szerkezetű indexnek megfelelő rekordcsoport).

Kompatibilitási mátrix

Ha az ABKR több zármódot használ, az ütemezőnek egy táblázatra van szüksége, ahhoz, hogy tudja, egy zárolási kérést egy adott adatbázis-elemen mikor engedélyezhet, ha azon már más zár van.

A *kompatibilitási mátrix* minden egyes zármódhoz egy-egy sort, illetve egy-egy oszlopot tartalmaz. A sorok egy másik tranzakció által az X elemre már érvényes záraknak, az oszlopok pedig az X-re kért zármódnak felelnek meg. Az ütemező C típusú zárat akkor engedélyezhet, ha a táblázat minden olyan S sorára, amelyre más tranzakció S módban már zárolta az X-et, a C oszlopban „Igen” szerepel (11.17. ábra).

		Kért zár	
		S	X
Érvényes zár	S	Igen	Nem
Ebben a módban	X	Nem	Nem

11.17. ábra: Osztott (S) és kizárólagos (X) zárak kompatibilitási mátrixa

A továbbiakban komplexebb kompatibilitási mátrixokat is fogunk látni. Lássuk, megoldottuk-e a három konkurrencia problémát a zárak segítségével?

Elveszett módosítás problémája

A 11.18 ábrán az elveszett módosítás problémájának osztott, illetve kizárólagos zárakkal való megoldási kísérletét láthatjuk. Betart minden kért feltételt, a kétfázisú lezárást is, ezért nincs $ul_1(P)$, mert utólag még kér zárakat, de amint látjuk, a probléma nem oldódott meg, holtponthoz vezetett.

Tranzakció 1	***Idő***	***Tranzakció 2***
$sl_1(P)$; $r_1(P)$	t_1	-
-	|	-
-	t_2	$sl_2(P)$; $r_2(P)$
-	|	-
$xl_1(P)$ **vár**	t_3	-
-	|	-
-	t_4	$xl_2(P)$ **vár**
holtpont	|	holtpont

11.18. ábra: Elveszett módosítás problémájának osztott/kizárólagos zárakkal való megoldási kísérlete

Tranzakció, mely nem véglegesített adatokat olvas

A 11.19. ábrán a „piszkos” adat olvasása problémát probáljuk megoldani osztott, illetve kizárólagos zárakkal. A probléma nem oldódott meg, az 1-es tranzakció „piszkos” adatot látott, a 2-es tranzakció által végzett módosításokat a rendszer utólag semmissé tette.

Tranzakció 1	***Idő***	***Tranzakció 2***
-	t_1	$sl_2(P)$; $r_2(P)$; $xl_2(P)$
-	|	$w_2(P)$; $u_2(P)$
-	|	-
$sl_1(P)$; $r_1(P)$; $xl_1(P)$	t_2	-
$w_1(P)$; $u_1(P)$	|	-
	|	-
	|	-
	t_3	ROLLBACK

11.19. ábra: Piszkos adat olvasása probléma megoldási kísérlete osztott/kizárólagos zárakkal

Az inkonzisztens analízis problémája

A 11.20. ábrán az inkonzisztens analízis problémát láthatjuk a bevezetett osztott és kizárólagos zárak felhasználásával. Amint látjuk ezt a problémát sem sikerült megoldjuk, ez is holtponthoz vezetett.

Tranzakció 1	***Idő***	***Tranzakció 2***
-	|	-
sl_1(BankSz1); r_1(BankSz1);	t_1	-
-	|	-
sl_1(BankSz2); r_1(BankSz2);	t_2	-
-	|	-
-	t_3	xl_2(BankSz3); r_2(BankSz3);w_2(BankSz3)
-	|	-
-	t_4	xl_2(BankSz1); **vár**
-	|	-
sl_1 (BankSz3); **vár**	t_5	-
holtpont	|	holtpont

11.20. ábra: Megoldási kísérlet az inkonzisztens analízis problémájának a megoldására

Holtpont, olyan állapot, mikor két vagy több tranzakció várási állapotban van, mindegyik a másik által lezárt objektumra vár. A System R ABKR-rel végzett kísérletek azt mutatták, hogy nem jelenik meg kettőnél több tranzakció holtpontban.

A *holtpont megoldására* több módszert is szoktak alkalmazni, egy pár ezek közül:

– módosítási zárak alkalmazása
– várakozási gráf (Wait-For-Graph)
– időkorlát mechanizmus.

11.5.7. Módosítási zárak

Az $ul_i(X)$ módosítási zár (update lock) a T_i tranzakciónak csak X adatbáziselem olvasására ad jogot, az X írására nem, viszont csak a módosítási zárat lehet később felminősíteni, az olvasásit nem. Ha egy tranzakciónak szándékában áll módosítani az X adatbáziselemet, akkor módosítási zárat kér rá. Módosítási zárat engedélyez a rendszer az X-en akkor is, ha már van osztott zár X-en. Ha már van egy módosítási zár X-en, akkor viszont a rendszer minden más zárat elutasít, legyen az osztott, módosítási vagy kizárólagos. Ha nem utasítaná el az újabb zárolásokat, akkor előfordulhat, hogy a módosításinak soha sem lenne lehetősége kizárólagossá való felminősítésre, mivel mindig valamilyen más zár lenne az X-en.

A kompatibilitási mátrix nem szimmetrikus (lásd 11.21. ábrát).

	S	*X*	*U*
S	Igen	Nem	Igen
X	Nem	Nem	Nem
U	Nem	Nem	Nem

11.21. ábra:. Osztott (*S*), kizárólagos (*X*) és módosítási (*U*) zárak kompatibilitási mátrixa

Lássuk a három konkurencia problémát:

Elveszett módosítás problémája

A módosítási zár megoldást ad az elveszett módosítás problémájára, amint a 11.22. ábra mutatja, a 2-es tranzakció késleltetve van, míg az 1-es felszabadítja a zárat.

Tranzakció 1	***Idő***	***Tranzakció 2***
$ul_1(P)$; $r_1(P)$	t_1	-
-	|	-
-	t_2	$ul_2(P)$ **vár**
-	|	-
$xl_1(P)$; $w_1(P)$; $u_1(P)$	t_3	-
-	|	-
-	t_4	$ul_2(P)$; $r_2(P)$
	|	$xl_2(P)$; $w_2(P)$; $u_2(P)$

11.22. ábra: Az elveszett módosítás problémájának a megoldása módosítási zárakkal

„Tranzakció, mely nem véglegesített adatokat olvas” probléma nem oldódik meg. A gond, hogy a zárat túl hamar felszabadítja; később visszatérünk erre a problémára.

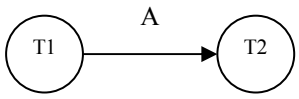
Az inkonzisztens analízis problémája sem oldódik meg, mivel a gond nem ott van, hogy mind a két tranzakció ugyanazt az adatbáziselemet módosítja.

11.5.8. Várakozási gráf

A gráf segítségével nyilvántartjuk, hogy melyik tranzakció melyikre vár. A gráf csúcsait a tranzakciók alkotják. Legyen T_1 tranzakció az egyik csúcsban és T_2 tranzakció egy másikban. T_1 és T_2 csúcs között élet rajzolunk, ha:

– T_1 lezárva tartja az A adatbáziselemet,
– T_2 kéri az A adatbáziselemet, hogy zárolhassa azt.

Az él irányítása a T_1-től a T_2 felé lesz és A az él címkéje. T_2 akkor kapja meg a számára megfelelő zárat, ha T_1 lemond róla.



Ha a gráf ciklust tartalmaz, azt jelenti, hogy van holtpont. Ebben az esetben valamelyik tranzakciót, mely a körben szerepel, meg kell szakítani és visszagörgetni. A rendszer ki kell derítse, hogy holtpont probléma áll fenn. A tranzakciót megszakítani azt jelenti, hogy valamelyik tranzakciót kiválasztja, mely a holtpontban szerepel – ez az áldozat (victim) –és visszagörgetni, így felszabadul a lock, amit ez a tranzakció adott ki, és lehetősége nyílik a többi tranzakciónak, hogy folytatódjon.

11.5.9. Időkorlát mechanizmus

A gyakorlatban nincs minden rendszernek holtpont felfedező mechanizmusa, csak egy *időtúllépés* (timeout) mechanizmusa, mely feltételezi, ha egy adott idő intervallumban a tranzakció nem dolgozik semmit, azt jelenti, hogy holtponton van.

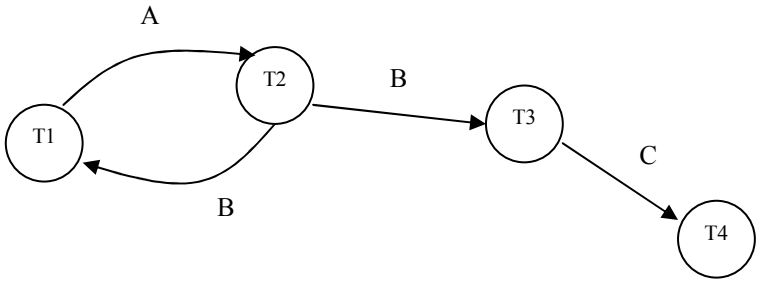
Az áldozatban nem lesz semmi hiba, egy pár rendszer újraindítja a tranzakciót, feltételezve, hogy változtak a feltételek, melyek a holtpontot okozták. Más rendszer olyan hiba kódot ad vissza, hogy „deadlock victim" az alkalmazásnak, és a programozó kell megoldja a feladatot, a tranzakciót indítsa újra. Mindkét esetben ajánlatos tudatni a felhasználóval a történteket.

11.16. példa: Négy tranzakció dolgozik egyidőben, *A*, *B* és *C* adatbáziselemekkel. Mindegyik tranzakció több adatbáziselemmel is szándékszik dolgozni, ezért nincsenek zár felszabadítások, hogy a kétfázisú lezárási protokoll legyen betartva. A futási tervet a 11.23 ábra szemlélteti.

Idő	T_1	T_2	T_3	T_4
t_1	$xl_1(A)$; $w_1(A)$;			
t_2		$xl_2(B)$; $w_2(B)$;		
t_3			$xl_3(C)$; $w_3(C)$;	
t_4				$xl_4(C)$; **vár**
t_5	$xl_1(B)$; **vár**			
t_6		$xl_2(A)$; **vár**		
t_7			$xl_3(B)$; **vár**	

11.23. ábra: A 11.16 példa futási terve

A 11.24 ábrán a várási gráfot láthatjuk, mely a fenti ütemezés esetén a tranzakciók egymástól való függését mutatja.



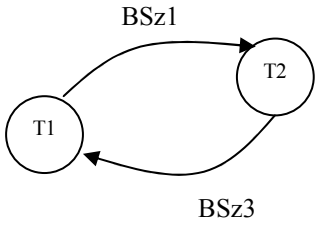
11.24. ábra: Várási gráf a 11.16 példa esetén

Tegyük fel, hogy a rendszer, miután felfedezte, hogy T_1 és T_2 vannak a körben, a T_1-et áldozatnak szemeli ki és visszagörgeti. A 11.25. ábrán az ütemezés alakulását szemléltetjük. A T_1 lesz az áldozat, a többi tranzakció be tudja fejezni a munkáját.

Idő	T_1	T_2	T_3	T_4
t_1	$xl_1(A)$; $w_1(A)$;			
t_2		$xl_2(B)$; $w_2(B)$;		
t_3			$xl_3(C)$; $w_3(C)$;	
t_4				$xl_4(C)$; **vár**
t_5	$xl_1(B)$; **vár**			
t_6		$xl_2(A)$; **vár**		
t_7			$xl_3(B)$; **vár**	
t_8	ABORT T_1 $u_1(A)$			
t_9		$xl_2(A)$; $w_2(A)$; $u_2(B)$; $u_2(A)$		
t_{10}			$xl_3(B)$; $w_3(B)$; $u_3(C)$; $u_3(B)$	
t_{11}				$xl_4(C)$; $w_4(C)$; $u_4(C)$;

11.25. ábra: A holtpont megoldása a 11.16. példa esetében

Lássuk, hogy az inkonzisztens analízis problémáját meg tudjuk-e oldani a várási gráf segítségével. A megfelelő várási gráf a 11.26. ábrán található.



11.26. ábra: Az inkonzisztens analízis problémájának megfelelő várási gráf

A gráf kört tartalmaz, az 1-es tranzakciót áldozatnak nevezzük ki és visszagörgetjük, a 11.27. ábrán a kialakult helyzetet láthajuk. Tehát a 2-es tranzakciónak sikerült elérni a COMMIT pontját, az áldozat tranzakció viszont még mindig probléma, újra kell indítani, mikor ismét problémák lehetnek.

Tranzakció 1	*Idő*	*Tranzakció 2*
-	|	-
sl_1(BankSz1); r_1(BankSz1);	t_1	-
-	|	-
sl_1(BankSz2); r_1(BankSz2);	t_2	-
-	|	-
-	t_3	xl_2(BankSz3); r_2(BankSz3);w_2(BankSz3)
-	|	-
-	t_4	xl_2(BankSz1); **vár**
-	|	-
sl_1 (BankSz3); **vár**	t_5	-
holtpont	|	holtpont
ABORT T_1	t_6	
u_1(BankSz1); u_1(BankSz2);	|	
	t_7	xl_2(BankSz1); r_2(BankSz1);w_2(BankSz1)
	|	u_2(BankSz1); u_2(BankSz3);
	|	COMMIT

11.27. ábra: Az inkonzisztens analízis problémájának a megoldása

11.5.10. Szigorú két-fázisú lezárási protokoll

Még mindig megoldatlan a „Tranzakció, mely nem véglegesített adatokat olvas” probléma. A *Szigorú két-fázisú lezárási protokoll* adja meg a megoldást:

1. Egy tranzakció nem írhat az adatbázisba, míg nem éri el a COMMIT pontját.
2. Egy tranzakció nem szabadíthatja fel a zárait, míg be nem fejezi az írást az adatbázisba, tehát a zárak felszabadítása is COMMIT pont után történik.

Tehát egy tranzakció, mely betartja a szigorú két-fázisú lezárási protokollt a következő sorrendben hajtja végre a munkát: minden szükséges zárat a tranzakció elején kér, mikor elérkezik a COMMIT ponthoz, beírja az adatbázisba a módosított adatbáziselemeket, majd felszabadítja a zárakat.

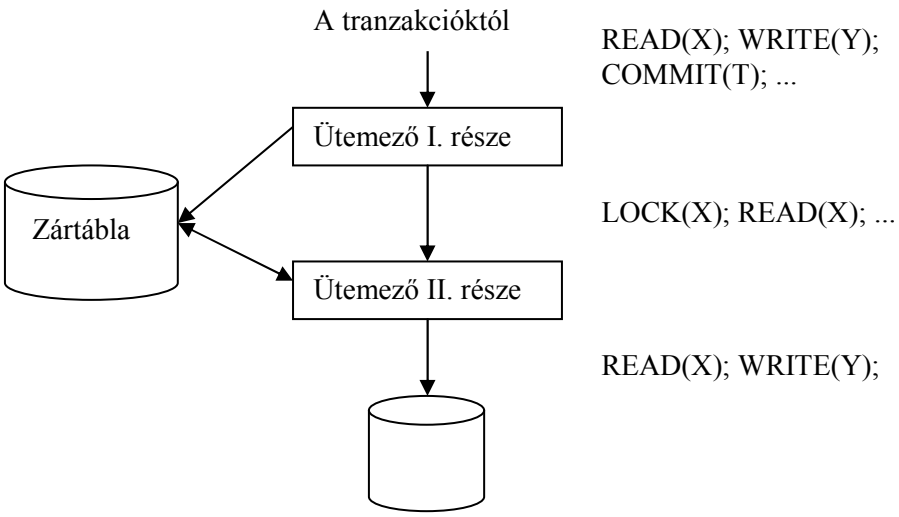
Lássuk a még megoldatlan feladatunk, a piszkos adat olvasása hogyan oldódik meg (11.28. ábra). Amint a futási terv szemlélteti, a 2-es tranzakció nem írhat az adatbázisba a ROLLBACK előtt, mert nem érte el a COMMIT pontját, így az 1-es tranzakció vár, mert a *P* adatbáziselemet még a 2-es tranzakció zárva tartja. Tehát nem olvas addig, míg a 2-es nem ér el a ROLLBACK-hez, amikor is a zárat felszabadítja, a *P* értéke nem változott meg, mivel az írás nem történt meg, az 1-es nem látott „piszkos” adatot.

Tranzakció 1	*Idő*	*Tranzakció 2*
-	t_1	$xl_2(P)$; $r_2(P,s)$;
-	|	$s := g(s)$;
$xl_1(P)$; **vár**	t_2	-
-	|	-
-	t_3	ROLLBACK;
-	|	$ul_2(P)$;
$r_1(P)$; $w_1(P)$;	t_4	
$u_1(P)$	|	

11.28. ábra: Piszkos adat olvasása probléma megoldása a szigorú két-fázisú lezárási protokoll segítségével

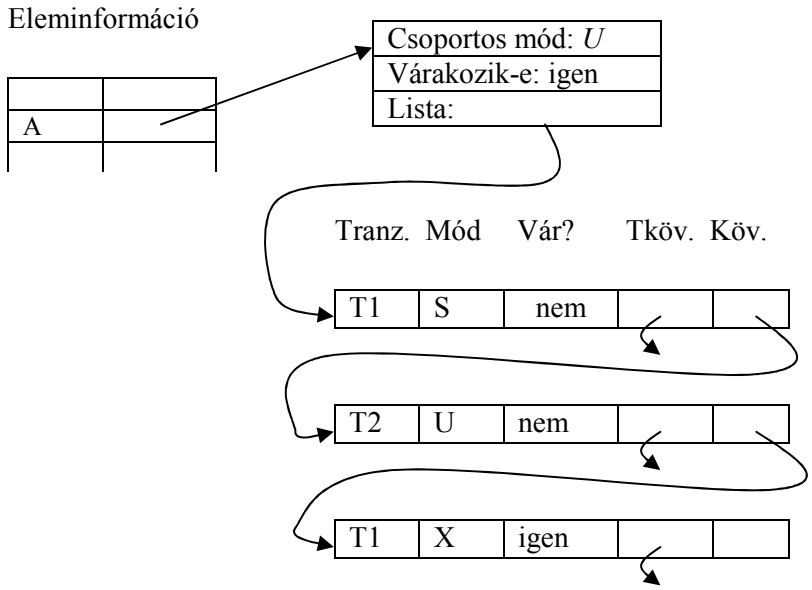
11.5.11. Az ütemező felépítése

Nem maguk a tranzakciók kérnek zárakat, az ütemező feladata, hogy zárolási műveleteket szúrjon be az adatokhoz hozzáférő olvasási és írási műveletek sorába. A zárakat is az ütemező oldja fel, amikor a tranzakciókezelő a tranzakció véglegesítésére vagy abortálására készül.



11.29. ábra: Ütemező és kapcsolata a zártáblához

- Az I. rész kapja a tranzakcióktól az adatbázis műveleteket és beszúrja a megfelelő zárolási műveleteket.
- A II. rész kapja a zárolási és adatbázis műveleteket és végrehajtja őket.
  - Ha a *T* tranzakció által kért adatbázis műveletet vagy a zárolást nem tudja végrehajtani, mivel a zár nem engedélyezhető, akkor késlelteti a műveletet és hozzáadja azoknak a műveleteknek a listájához, amiket a *T* tranzakciónak még végre kell hajtania.
  - Ha a *T* nem késleltetett (a kért zár már engedélyezve van), akkor
    - Ha a művelet adatbázis-hozzáférés, akkor azt átadja az adatbázisnak végrehajtásra.
    - Ha zárolási művelet érkezik, az ütemező megvizsgálja a zártáblát, hogy vajon a zár engedélyezhető-e:
      - ha igen, akkor a zártáblába beírja az aktuális zárat is,
      - ha nem, akkor a zárolási kérést beírja a zártáblába és az ütemező II. része ezután a *T* tranzakció további műveleteit mindaddig késlelteti, amíg nem tudja engedélyezni a zárat.
- Amikor a *T* tranzakciót véglegesítjük vagy abortáljuk, akkor a tranzakciókezelő értesíti az I. részt, hogy oldja fel az összes zárat. Ha a tranzakciók valamelyike ezen zárak valamelyikére várakozik, akkor az I. rész értesíti a II. részt.
- Amikor a II. rész értesül, hogy valamelyik *X* adatbáziselem felszabadul a zártól, amit más tranzakció tartott rajta, akkor eldönti, hogy melyik a következő tranzakció, amely megkapja a zárat az *X*-re. A tranzakció, mely megkapta a zárat, a késleltetett műveletei közül annyit hajt végre, amennyit csak tud, addig, amíg befejeződik vagy egy másik zárolási kéréshez érezik, amelyet nem lehet engedélyezni és a rendszer ismét késlelteti.



11.30. ábra: Zártábla-bejegyzések szerkezete

A zártábla egy olyan reláció, mely összekapcsolja az adatbáziselemet az elemre vonatkozó zárolási információval. Azok az elemek, amelyek nincsenek zárolva, a táblában nem fordulnak elő. A zártábla egy bejegyzésének egy adott *A* adatbáziselemhez való szerkezete a következő:

1. A *csoportos mód* (group mode) a legszigorúbb feltételek összefoglalása, amikor egy tranzakció szembesül, amikor egy új zárolást kér az *A*-n. Az osztott-kizárólagos-módosítási (S*XU*) zárolási sémához a szabály egy csoportos módban:
   a. S azt jelenti, hogy csak osztott zárak vannak.
   b. *U* azt jelenti, hogy egy módosítási zár van és lehet még egy vagy több osztott zár.
   c. *X* azt jelenti, hogy csak egy kizárólagos zár van, semmilyen más.

Más típusú zárolási sémához kell értemezni egy csoportos mód összegzésének megfelelő rendszert.

2. A *várakozási bit* (waiting bit) azt adja meg, hogy van-e legalább egy tranzakció, amely az *A* zárolására vár.
3. Mutató egy lista elejére, mely azon tranzakciókat tartalmazza, melyek jelenleg zárolják *A*-t, vagy *A* zárolására várakoznak. Egy listabejegyzés szerkezete:
   a. A zárolást fenntartó vagy a zárolásra váró tranzakció neve.
   b. A zár módja, melyet a tranzakció tart vagy kér.
   c. A tranzakció vár-e a zárra?
   d. Mutató, mely segítségével az adott tranzakciót (melynek neve az a. pontban adott) nyilvántartjuk a bejegyzéseit összekapcsoló listában. Tehát létezik minden tranzakció esetén egy lista, mely a tranzakció zárolási bejegyzéseit tartalmazza. Ez akkor használható, amikor a tranzakciót véglegesítjük vagy abortáljuk, így a rendszer megtalálja az összes zárat, amit fel kell oldania.
   e. Mutató a következő elemre ebben a listában, vagyis azon tranzakciók listájában, melyek az *A* adatbáziselemen zárat tartanak vagy zárra várnak.

11.5.12. Zárolási kérések kezelése

Legyen *T* a tranzakció, mely zárat kér az *A* adatbáziselemre. A kérés kezelésére a következő lépéseket követjük:

1. Keressük az *A* adatbáziselemet a zártáblában.
2. Ha nincs *A*-ra bejegyzés, akkor zárak sincsenek az *A*-n, létrehozzuk a bejegyzést és engedélyezzük a kérést.
3. Ha a zártáblában létezik bejegyzés az *A*-ra:

a. Megvizsgáljuk a csoportos módot, amely lehet:
   i. Módosítási (*U*), tehát semmilyen más zárat nem lehet engedélyezni az *A*-n (kivéve, ha maga *T* tartja fenn a zárat). A kérést elutasítjuk és egy bejegyzést helyezünk el a listában (Köv mutatót használva), amely szerint *T* zárat kért és a Vár? mezőt 'igen' értékkel töltjük ki.
   ii. Kizárólagos (*X*), ugyanúgy járunk el, mint *i*. esetben.
   iii. Osztott (*S*), lehet adni *S* vagy *U* zárat. Ha a kért zár:
      – *S*, akkor a *T* megkapja a zárat, új bejegyzés kerül az *A*-ból induló listába (Köv mutatót használva), a Vár? mező 'nem'.
      – *U*, akkor a *T* megkapja a zárat, a csoportos módot kicseréljük *U*-ra, új bejegyzés kerül az *A*-ból induló listába (Köv mutatót használva), a Vár? mező 'nem'.
      – *X*, a zárat nem lehet engedélyezni az *A*-n (kivéve, ha maga *T* tartja fenn a zárat). A kérést elutasítjuk és egy bejegyzést helyezünk el a listában (Köv mutatót használva), amely szerint *T* zárat kért és a Vár? mezőt 'igen' értékkel töltjük ki.

b. A *T* tranzakciót elhelyezzük a másik listában is, mely a *T* összes zárolását tárolja, a Tköv mutatót felhasználva.

Amint látjuk, a két listát és a csoportos módot használva az ütemező anélkül tudja kezelni a zárási kéréseket, hogy végig kelleni járja a zárolási kérelmek listáját.

11.5.13. Zárfeloldások kezelése

Legyen *T* a tranzakció, mely feloldja a zárat az *A* adatbáziselemen. A következő lépéseket kell az ütemezőnek követnie:

1. Törli *T*-t a listából, mely az *A* zárolásait tartja.
2. Összehasonlítjuk a *T* által tartott zárat a csoportos móddal, ami az *A* esetén érvényes, annak érdekében, hogy megállapítsuk az új csoportos módot.
   a) A *T* által fenntartott zár nem egyezik meg a csoportos móddal. A csoportos mód nem lehet *S*, mert a *T* által tartott zár, akkor csak *S* lehetne, de feltételeztük, hogy különböznek. A csoportos mód nem lehet *X*, mert a *T* által tartott zár, akkor csak *X* lehetne, de feltételeztük, hogy különböznek. Egyedüli eset, ami előfordul, ha a csoportos mód *U*, azt jelenti, hogy a *T* által tartott zár legfeljebb *S* (ha *U* lenne, megegyezne a csoportos móddal, *X* nem lehet, ha a csoportos mód *U*). Nem kell a csoportos módot változtassuk, marad *U*.
   b) A *T* által fenntartott zár megegyezik a csoportos móddal. Meg kell vizsgáljuk az egész listát, hogy megállapítsuk az új csoportos módot. A zár értékét tekintve, a következő esetek állnak fenn:
      – Ha a zár *S*, azt jelenti, hogy csak osztott zárak lehettek engedélyezve, esetleg még az *A*-n. Ha még van legalább egy elem a listában, a csoportos mód *S* marad.
      – Ha a zár *U*, mivel *U* csak egy lehet, ez most pont felszabadult, és ha még van osztott zár a listában, a csoportos mód *S* lesz.
      – Ha a zár *X*, akkor tudjuk, hogy nincs más zárolás az *A*-n, ez felszabadult, és a következő lépésben új zárakat fogunk engedélyezni, ha vannak még *A*-ra váró tranzakciók.

   Megjegyzés: amint látjuk feloldás után esetleg *S* lesz a csoportos mód, amit ebben a lépésben beállítunk. Ha első két esetben nincs több tranzakció a listában, amely zárat tart az *A*-n, illetve az utolsó esetben felszabadult a zár, mondjuk, hogy beállítjuk a csoportos módot *N*-re (Non).
3. Megvizsgáljuk a Várakozik-e mező értékét:
   a. Ha az érték 'igen', azt jelenti, hogy engedélyeznünk kell újabb zárakat. Több módszer is létezik:
      – *Első-beérkezett-első-kiszolgálása*: Azt a zárolási kérést engedélyezzük, amelyik a legrégebben várakozik, a zárolási kérések kezelése algoritmus 3.a.*iii*. pontját követve, ha a csoportos mód *S*-re lett beállítva a 2-es pontban. Ha a felszabadult zár után a csoportos mód *N*-re van beállítva, miután ez a módszer szerint a legelső tranzakciót, amelyik vár, a listából engedélyezzük, utána a csoportos módot is be

kell állítsuk. Ez a várakozó tranzakció kérhet: *S*, *U* vagy *X* zárat, ennek megfelelően a csoportos mód is *S*, *U* vagy *X* lesz. Ez a stratégia biztosítja, hogy ne legyen kiéheztetés, a tranzakció ne várjon örökké.

- *Osztott záraknak elsőségadás*: Először az összes várakozó osztott zárat engedélyezzük, majd egy módosítási zárolást, ha van ilyen a várakozók között. Kizárólagos zárat csak akkor engedélyezünk, ha semmilyen más igény nem várakozik. Azon tranzakciók, melyek *U* vagy *X* zárat kérnek, előfordulhat, hogy nagyon sokat várnak.
- *Felminősítésnek elsőségadás*: Ha van olyan *U* zárral rendelkező tranzakció, mely *X* zárrá való felminősítésre vár, akkor előbb ezt engedélyezzük, ha nincs, akkor a fenti közül valamelyik stratégiát követjük.

b. Ha a Várakozik-e mező értéke 'nem'

- Ha a csoportos mód *N*, azt jelenti, hogy épp az utolsó tranzakció is felszabadította a zárat az *A*-n, és a zártáblából *A*-t törölni lehet.
- Ha a csoportos mód *S*, mert − amint láttuk − csak ez az eset állhat még fenn a zár felszabadítása után, azt jelenti, vannak osztott zárak az *A*-n, más nem vár, nincs semmi tennivalónk.

4. Törljük a zárat abból a listából (TKöv), melyben a *T* tranzakció zárolási bejegyzései vannak.

11.6. Optimista konkurenciavezérlés

Léteznek zárolás nélküli módszerek is a tranzakciók sorba rendezhetőségének a biztosítására. Két ilyen módszert fogunk említeni: *időpecsét* és *érvényesítés*. Mindkét módszer optimistának tekinthető, mivel feltételezik, hogy nem fordul elő nem sorba rendezhető viselkedés. Ha netán mégis előfordul, akkor az optimista megközelítések egyetlen megoldása, hogy azt a tranzakciót, amelynek a viselkedése nem sorba rendezhető abortálja (visszagörgeti) és utána újraindítja. A zárolási ütemezők késleltetik a tranzakciókat.

11.6.1. Konkurenciavezérlés időpecsét módszerrel

Az időpecsét sorrenden alapuló ütemezés alapelve az, hogy minden tranzakcióhoz rendel egy születési dátumot, vagy időpecsétet, amely jelzi, hogy mikor is jött létre a tranzakció a többi tranzakcióhoz viszonyítva. Az időpecsét tehát egy sorszámnak is tekinthető, amely megadja, hogy hol helyezkedik el a tranzakció a többi tranzakcióhoz viszonyítva a létrehozási időpont tekintetében, így a korábban létrejött tranzakció időpecsétje kisebb, míg a később létrejött tranzakció időpecsétje pedig nagyobb. Az időpecsét generálását a rendszeróra segítségével oldják meg vagy az ütemező egy számlálót tart karban, mely 1-el növekszik, akárhányszor egy tranzakció indul. Jelöljük TS(*T*)-el a *T* tranzakcióhoz rendelt időpecsétet (timestamp).

A tranzakciókezelő filozófiája az időpecsét módszer esetén: A különböző tranzakciókhoz tartozó egymással konfliktusban álló műveletek esetén a műveletek végrehatási sorrendje feleljen meg a tranzakciók időpecsét sorrendjének. Vagyis azon műveleteknek kell előbb végrehajtódniuk, amelyek időpecsétje kisebb. Ha a tranzakciókezelő a végrehajtás során olyan kísérletet tapasztal, hogy egy tranzakció olyan adatbáziselemet akar olvasni vagy módosítani, melyet egy fiatalabb tranzakció már egyszer módosított, akkor ezen műveletet megakadályozza, méghozzá úgy, hogy ezen idősebb tranzakciót abortálja. A visszagörgetés után újraindítja a tranzakciót, amely most egy nagyobb időpecséttel újrapróbálkozhat a műveletek végrehajtásával.

A zárolás és időpecsét alapú módszerek egyik alapvető különbsége, hogy a zárolásnál a lefoglalás felengedése után bármely tranzakció hozzáférhet az adatbáziselemhez, addig az időpecsét módszer esetén nincs lezárás, az adatbáziselem bármikor elérhető, viszont csak azon tranzakciók férhetnek hozzá, melyek fiatalabbak, mint az utolsó foglalást végző tranzakció. A zárolás hátránya, hogy sokat várakoztat, és holtpont léphet fel, az időpecsét hátránya, hogy túlságosan könnyen kerülhet abortálásra egy tranzakció. Az időpecséten alapuló módszer akkor jobb, ha sok tranzakció csak olvasási.

Az időpecsét esetén nincs várakozás, de hibák léphetnek fel, mivel nem azt figyeli, hogy az adatbáziselem feldolgozás alatt áll-e vagy sem, hanem csak az utolsó hozzáférést végző tranzakció időpecsétjét figyeli.

Az időpecsét sorrenden alapuló ütemezés esetén a működés alapelve, hogy tudjuk, mely tranzakció használta az egyes adatbáziselemet utoljára. Ehhez nyilván kell tartani az utolsó hozzáféréseket. Az ütemező ezen jelző és az új igénnyel fellépő tranzakció időpecsétjének összehasonlítása alapján dönti el, hogy engedélyezi-e az adatbáziselemhez való hozzáférést, vagy abortálnia kell az igénylő tranzakciót. Külön nyilván kell tartani mind az olvasásra, mind az írásra, hogy mely időpecsétű tranzakció volt az utolsó engedélyezett művelet igénylője. Ezen jelzőknek egyre növekedő értékeket kell tartalmazniuk.

Minden *X* adatbáziselemhez két időpecsétet társítanak, és egy bitet.

1. RT(*X*), az *X* olvasási ideje (read time), mely a legnagyobb időbélyegző, ami ahhoz a tranzakcióhoz tartozik, mely utoljára olvasta az *X* adatbáziselemet.
2. WT(*X*), az *X* írási ideje (write time), mely a legnagyobb időbélyegző, ami ahhoz a tranzakcióhoz tartozik, mely utoljára írta az *X* adatbáziselemet.
3. C(*X*), az *X* véglegesítési bitje (commit bit), melynek értéke akkor 1, ha a legújabb tranzakció, amely az *X*-et írta, már véglegesítve van. Ennek a bitnek az a célja, hogy elkerüljük azt a helyzetet, amelyben egy *T* tranzakció egy másik *U* tranzakció által írt adatokat olvas be és utána az *U*-t abortáljuk. (piszkos adat olvasása).

Az időpecséten alapuló ütemezés esetén felmerülő problémák:

a) *Túl késői olvasás*: Egy *T* tranzakció megpróbálja olvasni az *X* adatbáziselemet, de ha az *X* írási ideje nagyobb, mint a *T* időpecsétje, vagyis WT(*X*) > TS(*T*), azt jelenti, egy fiatalabb tranzakció már módosította az *X*-et. Legyen *U* a tranzakció, mely módosította az *X*-et. Mivel *U*-t *T* után indítottuk, *T* az *U* által írt értéket olvasná, viszont *T*-nek azt az értéket kellett volna olvasnia, mely *U* módosítása előtt volt, ez viszont már nem ismert, ezért *T*-t az ütemező visszagörgeti.

b) *Túl késői írás*: A *T* tranzakció megpróbálja írni az *X* adatbáziselemet, de az *X* olvasási ideje azt mutatja, hogy egy másik tranzakció, mely a *T* által beírt értéket kellett volna olvasnia, mivel fiatalabb a *T*-nél, de már beolvasta az *X* értékét, még mielőtt *T*-nek sikerült volna írni az *X*-et, tehát TS(*T*) < RT(*X*).

c) *Piszkos adat olvasás*: A *T* tranzakció olvassa az *X* adatbáziselemet, melyet utoljára az *U* tranzakció írta, melynek az időpecsétje kisebb, mint a *T*-é és nem is lenne semmi baj, ha *U* tranzakciót nem pörgetné vissza valamilyen hiba miatt a rendszer. Erre mondják, hogy *T* piszkos adatot olvasott és ajánlott a *T* olvasását akkorra halasztani, mikor *U* véglegesítve van vagy abortálja a rendszer, a véglegesítési bit segítségével oldja meg az ütemező a feladatot.

Az időpecséten alapuló ütemezés szabályai:

Legyen egy *T* tranzakció, mely olvasási vagy írási kéréssel fordul az ütemezőhöz. Az ütemező választásai a következők lehetnek:

- Engedélyezi a kérést.
- Visszagörgeti *T*-t, ha az időpecsétje megsérti a szabályokat és egy új időbélyegzővel újraindítja.
- Késlelteti *T*-t és később dönti el, hogy abortálja-e vagy engedélyezi *T*-t.

Az időpecséten alapuló mechanizmus által kikényszerített abortálások számának csökkentésére számos elgondolás született, mindenik alapja a tranzakciók konkurenciájának a csökkentése.

Egyik megoldás: Az ütemező létrehoz egy várakozó listát, amelyben a végrehajtásra jelentkező műveletek foglalnak helyet. A módszer alapelve, hogy a várakozó műveletek végrehajtása során a bejegyzett műveletek közül mindig a legkisebb sorszámú műveletet veszi sorra. Az alapelvek betartásával biztosítható a műveletek soros végrehajtása, mert egy *T* tranzakció első művelete csak akkor kerülhet végrehajtásra, ha a rendszerben már nem létezik nála kisebb időpecséttel rendelkező és még futó tranzakció. Ha viszont a tranzakció megkapta a vezérlést, nem engedi el, míg ő maga be nem fejeződik, hiszen menet közben nem jöhet tőle kisebb sorszámú

művelet. A később keletkező tranzakciók mindig egyre növekvő időpecsétet, azaz sorszámot kapnak. De így már ez a módszer esetén is megjelenik a várakozás.

Szabályok, melyet az időpecséten alapuló ütemező betart:

1. Legyen $r_T(X)$ az ütemezőhöz érkező kérés.
   a) Ha TS(T) ≥ WT(X) az olvasás megvalósítható.
      Ha C(X) értéke 1, az ütemező engedélyezi a kérést.
         Ha TS(T) > RT (X), akkor RT(X) : = TS(T)
         különben RT(X) nem változik.
      különben késlelteti T-t addig, amíg C(X) értéke 1 lesz vagy az X-et író tranzakció abortál.
   b) különben az olvasás fizikailag nem megvalósítható. Az ütemező T-t visszagörgeti, majd újraindítja nagyobb időpecséttel.
2. Ha az ütemezőhöz érkező kérés $w_T(X)$,
   a) Ha TS(T) ≥ RT(X) és TS(T) ≥ WT(X), az írás fizikailag megvalósítható.
      Az X új értékét beírjuk.
         WT(X) : = TS(T).
         C(X) : = 0.
   b) Ha TS(T) ≥ RT(X) és TS(T) < WT(X), az írás fizikailag megvalósítható, de X-nek már egy későbbi értéke van.
      Ha C(X) értéke 1, az X előző írását végző tranzakció véglegesítve van, T írását egyszerűen figyelmen kívül hagyjuk (egy fiatalabb tranzakció által írt érték a helyes), az ütemező engedi, hogy T folytatódjon.
      különben késlelteti T-t addig, amíg C(X) értéke 1 lesz vagy az X-et író tranzakció abortál.
   c) Ha TS(T) < RT(X), az írás fizikailag nem megvalósítható és az ütemező T-t visszagörgeti.
3. Legyen az ütemezőhöz érkező kérés T véglegesítése.

– Az ütemező a karbantartási listája alapján megkeresi az összes olyan X adatbáziselemet, amelybe T írt és a C(X)-et 1-re állítja.
– Az ütemező egy másik listájában megkeresi azon tranzakciókat, melyek a T által írt X adatbáziselemek véglegesítésére vártak, ezeknek megengedi, hogy folytatódjanak.

4. Legyen az ütemezőhöz érkező kérés a T visszagörgetésére vonatkozó döntés. (1-es és 2-es esetben is előfordult).

Minden olyan tranzakcióra, amely T tranzakció írásaira várakozott, az ütemező meg kell ismételje azok írási vagy olvasási kísérleteit.

11.17. példa: Legyen három tranzakció T_1, T_2 és T_3, amelyek három adatbáziselemhez X, Y és Z-hez férnek hozzá. A 11.31 ábrán láthatjuk a három tranzakció időbeli futási tervét, az események előfordulásának valós ideje a lapon lefelé nő. A táblázat tartlamazza az adatbáziselemekhez rendelt írási és olvasási időket, melyek értéke kezdetben 0.

$T1$	$T2$	$T3$	X	Y	Z
15	20	25	RT = 0 WT = 0	RT = 0 WT = 0	RT = 0 WT = 0
$r1(Z)$;					RT = 15
	$r2(Y)$;			RT = 20	
		$r3(X)$;	RT = 25		
$w1(Z)$;					WT = 15
	$w2(X)$;				
	rollback				
		$w3(Z)$;			WT = 25

11.31. ábra: Három tranzakció időpecséten alapuló ütemező alatti végrehajtása

Az első három művelet különböző adatbáziselemekre vonatkozik, melyek kezdeti olvasási értéke 0 volt és az olvasási műveletek engedélyezhetőek, mivel a tranzakciók időpecsétje nagyobb, mint 0. (algoritmus 1. a) pont).

A $w1(Z)$ végrehajtható, mivel TS($T1$) = RT(Z) és TS($T1$) ≥ WT(Z), algoritmus 2.a. alapján. A $w2(X)$ esetén TS($T2$) < RT(X), algoritmus 2.c alapján az írás nem megvalósítható, a 2-es tranzakciót az ütemező visszagörgeti, egy fiatalabb tranzakció olvasta az X adatbáziselemet. A $w3(Z)$ esetén TS($T3$) > RT(Z) és TS($T3$) > WT(Z), az írás megvalósítható. □

A kereskedelmi rendszerek egy kompromisszumot alkalmaznak a zárolás és időpecséten alapuló módszer között. Lehetőséget adnak a felhasználónak, hogy a tranzakciót csak olvasásinak (READ ONLY) vagy olvasási/írási tranzakciónak deklarálja. A csak olvasási tranzakciókat az időpecséten alapuló módszerrel hatják végre, az olvasási/írási tranzakciók esetén pedig a kétfázisú zárolást alkalmazzák és a csak olvasási tranzakciók elől is lezárják az olvasási/írási tranzakciók által használt adatbáziselemeket.

11.6.2. Konkurenciavezérlés érvényesítéssel

Az *optimista* konkurenciavezérlés egyik típusa az *érvényesítés* (validation). Ezen vezérlés esetén a rendszer megengedi, hogy a tranzakciók zárolások nélkül férjenek az adatokhoz és a megfelelő időben ellenőrzi, a tranzakció sorba rendezhető viselkedését. Ezt úgy valósítja meg, hogy mielőtt egy tranzakció írni kezdene értékeket az adatbáziselemekbe, egy "érvényesítési fázison" megy keresztül, melynek során a rendszer összehasonlítja a beolvasott és írandó elemek halmazait más aktív tranzakciók írásainak a halmazaival. Ha a fizikailag nem megvalósítható viselkedés kockázata lép fel, akkor a tranzakciót visszagörgetjük.

11.6.2.1. Konfliktusok

Konkfliktus: olyan egymást követő műveletpár az ütemezésben, amelynek ha a sorrendjét felcseréljük, akkor legalább az egyik tranzakció viselkedése megváltozhat.

Nem cserélhetjük fel a műveletek sorrendjét a következő esetekben:

1. Ugyanannak a tranzakciónak két művelete konfliktus, pl: $r_i(X)$; $w_i(Y)$. Egy tranzación belül a műveletek sorrendje rögzített, az ABKR ezt a sorrendet nem rendezheti át.
2. Különböző tranzaciók ugyanarra az adatbáziselemre vonatkozó írása konfliktus. Pl. $w_i(X)$; $w_j(Y)$. Ebben a sorrendben X értéke az marad, melyet T_j ír, fordított sorrendben pedig az marad, melyet T_i ír. Ezek az értékek pedig nagy valószínüséggel különbözőek.
3. Különböző tranzakcióknak ugyanabból az adatbáziselemből való olvasása és írása konfliktus. Tehát $r_i(X)$; $w_j(X)$ konfliktus, mivel ha felcseréljük a sorrendet, akkor a T_i által olvasott X-beli érték az lesz, melyet a T_j ír és az nagy valószínűséggel nem egyezik meg az X korábbi értékével. Hasonlóan a $w_i(X)$; $r_j(X)$ is konfliktus.

11.6.3. Érvényesítésen alapuló ütemező felépítése

Az ütemező minden T tranzakció esetén megkapja a tranzakció által olvasott adatbáziselemek halmazát: RS(T) (read set) és a T által írt elemek hamazát WS(T) (write set). A tranzakciókat az ütemező három fázisban hajtja végre:

1. *Olvasás*. A tranzakciók beolvassák az adatbázisból az összes elemet az olvasási halmazba.
2. *Érvényesítés*. Az ütemező összehasonlítja a tranzakció olvasási és írási halmazait. Ha az érvényesítés hibát jelez, a tranzakciót visszagörgetjük, egyébként folytatódik a harmadik fázissal.
3. *Írás*. A tranzakció az írási halmazban lévő elemek értékét beírja az adatbázisba.

Az ütemező a következő halmazokkal dolgozik:

– KEZD: a már futó, de még nem teljesen érvényesített tranzakciók halmaza, melyben minden T tranzakcióhoz az ütemező tárolja a KEZD(T)-t, a T indításának az időpontját.

– ÉRV: a már érvényesített, de a 3. fázisban az írásokat még nem befejezett tranzakciók hamaza, melyben az ütemező karbantartja minden *T* tranzakció esetén a KEZD (*T*)-t és az ÉRV(*T*)-t a *T* érvényesítésekor. ÉRV(*T*) az az idő, amikor a *T* végrehajtását gondoljuk a végrehajtás soros sorrendjében.
– BEF: a befejezett tranzakciók halmaza, melyben az ütemező minden *T* tranzakcióra tárolja a KEZD (*T*)-t, az ÉRV(*T*)- és a BEF(*T*)-t a *T* befejezésekor. Az ütemező időnként tisztogatja ezt a halmazt. Egy T_d tranzakciót lehet törölni, ha bármely T_a aktív tranzakcióra (vagyis $T_a \in ÉRV$ vagy $T_a \in KEZD$):

$$BEF(T_d) < KEZD(T_a)$$

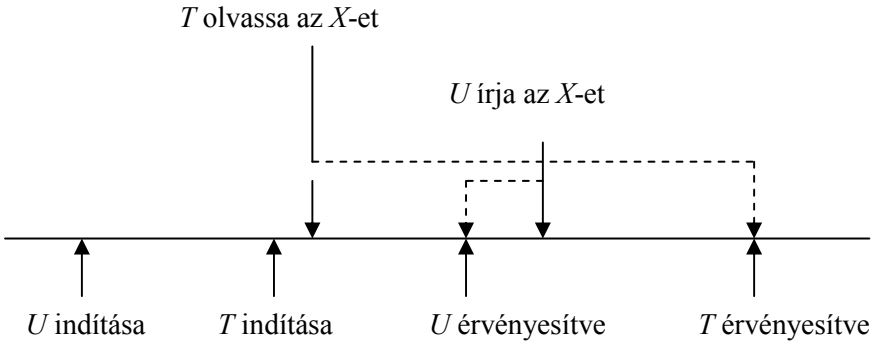
11.6.4. Érvényesítési szabályok

A szabályok megértése érdekében, lássuk, milyen hiba léphet fel, amikor egy *T* tranzakciót megpróbálunk érvényesíteni.

Feltételezzük, van olyan *U* tranzakció, hogy:

– *U* az ÉRV-ben vagy a BEF-ben van, vagyis az *U*-t már érvényesítettük.
– BEF(*U*) > KEZD(*T*), vagyis az *U* nem fejeződött be a *T* indítása előtt.
– RS(*T*) ∩ WS(*U*) nem üres, legyen *X* egy adatbáziselem a metszetből.

Ebben az esetben lehetséges, hogy *U* azután írja az *X*-et miután a *T* olvasta az *X*-et (Konfliktus 3. eset). Mivel nem tudjuk, hogy *T* tranzakciónak be kellett-e volna olvasnia az *U* által írt értéket vagy sem, (*U* indult először), *T*-t vissza kell pörgetnünk, hogy a soros sorrend megmaradjon.

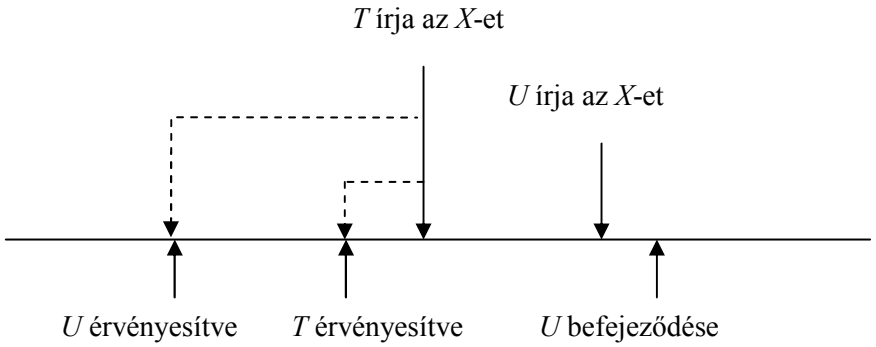


11.32. ábra: A *T*-t nem érvényesíthetjük, ha egy korábbi tranzakció, akkor ír valamit, amit a *T*-nek olvasniA kellett volna

Feltételezzük, van olyan *U* tranzakció, hogy:

– *U* az ÉRV-ben van; tehát az *U*-t sikeresen érvényesítette az ütemező;
– BEF(*U*) > ÉRV(*T*); tehát *U* nem lett befejezve mielőtt a *T* az érvényesítési fázisába lépett volna.
– $WS(T) \cap WS(U) \neq \emptyset$ és legyen *X* mind a két írási halmazban.

A felmerülő probléma a következő: úgy a *T*, mind az *U* írni fogja az *X* értékét (Konfliktus 2. eset) és előfordulhat, hogy *T* az *U* előtt írja. Mivel nem lehetünk biztosak, hogy a soros sorrend, mely szerint az *U* megelőzi a *T*-t, megmarad, *T*-t visszagörgetjük.



11.33. ábra: A *T* nem érvényesíthető, ha egy korábbi tranzakció, előtt tudna írni

Összefoglalás egy *T* tranzakció érvényesítésére vonatkozóan:

- minden olyan *U*-ra, amely még nem fejeződött be a *T* indítása előtt (vagyis BEF(*U*) > KEZD(*T*)) és *U* egy korábban érvényesített tranzakció, ellenőrizzük, hogy
  $$\mathrm{RS}(T) \cap \mathrm{WS}(U) = \varnothing .$$
- Minden olyan *U* tranzakcióra, mely még nem fejeződött be a *T* érvényesítése előtt (vagyis BEF(*U*) > ÉRV(*T*)) és *U* egy korábban érvényesített tranzakció, ellenőrizzük, hogy:
  $$\mathrm{WS}(T) \cap \mathrm{WS}(U) = \varnothing .$$

Tehát a *T* tranzakció érvényesíthető, ha mindkét feltétel teljesül.

11.7. A három konkurenciavezérlés működésének összehasonlítása

Három módszert láttunk a konkurenciavezérlésre: zárolás, időbélyegző és érvényesítés, mindegyiknek vannak előnyei.

Ami a tár felhasználását illeti, mindegyik az összes aktív tranzakciók által hozzáfért adatbáziselemek számának az összegével arányos megközelítőleg. Az időbélyegzés és érvényesítés kicsit több helyet használnak, mivel nyomon kell kövessék a a korábban véglegesített tranzakciók bizonyos hozzáféréseit.

Összehasonlíthatjuk a módszereket, abból a szempontból, hogy késleltetés nélkül fejeződnek-e be a tranzakciók. Egy módszer hatékonysága attól függ, hogy a tranzakciók közötti egymásra hatás erős vagy gyenge, vagyis milyen valószínűséggel akar egy tranzakció hozzáférni olyan elemhez, amelyhez egy konkurens tranzakció már hozzáfért.

A zárolás késlelteti a tranzakciókat, azonban elkerüli a visszagörgetéseket, még ha erős is az egymásra hatás. Az időbélyező és az érvényesítés nem késlelteti a tranzakciókat, azonban visszagörgetést okozhat, amely a késleltetésnek egy problémásabb formája és még erőforrásokat is pazarol.

Ha gyenge a tranzakciók közötti egymásra hatás, akkor sem az időbélyegzés, sem az érvényesítés nem okoz sok visszagörgetést és előnyösebb lehet a zárolásnál. Az időbélyegző módszer előnyösebb, mint az érvényesítés, ha visszagörgetésre kerül a sor, mivel az érvényesítés hagyja, hogy a tranzakció elvégezze az összes belső munkáját, mielőtt megnézné, hogy vissza kell-e görgetni a tranzakciót.

11.8. Tranzakciók az SQL-ben

A gyakorlatban általában nem lehet megkövetelni, hogy a műveletek egymás után legyenek végrehajtva, mivel túl sok van belőlük, ki kell használni a párhuzamosság által nyújtott lehetőségeket. Az ABKR-ek biztosítják a sorba rendezhetőséget, a felhasználó úgy látja, mintha a műveletek végrehajtása sorban történt volna, valójában nem sorban történik.

11.8.1. Csak olvasó tranzakciók

Ha a tranzakció csak olvas és nem módosítja az adatbázis tartamát, közölhetjük ezt az ABKR-el és akkor optimálisabb ütemezést valósít meg. Az utasítás amivel ezt közölhetjük az ABKR-el:

```
SET TRANSACTION READ ONLY;
```

11.8.2. Az elkülönítés szintjei SQL-ben

Az elmélet szerint a sorba rendezhető elkülönítési szint a biztos, amikor semmilyen konkurencia probléma nem merül fel., vagyis a tranzakció úgy fut le, mintha minden más tranzakció teljes egészében vagy előtte vagy utána ment volna végbe. A gyakorlatban viszont megengednek nagyobb interferenciát a tranzakciók között. A 11.34 ábra táblázata az elkülönítési szinteket, a megoldott és meg nem oldott problémákat összesíti.

ISOLATION LEVEL (Elkülönítési szint)	***Dirty read (Piszkos adat olvasása)***	***Unrepeteable read (Nem ismételhető olvasás)***	***Fantom***
READ UNCOMMITTED (nem olvasásbiztos)	Igen	Igen	Igen
READ COMMITTED (olvasásbiztos)	Nem	Igen	Igen
REPETEABLE READ (megismételhető olvasás)	Nem	Nem	Igen
SERIALIZABLE (sorba rendezhető)	Nem	Nem	Nem

11.34. ábra: Elkülönítési szintek és meg nem oldott problémák

A piszkos adat olvasása problémát láttuk a 11.4.2.-ben. *B* tranzakció módosítja a *P* adatbáziselemet, utána az *A* tranzakció olvassa a *P* adatbáziselemet, de időközben a *B* tranzakció abortál, így az *A* tranzakció olyan értéket látott, mely már nem aktuális.

Nem ismételhető olvasás akkor fordul elő, ha a tranzakciók párhuzamosan futnak, *A* tranzakció beolvas egy adatbáziselemet, időközben *B* tranzakció módosítja ugyanazt az adatbáziselemet, majd *A* tranzakció ismét beolvassa ugyanazt az adatbáziselemet, tehát *A* tranzakció „ugyanazt” az adatbáziselemet olvasta kétszer és két különböző értéket látott (11.35. ábra).

A Tranzakció	***Idő***	***B Tranzakció***
*sl*1(*P*); *r*1(*P*); *u*1(*P*)	t_1	-
-	|	-
-	t_2	*sl*2(*P*); *r*2(*P*); *xl*2(*P*)
-	|	*w*2(*P*); *u*2(*P*)
-	|	-
*sl*1(*P*); *r*1(*P*); *u*1(*P*)	t_3	-
-	|	

11.35. ábra: Nem ismételhető olvasás problémája

Fantomok: A rendszer csak létező adatbáziselemeket tud zárolni, nem könnyű olyan elemeket zárolni, melyek nem léteznek, de később beszúrhatók. T_1 azon sorok halmazát olvassa, melyek adott feltételnek tesznek eleget, T_2 új sort illeszt a táblába, mely kielégíti a feltételt, így a T_1 eredménye helytelen.

11.18. példa: Legyen T_1 a következő lekérdezés:

```
SELECT AVG(Atlag) FROM Diakok
WHERE CsopKod = '531'
```

T_2 pedig beszúr egy új diákot a Diakok táblába, tehát a T_1 által számított átlag nem helyes. Ha T_1 megismétli a kérést, akkor egy olyan sort lát, mely addig nem volt, ezt „fantom problémának" nevezik. □

Ez valójában nem konkurencia probléma, ugyanis a (T_1, T_2) soros sorrend ekvivalens azzal, ami történt.

Tehát van egy fantom sor a Diakok táblában, melyet zárolni kellett volna, de mivel még nem létezett, nem lehetett zárolni. A megoldás, hogy sorok beszúrását és törlését az egész relációra vonatkozó írásnak kell tekinteni és az egész relációra kizárólagos zárat kell kérni. Ezt nem kaphatja meg, csak ha a minden más zár fel van szabadítva, a példa esetén a *T*1 befejezése után. Egy más megoldás arra, hogy a rendszer megelőzze a fantom megjelenését: le kell zárnia a hozzáférési utat (acces path), mely a feltételnek eleget tevő adathoz vezet (láss részleteket a lekérdezés optimalizálása esetén a Garcia-Molina−Ullman−Widom 2000-ben).

Az izolálási szintet a tranzakciók esetén a felhasználó beállíthatja, a felhasználó nem ad semmi zárolást, rábízza a rendszerre. Ha a szerializálhatóságot kell garantálnunk, az izolálás szint a maximális kell legyen, vagyis sorba rendezhető (SERIALIZABLE). A

```
SET TRANSACTION ISOLATION LEVEL TO <elkülönítési szint>
```

parancs segítségével állíthatja be a felhasználó az elkülönítés szintjét. Amint a fenti táblázatból is láthattuk, ha a > erősebb feltételt jelent, akkor az elkülönítés szintjei között a következő relációk állnak fenn:

```
SERIALIZABLE > REPEATEBLE READ > READ COMMITTED >
     > READ UNCOMMITTED
```

Ha minden tranzakciónak sorba rendezhető (SERIALIZABLE) az elkülőnítés szintje, akkor több tranzakció párhuzamos végrehajtása esetén a rendszer garantálja, hogy az ütemezés sorba rendezhető. Ha egy ennél kisebb elszigeteltségi szinten fut egy tranzakció, a sorba rendezhetőség meg van sértve.

Ha a rendszer megenged sorba rendezhetőnél különböző elszigeteltségi szintet, azt jelenti, hogy megenged konkurenciát, akkor van saját explicit konkurenciakontrolt segítő mechanizmusa (zárak), amit a felhasználó explicit meg kell adjon. Ha nem garantálja a rendszer, akkor azt a felhasználó tudja kikényszeríteni.

Egy tranzakció, melynek az elszigeteltségi szintje sorba rendezhető (SERIALIZABLE), betartja a szigorú kétfázisú lezárási protokollt. Az ütemező lezárást alkalmaz, írás és olvasás előtt, tranzakció végéig tartja, azonkívül az objektum halmazt (indexen) is lezárja (ne jelenjen meg fantom).

A megismételhető olvasás (REPEATABLE READ) elszigeteltségi szint abban különbözik a sorba rendezhetőtől, hogy az ütemező indexet nem zárol, csak egyedi objektumokat, nem objektum halmazokat is. Olvasás előtt osztott, írás előtt kizárólagos zárat kér és tranzakció végéig tartja.

Az olvasásbiztos (READ COMMITTED) (ez szokott az implicit lenni) elszigeteltségi szint esetén a tranzakciókezelő az adatbázisból nem enged meg visszatéríteni olyan adatot, mely nincs véglegesítve („piszkos adat olvasás" nem fordulhat elő). Olvasás előtt osztott zárat kér a tranzakcióban szereplő objektumokra, utána rögtön felengedi, kizárólagos zárat kér írás előtt és azt a tranzakció végéig tartja.

Egy olyan tranzakció esetén, melynek az elszigeteltségi szintje nem olvasásbiztos (READ UNCOMMITTED), az ütemező nem kér semmilyen lezárást. Az a tranzakció, melynek ez az elszigeteltségi szintje, olvashatja egy más futó tranzakció által végzett változtatást, mely még nem volt véglegesítve. Más esetben, ha egy másik felkasználó időközben kitörli az adatot, melyet ez a tranzakció olvasott, nem jelez hibát. Nem ajánlott egyetlen alkalmazásnak sem, esetleg olyan statisztikai kimutatások esetén, ahol egy-két változtatás nem lényeges.

11.9. Gyakorlatok

11.1. Legyenek az A és B adatbáziselemek, a T_1 és T_2 tranzakciók és a következő műveleteik a megadott sorrendben:
BEGIN TRANSACTION T_1, $r_1(A)$, $r_1(B)$, BEGIN TRANSACTION T_2, $r_2(B)$, $r_2(A)$, $w_2(B)$, $w_1(A)$, COMMIT T_1, COMMIT T_2

a. Adjuk meg azt zároláson alapuló ütemezését a fenti tranzakcióknak, ahol csak a kizárólagos és osztott zárakat használjuk.
b. Adjuk meg azt a zároláson alapuló ütemezését a fenti tranzakcióknak, amikor módosítási, kizárólagos és osztott zárakat használunk.
c. Adjuk meg a tranzakciók időpecsétén alapuló ütemezését. Használjuk az X olvasási idejét (RT(X)), az X írási idejét, (WT(X)), az X véglegesítési bitjét (C(X)).
d. Hasonlítsuk össze a fenti 3 módszert.

11.2. Legyenek az A, B, C, D, E és F adatbázis elemek, a T_1, T_2, T_3, T_4 és T_5 tranzakciók és a következő műveleteik a megadott sorrendben:
BEGIN TRANSACTION T_1, $r_1(A)$, $r_1(E)$, BEGIN TRANSACTION T_2, $r_2(B)$, $r_2(F)$,
BEGIN TRANSACTION T_3, $r_3(B)$, $r_3(D)$, $r_1(D)$, $r_2(A)$, BEGIN TRANSACTION T_4, $r_4(E)$,
BEGIN TRANSACTION T_5, $r_5(F)$, $w_3(D)$, COMMIT T_3, $w_1(A)$, COMMIT T_1, $w_2(B)$, COMMIT T_2, $w_5(F)$, COMMIT T_5, $w_4(E)$ COMMIT T_4.

a. Adjuk meg azt zároláson alapuló ütemezését a fenti tranzakcióknak, ahol csak a kizárólagos és osztott zárakat használjuk.
b. Adjuk meg azt a zároláson alapuló ütemezését a fenti tranzakcióknak, amikor módosítási, kizárólagos és osztott zárakat használunk.
c. Adjuk meg a tranzakciók időpecsétén alapuló ütemezését. Használjuk az X olvasási idejét (RT(X)), az X írási idejét, (WT(X)), az X véglegesítési bitjét (C(X)).
d. Hasonlítsuk össze a fenti 3 módszert.

11.3. Legyenek az A, B, C, D, E, F adatbáziselemek, a T, U, V, W, Z tranzakciók és a következő műveleteik a megadott sorrendben:
BEGIN TRANSACTION T; $r_T(D)$; BEGIN TRANSACTION Z; $r_Z(B)$;
BEGIN TRANSACTION X; $r_X(C)$; $r_T(C)$; BEGIN TRANSACTION U; $r_U(E)$;
BEGIN TRANSACTION V; $r_U(F)$; $r_X(B)$; $r_Z(A)$; $w_X(B)$; $w_T(C)$; $w_U(F)$; COMMIT U; $w_Z(A)$;
$r_V(A)$; $w_X(C)$; COMMIT X; $w_T(D)$; COMMIT T; $w_Z(B)$; COMMIT Z; $w_V(A)$; COMMIT V.

a. Adjuk meg azt a zároláson alapuló ütemezését a fenti tranzakcióknak, ahol a kizárólagos és osztott zárakat használjuk. Ha probléma merül fel, oldjuk azt meg.
b. Adjuk meg azt a zároláson alapuló ütemezését a fenti tranzakcióknak, amikor módosítási, kizárólagos és osztott zárakat használunk.
c. Adjuk meg a tranzakciók időpecsétén alapuló ütemezését.

Hasonlítsuk össze a fenti 3 módszert.

11.4. Legyenek a következő bejegyzések a semmisségi naplózás naplójában:
<START S>; <S, A, 60>; <COMMIT S>; <START T>; <T, A, 61>; <START U>, <U, B, 20>;
<START CKPT (T, U)>; <T, C, 30>; <START V>; < U, D, 40>; <V, F, 70>; <COMMIT U>;
<T, E, 50>; <COMMIT T>; <END CKPT>; <V, B, 21>; <COMMIT V>.
Milyen tevékenységeket végez a helyreállítás-kezelő, ha hiba lép fel a

a. <T, E, 50 > bejegyzés után
b. <V, B, 21> bejegyzés után
c. <COMMIT V> bejegyzés után

11.5. Legyenek a következő bejegyzések a semmisségi naplózás naplójában:

<START *T*>; <*T*, *A*, 50>; <START *U*>; <*U*, *B*, 40>; <*T*, *C*, 20>; <START CKPT (*T*, *U*)>;
<*U*, *D*, 10>; <START *W*>; <COMMIT *T*>; <*W*, *E*, 10>; <*U*, *F*, 60>;
<COMMIT *U*>; <END CKPT>; <*W*, *G*, 70>; <START *X*>; <COMMIT *W*>; <*X*, *A*, 55>;
<START *Y*>; <*Y*, *B*, 45>;
<START CKPT (*X*, *Y*)>; <*X*, *D*, 15>; <START *Z*>; <*Y*, *E*, 15>;
<*Z*, *C*, 25>;
<COMMIT *X*>; <COMMIT *Y*>; <END CKPT>; <*Z*, *F*, 65>; <COMMIT *Z*>.
Milyen tevékenységeket végez a helyreállítás-kezelő, ha hiba lép fel a:

a. <COMMIT *T*> bejegyzés után
b. <COMMIT *U*> bejegyzés után
c. <*Y*, *B*, 45> bejegyzés után
d. <*Y*, *E*, 15> bejegyzés után
e. <COMMIT *Z*> bejegyzés után

11.6. Legyenek a következő bejegyzések a helyrehozó naplózás naplójában:
<START *S*>; <*S*, *A*, 61>; <COMMIT *S*>; <START *T*>; <*T*, *A*, 62>; <START *U*>, <*U*, *B*, 21>;
<START CKPT (*T*, *U*)>; <*T*, *C*, 31>; <START *V*>; < *U*, *D*, 41>; <*V*, *F*, 71>; <COMMIT *U*>;
<END CKPT>; <*T*, *E*, 51>; <COMMIT *T*>; <*V*, *B*, 22>; <COMMIT *V*>.
Milyen tevékenységeket végez a helyreállítás, ha hiba lép fel a

a. <*T*, *E*, 51> bejegyzés után
b. <*V*, *B*, 22> bejegyzés után
c. <COMMIT *V*> bejegyzés után

11.7. Legyenek a következő bejegyzések a helyrehozó naplózás naplójában:
<START *T*>; <*T*, *A*, 55>; <START *U*>; <*U*, *B*, 45>; <*T*, *C*, 25>; <START CKPT (*T*, *U*)>;
<*U*, *D*, 15>; <START *W*>; <COMMIT *T*>; <*W*, *E*, 15>; <*U*, *F*, 65>; <END CKPT>;
<COMMIT *U*>; <*W*, *G*, 75>; <START *X*>; <COMMIT *W*>; <*X*, *A*, 56>; <START *Y*>; <*Y*, *B*, 48>;
<START CKPT (*X*, *Y*)>; <*X*, *D*, 19>; <START *Z*>; <*Y*, *E*, 17>; <END CKPT>; <*Z*, *C*, 21>;
<COMMIT *X*>; <COMMIT *Y*>; <*Z*, *F*, 63>; <COMMIT *Z*>.
Milyen tevékenységeket végez a helyreállítás-kezelő, ha hiba lép fel a:

a. <COMMIT *T*> bejegyzés után
b. <COMMIT *U*> bejegyzés után
c. <*Y*, *B*, 48> bejegyzés után
d. <*Y*, *E*, 17> bejegyzés után
e. <COMMIT *Z*> bejegyzés után

11.8. Legyenek a következő bejegyzések a semmisségi/helyrehozó naplózás naplójában:
<START *S*>; <*S*, *A*, 60, 61>; <COMMIT *S*>; <START *T*>; <*T*, *A*, 61,62>; <START *U*>,
<*U*, *B*, 20, 21>; <START CKPT (*T*, *U*)>; <*T*, *C*, 30, 31>; <START *V*>; < *U*, *D*, 40,41>;
<*V*, *F*, 70, 71>; <COMMIT *U*>; <END CKPT>; <*T*, *E*, 50, 51>; <COMMIT *T*>; <*V*, *B*, 21, 22>;
<COMMIT *V*>.
Milyen tevékenységeket végez a helyreállítás-kezelő, ha hiba lép fel a:

a. <*T*, *E*, 50, 51> bejegyzés után
b. <*V*, *B*, 21, 22> bejegyzés után
c. <COMMIT *V*> bejegyzés után

11.9. Legyenek a következő bejegyzések a semmisségi/helyrehozó naplózás naplójában:
<START *T*>; <*T*, *A*, 50, 55>; <START *U*>; <*U*, *B*, 40, 45>; <*T*, *C*, 20, 25>;
<START CKPT (*T*, *U*)>; <*U*, *D*, 10, 15>; <START *W*>; <COMMIT *T*>; <*W*, *E*, 10, 15>;
<*U*, *F*, 60, 65>; <END CKPT>; <COMMIT *U*>; <*W*, *G*, 70, 75>; <START *X*>; <COMMIT *W*>;
<*X*, *A*, 55, 56>; <START *Y*>; <*Y*, *B*, 45, 48>; <START CKPT (*X*, *Y*)>; <*X*, *D*, 15, 19>;
<START *Z*>; <*Y*, *E*, 15, 17>; <END CKPT>; <*Z*, *C*, 25, 21>; <COMMIT *X*>; <COMMIT *Y*>;
<*Z*, *F*, 65, 63>; <COMMIT *Z*>.
Milyen tevékenységeket végez a helyreállítás-kezelő, ha hiba lép fel a:

a. <COMMIT *T*> bejegyzés után

b. <COMMIT *U*> bejegyzés után
c. <*Y*, *B*, 45, 48> bejegyzés után
d. <*Y*, *E*, 15, 17> bejegyzés után
e. <COMMIT *Z*> bejegyzés után

12. Osztott adatbázisok

Az osztott adatbázisok technológiája az adatfeldolgozás két különböző megközelítésének az egyesítése. Ez a két különböző megközelítés a hálózatok és adatbázis technológia. Amint az első fejezetben láttuk, az adatbázis technológia célja, hogy a külön-külön alkalmazások adatait központosítsa és központilag kezelje, létezzen logikai és fizikai adatfüggetlenség. A hálózatok technológiája szétosztja a munkát a csomópontok között. Hogyan lehet ezt a két ellentétes nézőpontot összeilleszteni úgy, hogy egy új technológiát kapjunk, mely erőteljesebb mindekettőnél? A megoldás, a két technológia integrálása.

12.1. Az osztott adatbázisrendszerek meghatározása

A 9. Fejezetben láttuk a kliens-szerver arhitektúráját az adatbázisoknak, több kliens is hozzáfér ugyanahhoz az adatbázis szerverhez. Egy cégnek vagy szervezetnek több helyen is lehet kirendeltsége, minden kirendeltségben találhatók számítógépek lokális hálózatba kötve. Minden kirendeltségnek meg van a maga feladata, adatai, és nagy valószínűséggel adatbázis szervere. A kirendeltségek kommunikációs hálózattal össze vannak kapcsolva, együtt egy rendszert alkotnak. Egy kirendeltség a rendszer egy csomópontja. Tehát minden csomópontban lehet több felhasználó, mely a saját csomópontjában vagy akármelyik más csomópontban levő adatbázist feldolgozni kívánja.

Egy *X* kliens az A csomópontból a saját csomópontjában található adatbázishoz a jogainak megfelelően a lokális hálózaton (LAN) keresztül hozzáférhet. Ha *X* felhasználónak szüksége van a B csomó adatbázisában található adatokra és van joga rá, akkor a távoli hálózaton (WAN) keresztül kell azt megtegye. Kétféleképpen teheti:

a) Megadja a hálózaton a *B* csomópontban található adatbázis „címét”, nevét, majd a tábla nevét. Ebben az esetben azt mondjuk, hogy a hegesztés látszik.
b) Használja a globális tábla nevet, melynek egy része az *A*, más része a *B* csomóban található, és az osztott adatbázis-kezelő rendszer keresi meg az adatokat az *A* és *B* csomóban. Azt mondjuk, a hegesztés nem látszik.

A b) megoldás az osztott adatbázis, az a) nem.

Az osztott adatbázis csomópontok összessége, melyek valamilyen kommunikációs hálózattal össze vannak kapcsolva, minden csomópont egy valódi adatbázis-kezelő rendszerrel és adatbázissal rendelkezik. A csomópontok hajlandók kell legyenek együtt dolgozni, hogy a felhasználó bármely csomópont adataihoz úgy férhessen hozzá, mint ha az a saját csomópontján levő adatok lennének.

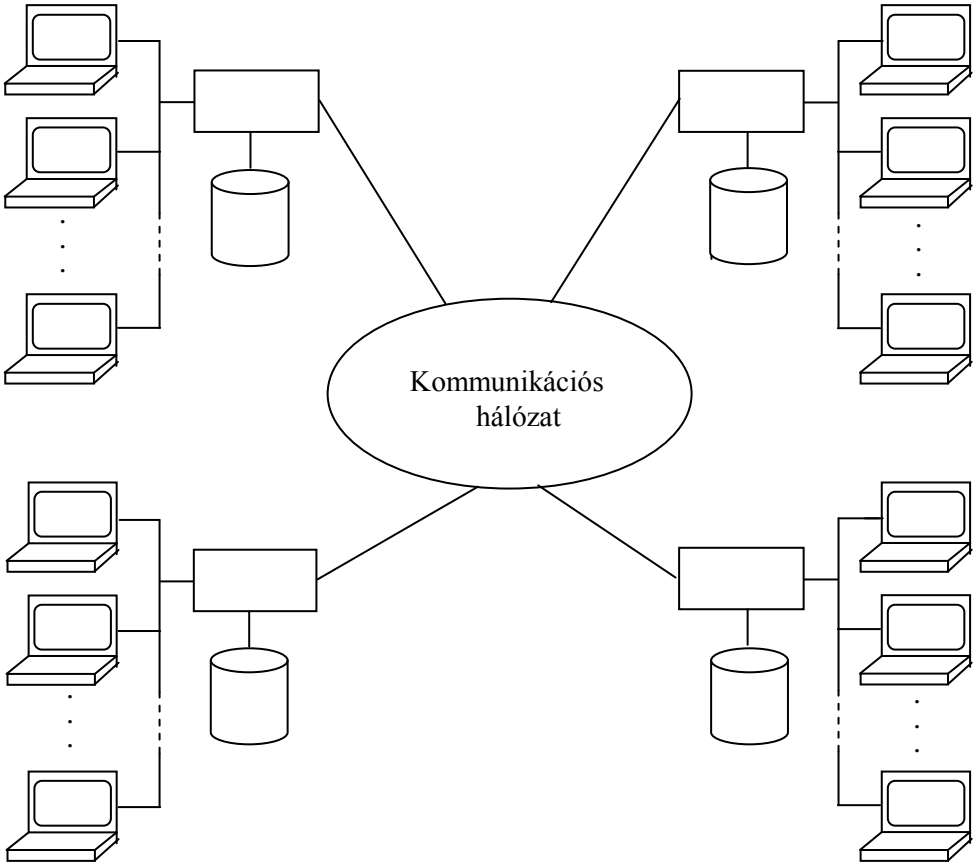
Az osztott adatbázis egy *virtuális objektum*, melynek komponensei fizikailag különböző csomópontokban vannak tárolva. Az osztott adatbázis ezeknek a logikai egyesítése.

Az osztott adatbázis fizikailag szétosztott, földrajzilag egymástól távol található több olyan adatbázis összessége, amelyekben különböző helyi ABKR-ek működhetnek, különböző operációs rendszereken. Az osztott adatbázis-kezelő rendszer (OABKR) feladata, hogy biztosítsa a csomópontok együttműködését.

Az angol könyvészet ([Da04], [ÖzVa91]) „site”-nak nevez egy csomópontot az osztott adatbázisrendszerben, mely több, mint egy hálózati „node”, mivel ABKR-t is tartalmaz. A magyar terminológiában a „site”-nak a munkaállomás felel meg .

Fontos, hogy különbséget tegyünk a valódi OABKR és azon rendszerek között, melyek megengedik a távoli adathozzáférést (vagyis azt, amit a kliens-szerver rendszerek megengednek). Egy „távoli adathozzáférésű” rendszerben a felhasználó dolgozhat olyan adatokkal, melyek egy távoli csomópontban vannak, vagy szimultán több csomópontban, de a „hegesztés” látszik, a felhasználó kell ezt kezelje. Ezzel ellentétben egy valódi osztott adatbázis-kezelő rendszerben a „hegesztések” rejtettek. A következőkben az osztott rendszer egy valódi osztott adatbázis-kezelő rendszert fog jelenteni és nem egy „távoli adathozzáférésű”-t.

Általában a csomópontok szét vannak osztva fizikailag, de ezt csak logikailag kell figyelemben venni, azonban két csomópont létezhet ugyanazon a fizikai gépen is. A kutatások során a földrajzilag szétosztott adatbázisokra tevődött a hangsúly, de a gyakorlatban előfordul, hogy ugyanabban az épületben lévő lokális hálózati rendszerben találhatóak. Az adatbázis szempontjából nem lényeges, hogy ugyanabban az épületben található-e vagy szét van osztva földrajzilag nagy területre, a távolági kapcsolat megoldása technikai probléma.



12.1. ábra. Osztott adatbázisrendszer

12.2. Az osztott adatbázisok előnyei és hátrányai

Előnyök:

- *Lokális autonómia*: az osztott adatbázisrendszerek megengedik az adatbázisok egy olyan felépítésének a használatát, amellyel tükrözni lehet a cég felépítését (ennek a részlegei földrajzilag már szét vannak osztva). A helyi adatokat lokálisan tárolják, ott ahol ezeknek logikailag van a helyük, és az esetek nagy részében abban a csomópontban felhasználják, amelyben találhatók.
- *Hatékonyság növelése*: az adatok visszakeresése tranzakciók segítségével lebontható több csomópontra, tehát a tranzakciókat végre lehet hajtani párhuzamosan. A párhuzamosság előnyös osztott rendszerek esetén, javítja az adatokhoz való hozzáférést.
- *Biztonság/elérhetőség növelése*: Az adatoknak több csomópontban létezhetnek másolatai. Ha egyik másolat meghibásodik, az adatok továbbra is elérhetők lesznek egy másolatból. Továbbá, rendszer hiba vagy kapcsolat megszakadás esetén nem válik az egész rendszer működésképtelenné. Annak ellenére, hogy esetleg egyes adatok nem elérhetők, az osztott adatbázis továbbra is nyújt bizonyos, korlátozott, szolgáltatásokat.

– *Gazdaságosság*: ha az adatok földrajzilag szét vannak osztva és ott tároljuk az adatokat, ahol a legtöbbet használják megtakaríthatók a kommunikációs költségek.
– *Terjeszthetőség*: osztott rendszerekben könnyebb az adatbázisok méreteit növelni.

Hátrányok:

– *Tapasztalat hiány*: az osztott adatbázisok még nincsenek elterjedve.
– *Bonyolultság*: az OABKR-eken belül felmerülő problémák sokkal összetettebbek, mint a központosítottak esetén, mert a központosított adatbázisok megvalósítási nehézségei mellett sok új probléma is megjelenik.
– *Költségek*: az osztott rendszerek újabb, kiegészítő hardvert (kommunikációs mechanizmusok, stb.) igényelnek, ez viszont növeli a költséget. A technikai problémák megoldására szükség van komplex szoftverre, további költségeket igényelve.
– *Vezérlés szétosztása*: ez a tulajdonság úgy szerepelt, mint előny. Sajnos, viszont a szétosztással felmerül a szinkronizálás és a koordinálás problémája. Így a szétosztás könnyen terhet jelenthet, ha nincs kidolgozva egy megfelelő politika, mely ezeket a problémákat megoldaná.
– *Védelem*: egy központosított adatbázis esetén, betartva a szabályokat a védelem könnyen megoldható az ABKR segítségével. Viszont egy OABKR-ben, ahol a helyzetet elbonyolítja a kommunikációs hálózat jelenléte, melynek meg vannak a saját védelmi követelményei, az OABKR-ek esetén a védelmi problémák sokkal bonyolultabbak, mint központosítottak esetén.
– *Változtatás nehézsége*: a legtöbb cég rengeteget fektetett be a jelenlegi központosított adatbázis rendszerébe, és jelenleg nem léteznek általános eszközök vagy módszerek, melyekkel a központosított adatbázist osztottá lehetne alakítani.

12.3. Az osztott adatbázisok célkitűzései

Az osztott adatbázis alapvető elve: A felhasználó számára az osztott rendszernek olyannak kell látszania, mint egy nem osztott rendszer. Vagyis más szavakkal, egy osztott rendszerben a felhasználók ugyanúgy dolgozhatnak, mintha a rendszer nem lenne osztott. Chris Date [Da04] tizenkét célkitűzést fogalmazott meg, amit egy osztott adatbázis–kezelő rendszernek be kellene tartania. Ezek a célkitűzések nem függetlenek egymástól, és nem egyformán fontosak. Az összes probléma ami az osztott rendszerhez kapcsolódik, belső vagy implementációs-szintű (vagy legalább is azok kell legyenek) és nem külső vagy felhasználó-szintű.

1. Helyi autonómia

Egy osztott rendszerben a csomópontok autonómok kell legyenek. A helyi autonómia azt jelenti, hogy bármilyen művelet egy adott csomópontban az illető csomópont által van vezérelve: egy *X* csomópont nem függhet egy *Y* csomóponttól saját műveletei végrehajtásában (különben az történne, hogy, ha az *Y* csomópont meghibásodik az *X* csomó sem tud működni, még akkor sem, ha az *X*-el semmi gond sincs). A helyi autonómia magába foglalja azt is, hogy a helyi adatok lokálisan vannak tárolva és kezelve. Minden adat csak egy adatbázishoz tartozik, függetlenül attól, hogy más, távoli csomópontból elérhető vagy sem. Olyan kérdések, mint biztonság, integritás és az adatok tárolása a lokális csomópont vezérlése alatt vannak.

Tulajdonképpen a helyi autonómiát nehéz teljes egészében megvalósítani, sok olyan eset van, amikor egy X csomópont bizonyos fokig egy Y csomópont vezérlése alatt áll. Így a cél inkább az, hogy az autonómia a lehető legnagyobb legyen.

2. Semmi bizalom egy központi csomópontban

A helyi autonómia maga után vonja, hogy minden csomópont egyenrangú, tehát nincs egy központi csomópont. Egy központi csomópont létezése nem kívánatos a következő két ok miatt: először is a központi csomópont nagyon le lenne terhelve, másodszor pedig, ami még fontosabb, a rendszer sebezhető lenne, vagyis, ha a központi csomópont meghibásodna, az egész rendszer működésképtelenné válna.

3. Folytonos műveletek

Az osztott rendszerek egyik előnye, hogy nagyobb megbízhatóságot és hozzáférhetőséget nyújthatnak. A rendszert úgy kell megtervezni, hogy legyenek az adatokról másolatok. Ez növeli a biztonságot, mert ha egy csomópont éppen nem működőképes, akkor az adatok még megvannak máshol is. A rendszer állandóan működőképes kell legyen.

4. Helytől való függetlenség

A felhasználó nem kell tudja, hogy fizikailag hol vannak az adatok, viszont úgy kell tudja őket kezelni (legalábbis logikai szempontból), mintha azok az ő saját helyi csomópontjában lennének. Ez a függetlenség azért szükséges, mert így a felhasználói programok egyszerűek. Az adat vándorolhat egyik csomópontból a másikba anélkül, hogy a rendszert megállítanánk. *Lokációtól való függetlenség*nek is nevezik vagy *átlátszóság*nak a *lokációval* szemben.

5. Tördeléstől (fragmentálástól) való függetlenség

Egy logikai reláció fizikai részekre bontását tördelésnek (fragmentálásnak) nevezzük. A tördelés hasznos, mivel a rendszer gyorsabb. A felosztott adat ott tárolható, ahol a legtöbbet használjuk, így a legtöbb művelet lokális és a hálózati forgalom a lehető legkisebb.

Többféle tördelést ismerünk: vízszintes, függőleges és levezetett.

Vízszintes tördelés: Legyen R egy reláció, n fragmensek száma és F_j (j = 1,..., n) konjunktív normál formában levő predikátum formulák. Vízszintes fragmenseket a vízszintes kiválasztás művelet segítségével a következőképpen kapunk:

$$R_j = \sigma_{F_j}(R); j = 1,\ldots,n$$

$$R_i \cap R_j = \varnothing \text{; ha } i \neq j;$$

$$R = \bigcup_{j=1}^{n} R_j .$$

Tehát, a töredékek diszjunktak és egyesítésükből visszakapjuk a globális R relációt.

12.1. példa: Legyen az Alkalmazottak tábla a NagyKer adatbázisból:

```
Alkalmazottak (SzemSzám, Név, Fizetés, Cím, RészlegID);
```

Az 1 és 2 azonosítóval rendelkező részleg adatai a kolozsvári csomópontban van tárolva, a 9-es részlegé pedig a nagyváradi csomópontban. Egy lehetséges parancs a tördelés leírására, mely az osztott adatbázis-kezelő rendszer katalógusában kerül tárolásra, a következő:

```
FRAGMENT Alkalmazottak INTO
    KolAlk AT SITE 'Kolozsvar' WHERE RészlegID = 1 OR RészlegID = 2
    NavAlk AT SITE 'Nagyvarad' WHERE RészlegID = 9;
```

Tehát a kolozsvári csomópontban a KolAlk nevű töredék van tárolva, a nagyváradi csomópontban pedig a NavAlk töredék (lásd 12.2 ábrát), a globális Alkalmazott tábla nincs tárolva, azt a rendszer előállítja a két töredék egyesítéséből.

```
Alkalmazottak = KolAlk ∪ NavAlk. □
```

Függőleges tördelés: Legyen R egy reláció, mely attribútumainak halmaza $A = \{A_1, A_2,..., A_n\}$. Legyen C az R elsődleges kulcsa, A_k olyan attribútumhalmazok, melyek nem tartalmazzák a kulcsot és diszjunktak, tehát:

$A_k \cap C = \emptyset$; k = 1,..., m;

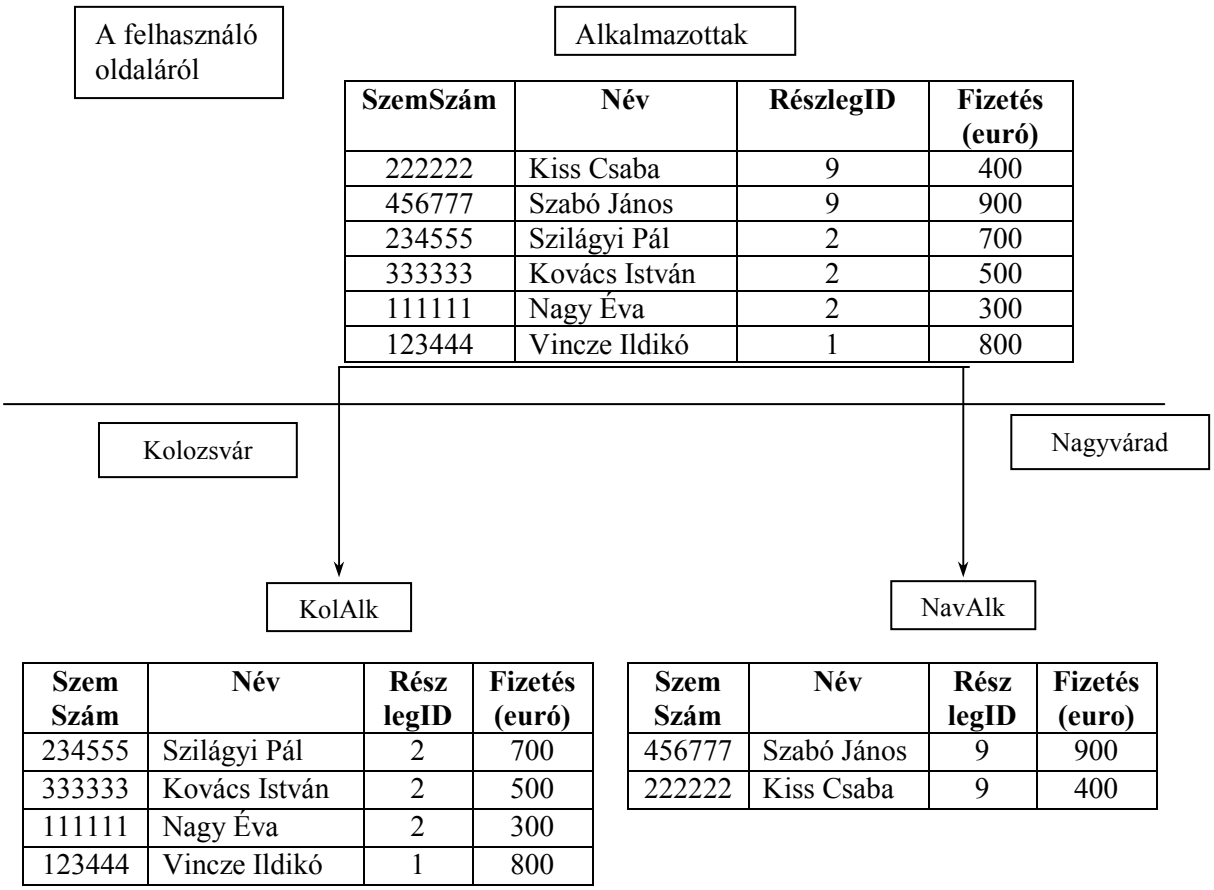
$A_k \cap A_l = \emptyset$; ha $k \neq l$.

Akkor egy függőleges töredéket a következőképpen kapunk:

$$R_k = \pi_{C,A_k}(R); k = 1,..., m;$$

A globális R relációt a természetes összekapcsolás műveletével kapjuk vissza:

$$R = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_m;$$



SzemSzám	Név	RészlegID	Fizetés (euró)
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500
111111	Nagy Éva	2	300
123444	Vincze Ildikó	1	800

Szem Szám	Név	Rész legID	Fizetés (euró)
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500
111111	Nagy Éva	2	300
123444	Vincze Ildikó	1	800

Szem Szám	Név	Rész legID	Fizetés (euro)
456777	Szabó János	9	900
222222	Kiss Csaba	9	400

12.2. ábra: Példa vízszintes tördelésre

12.2. példa: Legyen ismét az Alkalmazottak tábla a NagyKer adatbázisból:

```
Alkalmazottak (SzemSzám, Név, Fizetés, RészlegID);
```

Egy függőleges tördelését kapjuk az Alkalmazottak táblának a következőképpen (lásd 12.3. ábrát):

```
AlkAltalános (SzemSzám, Név, RészlegID);
AlkFizetés (SzemSzám, Fizetés).
```

A globális Alkalmazottak relációt megkapjuk, ha a két töredéket összekapcsoljuk:

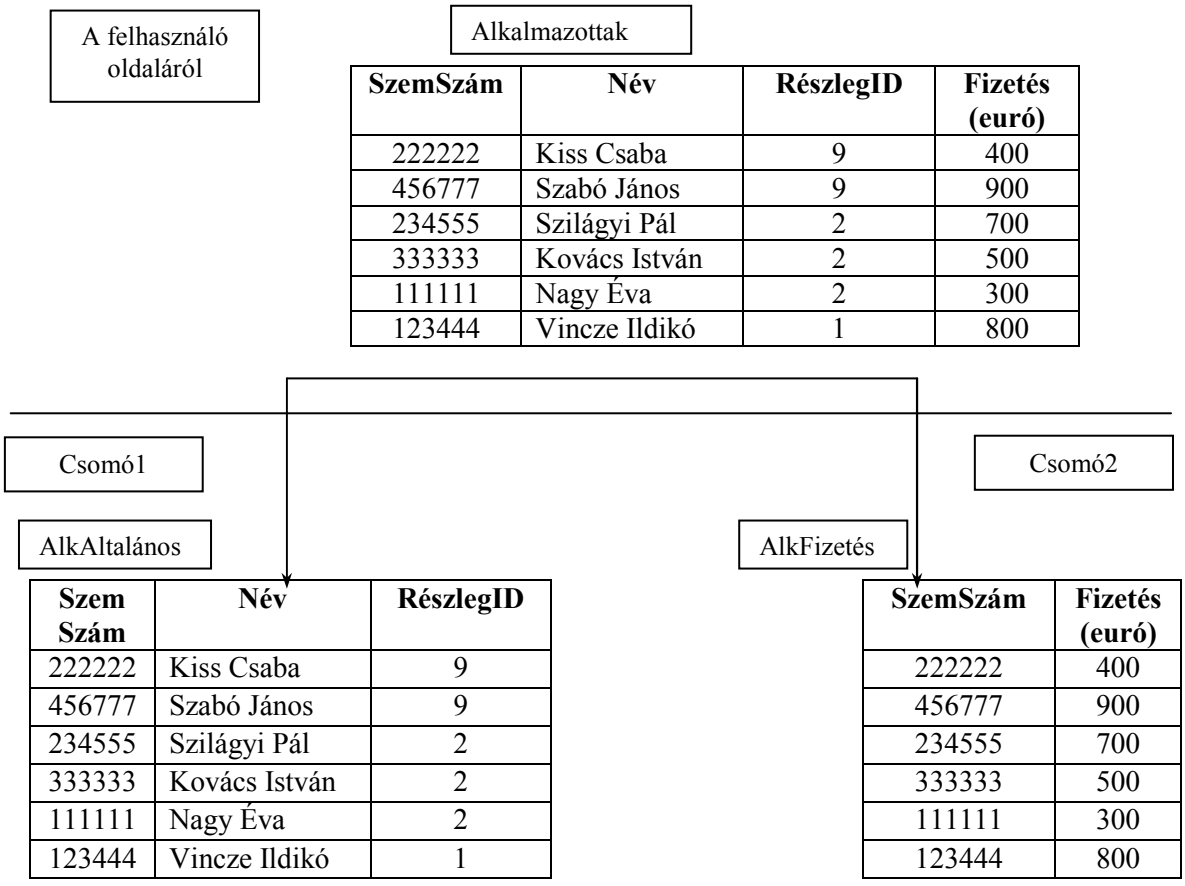
```
Alkalmazottak = AlkAltalános ⋈  AlkFizetés. □
```

Egy rendszer, mely megengedi a tördelést, biztosítania kell a *tördeléstől való függetlenség*et (nevezik még *átlátszóság*nak a *tördeléssel* szemben), azaz a felhasználók úgy kell lássák az adatokat (legalább is logikai szempontból), mintha azok egyáltalán nem lennének feldarabolva. A tördeléstől való függetlenség (hasonlóan a lokációtól való függetlenséghez) szükséges, mert így leegyszerűsödnek a felhasználói programok. Sajátos esetben megengedi, hogy a táblákat bármikor újra lehessen tördelni anélkül, hogy a felhasználói programokat módosítani kellene.

6. Másolatoktól való függetlenség

Ha egy osztott adatbázisrendszerben az adatbázis bizonyos részét több csomópontból is folyamatosan használják, akkor a kommunikációs költségek nagyok. Minden gép könnyen abba a helyzetbe kerülhet, hogy a különböző helyeken tárolt adatok folyamatos összerakásával foglalkozik, az alkalmazások felhasználóinak kiszolgálása érdekében. A teljesítmény elkerülhetetlenül romlik, ha az egyes gépeknek állandóan a tőlük távol tárolt adatokhoz kell

hozzáférniük. Az osztott adatbázisrendszerek célkitűzései között szerepel a kommunikációs költségek csökkentése. Másolatok készítésével jelentősen csökkenthetjük a lekérdezések feldolgozása során felmerülő kommunikációs költségeket.



SzemSzám	Név	RészlegID	Fizetés (euró)
222222	Kiss Csaba	9	400
456777	Szabó János	9	900
234555	Szilágyi Pál	2	700
333333	Kovács István	2	500
111111	Nagy Éva	2	300
123444	Vincze Ildikó	1	800

Szem Szám	Név	RészlegID
222222	Kiss Csaba	9
456777	Szabó János	9
234555	Szilágyi Pál	2
333333	Kovács István	2
111111	Nagy Éva	2
123444	Vincze Ildikó	1

SzemSzám	Fizetés (euró)
222222	400
456777	900
234555	700
333333	500
111111	300
123444	800

12.3. ábra: Példa függőleges tördelésre

Egy központi géptől való függőség megszüntetése szintén az osztott rendszerek célkitűzései között szerepel. Másolatok készítése segít megoldani ezt a problémát. Minden csomópont az általa gyakran használt táblákról, töredékekről készít helyi másolatot.

Másolatok létezése növeli az adatbázis elérhetőségét, mivel az egyes csomópontok akkor is működőképesek maradnak, ha az egyik csomópont meghibásodik. Az adatbázis elérhetősége veszélybe kerül, ha gyakran kell hozzáférni távoli adatokhoz, és a gyakran használt adatokat tároló gép meghibásodik. Az adatbázisnak lehetnek olyan részei, amelyek a cég számára létfontosságúak. Minden csomópontnak vannak saját biztonsági eszközei. Az osztott adatbázis helyreállíthatósága is javul, ha az adatokat egynél több gépen tároljuk.

A gyakorlatban a legtöbb osztott adatbázisrendszer bizonyos mértékig támogatja az adatreplikációt. Egy tárolt relációnak, vagy töredéknek több *másolata* lehet, melyek különböző csomópontokban vannak tárolva.

12.3. példa: Legyen a 12.1 példában bemutatott reláció, illetve annak a fragmensei. Készítsünk másolatot a kolozsvári csomópontban a nagyváradi fragmensről és fordítva. Egy lehetséges leírása a másolatok készítésének:

```
REPLICATE KolAlk
 NKolAlk AT SITE 'Nagyvarad';

REPLICATE NavAlk
 KNavAlk AT SITE 'Kolozsvar'; □
```

A másolat készítés két okból fontos:
- jobb teljesítményt jelent: az OABKR a helyi adatokon dolgozik, ahelyett, hogy távoli csomópontokkal kellene kommunikáljon;
- jobb hozzáférhetőséget biztosít: egy adott, másolattal rendelkező, objektum mindaddig elérhető lesz, amíg legalább egy másolata elérhető.

Hátránya a másolat készítésnek, ha adatkezelési műveleteket (új sor hozzaáillesztése, sor módosítása, sor törlése) végzünk a globális reláción, az összes másolatán el kell végezzük az adatkezelési műveleteket. Ez az *adatkezelési műveletek tovább terjedésé*nek a problémája. (lásd 12.5.2.-ben)

A másolat készítés ideális esetben a felhasználó számára „átlátszó" kell legyen. Vagyis egy rendszer, mely képes az adatok másolatának készítésére, kell biztosítsa a *másolatoktól való függetlenségé*t is (vagy más néven *átlátszóságot a másolással* szemben). Ez azt jelenti, hogy felhasználó úgy kell lássa az adatokat, legalább is logikai szempontból, mintha egyáltalán nem léteznének másolatok. A másolatoktól való függetlenség (hasonlóan a helytől, illetve fragmentálástól való függetlenséghez) fontos, mert így leegyszerűsödnek a felhasználói programok. Sajátos esetben megengedi, hogy bármikor létre lehessen hozni, illetve el lehessen törölni másolatokat, anélkül, hogy a felhasználói programokat módosítani kellene.

7. Lekérdezés osztott feldolgozása

A lekérdezés optimalizálása sokkal fontosabb egy osztott adatbázisrendszerben, mint egy központosítottban. Ha egy lekérdezés több csomópontban tárolt relációra vagy töredékre vonatkozik, több lehetőség is van szállítani a relációkat a csomópontok között. A központosított rendszerek lekérdezés optimalizálásához ez plusz feladatként jelenik meg.

A relációs adatbázisok megfelelőbbek az osztott adatbázisrendszerben, mint azok, melyek nem relációs adatmodellre épülnek. A relációs modell esetén egy lekérdezés részeredményei is relációk, a hálózaton eredményhalmazokat lehet küldeni, nem kell egyenként küldeni sorokat, mint például a hálós adatmodell esetén.

8. Tranzakciók osztott vezérlése

Egy osztott adatbázisrendszerben egy tranzakció módosításai több csomópontra is kiterjednek. Mindegyik tranzakció több ügynökből (ágens) épül fel. Egy ügynök az a folyamat, amit a tranzakció egy adott csomópontban végez el. A rendszernek biztosítania kell, hogy egy adott tranzakció összes ügynöke sikerrel fejeződjön be. Ha egy tranzakció egy ügynökének nem sikerült a munkáját befejezni, a tranzakciót visszagörgeti, ami azt jelenti, hogy az adott tranzakció többi ügynöke semmissé kell tegye amit a tranzakció végrehajtása érdekében dolgozott.

9. Hardwertől való függetlenség

A osztott rendszerben több különböző típusú gép is lehet, például IBM, Macintosh, HP, stb. Ezek a számítógépek egyenrangú partnerek kell legyenek az osztott rendszerben. Működhet rajtuk ugyanaz a típusú adatbázis-kezelő rendszer, például Oracle vagy különbözőek, lásd a 12-es célkitűzést.

10. Operációs rendszertől való függetlenség

A különböző típusú gépen, mely az osztott rendszerben együttműködik különböző operációs rendszer futhat. Ugyanazt az ABKR-t különböző operációs rendszeren is lehessen futtatni és ugyanabban az osztott rendszerben tudjanak együttműködni. Például az Oracle-nek a Windows operációs rendszer alatt futó változata és UNIX alatti változata tudjon az osztott rendszerben együttmőködni.

11. Hálózattól való függetlenség

Ha egy osztott rendszerben megengedettek a különböző csomópontokban a különböző típusú gépek, különböző operációs rendszerrel, természetes, hogy megengedett kell legyen a különböző kommunikációs hálózat is.

12. Adatbázis-kezelő rendszertől való függetlenség

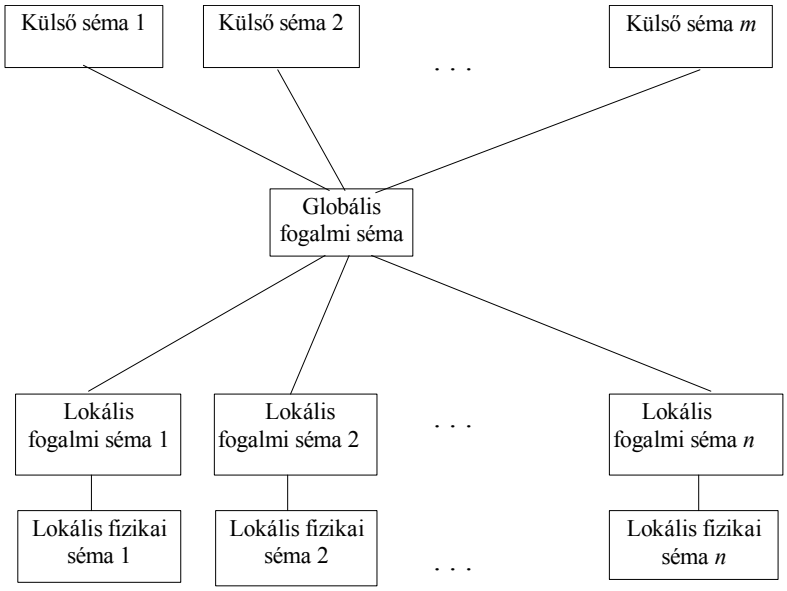
Egy osztott rendszerről azt mondjuk, hogy szigorúan homogén, ha minden csomópontjában ugyanaz az ABKR fut. Viszont ez egy túl erős megkötés, ezért az a cél, hogy ezt csökkentsük. Valójában csak az lesz szükséges, hogy a különböző csomópontokban található ABKR-ek *ugyanazt az interfészt támogassák*, és nem kell ugyanazon ABKR másolatai legyenek. Például, ha az MS SQL Server és az ORACLE ismerik a szabványos SQL-t, akkor elképzelhető, hogy egy MS SQL Server csomópont és egy ORACLE csomópont képes egymással kommunikálni egy osztott rendszer keretén belül. Vagyis lehetséges, hogy egy osztott rendszer *heterogén* legyen, legalább is bizonyos mértékig.

A heterogenitás támogatása kétségtelenül fontos. A valós rendszerekben nemcsak különböző gépek működnek, különböző operációs rendszerekkel, hanem a leggyakrabban különböző ABKR-eket működtetnek és az lenne jó, ha ezek a különböző ABKR-ek együtt tudnának működni egy osztott rendszerben. Tehát az ideális osztott rendszer kell biztosítsa az *ABKR-től való függetlenség*et.

12.4. Osztott adatbázis architektúrák

Az 1-es fejezetben láttuk az adatbázisok ANSI-SPARC arhitektúráját. Ezt az arhitektúrát kibővítették osztott adatbázisokra (lásd 12.4 ábra).

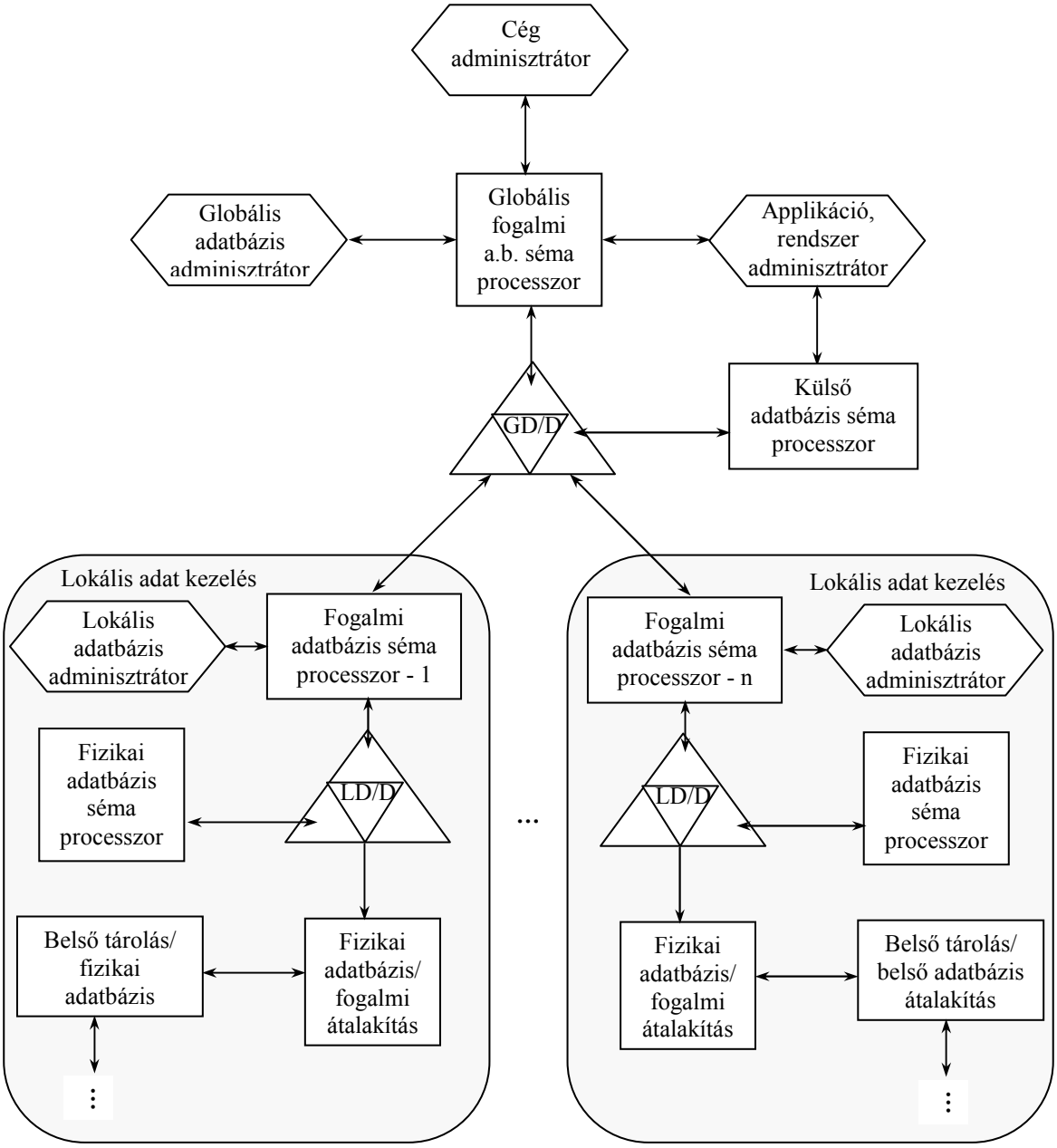
A központosított arhitektúra fogalmi szintjének a *globális fogalmi séma* felel meg, mely az osztott rendszer összes adatának általános szerkezetét írja le. A felhasználó ezen a szinten keresztül fér hozzá az adatokhoz, tehát nem látja, hogy az adat szét van darabolva, a globális relációt használhatja a lekérdezések, illetve adatkezelési műveletei esetén. A felhasználói alkalmazások és a felhasználó hozzáférését az adatbázishoz a *külső séma*k (*KS*) támogatják.



12.4. ábra: Osztott ANSI/SPARC arhitektúra

Egy osztott adatbázisban az adatok lehetnek tördelve, illetve a táblákról, töredékekről létezhetnek másolatok. Minden csomópontnak ismernie kell az adatok logikai szervezését, ezért létezik egy harmadik réteg az arhitektúrában, éspedig a *lokális fogalmi séma*. A globális fogalmi

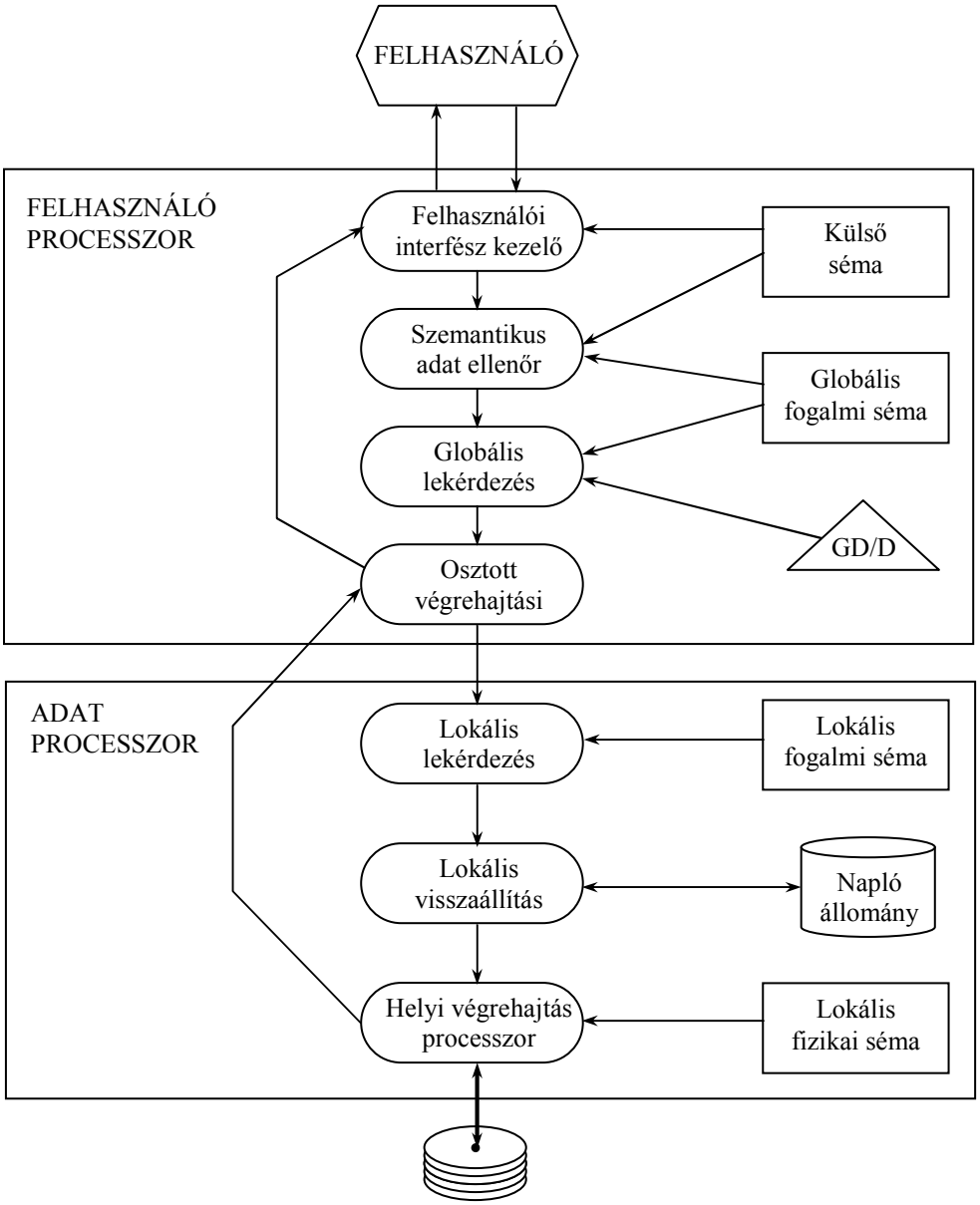
séma ezeknek a lokális fogalmi sémáknak az egyesítéséből jön létre. Fizikailag az adatok több csomópontban vannak tárolva, minden csomópontnak ismerie kell a saját adatainak a fizikai tárolási részleteit. Ez azt jelenti, hogy kell legyen egy egyedi fizikai (belső) séma definíció minden csomópontban, melyet *lokális fizikai* (*belső séma*nak) nevezhetünk.



12.5. ábra: Osztott ANSI-SPARC modell funkcionalitást figyelembe véve

Az itt bemutatott arhitektúra biztosítja a korábbiakban leírt átlátszóságokat. A logikai és fizikai adatfüggetlenséget biztosítja mivel az ANSI/SPARC arhitektúra kiterjesztése. A másolatoktól és a helytől való függetlenséget a lokális és globális fogalmi sémák biztosítják a köztük levő kapcsolatokon keresztül. A felhasználó lekérdezései nem veszik figyelembe az adatok helyét, illetve azokat a komponenseket, amelyek részt vesznek a lekérdezés végrehajtásában. A globális lekérdezéseket az osztott adatbázis-kezelő rendszer lebontja lokális lekérdezésekre, melyek más-más csomópontban hajtódnak végre.

Azt az ANSI/SPARC modellt is, mely figyelembe veszi a funkcionalitást, kiterjesztették osztott adatbázisokra (lásd 12.5 ábra). Kibővítették a *globális katalógus/szótárral* (*global directory/dictionary GD/D*), amely megengedi a szükséges globális leképezéseket. A lokális leképezéseket pedig a *lokális katalógus/szótár* (*local directory/dictionary LD/D*) kezeli. Így lokális adatbázis-kezelő komponensek be vannak építve a globális ABKR funkcióiba. A lokális fogalmi sémák a globális fogalmi séma leképezései minden egyes csomópontra. Az adatbázisok tipikusan fentről lefele stílusban vannak tervezve, és ezért az összes külső nézet globálisan van definiálva.



12.6. ábra: OABKR komponensei

A 12.6 ábra bemutatja egy OABKR komponenseit. Az egyik komponens (*felhasználói processzor*) kezeli az interakciót a felhasználóval, a másik (*adat processzor*) a tárolással foglalkozik.

A felhasználói processzor négy elemből áll:

– A *felhasználói interfész kezelő* felelős azért, hogy értelmezze a felhasználótól érkező parancsokat, továbbküldje végrehajtásra, majd az eredmény adatokat a felhasználónak minél érthetőbb formában szolgáltassa.

– A *szemantikus adat ellenőr* használja a globális fogalmi séma részeként definiált helyességi megszorításokat és hozzáférési jogokat, hogy ellenőrizze, hogy a felhasználói lekérdezés végrehajtható-e.
– A *globális lekérdezés optimalizáló* meghatároz egy végrehajtási stratégiát, mely minimalizálja a költség függvényt. A globális lekérdezéseket lefordítja lokális lekérdezésekre, felhasználva a globális és lokális fogalmi sémákat, illetve a globális könyvtárat/szótárat. A globális lekérdezés optimalizáló felelős, többek között, azért, hogy meghatározza a legjobb sorrendjét a *join* műveletek végrehajtásának.
– Az *osztott végrehajtási monitor* koordinálja a felhasználó kérésének osztott végrehajtását. Ezt nevezik *osztott tranzakció kezelő*nek is. A lekérdezések osztott módon történő végrehajtása során, a különböző csomópontokban található végrehajtási monitorok egymással tudnak kommunikálni.

A második fő komponense egy OABKR-nek az *adat processzor*. Ez a komponens az osztott adatbázis és ez a következő három elemből áll:
– A *lokális lekérdezés optimalizáló*, mely úgy működik, mint egy *hozzáférési út szelektor*, felelős bármelyik adat eléréséhez vezető legjobb hozzáférési út kidolgozásáért.
– A *lokális visszaállítás-kezelő* felelős azért, hogy a lokális adatbázis konzisztens maradjon, még akkor is, ha hiba lép fel.
– A *helyi végrehajtás processzor* fizikailag hozzáfér az adatbázishoz, a lekérdezés optimalizáló által generált végrehajtási tervben található fizikai parancsok szerint. Ez a processzor az interfész az operációs rendszer felé, és tartalmazza az *adatbázis puffer kezelő*t, mely a fő memória pufferek fenntartásáért felelős, és kezeli az adathozzáférést.

12.5. Osztott adatbázisrendszerek problémái

12.5.1. Osztott adatbázisok tervezése

Az osztott adatbázisok relációs adatmodellre alapulnak, láttuk a célkitűzések esetén, hogy mivel a lekérdezések eredménye szintén reláció, tehát a szállítandó részeredmények adathalmazok. Többek között ezért is jobb a relációs, mint más modellek. A mai kereskedelmi rendszerek zöme a relációs adatmodellt használja. Tehát a tervezésnél használhatjuk a normalizálást vagy egyed-kapcsolat diagram átírását relációs adatmodellé. Az így kapott adatbázis séma a globális séma.

Két stratégiát említünk az osztott adatbázisok tervezése szempontjából:
– *top-down* megközelítés,
– *bottom-up* megközelítés

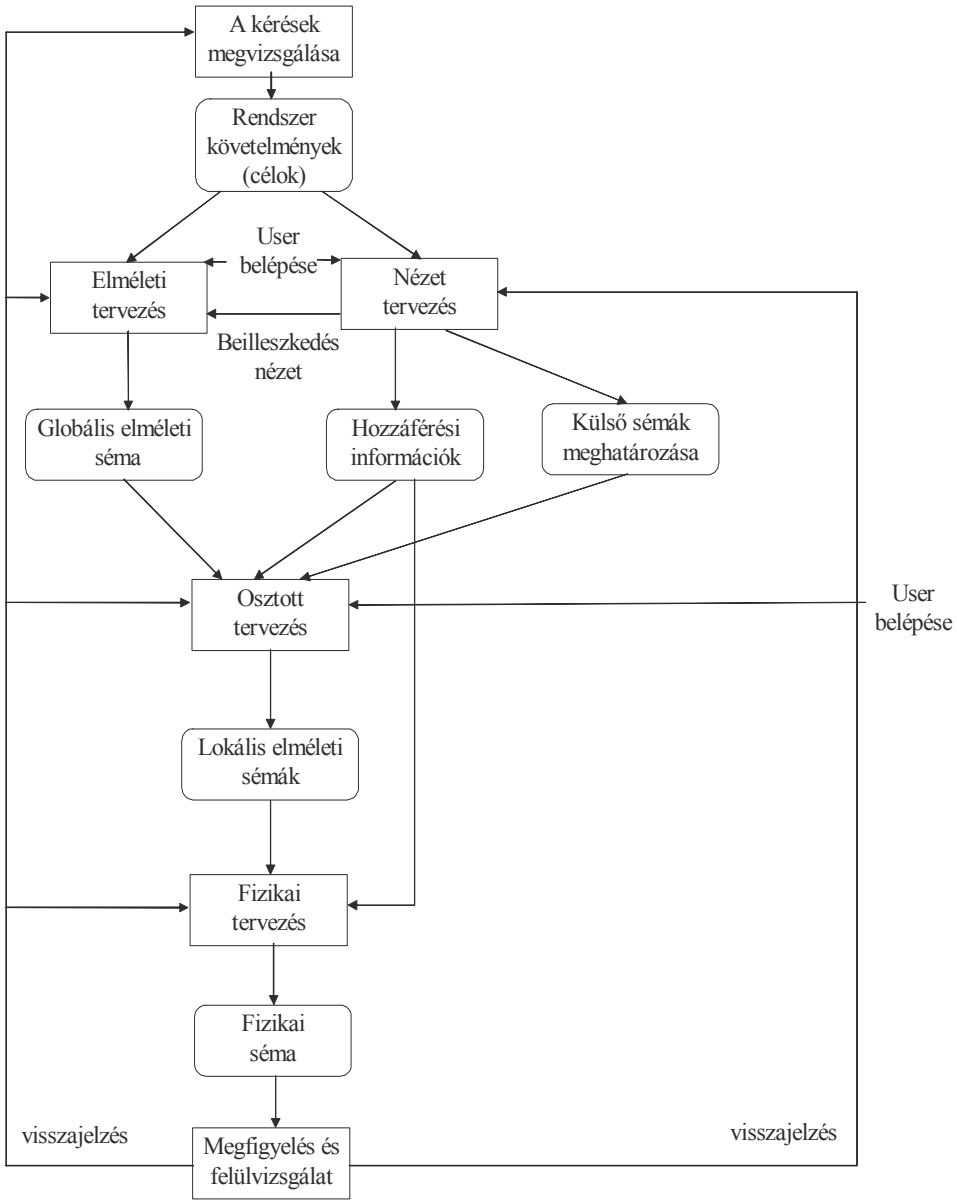
A top-down tervezési folyamat

Ezt a folyamatot a 12.7 ábra szemlélteti. A tevékenység a kérések elemzésével kezdődik, amelyek meghatározzák a rendszer környezetét. A fogalmi tervezés (conceptual design) eredménye a globális fogalmi séma lesz, mely az osztott rendszer minden egyedét és azok közötti kapcsolatokat tartalmazza. A kéréseket a nézetek tervezésénél is figyelembe kell vegyük. A nézetek tervezésének eredményeképp a felhasználói interfészeket kapjuk. A felhasználói nézetek a globális sémát, nem pedig lokális sémákat használnak.

Osztott rendszerek esetén a lokális sémát is meg kell határoznunk, ami a relációk tördelését illeti. Melyik relációt tördeljük és milyen feltétel alapján? Melyik töredéket, melyik csomópontban tároljuk? Kutatások ezen a területen matematikai programozást alkalmaznak, lásd [ÖzVa91]. A lokális séma tervezése esetén a másolatokat is meg kell határoznunk. Melyik táblát, melyik töredékét másoljuk és melyik csomópontban?

Az utolsó lépés a tervezési folyamatban a fizikai tervezés folyamata, mely figyelembe kell vegye a lokális fogalmi sémát.

A tervezés általános problémája NP-teljes. Az ajánlott megoldások heurisztikusak. Ezek közül egy: tároljunk minden adatot ott, ahol a legtöbbet használják.



12.7. ábra: Osztott adatbázis tervezése top-down módszerrel

A bottom-up tervezési folyamat

A top-down tervezési módszer alkalmazható, ha az osztott adatbázisból még semmi nem létezik. A gyakorlatban viszont gyakran az osztott adatbázis részei már léteznek, és a tervezési tevékenység ezek beillesztését is meg kell valósítsa. A bottom-up megközelítés alkalmasabb ilyen típusú környezetben.

Az indulási pont a bottom-up tervezés esetében a lokális fogalmi sémák. A folyamat abban áll, hogy a lokális sémákat beilleszti a globális fogalmi sémába.

12.5.2. Adatok másolásával felmerülő problémák

Ha egy rendszer megengedi a táblák, fragmensek másolatának létezését, gondoskodnia kell az adatkezelési műveletek továbbításáról az összes létező másolat felé. Ha a rendszer nem engedi meg másolatok létezését, a probléma nem merül fel, mivel minden adategységnek csak egy példánya létezik.

Az adatkezelési műveletek továbbításával az összes létező másolat felé az a gond, hogyha egyik csomópont azok közül, mely tárolja a sok közül az egyik másolatot nem elérhető, az adatkezelési művelet nem végezhető el. Ez részben megsérti az első célkitűzést. Egy *X* csomópont a saját adatán nem végezhet adatkezelési műveletet, ha létezik arról egy másolat egy olyan csomópontban, mely nem elérhető.

Egy megoldás az elsődleges másolat séma (primary copy scheme), mely a következőképpen működik: egy *X* logikai adatbázis objektumnak (tábla vagy fragmens), melynek vannak másolatai, az egyik másolatát kinevezi „elsődleges másolatnak" (primary copy), a többi másolat másodlagos (secondary) lesz. Különböző objektumok elsődleges másolatai különböző csomópontokban vannak, nincs egy központi, ahol tárolva van az összes elsődleges másolat, hogy a 2. célkitűzés be legyen tartva és ne legyen egy csomópont leterhelve. Tehát ez egy osztott séma.

Az elsődleges másolat séma esetén egy adatkezelési művelet logikailag teljes a rendszer szempontjából, ha az elsődleges másolaton végre lett hajtva. A csomópont, mely tárolja az elsődleges másolatot felelős azért, hogy továbbítsa az adatkezelési műveletet egy adott időn belül az összes létező másolat felé. Ahhoz, hogy a tranzakciók ACID tulajdonságai be legyenek tartva az adatkezelési művelet továbbítása az összes másolat felé a COMMIT előtt meg kell történjen. További részleteket lásd a tranzakciókezelés osztott adatbázisok paragrafusban.

Ezzel a megoldással is vannak problémák: ha egy *X* csomópont *A* fragmensének van több másolata és az elsődleges másolat nem az *X* csomópontban van, hanem egy távoliban, mely épp nem elérhető, az *X* nem tud dolgozni az *A* fragmenssel, mely a saját adata.

A kereskedelmi rendszerek elfogadják adatkezelési műveletek késleltetett továbbítását (delayed update propagation). Ezzel a megoldással az a gond, hogy nem lehet tudni az adatbázis mikor konzisztens és mikor nem az. Egyes rendszerek a SNAPSHOT fogalmát használják a másolat megvalósítására. Különböző rendszerek, különbözőképpen valósítják meg a másolatokat, illetve a tördelést.

12.5.3. Tranzakciókezelés osztott adatbázisok esetén

Amint a Tranzakciókezelés című fejezetben láttuk, a tranzakció a helyesség, a konkurencia és a helyreállítás egysége. Köpontosított adatbázis esetén a konkurencia problémákat általában zárolással oldják meg. Egy másolatokat engedélyező osztott környezetben hálózatszintű lezárásokat kell alkalmazni. Feltételezve, hogy minden csomópont felelős a saját objektumai lezárásáért, egy távoli csomópontban levő adatbáziselem módosítása a következő üzeneteket igényli:

- a zárolást kérni kell a távoli csomóponttól;
- a távoli csomópont küldi az engedélyt, ha a zárolás lehetséges;
- adatkezelési műveletet kell kérni a a távoli csomóponttól;
- a távoli csomópont küldi a művelet elvégzésének elismervényét (acknowledgment)
- zárolás feloldását kéri a távoli csomóponttól.

Legyen *T* egy tranzakció, mely egy *X* adatbáziselemet akar módosítani, melynek *k* darab másolata van egy távoli csomópontban. Egy lehetséges megvalósítása a *k* darab másolat módosításának 5*k* üzenetet von maga után:

- *k* számú zárolási kérés;
- *k* számú zárolási engedély;
- *k* számú módosítási művelet;
- *k* számú elismervény;
- *k* számú zárfeloldási kérés.

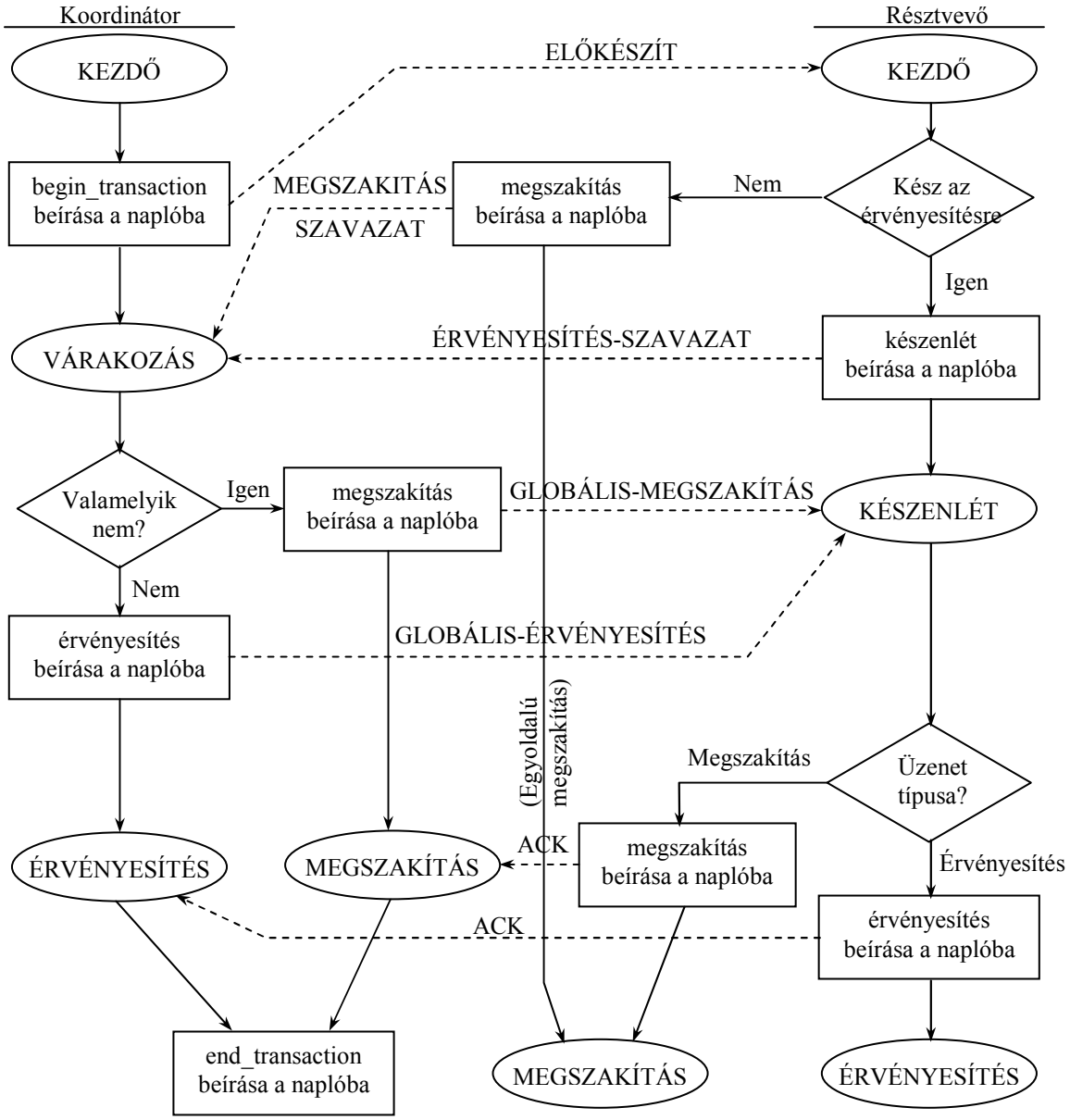
Egy megoldás lenne, összekapcsolni a zárolási kérést a módosítási művelettel, de így is sokkal több, mint a központosított rendszerben.

A kétfázisú véglegesítés (angolul: two-phase commit – 2PC) (lásd 12.8 ábra) egy nagyon egyszerű és elegáns protokoll, mely biztosítja az osztott tranzakció atomi véglegesítését. Kiterjeszti a lokális atomi véglegesítési műveletet osztott tranzakciókra, ragaszkodva ahhoz, hogy az osztott tranzakciókban résztvevő összes csomópont egyezzen bele a tranzakció véglegesítésébe, mielőtt annak hatásai maradandókká válnának.

Több ok van arra vonatkozóan, hogy miért szükséges a csomópontok közötti szinkronizálás. Az első, az alkalmazott konkurenciavezérlő algoritmustól függően, egyes ütemezők lehet, hogy

nincsenek készen a tranzakció befejezésére. Például, ha egy tranzakció egy olyan adatnak az értékét olvassa, melyet egy másik még nem véglegesített tranzakció módosított, az ütemező lehet, hogy még nem akarja az előzőt véglegesíteni.

Egy másik lehetséges ok, ami miatt egy résztvevő nem egyezhet bele a véglegesítésbe, a holtpont, mely megköveteli a résztvevőtől, hogy a tranzakciót megszakítsa. A résztvevőnek meg lehet engedve, hogy megszakítsa a tranzakciót. Ez a képesség meglehetősen fontos és *egyoldalú megszakítás*nak nevezzük.



12.8. ábra: Kétfázisú véglegesítés

A 2PC protokoll, koordinátor és résztvevő közötti műveletei, hiba hiányában a 12.8. ábrán láthatók. Az ellipszisek jelölik az állapotokat, a szaggatott vonalak mutatják a koordinátor és a résztvevők közötti üzeneteket. A szaggatott vonalakon levő címkék határozzák meg az üzenet természetét.

A 2PC protokoll rövid leírása, melyben nem vesszük figyelembe a hibákat a következő.

– a koordinátor beír egy begin_transaction bejegyzést a naplóba, elküldi az „ELŐKÉSZÍT” üzenetet mindegyik résztvevő csomópontnak és VÁRAKOZÁS állapotba kerül.

– Amikor egy résztvevő megkapja az „ELŐKÉSZÍT” üzenetet, megvizsgálja, hogy, tudja-e véglegesíteni a tranzakciót.
  - Ha igen, beír egy készenlét bejegyzést a naplóba, elküldi az „ VÉGLEGESÍTÉS-SZAVAZAT” üzenetet a koordinátornak, majd KÉSZENLÉT állapotba kerül;
  - Különben, egy megszakítás bejegyzést ír a naplóba és a „ MEGSZAKITÁS-SZAVAZAT”” üzenetet küldi el a koordinátornak. Ha a csomópont megszakítás döntést hozott, abba is hagyhatja a tranzakciót, mert egy megszakítás határozatnak döntő szerepe van (például az egyoldalú megszakítás).
– Miután a koordinátor minden résztvevőtől kapott választ, eldönti, hogy véglegesítse vagy megszakítsa a tranzakciót.
  - Ha legalább egy résztvevőtől negatív szavazatot kapott, a koordinátor meg kell szakítsa a tranzakciót globálisan. Így beír egy megszakítás bejegyzést a naplóba, elküldi a „GLOBÁLIS-MEGSZAKÍTÁS” üzenetet az összes résztvevőnek és MEGSZAKÍTÁS állapotba kerül;
  - Különben egy véglegesítés bejegyzést ír a naplóba, a „GLOBÁLIS-VÉGLEGESÍTÉS” üzenetet küldi el az összes résztvevőnek és VÉGLEGESÍTÉS állapotba kerül.
– A résztvevők a koordinátor utasításainak megfelelően vagy véglegesítik vagy megszakítják a tranzakciót és visszaküldenek egy visszaigazolást, melyre a koordinátor befejezi a tranzakciót beírva egy end_transaction bejegyzést a naplóba.

Megjegyzendő a mód, ahogy a koordinátor elér egy tranzakcióra vonatkozó globális befejezési döntéshez. Ezt a döntést két szabály irányítja, melyeket együttesen globális véglegesítési szabálynak nevezünk:

1) Ha legalább egy résztvevő a tranzakció megszakítására szavaz, a koordinátor egy globális megszakítás döntést kell hozzon.
2) Ha az összes résztvevő a tranzakció véglegesítésére szavaz, a koordinátor globális véglegesítési döntést kell hozzon.

Az ábrán egy pár fontos dolgot figyelhetünk meg. A 2PC megengedi, hogy egy résztvevő egyoldalúan megszakítson egy tranzakciót, mielőtt elhatározná, hogy igenlő szavazatot ad. Ha egy résztvevő azt szavazta, hogy véglegesíti a tranzakciót, utólag a szavazatát nem változtathatja meg. Amikor egy résztvevő a KÉSZENLÉT állapotban van, a tranzakciót vagy megszakíthatja vagy véglegesítheti a koordinátortól kapott üzenet természetének függvényében. A globális befejezési döntés a koordinátortól származik, a globális véglegesítési szabálynak megfelelően. A koordinátor és a résztvevő folyamatok biztos állapotokba kerülnek, melyben várakozniuk kell egymás válaszaira.

Amikor a frissítés véglegessé válik a frissítést végző gépen, akkor azt hálózatszintű véglegesítésnek kell követnie, amelynek során az összes gép megkísérli véglegesíteni a változást. Az adatnak mindaddig lezárva kell maradnia, amíg minden véglegesítés sikeres lesz. Ha valamelyik gépen a frissítés nem sikerül, akkor egy hálózatszintű visszagörgetést kell kibocsátani, hogy visszaállítsuk a változtatásokat ott, ahol az már véglegessé vált.

Részben enyhíti az elérhetőség problémáját az "elsődleges másolat" stratégiája. Ebben a helyzetben minden másolt (replikált) adatbázis-objektumhoz kijelölünk egy gépet, amely az objektum elsődleges másolatát tárolja. Amikor rendszerszintű frissítés történik, az elsődleges másolatot kezelő gép bocsátja ki a rendszerszintű lezárásokat. Amint az elsődleges másolat gépe véglegesíti a frissítést, a műveletet befejezettnek tekintjük.

Az elsődleges másolat stratégiája tovább javítható a kétfázisú véglegesítési protokollal. A kétfázisú véglegesítési protokollban az egyik gép vállalja a felelősséget a véglegesítések összehangolásáért, az elsődleges másolatot használó rendszerben ez az elsődleges másolatot kezelő gép lesz és a két fázis a következő:

Első fázis: A koordináló gép (elsődleges másoltatot tároló) elküldi a frissítési információt és egy "Kész a véglegesítésre?" kérdést az adatbázis-objektum másolatait tároló gépeknek. Mindegyik gép ellenőrzi, hogy tud-e véglegesíteni, és egy IGEN vagy NEM választ küld vissza.

Második fázis: Ha bármelyik gép a NEM választ küldi vissza, akkor a koordináló gép kibocsát egy rendszerszintű "Megszakít" üzenetet, vagyis az összes gép elfelejtheti a frissítési információt és továbbhalad. Ha minden válasz IGEN, akkor egy rendszerszintű véglegesítést

bocsátunk ki. Amint egy gép elvégezte a véglegesítést, azt befejezettnek tekintheti, és felszabadítja a helyi lezárásokat.

Az elsődleges másolat stratégiája a kétfázisú véglegesítéssel együtt nagyban növelik az adatelérhetőséget az osztott adatbázisban, viszont a helyi autonómiát megsértik, mivel kijelölnek egy gépet az adatfrissítés irányítására.

12.5.4. Katalóguskezelés

Az adatbázis-kezelő rendszerek egy adatszótárt (vagy katalógust) tartanak fenn, amely a rendszerben tárolt és kezelt adatobjektumokat írja le (például táblák szerkezete, index állományok értelmezése, helyességi megszorítások, nézetek értelmezése, felhasználókról szóló információk, azok jogosultságait, stb).

Az ANSI/SPARC modell esetén találkoztunk a *globális katalógus/szótár* (*GD/D*) és *lokális katalógus/szótár* (*LD/D*) fogalmával. Osztott környezetben minden résztvevő csomópontnak ismernie kell a rendszer összes objektumának (tábla, index, nézet, stb) értelmezését. A globális katalógus, tartalmazza a rendszerben található összes objektum leírását, melyre minden csomópontnak szüksége van. Egy adott csomópontnak van lokális katalógusa, mely az illető csomópontban tárolt objektumok leírását tartalmazza. Különböző megközelítések léteznek az osztott adatbázis-kezelő katalógusainak tárolására:

Teljesen szétszórt: E megoldás esetén minden csomópontja az osztott adatbázisnak csak azon objektumok leírását tartalmazza, amelyeket az illető csomópont tárol lokálisan. Tehát léteznek a lokális katalógusok, a globális katalógus csak virtuális, nincs tárolva, a lokális katalógusok egyesítéséből kapjuk. Ezzel a megközelítéssel az a probléma, hogy minden alkalommal, amikor nem helyi adathoz kell hozzáférni, meg kell vizsgálni minden más csomópont lokális katalógusát, amíg a keresett információt megtaláljuk. Ez súlyos teljesítmény problémához vezethet, amikor távoli adat elérésére van szükség.

Teljesen másolt: Ebben az esetben minden gép rendelkezik a globális katalógus egy teljes másolatával, ami felgyorsítja a távoli adatelérést. Azt jelenti azonban, hogy minden alkalommal, amikor valamelyik gépen egy új adatobjektum (tábla, index, nézet, stb.) keletkezik, vagy éppen megsemmisül, akkor a katalógus változását tovább kell küldeni a rendszer valamennyi katalógusának.

Központosított: E megoldás esetén a lokális katalógusokon kívül létezik egy „központi” csomópontban egy globális katalógus a rendszer összes objektumának a leírásával. A távoli adat elérésekor az osztott adatbázis-kezelő először a „központi” csomópont globális katalógusában keresi az objektumot, ahol annak a helye is megtalálható, majd magát az objektumot beolvassa. A lokális katalógusokban történő változásoknak el kell jutniuk a központi katalógusba. Ez tulajdonképpen a két előző megközelítés közötti kompromisszum, mindkettővel szemben vannak előnyei, viszont aláássa a központi csomóponttól való függetlenség elvét. Ha a „központi” csomópont meghibásodna, akkor lehetetlenné válna a hozzáférés az összes távoli adathoz.

Állandó azonosítók. Ezt az elrendezést elvileg számos rendszer használja, a részletek azonban különböznek. Új adatobjektum létrehozása esetén az objektum kap egy logikai azonosítót, ami az összes többi gép adatszótárába bekerül. Ez a logikai azonosító azonosítja a gépet, ahol megszületett az objektum. Az objektum "születési hely" katalógus bejegyzése nyomon követi a tulajdonképpeni tárolási helyet. Így ha egy objektum, mondjuk egy reláció a születési helyéről, átkerül egy másik gépre, a születési helyén lévő katalógus bejegyzése megváltozik az új tárolási helyet rögzítve. Ezen kívül hozzáadódik az új tárolóhely helyi katalógusához. Bármely gép, amelynek szüksége van az objektumra, először a születési helyen lévő katalógust vizsgálja meg, és az ott tárolt információt használja magának az objektumnak a megtalálására. Ha az objektumot ismét el kellene mozdítanunk, akkor az elhagyott gép helyi adatszótárból egyszerűen törölnénk és hozzáadnánk az új gép adatszótárához, valamint megváltoztatnánk a születési helyen lévő katalógus bejegyzését az új helyet rögzítve. Bármely olyan gépnek, amelynek az objektumra van szüksége, csak két gép katalógusához kell hozzáférnie: a születési helyen, vagyis azon a gépen, ahol létrejött és a jelenlegi tárolóhelyen. Ez azt jelenti, hogy az objektumok csak akkor válnak megtalálhatatlanná, ha a születési hely gépe meghibásodik.

12.5.5. Osztott lekérdezések feldolgozása

Osztott adatbázisrendszerben problémát jelenthet, ahogy már említettük az előző fejezetben, azon lekérdezések feldolgozása, amelyek különböző csomópontokban tárolt adatok vizsgálatát teszik szükségessé.

12.4. példa: Legyen a NagyKer adatbázisból Alkalmazottak és Részlegek táblák:

```
Részlegek (RészlegID, Név, Helység, ManSzemSzám);
Alkalmazottak (SzemSzám, Név, Fizetés, RészlegID).
```

Az Alkalmazottak tábla vízszintesen van fragmentálva, ahogy a 12.1 példában láttuk. A Részlegek táblát teljes egészében a "Kolozsvar" nevű csomópontban tároljuk. Az egyes csomópontokban tárolt adatok mennyisége a következő:

"Kolozsvár" csomópont:

Részlegek: 10 rekord, mindegyik 50 bájt hosszú: 500 bájt
Attribútumok: RészlegID (2 bájt), Név (20 bájt), Helység (20 bájt), ManSzemSzám (4 bájt)

KolAlk: 2000 rekord, mindegyik 100 bájt hosszú: 200.000 bájt
Attribútumok: SzemSzám (10 bájt), Név (30 bájt), Fizetés (4 bájt), Cím (50 bájt), RészlegID (2 bájt)
Minden Alkalmazott sor esetén a RészlegID értéke 1 vagy 2.

"Nagyvarad" csomópont:

NavAlk: 500 rekord, mindegyik 100 bájt hosszú: 50.000 bájt
Minden Alkalmazott sor RészlegID értéke 9.

Tegyük fel, hogy a kolozsvári csomóponton a következő lekérdezést adták ki:

```
SELECT SzemSzám, Alkalmazottak.Név, Részlegek.Név
FROM Alkalmazottak, Részlegek
WHERE Alkalmazottak.RészlegID = Részlegek.RészlegID
 AND RészlegID = 9;
```

A lekérdezés eredménye 500 sort fog tartalmazni, épp annyit ahány a 9-es részlegben található a nagváradi csomópontban. Egy sor hossza: SzemSzám (10 bájt), Alkalmazottak.Név (30 bájt), Részlegek.Név (20 bájt), összesen 60 bájt. Tehát az eredmény 30.000 bájtot tartalmaz.

A lekérdezés feldolgozása esetén a lekérdezést relációs algebrai műveletek sorozatává alakítja a rendszer, mely a következő lesz:

$$\text{Eredm} = \pi_{\text{SzemSzám, Alkalmazottak.Név, Részleg.Név}}(\sigma_{\text{RészlegID} = 9}(\text{Alkalmazottak} \bowtie \text{Részlegek}))$$

A globális Alkalmazottak relációt a lekérdezés feldolgozó helyettesíti a következő képlettel:

```
Alkalmazottak = KolAlk ∪ NavAlk.
```

Az így kapott relációs algebrai műveletek sorozatából álló lekérdezés végrehajtására több stratégiát is ki lehet dolgozni. A központosított lekérdezés feldolgozáshoz képest, itt a kommunikációs költségeket is figyelembe kell venni, sőt ezek képezik a lekérdezés feldolgozási költségének legnagyobb részét. A lemezolvasási költségek sokkal kisebbek, mint a kommunikációs költségek. A továbbiakban csak a kommunikációs költségeket vesszük figyelembe. A számítások esetén kerekítünk ami a szállítandó adatok méretét illeti.

1-es stratégia:

a) Szállítjuk a nagyváradi csomópontból a NavAlk fragmenset a kolozsvári csomópontba, 50.000 bájt-ot kell küldenünk.

A kolozsvári csomópontban most már az összes szükséges adat jelen van, végrehajtjuk a lekérdezést.

Szállítandó adatmennyiség: **50.000** bájt.

2-es stratégia:

a) Szállítjuk a kolozsvári csomópontból a KolAlk fragmenset a nagyváradi csomópontba, 200.000 bájt-ot kell küldenünk.

b) Szállítjuk a kolozsvári csomópontból a Részlegek táblát a nagyváradi csomópontba, 500 bájt-ot kell küldenünk.

A nagyváradi csomópontban most már az összes szükséges adat jelen van, végrehajtjuk a lekérdezést.

Eredmény relációt vissza kell küldenünk a kolozsvári csomópontba, 30.000 bájt.

Szállítandó adatmennyiség: **230.500** bájt.

3-as stratégia:

a) Átírjuk a lekérdezést, felhasználva az 5. fejezetben ismertetett tulajdonságait a relációs algebrai műveleteknek:

$$\text{Eredm} = \pi_{\text{SzemSzám, Alkalmazottak.Név, Részleg.Név}}(\sigma_{\text{RészlegID = 9}}(\text{KolAlk} \cup \text{NavAlk}) \bowtie \text{Részlegek})) =$$

$$\pi_{\text{SzemSzám, Alkalmazottak.Név, Részleg.Név}}((\sigma_{\text{RészlegID = 9}}(\text{KolAlk})) \cup (\sigma_{\text{RészlegID = 9}}(\text{NavAlk})) \bowtie \text{Részlegek}) =$$

(mivel a kolozsvári csomópontban nincs 9-es részlegnél dolgozó alkalmazott, a tördelési feltételben RészlegID = 1 OR RészlegID = 2, következik:

$$\sigma_{\text{RészlegID = 9}}(\text{KolAlk}) = \varnothing)$$

$$= \pi_{\text{SzemSzám, Alkalmazottak.Név, Részleg.Név}}((\sigma_{\text{RészlegID = 9}}(\text{NavAlk})) \bowtie \pi_{\text{Részleg.RészlegID, Részleg.Név}}(\text{Részlegek})$$

(Azért, hogy kevesebb információt szállítsunk a Részlegek táblából az az, csak annyit ami szükséges, a függőleges vetítés műveletét alkalmaztuk először a Részlegek táblára. Csak a Név szerepel az eredményben választott attribútumok között, viszont az összekapcsoláshoz szükségünk van a RészlegID-ra is, ezért kerül az is a a függőleges vetítés attribútumai közé)

b) Szállítjuk a Részlegek tábla RészlegID (2 bájt) és Név (20 bájt) attribútumát, összesen 250 bájt a kolozsvári csomópontból a nagyváradi csomópontba.

A nagyváradi csomópontban végrehajtjuk a lekérdezést.

Az eredmény relációt vissza kell küldenünk a kolozsvári csomópontba, (30.000 bájt).

Szállítandó adatmennyiség: **30.250** bájt.

4-es stratégia:

Hasonlóan a 3-as stratégiához, átírjuk az eredeti lekérdezést, tovább folytatjuk az átírást:

$$\text{Eredm} = \pi_{\text{SzemSzám, Alkalmazottak.Név, Részleg.Név}}((\pi_{\text{SzemSzám, Alkalmazottak.Név, RészlegID}}(\sigma_{\text{RészlegID=9}}(\text{NavAlk}))) \bowtie$$

$$\bowtie \pi_{\text{Részleg.RészlegID, Részleg.Név}}(\text{Részlegek}))$$

a) A nagyváradi csomópontban végrehajtjuk a következő lekérdezést:

$$\pi_{\text{SzemSzám, Alkalmazottak.Név, RészlegID}}(\sigma_{\text{RészlegID=9}}(\text{NavAlk}))$$

Az eredmény 500 sort fog tartalmazni, mivel az összes alkalmazott a nagyváradi csomópontban a 9-es részleghez tartozik, ez volt a tördelési feltétel: RészlegID = 9. Csak azokat az oszlopokat válogatjuk ki, melyek szükségesek az eredményhez: SzemSzám (10 bájt), Név (30 bájt), RészlegID (2 bájt). Szállítanunk 21.000 bájtot kell.

b) A kolozsvári csomópontban végrehajtjuk a lekérdezést, itt kell az eredményt a felhasználónak szolgáltassuk, tehát további szállításra nincs szükségünk.

Szállítandó adatmennyiség: **21.000** bájt. □

Osztott rendszerben a költségek nagy részét az adatok átvitele képezi. A fenti stratégiáknál az adatok átvitelének mennyisége 21.000-től 230.500 bájtig változik. A különbségek még

drámaibbak lennének, ha csak az adatok egy kis része elégítené ki a vízszintes kiválasztás feltételét. Például a feltételt kiegészítjük: Fizetés > 500. Tegyük fel, hogy csak az alkalmazottak 10 százalékának nagyobb a fizetése 500-ál. Ez azt jelenti, hogy a 4-es stratégia esetén 2.100 bájtnyi adatmennyiséget kell csak szállítanunk, a többi stratégia költségei is változnak, de a különbség még nagyobb lesz.

Egy osztott rendszer statisztikai bejegyzéseket tart fenn az egyes gépeken tárolt adathalmazok méretéről, hogy a lekérdezések végrehajtására a legjobb stratégiát meg tudja állapítani.

A lekérdezés feldolgozása a fragmentációt és az adatmásolását is ki kell tudja használni. A 12.3 példát figyelembe véve, a nagyváradi csomópontban levő NavAlk fragmensnek van másolata a kolozsvári csomóban KNavAlk. Ha a lekérdezés feldolgozó figyelembe veszi a másolatok létezését, használhatja a KnavAlk másolatot és adatátvitelre nincs szükség egyáltalán.

A fenti stratégiák feltételezik, hogy a lekérdezés processzor ismeri az adatbázis globális szerkezetét, hozzáfér a globális katalógushoz (GD/D). Ha a lekérdezés feldolgozó nincs birtokában ezeknek az információknak, akkor nem tehet mást, mint hogy megvizsgálja az összes gép lokális katalógusát, hogy megtalálja a szükséges adatokat.

Amikor több gépen lévő adatot kapcsolunk össze, az adatátviteli költség csökkentésére szolgáló egyik stratégia a félig-összekapcsolás (semijoin). További részleteket láss a [ÖzVa91]–ben.

A szerző eredményei az osztott adatbázisok lekérdezésének az optimalizálásában a [VaKa96], [Va97], [Va98], [Va99], [VaMo99], [VaMaMo01] és [VaDuGr04] dolgozataiban olvashatóak.

13. Objektumorientált adatbázisok

Az objektumorientált programozás segítségével, könnyebben, természetesebben modellezhetjük a valós világot, jobban szervezhetjük az adatainkat. Az adatbázisok világa is elmozdult az objektumorientáltság felé, lásd [LeRiVe88], [AtBaDe90], [BeMa93], [BeNePe92], [ChSt93], [LaStWa97].

Két irányban történt a fejlesztés (lásd Ramakrishnan 1998):

a) Objektumorientált adatbázisok (OOABKR), melyeket úgy fejlesztettek, hogy egy objektumorientált programozási nyelvet kibővítettek az adatbázis-kezelő rendszerek képességeivel:

- Adatok permamens tárolása. A hagyományos programozási nyelvekben a permamens adattárolás több ponton is eltér a memóriában tárolt változók kezelésétől, s egy sor extra programozási feladatot kíván meg a programozótól. Az ABKR ezzel szemben egy egyszerűbben kezelhető és gyorsabban programozható tárolási mechanizmust biztosít az adatok hosszú idejű tárolására.
- Függetlenségi szintek. Az ABKR-ek jellemzője, hogy a felhasználónak nem kell ismernie az adatok fizikai tárolási mechanizmusát, formátumát. A felhasználó egy emberközelibb kezelő nyelv segítségével kommunikálhat az ABKR-rel, amely majd lefordítja a kapott magas szintő utasítást a végrehajtáshoz szükséges fizikai szintre. E függetlenségi szint gyorsabb és kényelmesebb alkalmazás fejlesztést biztosít (lásd relációs adatmodell esetén az SQL nyelvet).
- Konkurrens adatkezelés biztosítása, adatvédelem, helyreállítás hiba esetén. Az ABKR egyidejűleg több párhuzamos kérés kiszolgálására alkalmas mechanizmussal rendelkezik. A hagyományos programfejlesztési eszközökkel a konkurrens igények kielégítése csak nehezen és részlegesen oldható csak meg. Az ABKR szolgáltatásai közé tartozik a hozzáférések jogosultságának ellenőrzése, az adatok sérülés elleni védelme is. (lásd Tranzakciókezelés fejezet).
- Ad-hoc lekérdező nyelv. Az ABKR-ek fontos eleme egy rugalmas lekérdezõ nyelv, amellyel egyszerűen és hatékonyan lehet a felhasználóhoz közel álló formátumban információt kinyerni a rendszerből. E nyelv lényeges eleme, hogy hozzáférést biztosít az adatbázis minden olyan eleméhez, melynek elérését a védelmi szabályok nem tilják. E rugalmasságra azért is szükség van, mert az adatbázisok tervezésének pillanatában még nem lehet biztosan tudni, hogy a távolabbi jövőben milyen igények merülnek majd föl, illetve a felhasználói igények igen széles skálán mozoghatnak, amiket szinte lehetetlen átfogni és előre megtervezni.

Ez a megoldás rendszerint egy speciális, saját könyvtár létrehozását jelenti, amely a permanens adatkezeléshez szükséges adattárolási funkciókat valósítják meg. Ekkor a fejlesztés menete megegyezik a normál programok készítésének menetével, azzal a különbséggel, hogy az adatok tárolását egy csatolt függvény gyűjtemény felhasználásával oldják meg. E rendszerekben sokszor hiányzik a védelem, a tranzakciókezelés, az optimalizálás és a rugalmas lekérdezõ nyelv is.

b) Objektum-relációs rendszerek (ORABKR), melyeket úgy fejlesztettek, hogy létező relációs adatbázis-kezelő rendszereket kibővítették az objektumorientált fogalmakkal:
   - Komplex típusokkal
   - Osztály fogalmával
   - Örökléssel

   Ezen fogalmak jellemzők úgy az objektumorientált, mint az objektum-relációs rendszerekre is, a továbbiakban részletezni fogjuk (lásd 13.1)

   Ennek a megoldásnak az előnye, hogy az ABKR szolgáltatások már rendelkezésre állnak, s a módosításnál, az új elemek bevonásánál is támaszkodhatnak az ABKR fejlesztők a korábbi szolgáltatásokra. Részletes leírását az objektum-relációs adatbázisoknak a [St96]-ban találunk.

Az első “tiszta” objektumorientált adatbázis-kezelő rendszerek (OOABKR) a 80-as évek végén jelentek meg. Ma már sok OOABKR van a piacon, de nem olyan sikeresek, mint várható lett volna.

A ma létezõ OOABKR pozitív tulajdonságai:

– igen népszerű, nagyon sok helyen foglalkoznak objektumorientált adatbázis rendszer fejlesztésével;
– nagyon sok egymástól független, párhuzamosan dolgozó műhely alakult ki az objektumorientált adatbázis-kezelő rendszerek kidolgozására és fejlesztésére;
– biztosítja az objektumorientált elvek megvalósítását;
– szoros kapcsolódást biztosít az egyéb objektumorientált fejlesztő eszközökhöz, fejlesztő nyelvekhez.

Az OOABKR gyenge pontjai:
– az objektumorientált adatmodell nem rendelkezik olyan elméleti megalapozottsággal, mint a relációs modell.
– az első OOABKR-ek teljesítményben, megbízhatóságban lényegesen alacsonyabb szinten álltak, mint a relációs ABKR rendszerek. Emiatt a felhasználók sem választották ezt a megoldást a kritikus feladatok megoldására. A piacon jelenleg is létező objektumorientált alapokat biztosító adatbázis-kezelő rendszerek igen különböző szolgáltatásokat nyújtanak és igen eltérő belső struktúrával rendelkeznek, melyekben az objektumorientáltság elvei is igen különbözőképpen valósulnak meg.

Az OOABKR-ek kialakulását követve megfigyelhető, hogy az objektumorientált adatmodellel szemben érezni bizonyos ellenállást. Megemlítjük Chris Date véleményét (lásd [Da04]) ezzel kapcsolatban. Date, mint a relációs modell támogatója, igen kritikusan vélekedik az objektumorientált modellről. Az általa megemlített lényegesebb hiányosságok:
– Az OOABKR-ek kezelő nyelve a fejlesztő eszközökhöz (C++) való simább illeszkedés miatt egy rekordorientált megközelítést mutat, amely egy lényeges visszalépést jelent a relációs modellben megszokott halmazorientált kezelési felülethez képest.
– Az OOABKR-ek több hiányossággal is rendelkeznek az adatok helyességének ellenőrzésére vonatkozólag. Így például nincs megfelelő mechanizmus a kapcsolatok számosságának az ellenőrzésére, s nem lehet deklaratív integritási feltételeket sem megadni. A metódusok segítségével kell ezeket a problémákat megoldani az objektumorientált adatbázisok esetén.
– A lekérdező nyelvek az OOABKR-ek esetén, melyek az SQL SELECT utasításhoz hasonló szolgáltatásokat tudnának nyújtani, még fejlesztés alatt vannak. Egy érdekes probléma ebből a szempontból, hogy mennyire illeszthető össze az általános lekérdezés által igényelt nyíltság az objektumok zártságának az elvével.

A fenti elméletibb jellegű kifogások mellett a gyakorlati szakemberek az OOABKR-ek megvalósításának a kérdőjeleit is fel szokták hozni az értékeléskor. Az olyan gyakorlati kételyek, mint hogy
– megfelelő lesz-e az OOABKR hatékonysága;
– tud-e majd a rendszer nagy adatmennyiséget is kezelni;
– mennyire megbízhatóak az OOABKR-t fejlesztő cégek;
– mennyire szabványosak az egyes OOABKR-ek.

A relációs kontra objektumorientált adatmodell vitában mára már nem az fő kérdés, hogy kell-e objektumokat tárolni az adatbázisokban, mivel ennek szükségességében mindenki egyetért, hanem azon folyik a vita, hogyan célszerű-e megvalósítani az objektumok tárolását. Itt két fő tábor létezik, az egyik szerint a tiszta OOABKR-t, a mások szerint az ORABKR-t (object relational DBMS) célszerű támogatni.

A továbbiakban mindkét tábor által elfogadott fogalmakat bemutatjuk.

13.1. A típusrendszer

Az objektumorientált nyelvek a típusok széles választékát kínálják. Az *alaptípusok* az egészek, valós számok, logikai értékek és karakterláncok. A típus-konstruktorok segítségével lehetőségünk van az alaptípusokból újabb típusok létrehozására. Segítségükkel például a következő összetett típusok hozhatók létre:

- *Rekordstruktúrák*. Ha adottak a T_1, T_1, … T_n típusok és az m_1, m_2,…, m_n mezőnevek, ahol az m_i mező T_i típusú, létrehozzuk a
  RECORDOF (m_1, m_2,…, m_n)
  típust, mely n komponensből álló rekord és típusa: [m_1:T_1, m_2 : T_2,…, m_n : T_n]
- *Kollekciótípusok*: Legyen T egy típus, új típusokat hozhatunk létre *kollekcióoperátor* alkalmazásával. Kollekcióoperátorok a tömbök, listák és halmazok.
- *Hivatkozástípusok*: Egy T típusra való hivatkozás egy olyan típus, amelynek értékei segítségével megkereshetünk egy T típusú értéket. C++-ban a hivatkozás egy mutató, amelyben a mutatott érték memóriabeli címét tároljuk. A két fogalom hasonló, az adatbázisrendszerekben viszont, ahol az adatok jó néhány lemezen, sokszor különböző számítógépeken helyezkednek el a hivatkozás sokkal bonyolultabb dolog, mint a mutató. A hivatkozás tartalmazhatja a számítógép nevét, a lemezegység számát, azon belül egy blokk azonosítóját és a blokkban egy pozíciót, ahol a keresett érték tárolva van.

A rekordstrukturát és kollekcióoperátorokat egymásba ágyazottan többször is alkalmazhatjuk, így összetett típusokat hozhatunk létre.

Osztályok és objektumok

Egy osztály struktúrálisan típusból, és opcionálisan egy vagy több metódusból (függvény vagy eljárás) áll. Az osztály, mint adathalmaz objektumokból áll. A metódusokat alkalmazhatjuk az osztály objektumain, s az osztály minden objektuma azonos típusú, mely típus az osztály struktúrálisan definiált típusával egyezik. Ezt a típust szokás az osztály nevével illetni.

Objektumazonosító

Minden objektumnak van egy objektumazonosítója (OID). Ez egyedi minden objektumra nézve. Úgy tekinthetjük az objektumazonosítót, mint egy memóriabeli mutató, amelyik az objektumra mutat. Az azonosító érvényes kell legyen mindaddig amíg az objektum létezik.

Metódusok

Az osztályhoz rendelt függvényeket vagy eljárásokat metódusoknak nevezzük. Például egy *C* osztályhoz tartozó metódusnak van legalább egy argumentuma, egy *C*-beli objektum.

Absztrakt adattípusok

Ha egy osztály megakadályozza az objektumaihoz való tetszőleges hozzáférést, azaz csak az osztály metódusain keresztül biztosít hozzáférést, akkor absztrakt adattípusról beszélünk. Csak az osztály metódusai módosíthatják közvetlenül az objektumokat. Így az csak olyan módon lehet módosítani, ahogy az megvolt tervezve.

Osztályhierarchiák

Egy osztályból származtathatunk alosztályokat. Az osztályt amiből származtattuk az alosztályt szuperosztálynak nevezzük. Az alosztály örökli a szuperosztály összes tulajdonságát és metódusait, és deklarálhatunk plusz tulajdonságokat is. Az alosztályoknak is lehet alosztálya, egy alosztályt származtathatunk több szuperosztályból, így az osztályok egy hierarchiájához jutunk.

13.2. Tervezés ODL (Object Definition Language) segítségével

Az ODL egy javasolt szabvány, segítségével adatbázisok struktúráját specifikálhatjuk objektumorientált terminológiával. Elsődleges célja, hogy támogassa az adatbázisok

objektumorientált tervezését, és utána ennek a transzformálását OOABKR-ek deklarációiba. Az ODL nem ad lehetőséget az adatbázis tartalmának megadására, az adatok manipulálására, lekérdezésekre. Az ODL egy adatdefiníciós nyelv, ugyanúgy, ahogy az SQL-ben a DDL.

13.2.1. Objektumorientált tervezés

Objektumorientált tervezésnél a valós világot objektumok segítségével modellezzük. Tekinthetjük például objektumoknak a személyeket, épületeket, kurzusokat, stb. Az objektumokról feltételezzük, hogy rendelkeznek egy sajátos azonosítóval (OID), ennek segítségével különböztetjük meg az objektumokat egymástól.

Az objektumokat osztályokba csoportosíthatjuk, azzal a megkötéssel, hogy egy osztályon belül minden objektum azonos típusú, s ugyanazok a metódusok vonatkoznak rájuk, mint amelyek az osztály definiálásánál adottak (például az SQL nyelv DDL nyelvének megfelelő ODL nyelven).

Az objektumok gyakran rekordoknak tekinthetők, ha sortípusúak, s a soron belül nincs más objektumtípusra hivatkozás (ekkor lehet hierarhikus vagy hálós struktúrájú az objektum). Ekkor beszélhetünk mezőkről, mezők értékeiről, melyek természetesen a sortípusnál definiált típusú értékek, továbbá rájuk is vonatkoznak a megfelelő metódusok.

Amikor specifikálunk az ODL-ben egy osztályt, a következő három jellemzőjét adjuk meg:

- Attribútumok: ezek az objektum tulajdonságai, s típussal rendelkeznek.
- Kapcsolatok: Típusa valamilyen osztály egy objektumára való hivatkozás, vagy ilyen hivatkozások gyűjteménye.
- Metódusok: ezek függvények, alkalmazhatjuk őket az osztály objektumaira.

13.2.2. Osztály deklarációja

Az ODL-ben az osztály deklarációja a következőképpen néz ki:

```
interface  <név> {
     < jellemzők listája >
}
```

Az interface kulcsszót az osztály neve követi, majd kapcsos zárójelek között megadjuk a jellemzők listáját. A jellemzők lehetnek attribútumok, kapcsolatok és metódusok.

13.2.3. Attribútumok az ODL-ben

Az attribútumok a jellemzők legegyszerűbb fajtái. Az objektumhoz adott típusú értéket rendelnek. Például a Diák osztálynak lehet egy attribútuma a név, melynek típusa karakterlánc, értéke a diák neve. Egy másik attribútuma lehet a lakcím mely sor típusú, azaz karakterláncokból álló kettes, amely reprezentálja a várost és az utcát. Az attribútum deklarálásakor az *attribute* kulcsszót használjuk. Például deklaráljuk a Diák osztályt:

```
interface Diák {
     attribute integer beiktatásiSzám;
     attribute string név;
     attribute struct cím
          {string város, string utca} lakcím;
};
```

A lakcím attribútum típusa sortípus, mely egy rekordstruktúra. Ennek a típusnak a neve cím és két karakterlánc típusú mezője van: a város és az utca. ODL-ben a struct kulcsszót használjuk a rekordstruktúra deklarációjára.

13.2.4. Kapcsolatok ODL-ben

Időnként szükségünk van megtudni, hogy az objektum hogyan kapcsolódik ugyanannak vagy egy másik osztály objektumához. Például szeretnénk a `Diák` osztályba felvenni egy jellemzőt,

ami a tantárgyak egy halmaza. A `Diák` osztály objektumainak nincsenek közös jellemvonásai a tantárgyakkal, ezért értelmezünk egy `Tantárgy` osztályt:

```
interface Tantárgy {
     attribute string név;
     attribute string leírás;
     attribute integer kreditszám;
};
```

Mivel a tantárgyak is egy osztályt alkotnak, mégpedig a `Tantárgy` nevű osztályt, ez az információ nem lehet egy attribútum, ugyanis ennek a típusa nem lehet egy osztály. Így a diák tantárgyainak a halmaza egy kapcsolat lesz. Ezt a következőképpen adhatjuk meg a `Diák` osztály deklarációjában:

```
relationship set<Tantárgy> tantárgyai;
```

Ez a kapcsolat azt jelenti, hogy a Diák osztály minden objektumának van egy hivatkozása a Tantárgy osztály objektumainak a halmazára. Fizikailag úgy fogható fel a tantárgy halmaz, mint mutatók egy listája, és mindegyik mutató egy Tantárgy objektumra mutat.

Lehet olyan kapcsolatot is deklarálni, amely egy objektumhoz csak egy objektumot rendel hozzá. Az előző példában a set kulcsszavat elhagyva minden Diák osztályba tartozó objektumhoz egy Tantárgy osztályba tartozó objektumot rendel hozzá. Ez ebben a megközelítésben nem helyes, mert egy diák több tantárgyat tanul, de ha a kedvenc tantárgyát szeretnénk eltárolni akkor helyes:

```
relationship Tantárgy kedvenctantárgy;
```

13.2.5. Inverz kapcsolatok

Hozzákapcsoltuk a tantárgyakat a diákokhoz, de most szeretnénk tudni, hogy a tantárgyakat milyen diákok tanulják. Ezt az információt a `Tantárgy` osztályban a következő sor tartalmazza:

```
relationship set<Diák> tanulja;
```

Azonban ez a sor és a hozzá hasonló deklaráció a Diák osztályban nem veszi figyelembe, azt, hogyha az *X* diák benne van az *Y* tantárgyhoz tartozó tanulja halmazban akkor kötelezően benne kell legyen az *Y* tantárgy a *X* diák tantárgyai halmazában. Ezt úgy tüntethetjük fel, hogy mindkét deklarációban használjuk az inverse kulcsszót és a másik kapcsolat nevét. Ha a kapcsolat egy másik osztályban van, akkor az osztály nevét is fel kell tüntetni, utána dupla kettőspont, majd a kapcsolat neve. Ezek után a Diák osztály a következőképpen néz ki:

```
interface Diák {
     attribute integer beiktatásiSzám;
     attribute string név;
     attribute struct cím
          {string város, string utca} lakcím;
     relationship set<Tantárgy> tantárgyai
               inverse Tantárgy::tanulja;
};
```

Az ODL megköveteli, hogy a kapcsolatoknak legyen inverze.

13.2.6. Kapcsolattípusok

Egy objektumhoz kapcsolhatunk egy objektumot. Az egyed/kapcsolat adatmodell esetén megismert egy-az egy-hez (1:1), egy a sokhoz (1:*n*) és sok a sokhoz (*n*:*m*) típusú kapcsolatok az ODL esetén is érvényesek. ODL-ben azzal specifikáljuk a sokat, hogy használunk kollekció típusú operátorokat, mint a set.

Például nézzük a következő osztályok közti kapcsolatokat:

```
interface Diák {
     attribute integer beiktatásiSzám;
     attribute string név;
     attribute struct cím
          {string város, string utca} lakcím;
relationship set<Tantárgy> tantárgyai
     inverse Tantárgy::tanulja;
relationship Tantárgy kedvenctantárgy
     inverse Tantárgy::kinekkedvence;
relationship Csoport tagja
     inverse Csoport::diákjai;
relationship Csoport felelős
     inverse Csoport::csoportfelelős;
};

interface Tantárgy {
     attribute string név;
     attribute string leírás;
     attribute integer kreditszám;
     relationship set<Diák> tanulja
          inverse Diák::tantárgyai;
     relationship set<Diák> kinekkedvence
          inverse Diak::kedvenctantárgy;
};

interface Csoport {
     attribute string csoportKód;
     relationship set<Diák> diákjai
          inverse Diák::tagja;
     relationship Diák csoportfelelős
          inverse Diák::felelős;
};
```

A fenti példában levő kapcsolattípusok:

1. *n*:*m* kapcsolat: a Diák és Tantárgy közötti tantárgyai (és annak inverze tanulja) kapcsolat, mivel egy diákhoz hozzárendeli az összes tantárgyat, melyet tanul és egy tantárgyhoz több diákot is társít, akik tanulják az illető tantárgyat. Amint láttuk, a Diák osztálynál a set operátorral adjuk azt meg, hogy egy diák több tantárgyat tanul, hasonlóan a Tantárgy osztálynál is set operátort használjuk.
2. 1:*n* kapcsolat: a Csoport és Diák osztály közötti diákjai (és annak inverze tagja) kapcsolat. Egy objektumhoz a Csoport osztályból több Diák osztályba tartozó objektumot rendelünk, viszont egy diákhoz csak egy csoportot, melynek tagja. Tehát a Csoport osztálynál a set operátorral adjuk azt meg, hogy több diák tartozik egy csoportba, a Diák osztály specifikálásában viszont nincs set, mert csak egy csoporthoz kapcsolódik. Egy másik példa 1:*n* kapcsolatra a Tantárgy és Diák közötti kinekkedvence (és annak inverze kedvenctantárgy) kapcsolat. Egy tantárgy több diáknak is kedvence, ezért a Tantárgy esetén használjuk a set operátort.
3. 1:1 kapcsolat: a Csoport és Diák osztály közötti csoportfelelős (és annak inverze felelős) kapcsolat. Egy objektumhoz a Csoport osztályból egy Diák osztálybeli objektumot rendelünk, aki az adott csoport felelőse, egy diákhoz csak egy csoportot, melynek esetleg a felelőse. A diák csak egy csoportnak lehet a felelőse, és minden csoportért csak egy diák felelhet.

13.2.7. Típusok ODL-ben

Az ODL típusrendszere hasonló más programozási nyelvek típusrendszeréhez. Alaptípusokból épül fel, és ezekből összetett típusokat építhetünk. Az ODL alaptípusai a következők:

1. *Atomi típusok*: integer, float, character, string, boolean és enum. Az enum felsorolás típus.
2. *Interfész típusok*: például a Diák és Tantárgy osztályok, melyek struktúrákat reprezentáló típusok.

Típuskonstruáló utasításokkal az alaptípusok strukturált típusokban kombinálhatóak. Ezek a következők:

1. *Halmaz*. Ha *T* egy típus, akkor Set<T> jelöli azt a típust melynek értéke *T* típusú elemek halmaza.
2. *Multihalmaz*. Ha *T* egy típus, akkor Bag<T> jelöli azt a típust melynek értéke *T* típusú elemek multihalmaza. A multihalmaz megengedi, hogy egy elem többször is előforduljon benne. Például {3, 4, 3} multihalmaz.
3. *Lista*. Ha *T* egy típus akkor List<T> jelöli azt a típust melynek értéke *T* típusú elemek véges listája.
4. *Tömb*. Ha *T* egy típus és *i* egy egész szám, akkor Array<T,i> jelöli azt a típust, amelynek az értéke egy *i* dimenziójú tömb, és az elemek *T* típusúak. Például array<integer,10> egy 10 dimenziójú és elemei integer típusúak.
5. *Struktúra*. Ha T_1, T_2, ..., T_n típusok, F_1, F_2, ..., F_n mezőnevek, akkor

   `Struct N {`T_1 F_1`,` T_2 F_2`, ...,` T_n F_n`}`

   jelöli azt az *N* nevű típust, amely egy *n* mezőből álló struktúra.

Az első négy típust *kollekciótípusnak* nevezzük. Egy kapcsolat típusa lehet interfész típus vagy kollekciótípus alkalmazva egy interfész típusra. Attribútum típusa atomi típus lehet vagy olyan struktúra, amely mezőinek típusa szintén típus, azaz lehet atomi vagy összetett típus. Az atomi típusra vagy a struktúrára kollekciótípust vagy struktúraképzést alkalmazhatunk.

13.2.8. Alosztályok az ODL-ben

Egy osztály rendelkezhet olyan objektumokkal, amelyeknek olyan tulajdonságaik vannak, amelyekkel az osztály más objektumai nem rendelkeznek. Ilyen esetekben hasznos az osztályt alosztályokra bontani úgy, hogy az osztályhoz tartozó attribútumok, kapcsolatok rájuk is érvényesek legyenek.

A diákok között vannak olyan diákok, akik kapnak ösztöndíjat, és vannak olyanok akik külföldről jöttek. Számukra definiálhatunk a Diák osztály egy alosztályát. ODL-ben egy *A* osztály *B* alosztályát úgy deklaráljuk, hogy a *B* deklarációjában a *B* neve után kettőspontot teszünk, majd az *A* osztály nevét. Az *A* osztály a *B* osztály szuperosztálya lesz.
Például:

```
interface ösztöndíjasdiák: Diák {
     attribute float ösztöndíj;
};
interface külföldidiák: Diák {
     attribute string ország;
};
```

Az alosztály örökli a szuperosztály összes tulajdonságát, a szuperosztály összes attribútuma, metódusa és kapcsolata automatikusan attribútuma, metódusa és kapcsolata lesz az alosztálynak is.

13.2.9. Többszörös öröklődés az ODL-ben

Mivel az alosztályoknak is lehet alosztálya, és egy osztálynak több alosztálya lehet osztályok hierarchiájához jutunk. Az is lehet, hogy egy osztálynak több szuperosztálya van. Például ha egy diák külföldi és van ösztöndíja akkor szükséges egy újabb osztály:

```
interface külföldiösztöndíjas: külföldidiák, ösztöndíjasdiák{ };
```

Így egy külföldiösztöndíjas objektumnak megvan minden tulajdonsága amivel a külföldidiák és ösztöndíjasdiák alosztályok rendelkeznek.

Előfordulhat, hogy mind a két szuperosztálynak van egy ugyanolyan nevű, de más típusú attribútuma. Ebben az esetben ennek a két osztálynak az alosztályában ennek az attribútumnak a típusa tisztázatlan. A következő módszerekkel lehet a típusát tisztázni:

1. Specifikálhatjuk, hogy a két definíció közül melyik kerüljön az alosztályba.
2. Az egyik szuperosztályban megváltoztatjuk az attribútum nevét.

3. Újra definiálhatjuk az alosztályban az attribútumot.

13.2.10. Kulcsok deklarálása ODL-ben

Egy K attribútum halmaz egy O osztály kulcsa, ha bármely két O_1 és O_2 különböző objektumok esetén a két objektum értéke nem egyezik meg a K attribútumhalmazán. ODL-ben a key kulcsszó segítségével lehet kulcsot deklarálni.

Legyen ismét a Diák osztály, deklaráljuk a beiktatásiSzám attribútumot kulcsnak:

```
interface Diák
     (extent Diákok
          key beiktatásiSzám)
{
     attribute integer beiktatásiSzám;
     attribute string név;
     attribute struct cím
          {string város, string utca} lakcím;
     relationship set<Tantárgy> tantárgyai
          inverse Tantárgy::tanulja;
     relationship Tantárgy kedvenctantárgy
          inverse Tantárgy::kinekkedvence;
     relationship Csoport tagja
          inverse Csoport::diákjai;
     relationship Csoport felelős
          inverse Csoport::csoportfelelős;
};
```

13.2.11. Osztályhoz tartozó objektumkészlet

Minden ODL osztályhoz deklarálható egy objektumkészlet az extent kulcsszó segítségével, ami az osztály aktuális objektumainak a halmazát jelöli. Egy osztály objektumkészlete megfelel a relációs adatmodell esetén a relációnak (táblának). A fenti Diák deklaráció már az objektumkészlet deklarálását is tartalmazza. Az OQL lekérdezések az objektumkészletre vonatkoznak.

13.2.12. Metódusok deklarálása ODL-ben

ODL-ben az osztály (interface) deklarálásában metódusokat is megadhatunk, melyek az adott osztály objektumaira alkalmazhatók. A metódusok paraméterei in, out vagy inout típusúak. Az in típusú paraméterek esetén érték szerint történik a paraméter átadás, míg az out és inout esetén hivatkozás szerint. A metódusok kiválthatnak kivételeket, melyek egy abnormális állapotot jeleznek, például nullával való osztás.

Egészítsük ki a Diák osztály deklarálását metódusokkal:

```
interface Diák
     (extent Diákok
          key beiktatásiSzám)
{
     attribute integer beiktatásiSzám;
     attribute string név;
     attribute string születésiDátum;
     attribute struct cím
          {string város, string utca} lakcím;
     relationship set<Tantárgy> tantárgyai
          inverse Tantárgy::tanulja;
     relationship Tantárgy kedvenctantárgy
          inverse Tantárgy::kinekkedvence;
     relationship Csoport tagja
          inverse Csoport::diákjai;
     relationship Csoport felelős
          inverse Csoport::csoportfelelős;
```

```
        integer kora() raises (nincsSzülDatum);
        hasonlóErdeklődésűDiákok(out Set<String>);
};
```

Figyeljük meg, hogy kiegészítettük a Diák osztály deklarációját a diák születési dátumával, mint attribútum és egy kora() nevű metódussal, mely megadja a diák korát. A metódus kivételt vált ki, ha a születésiDátum attribútumnak nincs értéke.

A hasonlóErdeklődésűDiákok metódus a kimenő paraméter értékekénk megadja azoknak a diákoknak a nevét, akiknek ugyanaz a kedvenc tantárgya, mint annak a Diák objektumnak, mely meghívta a metódust.

További részleteket az ODL-ről az [UlWi97]-ban találhat az olvasó.

13.3. OQL alapfogalmak

Az Object Query Language (OQL) objektum-orientált adatbázisok lekérdezésére szolgál. Az OQL a relációs adatmodell lekérdezésére használt SELECT SQL parancsnak felel meg. Azonban például abban is különbözik, hogy az OQL-t használhatjuk egy objektum-orientált befogadó nyelv, például C++, Java, stb. kiterjesztéseként. A befogadó nyelvi utasítások és az OQL parancsok keveredhetnek anélkül, hogy a lekérdezés eredményei és befogadó nyelvi változók között az értékeket explicit át kellene adni, mint a relációs model esetén, lásd 9.2.1.

13.1. példa: Legyen egy zenés CD-ket tartalmazó objektum-orientált adatbázis. Feltételezzük, hogy egy zenés CD típusa audio, vagy mp3-as, vagy videoklippeket tartalmaz és ezek nem keverednek. Ezen kívül tudjuk, hogy egy előadónak több albuma is van, egy albumon több zeneszám jelenik meg, viszont egy zeneszám csak egy albumon jelenik meg, egy albumon azonos stílusú zeneszámok jelennek meg. Feltételezzük még, hogy egy album teljes egészében egy CD-n található és egy zeneszám, csak egyszer jelenik meg a CD-ken. A továbbiakban ezt az adatbázist fogjuk lekérdezni ODL segítségével.

```
interface Előadó
     (extent Előadók
          key eNév)
{
     attribute string eNév;
     attribute string nemzetiség;
     relationship set<Album> albumai
          inverse Album::előadója;
};

interface Album
     (extent Albumok
          key (aCíme, megjelenEve))
{
     attribute string aCíme;
     attribute string stílus;
     attribute integer megjelenEve;
     relationship Előadó előadója
          inverse Előadó::albumai;
     relationship CD CDje
          inverse CD::CDalbumai;
     relationship set<Zeneszám> zeneszámai
          inverse Zeneszám::albuma;
};

interface Zeneszám
     (extent Zeneszámok
          key (szCíme, időtartam))
{
     attribute string szCíme;
     attribute float időtartam;
     relationship Album albuma
```

```
            inverse Album::zeneszámai;
        string stílus() raises (nincsStílus);
        ElőadóNeve(out string);
};

interface CD
        (extent CDk
            key azonosító)
{
        attribute string azonosító;
        attribute enum CDTípus {audio, mp3, videoklipp} típus;
        relationship set<Album> CDalbumai
            inverse Album::CDje;
 };∈
```

13.3.2. Típusok az OQL-ben

Az OQL-ben használt változókat a befogadó nyelvben deklaráljuk. Konstansokat a következő típusokból kiindulva építhetünk:

- alaptípusok
  - atomi típusok: egész számok, valós számok, karaktersorok és logikai értékek. A karakterláncokat dupla idézőjel közé helyezzük.
  - felsorolások: egy felsorolásban szereplő értékeket ODL-ben adjuk meg, ezek közül bármelyik használható konstans értékeként.
- összetett típusok
  - kollekciótípusok
    - Set()
    - Bag()
    - List()
    - Array()
  - Rekortípus
    - Struct().

A kollekciótípus és rekord szerkesztés alkalmazhatóak alap és összetett típusokra is. A rekord szerkesztés esetén mező neveket is kell adnunk, majd kettőspont után az értéket.

13.2. példa: A következő rekord szerkezetnek két mezője van: tantárgy és jegyek.

```
struct (tanárgy: "matematika", jegyek:bag (9,8,9,7,8,10,9)) □
```

13.3.3. Útkifejezések

Az összetett szerkezetű változók komponenseit a pont segítségével választjuk el, C-hez hasonlóan.

Legyen *o* egy *C* osztályba tartozó objektum és *k* az osztály valamely összetevője: attribútuma, kapcsolata vagy metódusa, akkor az *o.k* kifejezés az *o* objektum *k* komponensére hivatkozik.

Az összetevő típusa szerint a következő esetek állnak fenn:

- Ha *k* egy attribútum, akkor *o.k* az *o* objektum *k* attribútumának értéke.
- Ha *k* egy kapcsolat, akkor *o.k* az *o* objektumnak a *k* kapcsolaton keresztül kapcsolatban álló objektum vagy objektumok kollekciója.
- Ha *k* egy metódus, akkor az *o.k* a *k o*-ra való alkalmazásának az eredménye.

13.3. példa: Legyen a `kedvencZeneszám` egy Zeneszám típusú befogadó nyelvi változó, akkor:

- `A kedvencZeneszám.hossz` értéke, az adott zeneszám hossza, vagyis a hossz attribútum értéke abban a Zeneszám objektumban, mely a kedvencZeneszám változóban található.

- A kedvencZeneszám.stílus() értéke egy karaktersor, mely a zeneszám stílusát adja meg. Mivel a stílus nem a zeneszámnál, hanem a szám albumánál van megadva, írtunk egy metódust, mely megadja a zeneszám stílusát.
- A kedvencZeneszám.Albuma megadja azt az Album objektumot, melyhez a kedvencZeneszám tartozik.
- A kedvencZeneszám.ElőadóNeve(ElőadóNév) szintén metódus meghívás, mely az ElőadóNév paraméter értékeként karaktersor formájában magadja a kedvencZeneszám előadójának a nevét .

A pontot többször is alkalmazhatjuk egy kifejezésben. A fenti példát folytatva a kedvencZeneszám.albuma.előadója.eNév kifejezés megadja a kedvencZeneszám változóban található Zeneszám objektum előadójának a nevét, a következőképpen:

- Amint már láttuk fennebb, a kedvencZeneszám.Albuma azt az Album objektumot adja meg, melyhez a kedvencZeneszám tartozik. Tovább lépve,
- A kedvencZeneszám.albuma.előadója a fenti Album objektum Előadó objektumához vezet. Végül:
- A kedvencZeneszám.albuma.előadója.eNév megadja az keresett előadó nevét.

13.3.4. Select-from-where kifejezések OQL-ben

A relációs adatmodell esetén használt SELECT SQL-hez hasonlóan az OQL-ben is select kifejezéseket használunk. Ha az SQL-ben opcionális volt a tábla neve után egy sorváltozó használata, az OQL-ben kötelező. A select-from-where kifejezések általános alakja:

```
SELECT <kifejezések_listája>
FROM <változó_deklaráció>
WHERE <logikai_kifejezés>
```

A FROM kulcsszó után változókat adunk meg a következőképpen:

- egy kifejezés, amelynek értéke valamilyen kollekciótípusú (halmaz vagy multihalmaz). Az esetek többségében ez a kifejezés egy osztálynak az objektumkészlete, mint például az Album osztályhoz tartozó Albumok objektumkészlet. Az objektumkészletek hasonlóak a relációs táblákhoz. OQL-ben viszont tetszőleges kifejezést használhatunk egy változó deklarálásakor, amely kollekciót ad eredményül, például egy másik select-from-where kifejezés. SQL2-ben ez nem volt lehetséges, de kereskedelmi rendszerek (pl. Oracle) már megengedik az alkérdések használatát a FROM záradékban;
- opcionálisan az AS kulcsszót;
- a változó nevét.

A SELECT kulcsszó után a FROM záradékban deklarált változókat tartalmazó kifejezéseket adhatunk meg.

A WHERE kulcsszó utáni logikai kifejezésben olyan operandusok szerepelhetnek, amelyek csak konstansokat és a FROM záradékban deklarált változókat tartalmaznak. Az SQL-ből is ismert összehasonlító operátorokat (<, >, <=, >=, =) használjuk, azzal a különbséggel, hogy a különbözik operátor a "!=", a logikai operátorok pedig: AND, OR, NOT.

13.4. példa: Keressük a „Gyöngyhajú lány” című zeneszám időtartamát.

```
SELECT z.időtartam
FROM Zeneszámok z
WHERE z.szCíme = "Gyöngyhajú lány"
```

Figyeljük meg, hogy karaktersor esetén dupla idézőjelet használtunk. A z változó végigfut a Zeneszámok objektumkészleten. Minden objektum esetén a WHERE feltétel kifejezését (z.szCíme = "Gyöngyhajú lány") kiértékeli a lekérdezésfeldolgozó és ha a kifejezés igaz, akkor az objektum időtartam attribútum értéke az eredménybe kerül. □

Használhatnánk a FROM záradékban az AS kulcsszót is, de nem szokták.

A FROM záradékban több változó is szerepelhet. Lássuk a következő példát.

13.5. példa: Keressük az Omega 7 album zeneszámait.

```
SELECT z.szCíme, z.időtartam
FROM Albumok a, a.zeneszámai z
WHERE a.aCíme = "Omega 7"
```

Az *a* változó az Albumok objektumkészletet járja végig. Az Album osztálynak van egy zeneszámai kapcsolata és a *a.zeneszámai* egy adott *a* album zeneszám típusú objektumainak a halmazát adja meg. Minden *a* változó esetén, mely egy Album típusú objektum, a *z* változó végigfutja az *a* album összes zeneszámát. Tehát a *z* egy Zeneszám típusú objektum, mely *a.zeneszámai* halmazban veszi fel értékeit. Algoritmikusan a következőképpen írhatjuk le a select-from-where kifejezés kiértékelését:

minden a-ra az Albumok-ból
  minden z-re az a.zeneszámai-ból
    ha a.aCíme = "Omega 7"
      a z.szCíme, z.időtartam az eredménybe kerül

Amint látjuk, két ciklus segítségével (*a*, *z*) párok kerülnek kiértékelésre, ahol *a* Album és *z* az *a* Album zeneszámai. Az eredménybe azon (*a*, *z*) párok kerülnek, melyek kielégítik a WHERE feltételét. ∈

A select-from-where kifejezések eredményként multihalmazt állítanak elő, akárcsak az SQL nyelv esetén. Az ismétlődő sorokat megszüntethetjük a DISTINCT kulcsszó segítségével.

13.6. példa: Keressük azon előadókat, akik rock stílusú albumokat adtak ki.

```
SELECT DISTINCT e.eNév
FROM Előadók e, e.albumai a
WHERE a.stílus LIKE "%rock%"
```

Az *e* változó végigjárja az Előadó osztály objektumkészletét, minden *e* előadó esetén az *a* változó végigfutja az *e* albumait és ha a stílus rock, akkor az előadó neve az eredménybe kerül. A feltételt a LIKE kulcsszóval adtuk meg, mivel a hard rock, punck rock, gótikus rock stb. is rock. Egy előadónak több rock albuma is lehet és akkor többször is megjelenne a neve, a DISTINCT kulcsszó segítségével viszont megszüntetjük. □

Kvantort használó kifejezések

Az OQL lehetőséget ad logikai kvantorok: minden és létezik, használatára. Megvizsgálhatjuk, hogy egy halmaznak minden eleme eleget tesz-e az adott feltételnek. Legyen *A* egy halmaz, *x* egy változó és *F*(*x*) egy feltétel. A

```
FOR ALL x IN A : F(x)
```

kifejezés segítségével ellenőrizni tudjuk, hogy az *A* halmaz minden eleme kielégíti az *F*(*x*) feltételt. Ha minden *x* az *A*-ból kielégíti az *F*(*x*) feltételt, a kifejezés értéte TRUE (igaz), különben FALSE (hamis).

13.7. példa: Keressük azon előadókat, akik csak rock stílusú albumot adtak ki.

```
SELECT DISTINCT e.eNév
FROM Előadók e
WHERE FOR ALL a IN e.albumai :
a.stílus LIKE "%rock%"
```

Az *e* változó az Előadó osztály objektumkészletét járja végig, az e.albumai lesz az *A* halmaz az általános formából, melyet az *a* változó sgítségével járunk végig és az előadó bekerül az eredménybe, ha minden *a* albumra igaz a feltétel, hogy a stílusa rock. □

Legyen *A* egy halmaz, *x* egy változó és *F*(*x*) egy feltétel. Az

```
EXISTS x IN A : F(x)
```

kifejezés segítségével ellenőrizni tudjuk, hogy létezik legalább egy *x* az *A*-ból, mely kielégíti az *F*(*x*) feltételt. Ha létezik legalább egy *x* az *A*-ból, mely kielégíti az *F*(*x*) feltételt, a kifejezés értéke TRUE (igaz), különben FALSE (hamis).

13.8. példa: Keressük azon előadókat, akiknek van olyan zeneszáma, melyek címében szerepel a love szó.

```
SELECT DISTINCT a.előadója.eNév
FROM Albumok a
WHERE EXISTS z IN a.zeneszámai:
z.szCíme LIKE "%love%" OR z.szCíme LIKE "%loving%"
```

Az *a* változó az Album osztály objektumkészletét futja be és minden *a* esetén az EXISTS záradék segítségével ellenőrizzük, hogy létezik-e legalább egy olyan zeneszám az album zeneszámai között, melyen a *z* változó fut végig, mely tartalmazza a megadott szót.

Mivel egy albumnak csak egy előadója van, nem volt szükség még egy változóra, mely az Előadók objektumhalmazt járja végig, használhatjuk az Album előadója nevű kapcsolatát, hogy megkapjuk az album előadójának a nevét. Egy hasonló lekérdezés esetén használtunk plusz változót az Előadók végigjárására, lásd 13.6 példát. Abban az esetben, ha egy albumnak több előadója is lehetne, vagyis *m* : *n* típusú kapcsolat lenne az Előadók és Albumok között, akkor csak úgy helyes a lekérdezés, ha használunk plusz változót az Előadók végigjárására, mivel egy album esetén több előadó is lehet. □

Alkérdések

Amint láttuk a select-from-where kifejezés általános alakjánál, a FROM kulcsszó után amikor megadjuk a változó tartományát jelentő kollekciót, a kollekció lehet alkérdés is. Minden olyan helyen, ahol egy kollekció megjelenhet, használhatunk select-from-where kifejezést egy másik select-from-where kifejezésben.

13.9. példa: Keressük Celine Dion zeneszámait!

```
SELECT z.szCíme, z.időtartam, a.aCíme, a.megjelenEve
     FROM (SELECT a
               FROM Albumok a
               WHERE a.előadója.eNév = "Celine Dion") c,
          c.zeneszámai z
```

Az alkérdés eredménye tartalmazza a Celine Dion albumait, a *c* változó végigjárja ezen albumokat. Minden *c* albumra a *z* változó végigfut annak zeneszámain. Az eredmény tartalmazza a zeneszám címét, időtartamát, illetve az album címét és megjelenési évét, melyen a zeneszám található. □

Az SQL-hez hasonlóan, OQL-ben is használhatjuk a halmazműveleteket, melyek UNION, EXCEPT, illetve INTERSECT.

Az eredmény rendezése

Az eredményt rendezhetjük az SQL-hez hasonlóan az ORDER BY záradék segítségével.

13.10. példa: Adjuk meg a 2004-ben megjelent rap számok címét és előadójának a nevét előadók szerint rendezve, egy előadón belül pedig albumon belül rendezve.

```
SELECT DISTINCT e.eNév, a.aCíme, z.szCíme
FROM Előadók e, e.albumai a, a.zeneszámai.z
WHERE a.megjelenEve=2004 AND a.stílus LIKE "%rap%"
ORDER BY e.eNév, a.aCíme, z.szCíme
```

Egy előadónak több albuma is lehet, tehát ha az előadónév ugyanaz, akkor az album cím szerint rendezzük. Egy előadó ugyanazon albuma esetén pedig a zeneszámok kerülnek ábécé szerinti sorrendbe. □

Az alapértelmezett a növekvő sorrend (ASC), a csökkenő sorrendet a DESC kulcsszóval kérhetjük.

Összesítő kifejezések

OQL-ben is használhatjuk az AVG, COUNT, SUM, MIN, MAX összesítő operátorokat. Míg SQL-ben tábla oszlopára vagy kifejezésekre, melyek táblák oszlopaira vonatkoznak, alkalmazhatjuk az összesítő operátorokat, addig OQL-ben olyan kollekciókra alkalmazhatjuk, amelyek elemei megfelelő típusúak.

- A COUNT bármilyen típusú kollekcióra alkalmazható.
- A SUM és AVG aritmetikai típusú (egész számok, valós számok) kollekcióra.
- A MIN és MAX olyan típusú kollekciókra alkalmazhatóak, amelyeken értelmezett az összehasonlítás, például az aritmetikai típus és karaktersorok.

13.11. példa: Adjuk meg a CD-ken található zeneszámok összidejét!
A CD osztály esetén megadtuk a Cdalbumai kapcsolatot és annak inverzét is: Album::CDje, mely kikényszeríti, hogy minden album, mely az adatbázisba fel van vezetve, legyen eltárolva egy CD-n. Feltételezzük, hogy nem fordul elő, hogy az album zeneszámai, melyek fel vannak vezetve az adatbázisba ne legyenek eltárolva az albumokhoz rendelt CD-n, vagyis csak azokat a zeneszámokat vezetjük be az adatbázisba, melyek megtalálhatók a CD-inken, mivel az CD-ink adatbázisáról van szó. A megfogalmazott lekérdezést a következőképpen adjuk meg:

```
SUM (SELECT z.időtartam)
     FROM Zeneszámok z)
```

Az alkérdés segítségével időtartamokat választunk ki, melyek multihalmazt alkotnak, több zeneszám is lehet 3 perces, mindet össze kell adjuk. □

13.12. példa: Adjuk meg hány különböző gótikus metal album található a CD-ken.

```
COUNT (SELECT a.aCíme
          FROM Albumok a
          WHERE a.stílus = "gótikus metál") □
```

Csoportosító kifejezések

Az SQL-ből ismert GROUP BY-t az OQL másképp használja, egy példán keresztül fogjuk ilusztrálni.

13.13. példa: Adjuk meg zenénk összidejét stílusokra, évekre leosztva a következőképpen: StílusNév, Ev, Időtartam.

```
SELECT stl, év, összidő: SUM (SELECT p.z.időtartam
              FROM partition p)
FROM Zeneszámok z
GROUP BY stl:z.albuma.stílus, év:z.albuma.megjelenEve
```

A *z* változó végigfut a Zeneszámok objektumkészleten. A GROUP BY záradékban is megadunk változókat, a *stl* és *év* a z.albuma.stílus, illetve a z.albuma.megjelenEve kifejezések felelnek meg. A GROUP BY záradék szerepe, hogy csoportosítsa stíluson belül, megjelenési éven belül a zeneszámokat. Minden egyes *stl* stílus esetén, minden *év* esetén, azon zeneszámok, melyek stílusa *stl* és megjelenési éve *év* bekerülnek a Partition nevű multihalmazba, melyre összesítő függvényt alkalmazhatunk, a mi esetünkben összegezést. Tehát a partition olyan típusú objektumokat tartalmaz, amilyen a FROM után szerepel. A *p* változó végigfutja a partition multihalmazt. □

Általános esetben, a FROM után *k* multihalmaz is szerepelhet. Legyen a select-from-where kifejezés általános alakja:

```
SELECT <kifejezés_lista>
FROM M1 m1, M2 m2, ..., Mk mk
GROUP BY v1:z1, v2:z2, ..., vn:zn
```

Az m_1, m_2, ..., m_k változók mindegyike használható az z_1, z_2, ..., z_n kifejezésekben. A partition mező értékét adó multihalmaz struktúrái *k* darab mezőből állnak, amelyekhez rendelt mezőnevek m_1, m_2, ..., m_k.

Legyenek e_1, e_2, ..., e_k az m_1, m_2, ..., m_k változók értékei, melyek a WHERE feltételét kielégítik. Ekkor a GROUP BY eredményeként adódó halmaz struktúrája:

```
Struct (v1: z1(e1, e2, ..., ek), ..., vn: zn (e1, e2, ..., ek),
     partition: P)
```

A GROUP BY-t tartalmazó select-from-where kifejezés SELECT záradékában csak a GROUP BY eredményében szereplő struktúrák mezőire lehet hivatkozni. A *P* multihalmaz struktúrája:

```
Struct (m1:e1, m2:e2, ..., mn:en).
```

Akár SQL-ben, OQL-ben is használhatjuk a HAVING záradékot, melynek segítségével a GROUP BY által létrehozott csoportok közül csak azokat választjuk ki, melyek a HAVING feltételét kielégítik. A HAVING feltételében szintén a partition multihalmazra hivatkozhatunk.

13.14. példa: Egészítsük ki utolsó példánkat azzal, hogy csak akkor érdekel zenénk összideje stílusokra, évekre leosztva, ha az adott stílusból, az adott évben van legalább 10 perc zene.

```
SELECT stl, év, összidő: SUM (SELECT p.z.időtartam
                         FROM partition p)
FROM Zeneszámok z
GROUP BY stl: z.albuma.stílus, év: z.albuma.megjelenEve
HAVING SUM (SELECT p.z.időtartam FROM partition p) >= 10 □
```

Objektumok létrehozása és kollekció végigjárása

A továbbiakban a C++-t fogjuk használni befogadó nyelvnek, melyben az OQL select-from-where kifejezéseit természetes módon lehet használni, nincs szükség adatátvitelre, mint SQL és befogadó nyelv között.

A select-from-where kifejezések eredményként objektumokat állítanak elő, melyet megfelelő típusú változónak értékül adhatunk.

13.15. példa: Legyen SarahSzamai egy set<Zeneszám> típusú változó. OQL-el kiegészített C++-ban írhatjuk a következő értékadást:

```
SarahSzamai = SELECT DISTINCT z
              FROM Albumok a, a.zeneszámai z
              WHERE a.előadója.eNév = "Sarah Connor";
```

melynek következtében a SarahSzamai változó a Sarah Connor zeneszámait fogja tartalmazni. □

Egy select-from-where kifejezés eredményéhez soronként is hozzáférhetünk, ha listává alakítjuk, ezt az ORDER BY kulcsszóval tehetjük meg, melynek következtében az eredmény egy lista típusú lesz. Egy ilyen lista elemeihez a tömböknél megszokott módon férhetünk hozzá.

13.16. példa: Legyen `RasmusSzamok` egy `list<Zeneszám>` és `szam` egy `Zeneszám` típusú változó. OQL-el kiegészített C++-ban kiírhatjuk a The Rasmus együttes dalait a következőképpen:

```
RasmusSzamok = SELECT DISTINCT z
                    FROM Albumok a, a.zeneszámai z
                    WHERE a.előadója.eNév = "The Rasmus"
                    ORDER BY z.szCíme;
NrSzamok = COUNT(SELECT DISTINCT z
                    FROM Albumok a, a.zeneszámai z
                    WHERE a.előadója.eNév = "The Rasmus");
for (i=0; i<NrSzamok; i++) {
     szam = RasmusSzamok[i];
     cout << szam.szcíme << " " << szam.idotartam << "\n";
}
```

Amint látjuk, egy zeneszámhoz a `RasmusSzamok[i]` kifejezés segítségével hozzá tudunk férni. □

13.4. Gyakorlatok

13.1.. *i*) Tervezzük meg ODL segítségével egy könyvtár objektum-orientált adatbázisát. Egy könyvről szóló információk a: cím, megjelenési év, kiadó, ISBN, ár, KönyvKod (azonosít egy könyvet a könyvtárban). Egy kiadó több könyvet is kiad, egy könyvet egy kiadó ad ki. Egy könyvnek több szerzője is lehet, egy szerző több könyvet is ír. Egy könyv egy témakörre vonatkozik (példák témakörökre: matematika, informatika, ifjúsági regény, történelmi könyv stb.). A könyvtárban egy könyvből több példány is van, egy példányt a raktári szám azonosít.

ii) OQL segítségével adjuk meg:
A Teora nevű kiadó által kiadott informatika könyvek címét és a megjenelési évet.
Az „Adatbázisrendszerek megvalósítása" című könyv szerzőinek a nevét.
Adjuk meg a matematika könyvek címét és szerzőiknek a nevét!
Adjuk meg hány különböző informatika könyv található a könytárban!
Adjuk meg a könytárban található könyvek összértékét!
(1 könyv értéke = PéldánySzám * Ar)

13.2. *i*) Tervezzük meg ODL segítségével egy ingatlan ügynökség objektum-orientált adatbázisát, mely a következő ingatlanok: lakások, házak és telkek eladását irányítja. Az ingatlanoknak vannak tulajdonosai, akiknek neve, címe, telefonszámai az adatbázisba kell kerüljenek. Az ingatlan ügynökségnek vannak alkalmazottai, akiknek szintén érdekel a neve, címe, telefonszámai. Minden ingatlan esetén el akarjuk tárolni, hogy milyen helységben van, a város melyik negyedében, a negyednek lehet egy jegye, hogy mennyire jó, keresett (pl. Györgyfalvi negyed 9), az ingatlan címe, ingatlan ára. A telkeknek a területére vagyunk kíváncsiak. A lakásoknak is a területét, viszont a ház esetén telek terület és lakható terület is tárolandó adatok. A lakások és házak esetén is fontos, hogy új-e, szobák száma, fürdőszobák száma, fűtés típusa, van-e telefon és extrák, amivel esetleg rendelkezik a lakás, illetve ház. Extrák lehetnek: termopán ablakok, felszerelt konyha, csempe típusa, stb. A vásárlók is bekerülnek az adatbázisba, nekik is van nevük, címük, telefonszámaik. Az adatbázisban az eladott ingatlanok is jelen vannak még egy ideig.

ii) OQL segítségével adjuk meg:
a) Adjuk meg azon alkalmazottak nevét, akik ügynök beosztásban dolgoznak.
b) Adjuk meg azon lakásokat a címükkel, árukkal és tulajdonosuk nevével, melyek eladók Kolozsváron a Hajnal negyedben, van saját központi fűtésük, van garázsuk, de nincsenek termopán ablakaik.
c) Adjuk meg azon alkalmazottak nevét, akik 8-kor kezdenek reggel.
d) Adjuk meg azokat a tulajdonosokat, akik több, mint 1 ingatlant árulnak.
e) Adjuk meg a címükkel, árukkal és tulajdonosuk nevével azon eladó házakat Gyaluban, melyek újak, van saját központi fűtésük, emeletesek, lakható terület legalább $100m^2$.
f) Töröljük az adatbázisból az eladott ingatlanokat.
g) Adjuk meg Kolozsvár eladó telkeinek összértékét negyedenként a következő formában:
NegyedNév, Erték.
h) Adjuk meg Kolozsvár legdrágább eladó házát az összes információjával.
i) Adjuk meg Kolozsvár legolcsóbb 2 szobás eladó lakását az összes információjával.

14. A féligstruktúrált adat

Ahogy a világhálón levő adat-mennyiség napról-napra nő, mind több és több adat jelentkezik féligstruktúrált formában. Ezalatt azt értjük, hogy bár az adatnak lehet valamilyen értelemben struktúrája, ez a struktúra nem annyira merev vagy teljes, mint ami szükséges a hagyományos adatbáziskezelő rendszerekhez.

14.1. Mi is a féligstruktúrált adat?

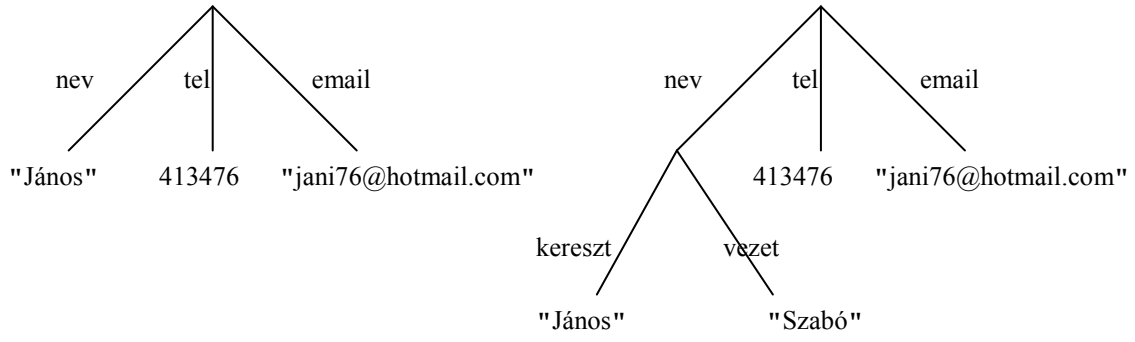
A féligstruktúrált adatról sokszor úgy beszélnek, mint „sémamentes" vagy „önleíró" adat, fogalmak, melyek jelzik, hogy nincs külön leírása az adat típusának, illetve szerkezetének. Adatbáziskezelés esetén előbb meghatározzuk az adatok szerkezetét, amit az ABKR a katalógusban tárol, nem az adatokkal együtt, azoktól külön, majd feltöltjük az adatbázist adatokkal. Féligstruktúrált adat esetén az adat szerkezete nincs külön, az adattal együtt írjuk le. Kezdjük a féligstruktúrált adat bemutatását Lisp programozóknak ismerős megfeleltetési listával, mely nem más, mint (címke-érték) párok halmaza, ezekkel ábrázolhatjuk a rekordszerű struktúrát:

```
{nev: "János", tel: 413476, email: "jani76@hotmail.com"}
```

Maguk az értékek is lehetnek struktúrák:

```
{nev: {kereszt: "János", vezet: "Szabó"}, tel: 413476,
       email: "jani76@hotmail.com" }
```

Grafikusan ábrázolva egy fa szerkezetet kapunk, ahol egy érték a fa egy csomópontjába kerül, az élekre pedig a címkék (lásd 14.1 ábra).



14.1. ábra: Egyszerű struktúrák ábrázolása gráf segítségével

A címke nem feltétlenül egyedi, megengedett a következő szerkezet:

```
{nev: "János", tel: 413476, tel: 0740334488}
```

Több sort is írhatunk ugyanazzal a szintaxissal:

```
{szemely:
   {nev: "János", tel: 413476, email: "jani76@hotmail.com"},
szemely:
   {nev: "István", tel: 405406, email: "isti91@hotmail.com"},
szemely:
   {nev: "Anna", tel: 113178, email: "ani87@yahoo.com"}
}
```

Nem szükséges, hogy az összes személy sor szerkezete ugyanaz legyen. A féligstruktúrált adat egyik erőssége éppen az, hogy képes alkalmazkodni a szerkezetbeli változásokhoz. Igaz, hogy

semmi sem állít meg abban, hogy teljesen véletlenszerű gráfokat építsünk és kérdezzünk le, mégis többnyire hasonló struktúrákkal dolgozunk. A különbségek általában hiányos adatokban, ismétlődő mezőkben vagy apró szerkezeti változtatásban vannak. Például:

```
{szemely:
     {nev: {kereszt: "János", vezet: "Szabó"}, tel: 413476,
           tel: 0740334488, email: "jani76@hotmail.com"},
 szemely:
     {nev: "István", tel: 405406, email: "isti91@hotmail.com"},
 szemely:
     {nev: "Anna", tel: 113178, magassag: 168}
 }
```

Ha a fenti három személy sort egy relációs adatbázisban szeretnénk eltárolni, három különböző típusú szerkezetet kellene értelmezni. Féligstruktúrált adat esetén tudatosan elfelejthetünk bizonyos mezőket, viszont mindegyik sor esetén meg kell adjuk a mezőnevet és az értéket is. Az ilyen adatot nevezzük „önleíró" adatnak. A sorozatosítás (serialization) azt jelenti, hogy az adatot olyan byte-sorrá alakítjuk, ami könnyen továbbítható és amiből a fogadó fél könnyen újraépítheti az adatot. Természetesen az önleíró adat helyigényes, de ezáltal hordozhatóságot nyerünk, ami a világháló környezetében létfontosságú.

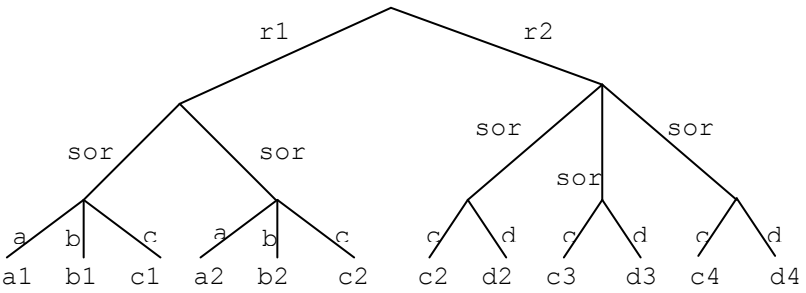
A féligstruktúrált adatok esetén használt alaptípusok a karaktersorok, számok, dátum és kép típusok.

14.1.2. Relációs adatbázis ábrázolása féligstruktúrált adat modell segítségével

Egy relációs adatbázis leírása az `r1(a,b,c), r2(c,d)` –hez hasonló sémával történik. Egy ilyen kifejezésben `r1` és `r2` a relációk nevei, és `a,b,c` illetve `c,d` az oszlopok nevei a két relációban. A gyakorlatban az oszlopok típusait is meg kell adni. Egy ilyen séma példánya olyan adat, amely igazodik ehhez a specifikációhoz, és, bár nincs egy egyezményes szintaxis a relációs példányok leírására, általában táblák soraiként fogjuk fel.

Féligstruktúrált adat formájában ezt a következőképpen írhatjuk:

```
{r1: {sor: {a: a1, b:b1, c:c1},
 sor: {a: a2, b:b2, c:c2}
          },
 r2: {sor: {c: c2, d:d2},
            sor: {c: c3, d:d3},
            sor: {c: c4, d:d4}
          }
}
```



14.2. ábra: Relációs adatbázis ábrázolása fa segítségével

14.1.3. Objektumokra való hivatkozás

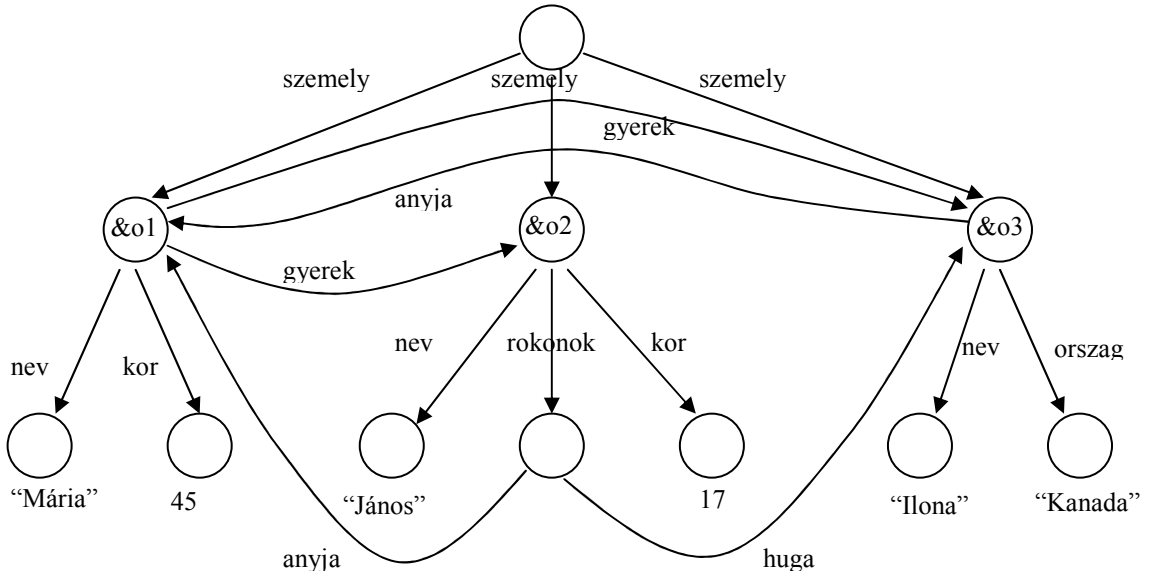
A mai adatbázis alkalmazások tudnak objektumokat kezelni, objektum-relációs vagy objektum-orientált adatbázisok segítségével. Ezen típusú adatbázisok által használt adatok is ábrázolhatóak féligstruktúrált adatként. Legyen a következő példa:

14.1. példa: Máriának két gyereke van, János és Ilona. Objektumazonosítókat használhatunk akkor, ha olyan struktúrákat szeretnénk felépíteni, melyekben objektumokra hivatkozunk:

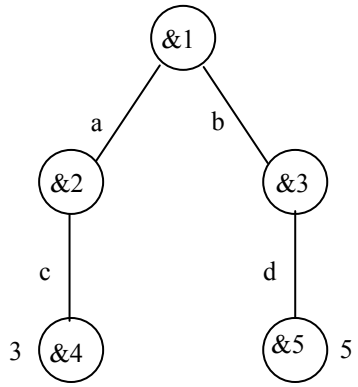
```
{szemely: &o1{nev: "Mária",
     kor: 45,
     gyerek: &o2,
     gyerek: &o3
     },
 szemely: &o2{nev: "János",
     kor: 17,
     rokonok: {anyja: &o1,
               huga: &o3}
     },
 szemely: &o3{nev: "Ilona",
     orszag: "Kanada",
     anyja: &o1
     }
}
```

A példát ábrázolva egy gráfot kapunk. A gráf bizonyos csúcspontjaiban használhatunk objektumazonosítót, mint címkét. Ezután használhatjuk a címkét (mint egy érték), annak a szerkezetnek az azonosítására, mely az irányított gráfban utána következik (alatta található). A reprezentációs gráfban így megjelennek osztott alstruktúrák (14.3 ábra). A `&o1`, &o2, &o3 neveket *objektumazonosítóknak* hívjuk (röviden OID). Az ábrán az élek végére nyílakat helyezünk, így jelezve a hivatkozás irányát. Ettől a ponttól kezdve az adat már nem fa, hanem gráf, melyben minden csúcsnak egyedi objektumazonosítója van. □

Az 14.4 ábrán a leveleknek is adtunk azonosítót. A levelekben csak atomi értékek lehetnek.



14.3. ábra Az 14.1 példa ábrázolása OEM gráf segítségével



14.4. ábra: Struktúra objektumazonosítókkal

Az objektumazonosítók értéke a háttértárolón tárolt adatok esetén lehet:

- egy cím a háttértárolón, ahol a csúcs a lemezen tárolva van;
- bemenet egy átalakítási táblába, mely az effektív címet megadja.

Az objektumazonosító a memoriába betöltve pointer lesz, mely arra a memória helyre mutat, ahol a csúcs tárolva van. Ez adott alkalmazás esetén érvenyes csak.

A Web esetén az objektumazonosító egy globális névterület (namespace) része. Lehet egy URL, melyet egy lekérdezés követ, mely az objektumot megadja. Az objektumazonosítók a különböző doméniumokból egyidejűleg létezhetnek és adatcsere esetén átalakítódnak egyik formából a másikba. Például a Web-en található objektum objektumazonosítója átalakítható egy lemez lokális objektumazonosítójává, mikor az objektumot a lemezre tároljuk, majd memóriabeli pointerré mikor betöltődik a memóriába.

A féligstruktúrált adatmodell esetén egy csomónak lehet explicit objektumazonosítója vagy nem. A rendszer automatikusan fog neki generálni egy egyedi objektumazonosítót, mikor feldolgozza.

Például az {a: {b: 5}} adatszerkezetet megadhatjuk:

{a: &o1 {b: &o2 5}} formában vagy {a :&o1{b: 5}},

ezek izomorf gráfokat jelölnek.

A következő adatszerkezet {a: {b: 2}, a: {b: 2}} objektumazonosítókkal:
{a: &o1{b: &o2 2}, a: &o3{b: &o4 2}}.

14.1.4. Féligstruktúrált kifejezések értelmezése

A szintaxis meghatározásában a féligstruktúrált adat kifejezést rövidítve *fsa-kifejezés*-ként használjuk. A címkéknek és atomi értékeknek (számok, karakterláncok) szabványos szintaxist használunk. Az objektumazonosítokat (OID-ket) egy „&” jellel kezdjük, pl. &123.

<*fsa-kifejezés*>:: = <*érték*> | OID <*érték*> | OID
<*érték*> :: = atomiérték | <*komplexérték*>
<*komplexérték*> :: ={címke : <*fsa-kifejezés*>, ... , címke: <*fsa-kifejezés*>}

Azt mondjuk, hogy egy *o* objektumazonosító egy *k* fsa-kifejezésben van *értelmezve*, ha:

- a *k* kifejezés *o e* alakú ahol *e* egy érték, vagy
- a *k* kifejezés $\{c_1: e_1, \ldots, c_n: e_n\}$ alakú és *o* értelmezve van az $e_1, \ldots, e_n$ valamelyikében.

Ha bármilyen más módon van jelen *o* a *k* kifejezésben, azt modjuk, hogy az *o használva van k*–ban. Ahhoz, hogy egy *k* fsa-kifejezést helyesnek tekintsünk, teljesítenie kell a következő feltételeket:

- egy *o* objektumazonosító legföljebb egyszer értelmezett a *k*-ban;
- ha egy *o* objektumazonosító használva van *k*-ban, akkor értelmezve kell legyen *k*-ban.

14.1.5. Az Object Exchange Model (OEM)

Az OEM a Tsimmis-ben volt értelmezve, rendszer, melyet heterogén adatforrások integrálására terveztek.

Egy OEM objektum a következő négyes: {*címke*, *OID*, *típus*, *érték*}, ahol

- *címke* egy karaktersor;
- *OID* egy objektumazonosító;
- *típus* lehet atomi (számok, karaktersorok, kép típus, stb.) vagy összetett (objektumazonosítók listája);.
- *érték*, mely atomi, ha a típus atomi, illetve összetett érték, ha a típus összetett.

OEM adat egy gráf, akárcsak a féligstruktúrált adat, de a címkék a csúcsokhoz és nem az élekhez vannak rendelve. Több változata is volt az OEM-nek, volt olyan is, amelynél a címke az élhez volt rendelve. A féligstruktúrált adat ábrázolására alkalmasabb az él-címkézett gráf, mi továbbra is ezt fogjuk használni.

14.1.6. Objektum-orientált adatbázis ábrázolása

Modern alkalmazások objektum-orientált modellt használnak adataik leírására. E modell alapjául az ODMG specifikációt vesszük. Röviden leírjuk, hogyan fejezhető ki ODMG adat a féligstruktúrált adat sintaxisával. (Itt eltekintünk a metódusok ábrázolásától.)

Legyen a következő, ODL-ben (Object Definition Language) kifejezett séma:

```
class Orszag
     (extent Orszagok)
{    attribute string oKode;
     attribute string oNev;
     relationship Varos fovaros;
     relationship set<Varos> varosai
          inverse Varos::orszaga;
}
class Varos
     (extent Varosok)
{    attribute string vKod;
     attribute string vNev;
     relationship Orszag orszaga
          inverse Orszag::varosai;
}
```

Ebben a példában az `Orszag` és `Varos` osztályok objektumai objektumok kollekciója. A `Varosok` halmazban az elemek `set<Varos>` típusúak. A halmazban olyan elemek vannak, amit a `Varos` konstruktora hoz létre. A `Varos` típusú objektumoknak pontos struktúrájuk van, mely meghatározza a jellemzőiket (attribútumaikat), azaz `vKod`, `vNev`, `orszaga`. A relationship jellemző kapcsolatot fejez ki. A kapcsolatoknak vannak inverz kapcsolatai, amint ezt láttuk az ODL részletes leírása esetén (lásd 13.2). A tény, hogy az `orszaga` kapcsolat a `Varos`-ban és a `varosai` az `Orszag`-ban egymásnak fordítottjai, azt jelzi, hogy minden *o* `Orszag` objektumra, ha a *v* `Varos` objektum benne van az `o.varosai`–ban, akkor a `v.orszaga` egyenlő *o*–val.

Féligstruktúrált adat szintaxisával ezt a következőképpen írhatjuk le:

```
{orszagok: {orszag: &o1{oKod: "HU",
                    oNev: "Magyarország",
                    fovaros: &v1,
                    varosai: {varos: &v1, varos: &v3, …}
               },
               orszag: &o2{oKod: "RO",
                    oNev: "Románia",
                    fovaros: &v2;
                    varosai: { varos: &v2, varos: &v4 …}
               },
          ...
```

```
      },
   varosok:
    { varos: &v1{vKod: "BUD", vNev: "Budapest", orszaga: &o1},
      varos: &v2{vKod: "BUC", vNev: "Bucuresti", orszaga: &o2},
      varos: &v3{vKod: "DEB", vNev: "Debrecen", orszaga: &o1},
      varos: &v4{vKod: "CLJ", vNev: "Cluj", orszaga: &o2},

      ...
      }
}
```

14.1.7. Terminológia

Lássunk egy rövid áttekintését, azon gráfelméleti fogalmaknak, melyekre szükségünk lesz a továbbaikban.

Egy (*Cs*, *E*) *gráf csúcs*ok (*Cs*) és *él*ek halmazából (*E*) áll. Minden $e \in E$ élhez hozzátartozik egy rendezett csúcs-pár, a *kezdő-csúcs* (source node) $s(e)$ és a *cél-csúcs* (target node) $t(e)$, amennyiben a gráf irányított.

Egy *út* olyan $e_1, e_2, ..., e_k$ élek sora, amelyekre $s(e_{i+1}) = t(e_i)$, ahol $1 \le i \le k-1$. Az útban szereplő élek száma, *k*, az út *hossz*a.

Egy *r* csúcs a (*Cs*, *E*) irányított gráf *gyöker*e, ha *r*-ből a gráf bármely $c \in Cs$ csúcsába létezik út, ahol $c \neq r$, amennyiben a gráf egyszeresen összefüggő.

Egy *kör* egy gráfban egy út egy csúcsból önmagába.

Egy gyökeres gráf akkor *fa*, ha a gyökérből minden csúcsba van egy és csakis egy út.

Egy csúcsot *levél*nek nevezünk, ha ez a csúcs egyetlen élnek sem kezdő-csúcsa.

A féligstruktúrált adat modellje egy *él-címkézett gráf*. Az *adatgráf* kifejezés alatt az adat gráfreprezentációját értjük.

14.2. Féligstruktúrált adatok lekérdezése

A dokumentum és az adatbázis nézet között lényeges különbség van abban, ahogyan az adatot elérik. A dokumentumok világában megadunk egy URL címet és kapunk egy teljes HTML oldalt. Az adatbázisok világában felépítünk egy lekérdezést, a lekérdezést elküldjük a szerverhez, mely meghatározza a választ és elküldi a kliensnek.

A lekérdező nyelvek nagyon fontosak a világhálón levő adatok szemszögéből is. A lekérdező nyelvek segítségével nagy mennyiségű adatból csak annyit válogatunk ki, melyre az adott pillanatban szükségünk van. A lekérdezés feldolgozó pedig egy hatékony hozzáférési tervet kell kidolgozzon, hogy minél gyorsabban kapjunk választ a feltett kérdésre.

Több lekérdező nyelvet kidolgoztak már a féligstrukturált adatok lekérdezésére, többnyire az OQL-hez (Object Query Language) hasonlítanak.

14.2.1. Elérési-út kifejezések (path expressions)

A féligstruktúrált adatokat lekérdező nyelvek egyik legmegkülönböztetőbb jellemvonásuk az a képességük, hogy tetszőleges mélységig mehetnek le az adatgráfban. Ennek megvalósítása érdekében az ún. *elérési-út kifejezés*t használják.

Emlékezzünk, hogy az adatunkat egy él-címkézett irányított gráf segítségével modelleztük. Egy $c_1.c_2. ... .c_n$ élcímke sorozatot *elérési-út kifejezés*nek nevezünk. Egy elérési-út kifejezést egy egyszerű lekérdezésnek tekintünk, amelynek eredménye, egy adott adatgráf esetén, csúcsok halmaza.

14.2. példa: Legyen egy könyvészetet tartalmazó adatgráf az 14.5a ábrán.

- A "`biblio.könyv`" elérési-út kifejezés eredménye az {`n1, n2`} halmaz.
- A "`biblio.könyv.szerző`" elérési-út kifejezés eredménye azon csúcsok halmaza, amelyekhez a hozzárendelt értékek a {"`Ullman`", "`Widom`", "`Smith`"} halmazból vannak, lásd 14.5b ábra. □

Általánosan fogalmazva: a $c_1.c_2. \ldots .c_n$ útkifejezés eredménye egy adott adatgráf esetén azon v_n csúcsok halmaza, amelyekre léteznek az

$$(r, c_1, v_1), (v_1, c_2, v_2), \ldots (v_{n-2}, c_{n-1}, v_{n-1}), (v_{n-1}, c_n, v_n)$$

élek a gráfban, ahol *r* a gráf gyökere.

Ahelyett, hogy leírnánk az egész utat, sokszor csak egy tulajdonságát adjuk meg. Például az út át kell haladjon egy `cikk` élen, vagy az él címkéjében szerepelnie kell a "`bib`" szótöredéknek. Ezen tulajdonságok leírására *reguláris kifejezés*eket használunk. Két szinten használunk reguláris kifejezéseket: az elérési-út kifejezéseket alkotó élcímkék ábécéjén és azon karakterek ábécéjén, melyekből a címkék képződnek.

Kezdjük az élcímkékkel, a `könyv|cikk` reguláris kifejezésre illeszkedik a `könyv` és a `cikk` él is. Ez használható a `biblio.(könyv|cikk).szerző` lekérdezésben és megadja mind a könyvek, mind a cikkek szerzőit, melyek az adatgráfban találhatóak.

Egy másik hasznos minta az alulvonás: _ , amelyre illeszkedik bármilyen élcímke. Például a `biblio._.szerző` mintára azok az útak illeszkednek, melyek esetén egy `biblio` élet akármilyen él követhet, majd egy `szerző` éllel ér véget. Ebben az esetben is az adatgráfból a szerzőket kapjuk meg, de nem csak könyv és cikk szerzői lehetnek, hanem ha az adatgráf más típusú könyvészetet is tartalmaz, azok szerzői is bekerülnek az eredménybe.

A * azt jelzi, hogy egy reguláris kifejezés tetszőleges számszor ismétlődhet, a művelet neve: Kleene lezárás. Például a `biblio._*.szerző` mindazokat a csúcsokat eredményezi, amelyek olyan út végén vannak, amely `biblio` címkével kezdődnek, `szerző` címkével végződnek és köztük akárhány él lehet.

A reguláris kifejezés általános szintaxisa:

`e::= c | ∈ | _ | e.e | (e) | e|e | e* | e+ | e?`

ahol *c* a címkékből veszi az értékeket, *e* a kifejezésekből, ∈ az üres kifejezést jelöli, a | jelet pedig reguláris kifejezések különböző formájának elhatárolására használjuk.

A pont az úton belül az élek elhatárolására használatos, ezt már az előbbi példákban is láthattuk.

A * 0 vagy több ismétlését, + egy vagy több ismétlést, és ? 0 vagy egyszeri előfordulást jelent.

Például az `e*` a következő formájú lehet:

`∈ |e|e.e|e.e.e|e.e.e|...`

Lássunk még egy pár komplexebb példát is.

14.3. példa: A következő mintára

```
((s|S)ection|paragraph)(s)?
```

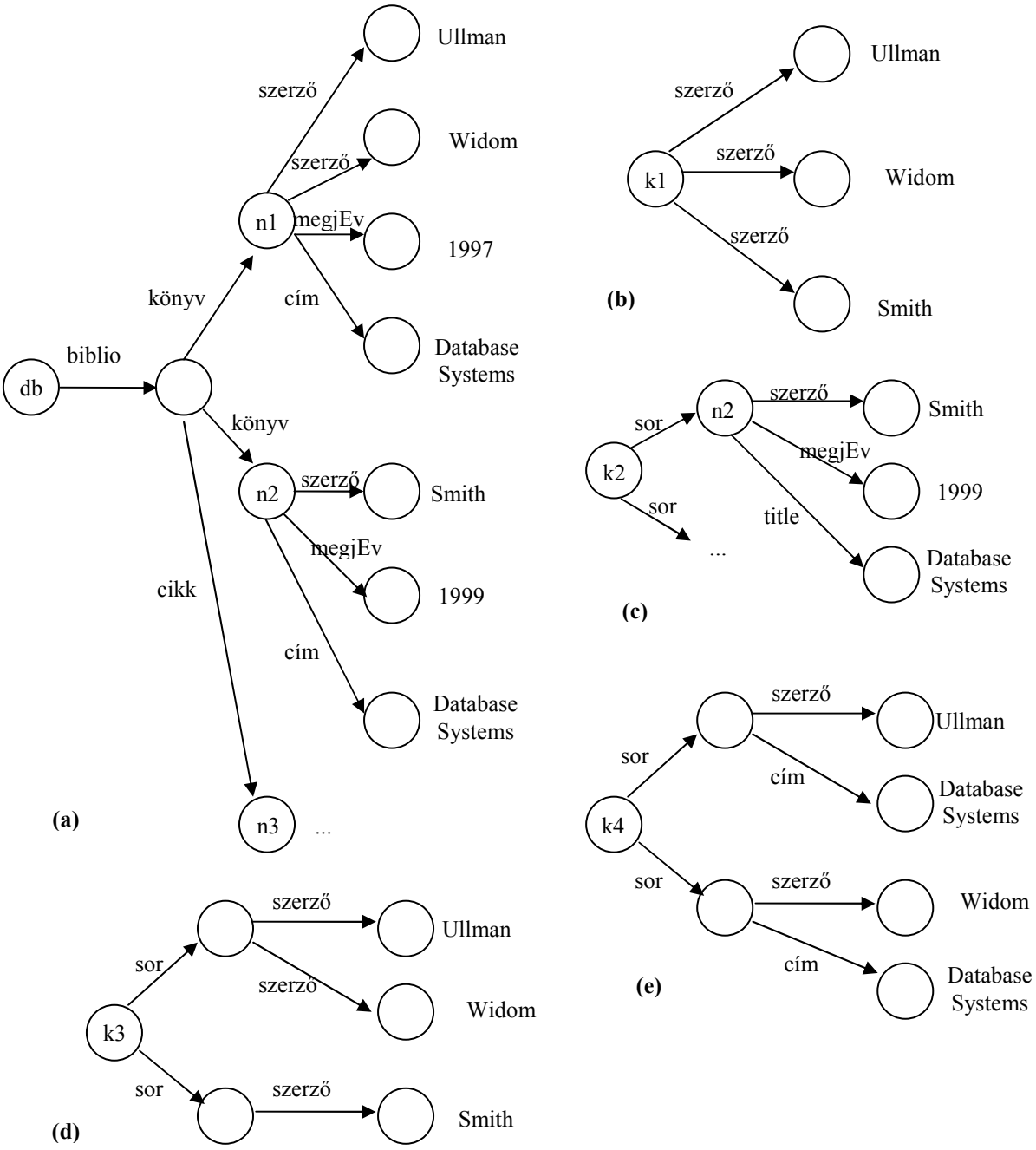
illeszkednek:

`section, Section, sections, Sections, paragraph, paragraphs.` □

Annak érdekében, hogy elkerüljük a kétértelműséget a címkékre, illetve elérési-út kifejezésekre vonatkozó reguláris kifejezések esetén, az előbbi esetén kettős idézőjelet használunk. Egy általános elérési-út kifejezés, melyben mindkét típusú reguláris kifejezés előfordul:

```
biblio._*.section.("[t|T]itle"|paragraph)."*heading.*"
```

A fenti mintára minden olyan elérési-út kifejezés illeszkedik, mely `biblio` éllel kezdődik, `section` éllel végződik, melyet `title` (esetleg nagybetűvel) vagy `paragraph` követ, majd olyan él következik, mely tartalmazza a `heading` karaktersort.



14.5. ábra: Könyvészetet tartalmazó féligstruktúrált adatok és lekérdezések eredményei

Amennyiben az adatgráf nem körmentes, több figyelemmel kell megszerkesztenünk a reguláris kifejezéseket. Az országok és városokról szóló példa esetén a gráf adatai illeszkednek a következő elérési-út kifejezésekre:

```
varosok.varos.orszaga
varosok.varos.orszaga.fovaros
varosok.varos.orszaga.fovaros.orszaga
varosok.varos.orszaga.fovaros.orszaga.fovaros
```

Ha az adatgráfban van kör, akkor végtelen sok út illeszkedik a _* reguláris kifejezés mintájára.

14.3. A lekérdező nyelvek alapjai

Annak ellenére, hogy az elérési-út kifejezések a féligstruktúrált lekérdezőnyelvek fontos jellemzője, mégis csak részben nyújták azt, amit egy lekérdező nyelv adhat. Csak egy csúcs halmazt adnak meg az adatbázisból. Nem képesek új csúcsok felépítésére, nem tudják elvégezni a relációs adatbázisokból jól ismert join műveletet, és nem tudják tesztelni az adatbázisban tárolt információt. Az elérési-út kifejezések képezik a lekérdező nyelvek alapját, a lekérdező nyelv végzi el majd a többit.

Sok lekérdezőnyelvet dolgoztak már ki a féligstruktúrált adatok lekérdezésére és még tovább folyik ez a munka. Mi a nyelvek egy magját mutatjuk be, mely a Lorel és UnSQL alapszintaxisára hasonlít.

14.3.1. Alapszintaxis

Az alapszintaxis az OQL-re épül és a következő lekérdezés példázza:

14.4. példa:

```
% k1 lekérdezés
select szerző: S
from biblio.könyv.szerző S
```

amely a könyvszerzők halmazát eredményezi. Az `S` változó végigfut a könyvszerzőkön. A lekérdezés feladata, hogy létrehozzon egy új csomót és ezt `szerző` –vel címkézett élek által összekösse azokkal a csúcsokkal, amelyek a `biblio.könyv.szerző` elérési-út kifejezés kiértékeléséből származnak. A lekérdezés eredménye egy új adatgráf, amely a 14.5.b. ábrán látható, az eredmény a

```
{szerző: "Ullman", szerző: "Widom", szerző: "Smith"}
```

féligstruktúrált adat kifejezésnek felel meg. □

A következő példában leszűkítjük a kimenetet egy feltétel bevezetésével, a where segítségével.

14.5. példa:

```
% k2 lekérdezés
select sor: S
from biblio._ S
where "Smith" in S.szerző
```

Ennek a lekérdezésnek az eredménye az 14.5.c. ábrán látható, és a neki megfelelő fsa-kifejezés:

```
{sor: {szerző: "Smith",
       megjEv: 1999,
       cím: "Database Systems"}, …}
```

Az `S` változó befutja azokat a csomópontokat, melyek elérési-út kifejezése a `biblio` éllel kezdődnek és akármilyen él követhet, az ábrán szereplő adatgráf esetén lehet `könyv` vagy `cikk`. Az `S.szerző` szintén egy elérési-út kifejezés, amelynek a gyökere az `S` által jelölt csúcs. Minden `S` esetén lehet egy vagy több szerző él, tehát az `S.szerző` egy halmaz, tehát a feltétel egy halmazhoz való hozzátartozást jelent. □

Általános esetben, legyen a következő alakú lekérdezés:

```
select E
from B
where F
```

A lekérdezés értelmezése három lépésben történik. A `from` kulcsszó után változókat adhatunk meg OQL stílusban, melyek értelmezése egy-egy elérési-út kifejezés. Legyenek ezek a változók a mi esetünkben *X*, *Y*, *Z*.

Első lépésben e három változó értékeinek a halmazát határozzuk meg, kiértékelve a megfelelő elérési-út kifejezéseket , melyek adatgráfbeli objektumazonosítók lesznek.

A második lépés az *X*, *Y*, *Z* értékeinek a halmazából kiválogatja azokat, amelyek kielégítik az *F* feltételeit. Legyen ez a halmaz

$$\{(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\}.$$

ahol minden x_1, ... , z_n érték egy objektumazonosító a bemeneti gráfból. Általános esetben a *B*–ben értelmezett változók számosságától függ az értékhalmaz elemeinek a száma, esetünkben ez érték hármas.

Az utolsó lépésben felépítjük az fsa-kifejezést, amely az eredményt képezi:

$$\{\mathtt{E}(x_1, y_1, z_1), \ldots, \mathtt{E}(x_n, y_n, z_n)\},$$

melyet úgy kapunk, hogy az `E` kifejezésbe behelyettesítjük *X*, *Y* és *Z* helyett a második lépesben kapott x_i, y_i, z_i, $i = 1,..., n$ értékeket.

14.6. példa: Legyen a következő lekérdezés:

```
select szerző: Y
from biblio._ X,
     X.szerző Y,
     X.cím Z
where matches(".*(D|d)atabase.*", Z)
```

Az `X` változó befutja azokat a csomópontokat, melyek elérési-út kifejezése a `biblio` éllel kezdődnek és akármilyen él követhet, a mi esetünkben lehet `könyv` vagy `cikk`. Az `X.szerző` szintén egy elérési-út kifejezés, melyet az `Y` segítségével járunk végig. A `Z` változó befutja a `könyv` vagy `cikk` címeket. A `where` záradék feltétele az SQL2-ben ismert LIKE operátorhoz hasonló, segítségével a kapott értékhalmazból, mely könyv vagy cikk, szerzők és címek értékhármasokat tartalmaz, kiválogatjuk azokat, ahol a könyv vagy cikk címében szerepel a Database vagy database szó. A select záradék után megadjuk, hogy ezekből az értékhármasokból a szerzőket választjuk ki, tehát azon könyvek vagy cikkek szerzőit, kik adatbázisról írtak. □

Egy másik lehetőség új csúcsok létrehozására az alkérdések beágyazása a select parancsba. Ezt példázza a következő két lekérdezés:

14.7. példa:

```
% k3 lekérdezés
select sor:( select szerző: Y
                    from X.szerző Y)
from biblio.könyv X
```

Az `X` változó befutja a könyv csomópontokat, az `Y` változó az alkérdésen belül a könyvszerzőket járja végig. Azon könyv esetén, melynek 2 szerzője van, ezek egy eredmény sorba kerülnek.

A lekérdezés eredménye megfelel a következő fsa-kifejezésnek:

```
{ sor: { szerző: "Ullman", szerző: "Widom"},
  sor: { szerző: "Smith"} }
```

Az eredmény adatgráfja az 14.5.d ábrán látható, hasonlítsuk össze a k1 lekérdezés eredményével. □

14.8. példa:

```
% k4 lekérdezés
select ( select sor: { szerző: Y, cím: Z }
               from X.szerző Y,
                    X.cím Z)
from biblio.könyv X
where "Ullman" in X.author
```

A `k4` lekérdezés a vetítés (projekció) és kiválasztás (szelekció) általánosított formáját mutatja be. Az eredmény az 14.5.e ábrán látható.

A főlekérdezés kiválasztja azokat a könyveket, ahol Ullman a szerzők között szerepel és ezeket az `X` változó fogja végigjárni. A mi példánk esetében csak egy ilyen könyv van. Ez a kiválasztás.

Az alkérdés tartalmazza a vetítést. Az eredmény szerző és cím értékpárokból áll, ahol `Y` a főlekérdezésben kiválasztott könyv szerzője és `Z` ugyanazon könyv címe. Így, mivel a kiválasztott könyvnek 2 szerzője van, mindkét szerző mellett megjelenik ugyanaz a könyvcím. □

Ebben a lekérdező nyelvben a join műveletet is ki tudjuk fejezni.

Legyen az `r1(a,b), r2(b,c)` relációs séma, amelynek sorait féligstruktúrált adatként a következőképpen írhatjuk :

```
{ r1: { sor: {a: 1, b: 2},
             sor: {a: 1, b: 3} },
  r2: { sor: {b: 2, c: 4},
       sor: {b: 2, c: 3} } }
```

A lekérdezés pedig:

```
select a: A, c: C
from r1.sor X,
          r2.sor Y,
          X.a A, X.b B, Y.b B', Y.c C
where B = B'
```

ahol `X` és `Y` az `r1` és `r2` sorok értékeit veszik fel rendre. A lekérdezés a két reláció `B` attribútum szerinti összekapcsolását végzi el, és az `A,` illetve `C` jellemzőket választja ki.

14.3.2. Lorel

Az eddig leírt nyelv a Lorel egy része. A Lorel a Lore (Lightweight Object REpository) nevű, féligstruktúrált adatot kezelő rendszer lekérdező nyelve. A Lore-t a stanfordi egyetemen fejlesztették és a 2000-es év egyik sikeres rendszere volt. A Lorel-t az OQL-ból kiindulva fejlesztették, átalakítva a féligstruktúrált adatmodellnek megfelelően. A következőkben kiegészítjük a Lorel leírását. A szintaktikai rövidítések által nyújtott tömörség vonzóbbá teszi a nyelvet.

Az első ilyen rövidítés a címkék elhagyása.

14.9. példa: Az 14.4 példa k1 lekérdezése címkék nélkül:

```
select S
from biblio.könyv.szerző S
```

A Lorel hozzáad egy alapértelmezett címkét. Annak érdekében, hogy a példáink következetesek legyenek, feltételezzük hogy `sor` az alapértelmezett címke. (A Lorelben valójában ez az `answer`.) Ezáltal a lekérdezés eredménye:

```
{sor: "Ullman", sor: "Widom", sor: "Smith"} □
```

A Lorel megengedi, hogy elérési-út kifejezéseket használjunk a `select` záradékban.

14.10. példa: Keressük a könyvek szerzőit! Nem szükséges plusz változót bevezesssünk a szerzők kiválogatására, mint ahogy a 14.7 példa k3 lekérdezésében tettük.

```
% k3' lekérdezés
select X.szerző
from biblio.könyv X
```

Minden `X` esetén az `X.szerző` csúcsok halmazát jelenti. Valóban, az `X.szerző` kifejezést a következő (beágyazott) lekérdezésként foghatjuk fel:

```
select szerző: Y
from X.szerző Y
```

Ezek alapján a példaként felírt `k3'` lekérdezés így is írható:

```
select sor:( select szerző: Y
               from X.szerző Y)
from biblio.könyv X
```

ami teljesen azonos a 14.7 példa k3 lekérdezésével. □

Általánosítva, az X.u.c kifejezés, ahol u egy tetszőlegesen hosszú elérési-út kifejezés, a következő beágyazott lekérdezéssel egyenértékű:

```
select c: Y
from X.u.c Y
```

Egy más fajta rövidítés is megjelenik Lorelben az összehasonlító feltételnél, leginkább az egyenlőség esetén, lásd a következő példát.

14.11. példa:

```
select sor: X
from biblio.cikk X
where X.szerző = "Ullman"
```

Mivel az X.szerző valójában egy halmaz, több értékre is alkalmazzuk az összehasonlítást. A feltétel valójában:

```
"Ullman" in X.szerző,
```

ami egy halmazhoz való hozzátartozás vizsgálatát jelenti. Részletezve a lekérdezés:

```
select sor: X
from biblio.cikk X
where exists Y in X.szerző (Y = "Ullman") □
```

Ez a szintaktikai kiegészítés azt jelenti, hogy a lekérdezés írója nem kell törődjön az elemi típusokkal. Az összehasonlítás szemantikailag helyes, ha mindkét érték átírható olyan közös típusra, amelyen az összehasonlítás elvégezhető.

14.12. példa: Keressük azon cikkeket, melyek 1993 után jelentek meg. Írhatjuk a következőképpen:

```
select sor: X
from biblio.cikk X
where X.megjEv > 1993
```

A feltételt az exists Y in X.megjEv (Y > 1993) formában kell érteni. □

14.3.3. UnQL

Az UnQL (Unstructured Query Language) nyelv adatmodellje különbözik a Lorel adatmodelljétől. Az UnQL nyelv nem más, mint az alap nyelv *mintá*kkal való kiegészítése, lehetővé téve így bonyolult szerkezeti feltételek rövid megfogalmazását. A Lorel-től eltérően az UnQL nem írja át a típusokat, ezért jobban kell ismernünk az adat szerkezetét a lekérdezések megírásához.

Itt az UnQL-nek csak a select részét említjük meg, további leírás található az AbBuSu2000-ben.

UnQL-ben egy mintát úgy kapunk, hogy a fsa-kifejezés szintaxisát változókkal egészítjük ki. Az egyszerűség kedvéért a példákban a változókat nagybetűkkel, a címkéket pedig kisbetűkkel jelöljük. A mintákat arra használjuk, hogy a változókat a csúcsokhoz kössük. Példák mintára:

```
{biblio: X}
{biblio: {könyv: {szerző: Y, cím: Z}}}
```

Egy minta olyan adatgráfot határoz meg, amely illeszkedik egy nagyobb gráfhoz. Ehhez az m in e szintaxist használjuk, ahol az m minta illeszkedik az e adatgráfhoz. Például a

```
{biblio: {cikk: X}} in db
```

az X változót az 14.5.a. ábra cikk csúcsához illeszti

Legyen db a gyökér neve. A következő példában láthatjuk, hogyan lehet a mintákat UnQL lekérdezésben használni:

```
select cím: X
where {biblio: {cikk:{{cím: X, megjEv: Y}}} in db,
  Y > 1989
```

A korábban használt `from` záradékra, ahol a változókat vezettük be, itt nincs szükségünk, mivel itt a minta keretén belül adjuk meg a változókat. A `where` záradék kétféle szintaktikai struktúrát tartalmaz: *generátort*, mint a `{biblio: {cikk: X}} in db`, ami bevezeti a változókat, és *feltételt*, ilyen az `Y > 1989`, ami az eddig leírtak szerint működik. A vesszőt használtuk a generátor és a feltétel elválasztására. Kiterjesztették az `in`-t, hogy mintára való illeszkedést ellenőrizzenek. Ez is kiegészítése az alapszintaxisnak.

Ha a minta helyett a `from` záradékot használnák, a lekérdezés a következő lenne:

```
select cím: X
from biblio.cikk Z,
          Z.cím X,
          Z.megjEv Y
where Y > 1997
```

Azokat a mintákat, amelyek egyetlen útból álló fákat írnak le, mint például a `{biblio: {paper: X}}`, *lineáris*aknak nevezzük. A `{c1:{c2:…{cn:e}…}}` mintára értelmezzük az $l_1.l_2.\ldots l_n.e$ szintaxist, amit *útmintá*nak nevezünk.

14.13. példa: A következő UnQL lekérdezés az Ullman általt írt dolgozatokat eredményezi:

```
select cím: X
where biblio.cikk.{szerző: "Ullman", cím: X} in db □
```

A minták használhatók a többszörös összekapcsolásra is, ami szükséges a `join` kifejezésére, amint a következő példában láthatjuk.

14.14. példa:

```
select sor: {a: X, c: Z}
where r1.sor.{a: X, b: Y} in db,
          r2.sor.{b: Y, c: Z} in db □
```

További részleteket a Lorel nyelvvel kapcsolatban lásd a [AQMW97]-ben.

14.4. XML

A mai webalkalmazások számos olyan problémával szembesülnek, amire tíz évvel ezelőtt még senki sem gondolt. Egymástól ezer kilométeres távolságra elosztott rendszereknek kell gyorsan és hiba nélkül együttműködniük. Összetett rendszerekből, adatbázisokból és alkalmazásokból származó adatokat kell továbbítani anélkül, hogy akár egyetlen tizedesvesző elveszne. Az alkalmazásoknak képeseknek kell lenniük arra, hogy kommunikáljanak, és nemcsak más üzleti komponensekkel, hanem összességében egész üzleti rendszerekkel, gyakran más vállalatokon vagy más technológiákon keresztül is. Az ügyfelek most már nemcsak a behatárolt, ún. vastag (thick) kliensek lehetnek, hanem lehetnek HTML-t támogató webböngészők, Wireless Application Protocolt (Vezeték nélküli alkalmazási protokoll - WAP) támogató rádiótelefonok vagy egészen más jelölő nyelvvel rendelkező menedzserszámítógépek. Az adatok, és ezen adatok átalakítása vált minden ma fejlesztett alkalmazás döntő fontosságú részévé.

Az XML lehetőséget nyújt a programozók számára mindezen követelmények kielégítésére. Az XML azt ígéri az adatok hordozhatósága terén, amit a Java tett a programok hordozhatósága érdekében.

14.4.1. Mi az XML?

Az XML az Extensible Markup Language (Kiterjeszthető jelölő nyelv) szavak rövidítése. Az XML a World Wide Web Konzorcium (W3C) ajánlása, amely kompatíbilis egy sokkal régebbi, SGML (Standard Generalized Markup Language = szabványos általánosított jelölőnyelv) nevű ISO

8879/1986 szabvánnyal. Az SGML megfelelőség azt jelenti, hogy minden XML dokumentum egyben SGML dokumentum is, de fordítva már nem igaz, azaz van olyan SGML leírás, ami nem XML megfelelő. Az XML azonban sokkal egyszerűbb és jobban érthető, mint az SGML. Az XML jelenleg egy kész W3C ajánlás, ami azt jelenti, hogy végleges, és nem fog változni egy újabb verzió kiadásáig. A teljes XML specifikáció egy kiváló, magyarázó szöveggel ellátott változata megtalálható a http://www.xml.com címen.

Az XML megjelenését az világhálón történő adatcsere tette szükségessé. Az XML kiegészíti a HTML-t. A HTML az web-oldalak leírásának nyelve. Egy HTML dokumentumban a szöveget tag-ek közé írjuk, a tag-ek ... alakúak, segítségükkel írjuk le a szöveg megjelenítésének módját, a hyperlinkeket, képek beszúrását, stb. A világhálón megjelenített adat nagyrésze HTML oldalak segítségével történik. Az adatok sokszor relációs adatbázisból származnak. A megjelenített adat az ember számára olvasható ugyan, de semmi sem könnyíti a programok számára az adatok struktúrájának és tartalmának feldolgozását. Az XML-t kimondottan az adatok tartalmának leírására tervezték, és nem annak megjelenítésére.

Az XML nyelvi szerkezetei biztosítják, hogy az XML dokumentum tagoltsága mind az emberi szem, mind a számítógépes programok számára jól felismerhető és azonosítható legyen. Nem mellékes körülmény az sem, hogy teszi mindezt úgy, hogy eközben csak ábrázolható (az emberi kézírásban is alkalmazható és alkalmazott) karaktereket, szimbólumokat használ. Ez egyben azt is jelenti, hogy az XML dokumentumok készítése nem igényel semmiféle speciális dokumentáló rendszert, speciális szövegszerkesztő programot, jóllehet ma már ilyenek szinte minden software környezetben elérhetők.

Az XML dokumentumok (hasonlóan a Java forráskódhoz) unicode alapú szöveges karaktersorozatok, ahol az alapértelmezés az UTF-8 (tömörített unicode), de a dokumentum elején ettől eltérő kódolási eljárást is választhatunk. Az XML dokumentumokban az adatok értékein túl olyan további címkéket és hivatkozásokat helyezhetünk el, amik utalnak az adat természetére, a dokumentum szerkezeti és tartalmi felépítésére, továbbá ezek az információk felhasználhatók a dokumentum érvényességének vizsgálatához is.

Az XML három lényeges ponton különbözik a HTML-től:

- tetszés szerint új elemek (tag) vezethetők be;
- a struktúra bármilyen mélységig beágyazható;
- az XML dokumentum tartalmazhat leírást a nyelvtanáról.

Alapformájában az XML egy egyszerű, az előbbi fejezetben leírthoz hasonló szintaxis az adatok továbbítására.

14.4.2. Az XML állományok felépítése

XML-ben az adatokat elemek (tag-ek) jelölik. Az elemeket kisebb, illetve nagyobb jelekkel („<”, „>”) határoljuk el, a határoló elemek közötti szó – címke – írja le magát az elemet. HTML-ben az elemek köre zárt és jelentésük előre meghatározott. Az XML-ben ezzel szemben nincsenek meghatározva az elemek nevei, sem jelentésük, sem alkalmazásuk módja. Ez lehetővé teszi, hogy mindenki a saját maga számára legmegfelelőbb elnevezéseket válassza.

Minden XML dokumentum egy vezérlési utasítással kell kezdődjön, melyben legalább azt meg kell adni, hogy a dokumentum melyik XML verzió szerint készült.

Vezérlési utasítás:

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
```

Annak érdekében, hogy a leírtak ne legyenek túlságosan elvontak, nézzünk előbb egy konkrét példát! Legyen egy relációs adatbázisunk, amelyben a `SZEMELYEK` tábla felépítése a következő:

```
NEV VARCHAR(50)
TELEFON VARCHAR(10)
EMAIL VARCHAR(50)
```

Vizsgálódásunk pillanatában a `SZEMELYEK` tábla a következő 3 bejegyzést tartalmazza:

SzabóJános	413476	jani76@hotmail.com
Kiss István	405406	isti91@hotmail.com
Nagy Anna	113178	ani87@yahoo.com

Féligstruktúrált adat formájában:

```
szemely:
{nev: "Szabó János", telefon: 413476, email:
 "jani76@hotmail.com"},
szemely:
{nev: "Kiss István", telefon: 405406, email:
 "isti91@hotmail.com"},
szemely:
     {nev: "Nagy Anna", telefon: 113178, email:
 "ani87@yahoo.com"}
```

Az XML jellegzetessége, hogy az adatokat hierarchikus szerkezetben képes ábrázolni, így egy XML adatfolyam (állomány) egyben mindig egy fát is definiál. A SZEMELYEK tábla tartalmát a következő XML formában állíthatjuk elő:

14.15. példa:

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<SZEMELYEK>
     <SZEMELY sorszam="1">
          <NEV>Szabó János</NEV>
          <TELEFON>413476</TELEFON >
          <EMAIL>jani76@hotmail.com</EMAIL>
     </SZEMELY>
     <SZEMELY sorszam="2">
          <NEV>Kiss István</NEV>
          <TELEFON>405406</TELEFON >
          <EMAIL>isti91@hotmail.com</EMAIL>
     </SZEMELY>
     <SZEMELY sorszam="3">
          <NEV>Nagy Anna</NEV>
          <TELEFON>113178</TELEFON >
          <EMAIL>ani87@yahoo.com</EMAIL>
     </SZEMELY>
</SZEMELYEK> □
```

A SZEMELYEK tábla tartalmának XML exportját a következőképpen is megtehettük volna:

14.16. példa:

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<SZEMELYEK>
     <SZEMELY sorszam="1" NEV="Szabó János">
          <TELEFON>413476</TELEFON >
          <EMAIL>jani76@hotmail.com</EMAIL>
     </SZEMELY>
     <SZEMELY sorszam="2" NEV="Kiss István">
          <TELEFON>405406</TELEFON >
          <EMAIL>isti91@hotmail.com</EMAIL>
     </SZEMELY>
     <SZEMELY sorszam="3" NEV="Nagy Anna">
          <TELEFON>113178</TELEFON >
          <EMAIL>ani87@yahoo.com</EMAIL>
     </SZEMELY>
</SZEMELYEK> □
```

A második megoldás többször használja a tag-ek tulajdonság (attribútum) megadási lehetőségeit, így csökkentve a tag-ek számát. Amint látjuk egy tábla tartalmát többféle XML adatfolyamként is elő lehet állítani. A tervezés során el kell döntenünk, mely XML formátum felel meg az adott feladat szempontjainak, majd el kell készítenünk hozzá azokat a nyelvtani szabályokat (lásd DTD), amik egyértelműen rögzítik azt, hogy milyen formában fogjuk az adatainkat az XML dokumentumban leírni.

Egy XML dokumentumnak csak egy gyökere lehet, de minden dokumentumnak kötelezően rendelkeznie kell egy gyökér elemmel.

Az XML dokumentum szerkezete hierarhikus, egy gyökér csomópontból és ennek gyerek csomópontjaiból épül fel.

Minden XML dokumentum esetén kötelező:

- vezérlési utasítás + verzió információ,
- gyökér elem,
- kötelező bezáró pár,
- zárójelezési szabály.

Egy XML dokumentumban a nyitó és záró elemre a hagyományos zárójelezés szabályai érvényesek, a HTML-ben megengedett ... szerkezet nem érvényes. Ez a HTML-énél bonyolultabb szerkezetet eredményez, de az értelmezők, illetve a világosabb dokumentum szerkezetek létrehozásakor előnyösebb.

14.4.3. Karakterek, elnevezések

Az XML dokumentumokban használt elemek neveiben, illetve az adatokban használt karakterek a következő szabályoknak kell megfeleljenek:

- A szabvány szerint az XML megkülönbözteti a kis és nagy betűket (case sensitive).
- A HTML-el ellentétben az XML értelmezők nem veszik ki az adatokból a többszörös kihagyás jellegű (white space) karaktereket.
- Elemek nevei a következő szabályokat kell betartsák:
  - Elem név nem tartalmazhat kihagyást (space).
  - Elem név nem kezdődhet számmal vagy aláhúzás karakterrel.
  - Elem név nem kezdődhet az „XML” karatersorral.
  - Elem névben nem ajánlott a mínusz (-), kettőspont (:) és a pont (.) karakter.

Nincs fenntartott szó, a fentieknek megfelelő bármilyen karaktersor szerepelhet egy elem nevében.

Adatok neveiben tetszőleges karakterek lehetségesek, azzal a megkötéssel, hogy az elemzett adatokban kerülni kell a kisebb, a nagyobb, valamint az idéző jeleket. Ezek a jelek karakter kódokkal vagy speciális egyedekkel (&lt, &gt, &amp, stb.) adhatók meg.

14.4.4. Névterületek (namespaces)

A névterületek célja, hogy egyértelműen tudjunk elem és attribútum neveket használni XML és XML Schema dokumentumokban. Egy névterület értelmezés alapján neveknek a kollekciója, melyet egy URI (Universal Resource Identifier) azonosít, tehát egyedi.

A névterületek használata lehetővé teszi a névkonfliktusok elkerülését két azonos elnevezésű és különböző célú elem egyazon XML dokumentumban való használata esetén. Név konfliktusok számos okból fordulhatnak elő, elsősorban azért, mert az XML nem tesz megkötést az elemek neveire vonatkozóan. Könnyen előfordulhat, hogy amikor két kölönböző forrásból származó XML dokumentumból egy új XML dokumentumot hozunk létre, akkor az új dokumentumban név konfliktusok adódnak.

Név konfliktus

Legyen egy cég részlegeinek és azok alkalmazottainak az adatai XML dokumentum formájában:

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<RESZLEGEK>
 <RESZLEG>
  <NEV>Tervezés</NEV>
  <CIM>
   <VAROS>Kolozsvár</VAROS>
   <UTCASZAM>Bolyai utca 5</UTCASZAM>
  </CIM>
  <ALKALMAZOTTAK>
   <ALKALMAZOTT>
    <NEV>Szabó János</NEV>
    <TELEFON>413476</TELEFON >
    <CIM>
          <VAROS>Kolozsvár</VAROS>
     <UTCASZAM>Béke tér 3</UTCASZAM>
    </CIM>
    <EMAIL>jani76@hotmail.com</EMAIL>
   </ALKALMAZOTT>
   <ALKALMAZOTT>
    <NEV>Kiss István</NEV>
    <TELEFON>405406</TELEFON >
    <CIM>
     <VAROS>Kolozsvár</VAROS>
          <UTCASZAM>Bucegi utca 4</UTCASZAM>
    </CIM>
    <EMAIL>isti91@hotmail.com</EMAIL>
   </ALKALMAZOTT>
  </ALKALMAZOTTAK>
 </RESZLEG>
 <RESZLEG>
  <NEV>BESZERZES</NEV>
  <CIM>
   <VAROS>Temesvár</VAROS>
   <UTCASZAM>Petőfi utca 15</UTCASZAM>
  </CIM>
  <ALKALMAZOTTAK>
   <ALKALMAZOTT>
    <NEV>Nagy Anna</NEV>
    <TELEFON>113178</TELEFON>
    <CIM>
          <VAROS>Temesvár</VAROS>
     <UTCASZAM>Horea utca 44</UTCASZAM>
    </CIM>
        <EMAIL>ani87@yahoo.com</EMAIL>
   </ALKALMAZOTT>
  </ALKALMAZOTTAK>
 </RESZLEG>
</RESZLEGEK>
```

A fenti példában a részleg és az alkalmazott is rendelkezik <NEV> és <CIM> elemekkel, ez pedig név konfliktushoz vezet.

Megoldás név előtéttel (prefix)

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<RESZLEGEK>
 <reszl:RESZLEG>
  <reszl:NEV>Tervezés</reszl:NEV>
  <reszl:CIM>
   <reszl:VAROS>Kolozsvár</VAROS>
        <reszl:UTCASZAM>Bolyai utca 5</UTCASZAM>
  </reszl:CIM>
```

```
  <alk:ALKALMAZOTTAK>
   <alk:ALKALMAZOTT>
    <alk:NEV>Szabó János</alk:NEV>
    <alk:TELEFON>413476</alk:TELEFON >
    <alk:CIM>
     <alk:VAROS>Kolozsvár</alk:VAROS>
     <alk:UTCASZAM>Béke tér 3</alk:UTCASZAM>
    </alk:CIM>
    <alk:EMAIL>jani76@hotmail.com</alk:EMAIL>
   </alk:ALKALMAZOTT>
   <alk:ALKALMAZOTT>
    <alk:NEV>Kiss István</alk:NEV>
    <alk:TELEFON>405406</alk:TELEFON >
    <alk:CIM>
     <alk:VAROS>Kolozsvár</alk:VAROS>
     <alk:UTCASZAM>Bucegi utca 4</alk:UTCASZAM>
    </alk:CIM>
    <alk:EMAIL>isti91@hotmail.com</alk:EMAIL>
   </alk:ALKALMAZOTT>
  </alk:ALKALMAZOTTAK>
 </reszl:RESZLEG>
 <reszl:RESZLEG>
  <reszl:NEV>BESZERZES</reszl:NEV>
  <reszl:CIM>
   <reszl:VAROS>Temesvár</reszl:VAROS>
   <reszl:UTCASZAM>Petőfi utca 15</reszl:UTCASZAM>
  </reszl:CIM>
  <alk:ALKALMAZOTTAK>
   <alk:ALKALMAZOTT>
    <alk:NEV>Nagy Anna</alk:NEV>
    <alk:TELEFON>113178</alk:TELEFON >
    <alk:CIM>
     <alk:VAROS>Temesvár</alk:VAROS>
     <alk:UTCASZAM>Horea utca 44</alk:UTCASZAM>
    </alk:CIM>
    <alk:EMAIL>ani87@yahoo.com</alk:EMAIL>
   </alk:ALKALMAZOTT>
  </alk:ALKALMAZOTTAK>
 </reszl:RESZLEG>
</reszl:RESZLEGEK>
```

A dokumentum különböző részeit különböző előtéttel kiegészítve biztosítjuk a név egyediségét.

Megoldás névterület segítségével

A névterület használata annyiban különbözik az előtét használatától, hogy az előtét első előfordulásakor definiálni kell, hogy az egy névterület és hozzá kell rendelni egy egyedi névterület nevet.

```
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<reszl:RESZLEGEK xmlns:reszl="http://www.cs.ubbcluj.ro/reszl.html">
 <reszl:RESZLEG>
  <reszl:NEV>Tervezés</reszl:NEV>
  <reszl:CIM>
   <reszl:VAROS>Kolozsvár</VAROS>
        <reszl:UTCASZAM>Bolyai utca 5</UTCASZAM>
  </reszl:CIM>
  <alk:ALKALMAZOTTAK xmlns="http://www.cs.ubbcluj.ro/alk.html">
   <alk:ALKALMAZOTT>
    <alk:NEV>Szabó János</alk:NEV>
    <alk:TELEFON>413476</alk:TELEFON >
    <alk:CIM>
```

```
      <alk:VAROS>Kolozsvár</alk:VAROS>
      <alk:UTCASZAM>Béke tér 3</alk:UTCASZAM>
     </alk:CIM>
     <alk:EMAIL>jani76@hotmail.com</alk:EMAIL>
    </alk:ALKALMAZOTT>
    <alk:ALKALMAZOTT>
     <alk:NEV>Kiss István</alk:NEV>
     <alk:TELEFON>405406</alk:TELEFON >
     <alk:CIM>
      <alk:VAROS>Kolozsvár</alk:VAROS>
      <alk:UTCASZAM>Bucegi utca 4</alk:UTCASZAM>
     </alk:CIM>
     <alk:EMAIL>isti91@hotmail.com</alk:EMAIL>
    </alk:ALKALMAZOTT>
   </alk:ALKALMAZOTTAK>
  </reszl:RESZLEG>
  <reszl:RESZLEG>
   <reszl:NEV>BESZERZES</reszl:NEV>
   <reszl:CIM>
    <reszl:VAROS>Temesvár</reszl:VAROS>
    <reszl:UTCASZAM>Petőfi utca 15</reszl:UTCASZAM>
   </reszl:CIM>
   <alk:ALKALMAZOTTAK xmlns="http://www.cs.ubbcluj.ro/alk.html">
    <alk:ALKALMAZOTT>
     <alk:NEV>Nagy Anna</alk:NEV>
     <alk:TELEFON>113178</alk:TELEFON >
     <alk:CIM>
      <alk:VAROS>Temesvár</alk:VAROS>
      <alk:UTCASZAM>Horea utca 44</alk:UTCASZAM>
     </alk:CIM>
     <alk:EMAIL>ani87@yahoo.com</alk:EMAIL>
    </alk:ALKALMAZOTT>
   </alk:ALKALMAZOTTAK>
  </reszl:RESZLEG>
</reszl:RESZLEGEK>
```

Két névterületet definiáltunk, melyeknek egy-egy URL-t adtunk meg. Az XML értelmező ezt az URL-t nem használja, csak egy egyedi névnek tekinti. Sok esetben az XML készítői az URL által hivatkozott oldalon helyeznek el információt az adott névterületet illetően, de előfordul, hogy valamely technológia (pl. XSLT, XHTML) névterületének nevét a technológia értelmezője felhasználja a technológia azonosítására.

14.4.5. **Elemek** (Elements)

Az elemek a legalapvetőbb részei az XML dokumentumoknak. A `<NEV>Szabó János</NEV>` részlet egy konkrét elem.

Egy XML elem általában a következő részeket tartalmazza:

- Az elemet bevezető `<ElemCimke>` tag, amit csúcsos zárójelek között kell megadni.
- Az elemhez tartozó adat (Pl. `Szabó János`). Nem minden elem tartalmaz adatot.
- Az elemet záró `</ElemCimke>` tag.

A nyitó és záró tag-ek feladata az, hogy megjelöljék és elnevezzék az általuk közrefogott adatot. Ezzel az adataink természete és tartalma is nyomonkövethetővé válik, emiatt fontos, hogy a címke nevek kifejezők legyenek. Lényeges kiemelni, hogy a hagyományos XML meghatározás nem ad olyan precíz adatábrázoló és kezelő eszközt a kezünkbe, mint a programozásban megismert típus fogalma, azonban már létezik olyan W3C ajánlás (XML Schema definíció), ami ennek a szigorú követelménynek is eleget tesz.

Időnként előfordul, hogy egy tag nem fog közre semmilyen adatot. Erre példa az, ha valakinek nincs e-mail címe: <EMAIL></EMAIL>. Ezt XML-ben a következő módon rövidíthetjük: <EMAIL/>. Az elem nevekre vonatkozó szabályokat láttuk a 14.4.3-ban.

Az elemek kezdetét és végét határoló tagok azt mutatják, hogy az elem tartalma milyen típusú. Az XML dokumentumokban megjelenő elemek valójában (nem feltétlenül különböző) tartalommal rendelkező adat-előfordulások. Ezért lényeges, hogy elemtípus névről és ne tagnévről, vagy elemnévről beszéljünk.

A fenti példa-dokumentum szerkezetét egy olyan fával reprezentálhatjuk, amelynek gyökere a RESZLEGEK nevet viseli, levelei pedig a részleg neve, városa, utcája, a részleg minden alkalmazotta esetén annak neve, telefonszáma, városa, utcája és e-mail címe. A fa gyökerével reprezentált elemet a dokumentum gyökéreleménének nevezzük, de szokásos a dokumentumelem elnevezés is.

Minden XML dokumentumnak *jól formázott*nak (well-formated) kell lennie, hogy használhatók és helyesen értelmezhetők legyenek. Jól formázott az a dokumentum, amelynél minden megnyitott elem le is van zárva, nem tartalmaz a sorrendből kiágazó elemeket és szintaktikusan megfelel a specifikációnak. Általános szabály, hogy egy *jól formázott* XML dokumentumban az elemek nem nyúlhatnak egymásba, minden elemnek kell kezdő- és záró tag-jének lennie, továbbá, hogy a dokumentumelem kivételével minden elemhez tartozzék befogadó elem: más szavakkal egy XML dokumentum mindig reprezentálható legyen egy faszerkezettel, ahol a csomópontok az elemeket reprezentálják. A valós világ dokumentumaira erős megszorítás az, hogy szerkezetük fával legyen reprezentálható. Az XML dokumentumok világában a jól formáltság ezzel szemben (nem túl erős) követelmény. Az XML dokumentumokra előírt jól formáltság látszólag igen sok valós életbeli dokumentumot zár ki az XML dokumentumok világából, de csak látszólag. A nyelvi elemek részletes megismerésekor látni fogjuk, hogy a faszerkezetnél bonyolultabb szerkezettel leírható dokumentumok is előállíthatók XML dokumentumként.

14.4.6. XML és a félig-struktúrált adatmodell

A féligstruktúrált adatot át tudjuk írni XML dokumentummá a következő *T* függvény segítségével, ha a féligstruktúrált adat OEM adatgráfja fa:

T (atomiérték) = atomiérték;
T ({e_1:v_1,…, e_n:v_n}) = <e_1> *T* (v_1) </e_1>…<e_n> *T* (v_n) </e_n>.

14.17. példa: Legyen a következő féligstruktúrált adat:

```
{nev: {kereszt: "János", vezet: "Szabó"}, tel: 413476,
 email: "jani76@hotmail.com"}
```

Alkalmazva a *T* függvényt:

```
T ({nev: {kereszt: "János", vezet: "Szabó"}, tel: 413476,
 email:"jani76@hotmail.com"}) =
<nev> T ({kereszt: "János", vezet: "Szabó"}) </nev>
<tel>413476</tel>
<email>jani76@hotmail.com</email> =

<nev>
  <kereszt>János</kereszt>
  <vezet>Szabó</vezet>
</nev>
<tel>413476</tel>
<email>jani76@hotmail.com</email> □
```

14.4.7. XML gráf modellje

Láttuk, hogy a félig-struktúrált adatot , mely (címke, részfa) párok halmazaként jelenik meg, él-címkézett gráffal ábrázoltuk. Az XML adat ábrázolása esetén a címkék a gráf csúcsaiba kerülnek.

14.4.8. Karakterkészlet és kódrendszerek

A kezdeti számítógépes környezetekben a karakterkészlet elemeire különböző kódrendszereket használtak. Ezek között a legismertebbek az IBM által bevezetett EBCDIC, a hozzá nagyon hasonló kártyakód, illetőleg az ISO által rögzített ASCII. Ezek a kódrendszerek 7-8 bitet használtak egy-egy karakter digitális kódjához (ASCII 7 bit – 128 karakter, EBCDIC, kártyakód 8 bit – 256 karakter).

Napjainkra természetes elvárás lett a legkülönbözőbb természetes nyelvekhez tartozó jelek (jelrendszerek) hozzávétele az elfogadható karakterek halmazához (ékezetes betűk, az arab nyelv betűi, cirill betűk, az ázsiai képi jelek, stb.). Ezek együttes egymás melletti jelenléte szükségessé tette egy új kódrendszer az UTF-8 (Unicode) kidolgozását. Ez egy 16 bites kódrendszer, amely természetes kiterjesztése az ASCII 7 bites kódrendszernek. Az XML a Unicode kódrendszert és az általa reprezentált karakterkészletet használja.

14.4.9. Megjegyzések

Az XML dokumentumban a HTML nyelvben megszokott megjegyzést használhatjuk.

```
<!--
Ez egy XML megjegyzés
-->
```

A megjegyzéseket a `<!--` négy karakterből álló összetett szimbólum vezeti be és a következő `-->` három karakterből álló összetett szimbólum zárja le. Megszorítás a megjegyzés szövegére, hogy ne tartalmazzon két egymást követő kötőjel karaktert. Ezeket a részeket az XML feldolgozó (elemző, érvényesség vizsgáló, megjelenítő) alkalmazások figyelmen kívül hagyják.

14.4.10. Tulajdonságok (attributes)

Az XML elemek tetszőleges számú tulajdonsággal (attribútummal) is rendelkezhetnek. A `SZEMELYEK` XML példában tekintsük a következő részletet:

```
<SZEMELY sorszam="1" nev="Szabó János">
```

Itt két tulajdonságot adtunk meg a `SZEMELY` elem részére: `sorszam` és `nev`.
A tulajdonságok általában így néznek ki:

```
<element nev1="érték1" nev2="érték2" ...  >
  …
</element>
```

XML-ben az attribútumok értékeinek idézőjel között kell lenniük. Az idézőjel lehet egyszeres és kettős időzőjel.

A tulajdonságok hasonló szerepet töltenek be, mint az adott elem gyermek elemei: valamilyen információval bővítik annak jelentéstartalmát. Láttuk a 14.15 és 14.16 példa esetén a NEV szerepelt elemként, illetve tulajdonságként. Nincs előírás mikor kell egy adott adat tárolására gyermek elemet és mikor tulajdonságot használni.
Különbségek az elemek és attribútumaik között:

- Egy adott attribútum egy elemen belül csak egyszer jelenhet meg, míg egy elemen belül gyerek elemek ismétlődhetnek.
- Az attribútumoknak nincsenek „elemeik” (nincsenek egymásba ágyazott attribútumok)
- Az attribútumok neveinek csak egy elemen belül kell egyedieknek lenniük.
- Az attribútumok nem bővíthetők olyan egyszerűen, mint a gyermek elemek.
- Az attribútumok adatai nem rendezhetők struktúrákba (a gyermek adatai igen).
- Az attribútumokat nehezebb kezelni a programokban.
- Az attribútumok helyességét nehezebb ellenőrizni.

Tulajdonságokat akkor érdemes használni, ha azok nem az adatok, hanem az adatokat feldolgozó alkalmazások szempontjából jelentenek plusz információt. Ha az adat fontos, akkor elem formájában érdemes megadni.

A következő példában nem lényeges a kép neve, sem a típusa, csak maga a kép, a megjelenítéséhez viszont a megjelenítést végző alkalmazásnak szüksége van ezekre az információkra, ezért tulajdonság formájában adjuk meg:

```
<image type="gif" name="picture.gif"/>
```

Egy más példa arra, mikor érdemes a tulajdonságokat használni, ha az adat rekord halmazban az egyes rekordokat egyedi azonosítóval kell ellátni. Ekkor az azonosító a felhasználó szempontjából nem lényeges, azt csak a feldolgozó alkalmazás használja.

14.4.11. Feldolgozási utasítások

A feldolgozási utasítások nem részei az XML-nek. Szerepük a fordítóprogramoknak küldött direktívákhoz hasonlíthatók. Ezek, a megjegyzésekhez hasonlóan, nem részei a dokumentum szöveges tartalmának, de követelmény az XML feldolgozókkal szemben, hogy továbbadják ezeket az utasításokat az alkalmazások számára.

Megadási módjuk: `<?target name?>`. Egy alkalmazás csak azokat az utasításokat veszi figyelembe, amelynek a célját felismeri, az összes többit figyelmen kívül hagyja. A célt (target) követő adatok az alkalmazásnak szólnak, megadásuk opcionális.

14.4.12. Dokumentum séma definició

A másik alapvető fogalom, mely az XML dokumentumokra vonatkozik az, hogy a dokumentumok *érvényes*ek (valid) lehetnek, de nem kell feltétlenül érvényesnek lenniük. Érvényes dokumentum az, amely megfelel a dokumentum séma definíciójának.

Egy XML dokumentum tetszőleges elemeket és tulajdonságokat tartalmazhat, tetszőleges felépítéssel, amelyeket leírva megkapjuk a dokumentum séma definícióját. A dokumentum séma definícióját többek között arra használhatjuk, hogy ellenőrizzük a különböző forrásból származó XML dokumentumok azonos típusúak, felhasználhatóak-e ugyanabban az alkalmazásban. Felhasználható arra, hogy egy készítendő dokumentumnak milyen szabályokat kell betartania, hogy egy adott alkalmazásban felhasználható legyen. Ezen kívül a dokumentum érvényességének a vizsgálatára is használható.

Több séma leíró módszer is létezik. A legelterjedtebb a DTD (Document Type Definition), szabványos, hátránya azonban, hogy nem XML alapú, és valószínű a jövőben nem fogják használni.

XML alapú az XSD (XML Schema Definiton), mely hivatalosan is ajánlott, mint dokumentum típus leíró nyelv az XML dokumentumokhoz.

DTD (Document Type Definition)

Egy DTD meghatározza a nyelvtant és az elem-készletet az adott XML-formázáshoz. Ha egy dokumentum megfelel egy DTD-nek és követi ennek a DTD-nek a szabályait, akkor érvényes XML dokumentumnak nevezzük.

Az XML dokumentumokra sémával is érvényesíthetők szabályok. Ez az XML formátum meghatározásának egy módja, amely helyettesíteni fogja a DTD-t. Amikor egy dokumentum megfelel egy sémának, akkor séma-érvényesnek nevezhetjük.

A DTD a dokumentum típus definíció szavak rövidítése. Egy DTD egy XML dokumentumra vonatkozó szabálykészletet állít fel. A DTD önmagában nem egy specifikáció, hanem az XML specifikáció részeként szerepel. Egy XML dokumentumban egy dokumentum típus deklaráció tartalmazhat jelölő szabályokat és hivatkozhat egy jelölő szabályokat tartalmazó külső dokumentumra is. E két szabálykészlet együttese a dokumentum típus definíció.

Egy fejlesztő vagy tartalomszerző a DTD-t is olyan dokumentumként hozza létre, amelyre az XML állományaiban hivatkozik, vagy magában az XML állományban helyezi el, tehát semmilyen módon nem korlátozza az XML dokumentumokat. Valójában a DTD az, ami az XML adatok hordozhatóságát biztosítja. A DTD teszi lehetővé, hogy az XML állományt fogadó másik alkalmazás felismerje, hogyan kell a kapott állományt feldolgozni, és hogyan kell benne keresni. A

DTD adja a hordozhatóságot egy XML dokumentum kiterjeszthetőségéhez, ami azután nemcsak az adatok hajlékonyságát eredményezi, hanem olyan adatokat, amelyek bármely gépen feldolgozhatók és érvényesíthetők, amelyek el tudják érni a dokumentum DTD-jét.

Elemek deklarálása DTD-ben

A `!DOCTYPE` kulcsszó vezeti be a dokumentum típusának a deklarációját, mely után közvetlen a gyökér elem nevének kell következnie. Ezután következnek a gyermek elemek.

Legyen ismét az 14.15 példabeli XML dokumentumunk, amely személy elemeket tartalmaz, egy ilyen személy elem:

```
<SZEMELYEK>
     <SZEMELY>
          <NEV>Szabó János</NEV>
          <TELEFON>413476</TELEFON >
          <EMAIL>jani76@hotmail.com</EMAIL>
     </SZEMELY>
</SZEMELYEK>
```

Egy megfelelő DTD lehetne:

```
<!DOCTYPE SZEMELYEK [
     <!ELEMENT SZEMELYEK (SZEMELY*)>
     <!ELEMENT SZEMELY (NEV,TELEFON,EMAIL)>
     <!ELEMENT NEV (#PCDATA)>
     <!ELEMENT TELEFON (#PCDATA)>
     <!ELEMENT EMAIL (#PCDATA)>
] >
```

Az első sor adja meg a gyökérelemet. (`<SZEMELYEK>`). A következő sor jelzi, hogy a `<SZEMELYEK>` tetszőleges számú `<SZEMELY>` elemet tartalmazhat. Mindegyik `<SZEMELY>` elem tartalmazza a `<NEV>`, `<TELEFON>`, `<EMAIL>` elemeket, melyek elemzett karakteres adatot (`#PCDATA` - Parsed Character Data) tartalmaznak.

A levél elemek tartalmazhatnak nem elemzett karakteres adatot is, melyet a `#CDATA` (Character Data) kulcsszóval adunk meg és interpretálás nélkül adódik át az XML adatfolyamot fogadó alkalmazásnak. A következő formában adhatjuk meg az adatot az XML dokumentumban:

```
<![CDATA [Ez egy tetszoleges karaktersor]]>
```

A DTD, mint nyelvtan

Legyen a következő példa:

14.18. példa: Egy komplexebb példa, mely postai címeket tartalmaz:

```
<?xml version="1.0"?>
<ab><cim><szemely>
<cimzes>Dr.</cimzes>
               <csaladnev>Kelemen</csaladnev>
               <keresztnev>Imre</keresztnev>
          </szemely>
          <utca>Tavasz 12</utca>
          <varos>Kolozsvar</varos>
          <megye>Kolozs</megye>
          <iranyitoszam>400231</iranyitoszam>
     </cim>
     ...
</ab>
```

Ennek a DTD-je a következő lenne:

```
<!DOCTYPE ab [
 <!ELEMENT cim (szemely,utca,varos,megye,iranyitoszam)*>
 <!ELEMENT szemely (cimzes?,csaladnev,keresztnev+)>
 <!ELEMENT cimzes (#PCDATA)>
 <!ELEMENT csaladnev (#PCDATA)>
 <!ELEMENT keresztnev (#PCDATA)>
]>
```

Az első sor adja meg, hogy a gyökérelem az <ab>. A következő öt sor jelölő deklarációk, melyek jelzik, hogy az <ab> tetszőleges számú <cim> elemet tartalmazhat, mindegyik tartalmazza a <szemely>, <utca>, <varos>, <megye>, <iranyitoszam> elemeket. A <szemely> további három elemből áll, a többi csak karakteres adatot (parsed character data) tartalmaz. Itt a

```
cim (szemely, utca, varos, megye, iranyitoszam)*
```

egy k* alakú reguláris kifejezés, amely tetszőleges számú, adott elemekből felépített cím elemet jelöl. □

Használható reguláris kifejezések:

- k* (tetszőleges számú előfordulás)
- k+ (legalább egy előfordulás)
- k? (egy vagy egysem),
- k | k′ (egyik a kettőből)
- k,k′ (összetevés).

A DTD valójában egy nyelvtan a dokumentum számára. A fenti példában a DTD megköveteli, hogy a <szemely>, <utca>, <varos>, <megye>, <iranyitoszam> a megadott sorrendben jelenjenek meg a <cim> elemben. A nyelvtanok lehetnek rekurzívek is, mint például a következő, bináris fát leíró DTD-ben:

```
<!DOCTYPE csucs [
<!ELEMENT csucs ( level | (csucs,csucs) )>
<!ELEMENT level (#PCDATA)>
]>
```

A fentire nyelvtanra egy XML dokumentum:

```
<csucs>
     <csucs>
          <csucs> <level> 1 </level> </csucs>
          <csucs> <level> 2 </level> </csucs>
     </csucs>
     <csucs>
          <level> 3 </level>
     </csucs>
</csucs>
```

14.19. példa: Legyen egy relációs adatbázis séma:

```
r1(a, b, c);
r2(c,d).
```

Láttuk a sorokat a 14.1.2 –ben. Egy XML reprezentációja a következő:

```
<db> <r1><a> a1 </a> <b> b1 </b> <c> c1 </c> </r1>
<r1><a> a2 </a> <b> b2 </b> <c> c2 </c> </r1>
<r2><c> c2 </c> <d> d2 </d> </r2>
<r2><c> c3 </c> <d> d3 </d> </r2>
<r2><c> c4 </c> <d> d4 </d> </r2>
</db>
```

Egy DTD a fenti XML adatfolyamnak:

```
<!DOCTYPE db [
 <!ELEMENT db (r1*, r2*)>
 <!ELEMENT r1 (a,b,c)>
 <!ELEMENT r2 (c,d)>
```

```
 <!ELEMENT a (#PCDATA)>
 <!ELEMENT b (#PCDATA)>
 <!ELEMENT c (#PCDATA)>
 <!ELEMENT d (#PCDATA)>
]>
```

Ezzel a DTD-vel az `r1` komponensei az `a, b` és `c`, illetve az `r2` komponensei a `c` és `d`, mégpedig pontosan ebben a sorrendben. Elméletileg a relációs adatmodell esetén az oszlopok sorrendje nem számít, tehát a következő forma is megfelel:

```
<!ELEMENT r1 ((a,b,c)|(a,c,b)|(b,a,c)|(b,c,a)|(c,a,b)|(c,b,a)>
<!ELEMENT r2 ((c,d)|(d,c))>
```

A fenti formában adott DTD esetén először kell az `r1` elemeknek megjelenniük és utána jöhetnek csak az `r2` elemek. A következő forma megengedi, hogy az `r1` elemek keveredjenek az `r2` elemekkel.

```
<!ELEMENT db ((r1|r2)*)> □
```

Eddig a DTD-t magában az XML dokumentumban írtuk le, de lehetőségünk van arra is, hogy egy külön állományban tároljuk, mondjuk a `sema.dtd` –ben, és akkor a következőképpen hivatkozunk rá:

```
<!DOCTYPE db SYSTEM "sema.dtd">
```

A séma egy különböző URL-en is lehet, és akkor a hivatkozás:

```
<!DOCTYPE db SYSTEM "http://www.schemaautorithy.com/sema.dtd">
```

Ez lehetővé teszi, hogy több web-oldal is ugyanazt a közös sémát használja, megkönnyítve így az adatcserét.

Tulajdonságok deklarálása a DTD-ben

A DTD-ben lehetőség van a tulajdonságok (attribútumok) leírására a következőképpen:

```
<!ATTLIST elem-név
attribútum-név1 attributum-típus1 jelző
...
attribútum-név-n attributum-típus1 jelző
>
```

14.20. példa:

```
<!ATTLIST SZEMELY sorszam CDATA #REQUIRED>
```

A példa azt tudatja az elemző programmal, hogy a SZEMELY elemnek van egy sorszám nevű tulajdonsága, mely egy tetszőleges karakter sor (CDATA) és kötelezően szerepelnie kell az elem előfordulásában (#REQUIRED). □

Az elem-név annak az elemnek a neve, amelyhez a tulajdonság tartozik. A tulajdonság típus lehetséges értékeit a következő táblázat adja meg:

Típus	**Jelentés**
CDATA	Nem ellenőrzött karakter sor
`(en1|en2|...)`	Az érték egy számozott lista valamely eleme lehet
ID	Az érték egy egyedi azonosító
IDREF	Az érték egy másik elem egyedi azonosítója
IDREFS	Más elemek egyedi azonosítóinak listája
NMTOKEN	Az érték egy érvényes XML név
NMTOKENS	Az érték érvényes XML nevek listája
ENTITY	Az érték egy egyed
ENTITIES	Az érték egyedek listája
NOTATION	Az érték egy megjegyzés neve
Xml:	Az érték előre definiált xml: érték

A jelző lehetséges értékei:
- #REQUIRED: Az attribútumot kötelezően, explicit módon meg kell adni az elem minden előfordulása esetén.
- #IMPLIED: Az attribútum megadása nem kötelező. Ha nincs érték megadva, az elemző tovább halad.
- Egy érték: Ha megadunk egy értéket, az lesz az attribútum alapértelmezett értéke.
- #FIXED „érték”: Az attribútum nem kötelező, de ha megadjuk, akkor az itt megadott értéknek kell lennie.

Egyedek (ENTITIES)

DTD-ben az egyedek hasonlítanak a C nyelv #include, illetve #define lehetőségeire, segítségükkel előre definiált vagy külső erőforrásban levő adatokat (szövegek, XML dokumentum részletek, stb.) újra felhasználhatunk.

Minden egyednek egy egyedi névvel kell rendelkeznie. Legyen az „EgyedNev” egy egyednév.. A dokumentumban „&EgyedNev;” („&” jellel kezdődik és „;”-vel fejeződik be) formájában adjuk meg, viszont az egyedhez rendelt karakter fog megjelenni a dokumentumban.
Léteznek előre definiált egyedek, mint például a „gt” nevű. Az „>” hatása olyan, mintha az XML adatfolyamba egyből a „>” jelet írtuk volna. Az egyedek fogalma lehetővé teszi azt is, hogy tetszőleges unicode karaktert helyezzünk el a szövegbe. Ekkor az egyednév „#szám” vagy „#xszám” alakú, ahol az első a decimális, a második a hexadecimális megadási mód. A kis „o” hexa unicode kódja 0x0151. Ezt mint egyedet a következő módon hívhatjuk meg: „ő”.

Előre definiált egyed	**Egyed név**
< (a kisebb jel)	Lt
> (a nagyobb jel)	Gt
& (AND jel)	Amp
’ (egyszeres idézőjel)	Apos
” (dupla idézőjel)	Quot

Az XML ajánlás lehetővé teszi, hogy saját egyedeket is létrehozzunk. Ez az `<!ENTITY myEgyed "az egyedhez rendelt szöveg">` szintaxis-sal lehetséges, amit „&myEgyed;”-vel hívhatunk meg és az a hatása, mintha közvetlenül az „Érték” karakterfűzért írtuk volna le.
Legyen a következő egyed:

```
<!ENTITY DOM "Document Object Model">
```

A példa esetén, ahol az XML dokumentumban az &DOM; karakter sor található, ott a Document Object Model szöveg jelenik meg.
A paraméter egyedeket csak a DTD-ben használhatjuk, az XML dokumentumban nem. A DTD-ben annak megkönnyítésére használják.

```
<!ENTITY % paraméter-egyed-név "az egyedhez rendelt szöveg">
```

Hivatkozni %paraméter-egyed-név; utasítással lehet egy paraméter egyedre.

14.21. példa: Legyen a következő két egyed:

```
<!ELEMENT Novella (Bevezető, Fejezet+, Befejezés)>
<!ELEMENT Regény (Bevezető, Fejezet+, Befejezés)>
```

Kihasználva a hasonlóságot, létrehozunk egy paraméter egyedet:

```
<!ENTITY % TARTALOM " Bevezető, Fejezet+, Befejezés">
```

Az így kapott paraméter egyeddel a `NOVELLA` és `REGENY` elemek nyelvtani szabályai a következők lesznek:

```
<!ELEMENT Novella (%TARTALOM)>
<!ELEMENT Regény (%TARTALOM)> □
```

A belső egyedeken kívül léteznek külső egyedek is, melyek lehetővé teszik, hogy egy másik állomány tartalmára hivatkozzunk az XML dokumentumból. A külső állomány tartalmazhat szöveges adatot, illetve bináris adatot. Ha a külső file szöveges adatot tartalmaz, akkor az az eredeti dokumentumba illesztődik a referencia helyére és úgy kerül feldolgozásra, mintha a hivatkozó dokumentum része lenne. A bináris adat nem kerül értelmezésre.

Külső egyed esetén az egyedhez rendelt szöveg helyett egy erőforrás (URI/URL) mutat a bemásolandó adatra.

```
<!ENTITY pelda SYSTEM "http://www.cs.ubbcluj.ro/pelda.xml">
```

Legyen például a következő XML dokumentum termékekről:

```
<termek>
     <nev nyelv="Magyar" reszleg="zene"> fuvola </nev>
     <ar penznem="Euro"> 420.12 </ar>
</termek>
```

A DTD-ben az elemekre vonatkozó szabályok mellett a következőket is megadjuk:

```
<!ATTLIST nev nyelv CDATA #REQUIRED
          reszleg CDATA #IMPLIED>
<!ATTLIST ar penznem CDATA #IMPLIED>
```

A deklaráció szerint a `<nev>` -nek két tulajdonsága van, a `nyelv` és a `reszleg`, az első kötelező (`#REQUIRED`) és a második opcionális (`#IMPLIED`). Az attribútumnak adhatunk egy értéket is (ekkor ez alapértelmezett érték lesz), vagy használhatjuk még a `#FIXED 'érték'` szintaxist, ekkor nem kötelező az attribútum, de ha megadjuk, ennek az értéknek kell lennie. Továbbá az `<ar>` a `penznem` tulajdonsággal rendelkezik, ami opcionális. Mindegyik tulajdonság CDATA típusú, mondhatjuk, hogy idézőjelezett karaktersor.

Külön jelentőséggel bírnak az ID, IDREF és IDREFS típusú deklarációk. Az ID típus azt jelenti, hogy az illető atrribútum az elem egyedi azonosítóját tartalmazza. Az ID egy karaktersor. Ennek párja, az IDREF típus azt jelenti, hogy az attribútum értéke egy másik elem azonosítója (ID-je); IDREFS pedig azonosítók listájára vonatkozik. Ezen tulajdonságok segítségével az XML dokumentumnak gráf szerkezetet adhatunk, melynek nem kell feltétlenül fának lennie. Amint látjuk ezen típusú tulajdonságokkal megszorításokat adhatunk meg az XML adatokra, mégpedig egyedi, illetve hivatkozási épség megszorítást. A relációs adatmodell megszorításaihoz hasonlítva, az ID elsődleges kulcs, az IDREF pedig külső kulcs, de a következő eltérésekkel:

- Az ID típusú tulajdonság csakis karaktersor lehet, nincs rá lehetőség, hogy más típust deklaráljunk neki.
- Nincs lehetőség összetett kulcsok deklarálására, tehát ID típusa csak egy tulajdonságnak lehet.
- Ha egy XML dokumentumban több ID típusú tulajdonságot deklarálunk, mindegyik értéke egyedi lesz az egész dokumentumban.
- Az IDREF típusú tulajdonság esetén nem tudjuk megadni, hogy a dokumentumban létező, melyik ID típusú tulajdonságra hivatkozik, csak annyit ellenőriz a rendszer, hogy az IDREF értéke legyen egyenlő egyik ID típusú attribútum értékével.
- IDREF típusú hivatkozás csakis dokumentumon belül lehetséges, dokumentumok között nem.

14.22. példa: A hivatkozások használatára egy egyszerű példa:

```
<!DOCTYPE csalad [
     <!ELEMENT csalad (szemely*) >
     <!ELEMENT szemely (nev)>
     <!ELEMENT nev (#PCDATA)>
     <!ATTLIST szemely az ID #REQUIRED
          anya IDREF #IMPLIED
          apa IDREF #IMPLIED
          gyerekek IDREFS #IMPLIED>
]>
```

Egy, ehhez alkalmazkodó egyszerű XML elem:

```
<csalad>
     <szemely az="julia" anya="maria" apa="attila">
          <nev> Kiss Júlia </nev>
     </szemely>
     <szemely az="attila" gyerekek="julia zoltan">
          <nev> Kiss Attila </nev>
     </szemely>
     <szemely az="maria" gyerekek="julia zoltan">
          <nev> Kiss Maria </nev>
     </szemely>
     <szemely az="zoltan" anya="maria" apa="attila">
          <nev> Kiss Attila </nev>
     </szemely>
</csalad> ∈
```

14.23. példa: Egy másik példa egy zenés albumokat leíró DTD. Feltételezzünk, hogy egy előadónak több albuma is van, egy albumon több zeneszám jelenik meg, viszont egy zeneszám csak egy albumon jelenik meg, egy albumon azonos stílusú zeneszámok jelennek meg.

```
<!DOCTYPE AlbumAB[
     <!ELEMENT AlbumAB (Album*, Előadó*, Zeneszám*) >
<!ELEMENT Album (ACíme, Stílus, MegjelenEve) >
<!ELEMENT ACíme (#PCDATA)>
<!ELEMENT Stílus (#PCDATA)>
<!ELEMENT MegjelenEve (#PCDATA)>
<!ATTLIST Album AlbumAzon ID #REQUIRED
          Előadója IDREF #IMPLIED
          Zeneszámai IDREFS #IMPLIED>
<!ELEMENT Előadó (ENév, Nemzetiség) >
<!ELEMENT ENév (#PCDATA)>
<!ELEMENT Nemzetiség (#PCDATA)>
<!ATTLIST Előadó ElőadóAzon ID #REQUIRED
          Albumai IDREFS #IMPLIED>
<!ELEMENT Zeneszám (SzCíme, Időtartam) >
<!ELEMENT SzCíme (#PCDATA)>
<!ELEMENT Időtartam (#PCDATA)>
<!ATTLIST Zeneszám ZeneszámAzon ID #REQUIRED
          Albuma IDREF #IMPLIED>
]>
```

XML adat, mely szerkezetét a fenti séma adja és az Zene.xml állományban van tárolva:

```
<AlbumAB>
 <Album AlbumAzon ="DeadLetters" Előadója="Rasmus"
Zeneszámai="Shadows Guilty">
  <ACíme>Dead Letters</ACíme>
  <Stílus>rock</Stílus>
  <MegjelenEve>2003</MegjelenEve>
 </Album>
 <Előadó ElőadóAzon="Rasmus" Albumai="DeadLetters">
  <ENév>The Rasmus</ENév>
  <Nemzetiség>finn</Nemzetiség>
 </Előadó>
 <Zeneszám ZeneszámAzon="Shadows" Albuma="DeadLetters">
  <SzCíme>In the Shadows</SzCíme>
  <Időtartam>3.35</Időtartam>
 </Zeneszám>
 <Zeneszám ="Guilty" Albuma="DeadLetters">
  <SzCíme>Guilty</SzCíme>
  <Időtartam>3.4</Időtartam>
 </Zeneszám>
</AlbumAB>
```

Tárgyaljuk újra az elsődleges és külső kulcsok problémáját a fenti példa esetén. Amint látjuk, több ID típusú tulajdonság is van az XML dokumentumban: ZeneszámAzon, AlbumAzon, ElőadóAzon, ezek értékei az egész dokumentumban egyedieknek kell leniük, azt jelenti nem lehet például "Shadows" azonosítójú album is és zeneszám is. A zeneszám esetén az Albuma IDREF típusú, de nem tudtuk megadni, hogy ez egy AlbumAzon-nak kell lennie, következésképpen, a rendszer egy Albuma típusú tulajdonság esetén ellenőrzi, hogy a megadott érték egy létező ID típusú tulajdonság értéke-e, mely lehet ZeneszámAzon, AlbumAzon vagy ElőadóAzon is. □

A leírt séma segítségével minden albumról szóló információ csak egyszer kerül tárolásra, hasonlóan a zeneszámok, illetve előadók esetén is. Ugyanezt a sémát megadhatjuk attribútumok nélkül, csak elemekkel, de vigyáznunk kell, hogyan tervezzük meg, ne ismételjünk információt. Ha esetleg a zeneszámot helyezzük a gyökérbe és azon belül gyerek elem az album, egy albumon található minden szám esetén ismételjük az albumról szóló adatokat. Hasonlóan, ha az albumok keretén belül tároljuk az előadót, ismételni fogjuk az előadó információit, az összes zeneszáma, összes albuma esetén. Fontos, hogy ne fordítsuk meg a hierarchiát. Lásd a faszerkezetet a következő példában:

14.24. példa:

```
<!DOCTYPE AlbumAB2[
     <!ELEMENT AlbumAB2 (Előadó*,) >
<!ELEMENT Előadó (Enév, Nemzetiség, KezdoEv, Album*) >
<!ELEMENT Enév (#PCDATA)>
<!ELEMENT Nemzetiség (#PCDATA)>
<!ELEMENT KezdoEv (#PCDATA)>
<!ELEMENT Album (Acíme, Stílus, MegjelenEve, Zeneszám*) >
<!ELEMENT Acíme (#PCDATA)>
<!ELEMENT Stílus (#PCDATA)>
<!ELEMENT MegjelenEve (#PCDATA)>
<!ELEMENT Zeneszám (SzCíme, Időtartam) >
<!ELEMENT SzCíme (#PCDATA)>
<!ELEMENT Időtartam (#PCDATA)>
]>
```

XML adatok, melyek megfelelnek a fenti DTD-nek:

```
<AlbumAB2>
 <Előadó>
  <ENév>Linkin Park</ENév>
  <Nemzetiség>amerikai</Nemzetiség>
  <KezdoEv>2001</KezdoEv>
  <Album>
   <ACíme>Meteora</ACíme>
   <Stílus>hard rock</Stílus>
   <MegjelenEve>2003</MegjelenEve>
   <Zeneszám>
    <SzCíme>Numb</SzCíme>
    <Időtartam>3.06</Időtartam>
   </Zeneszám>
   <Zeneszám>
    <SzCíme>Faint</SzCíme>
    <Időtartam>2.42</Időtartam>
   </Zeneszám>
  </Album>
  <Album>
   <ACíme>Hybrid Theory</ACíme>
   <Stílus>hard rock</Stílus>
   <MegjelenEve>2001</MegjelenEve>
   <Zeneszám>
    <SzCíme>In the End</SzCíme>
    <Időtartam>3.37</Időtartam>
   </Zeneszám>
```

```
   <Zeneszám>
    <SzCíme>Papercut</SzCíme>
    <Időtartam>3</Időtartam>
   </Zeneszám>
  </Album>
 </Előadó>
 <Előadó>
     ⋮
 </Előadó>
</AlbumAB2> ∈
```

A második séma egy fát ír le, viszont az első séma nagyobb szabadságot ad a tervezésre. Az első séma segítségével megengedhetjük, hogy egy albumnak több előadója is legyen, vagy egy szám több albumon is megjelenjen, vagyis az adatok már nem fa szerkezetben, hanem gráf formájában vannak. Az adatok csak egyszer kerülnek tárolásra, de több helyről is hivatkozunk ugyanarra az adatra.

14.25. példa:

```
<!DOCTYPE AlbumAB3[
     <!ELEMENT AlbumAB3 (Album*, Előadó*, Zeneszám*) >
<!ELEMENT Album (ACíme, Stílus, MegjelenEve) >
<!ELEMENT ACíme (#PCDATA)>
<!ELEMENT Stílus (#PCDATA)>
<!ELEMENT MegjelenEve (#PCDATA)>
     <!ATTLIST Album AlbumAzon ID #REQUIRED
          Előadói IDREFS #IMPLIED
          Zeneszámai IDREFS #IMPLIED>
<!ELEMENT Előadó (ENév, Nemzetiség) >
<!ELEMENT ENév (#PCDATA)>
<!ELEMENT Nemzetiség (#PCDATA)>
     <!ATTLIST Előadó ElőadóAzon ID #REQUIRED
          Albumai IDREFS #IMPLIED>
<!ELEMENT Zeneszám (SzCíme, Időtartam) >
<!ELEMENT SzCíme (#PCDATA)>
<!ELEMENT Időtartam (#PCDATA)>
     <!ATTLIST Zeneszám ZeneszámAzon ID #REQUIRED
          Albumai IDREFS #IMPLIED>
]>
```

Az XML séma

Egy másik lehetőség egy XML dokumentum struktúrájának megadására az XML séma. A séma segítségével pontosabban meg lehet határozni a dokumentum struktúráját, és a programozási nyelvekből ismert adattípusokhoz hasonló típusokat is lehet használni. A DTD 10 típust ismer, az XML séma 44-et.

Tetszőleges típus is definiálható a sémában, illetve megszorításokat is értelmezhetünk. Bizonyos szinten objektum-orientált elemeket is tartalmaz, mivel típusokat ki lehet terjeszteni, tehát létező típusokból újak levezethetők, illetve létező típusok leszűkíthetők.

Egy másik előnye, hogy egy bizonyos elem előfordulásának számát is pontosan meg lehet határozni, a `minOccurs` és `maxOccurs` segítségével. Továbbá, az XML séma szintaxisa megegyezik az XML dokumentum szintaxisával, így könnyebb elemezni mint a DTD-t, amelynek saját szintaxisa van.

14.26. példaként tekintsük a következő egyszerű DTD-t, mely könyvekről szóló XML adatok szerkezetét adja meg:

```
<!ELEMENT bibliográfia (könyv+)>
<!ELEMENT könyv (cím, szerző, év, ISBN, kiadó)>
<!ELEMENT cím (#PCDATA)>
<!ELEMENT szerző (#PCDATA)>
<!ELEMENT év (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT kiadó (#PCDATA)>
```

Ugyanezt a szerkezetet XML séma segítségével a következőképpen adhatjuk meg:

```
<?xml version="1.0"?>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.cs.ubbcluj.ro/bibl"
      xmlns="http://www.cs.ubbcluj.ro/bibl"
      elementFormDefault="qualified">
  <xsd:element name="bibliográfia">
   <xsd:complexType>
    <xsd:sequence>
<xsd:element ref="könyv" minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="könyv">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element ref="cím" minOccurs="1" maxOccurs="1"/>
     <xsd:element ref="szerző" minOccurs="1" maxOccurs="1"/>
     <xsd:element ref="év" minOccurs="1" maxOccurs="1"/>
     <xsd:element ref="ISBN" minOccurs="1" maxOccurs="1"/>
     <xsd:element ref="kiadó" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="cím" type="xsd:string"/>
  <xsd:element name="szerző" type="xsd:string"/>
  <xsd:element name="év" type="xsd:integer"/>
  <xsd:element name="ISBN" type="xsd:string"/>
  <xsd:element name="kiadó" type="xsd:string"/>
 </xsd:schema>
```

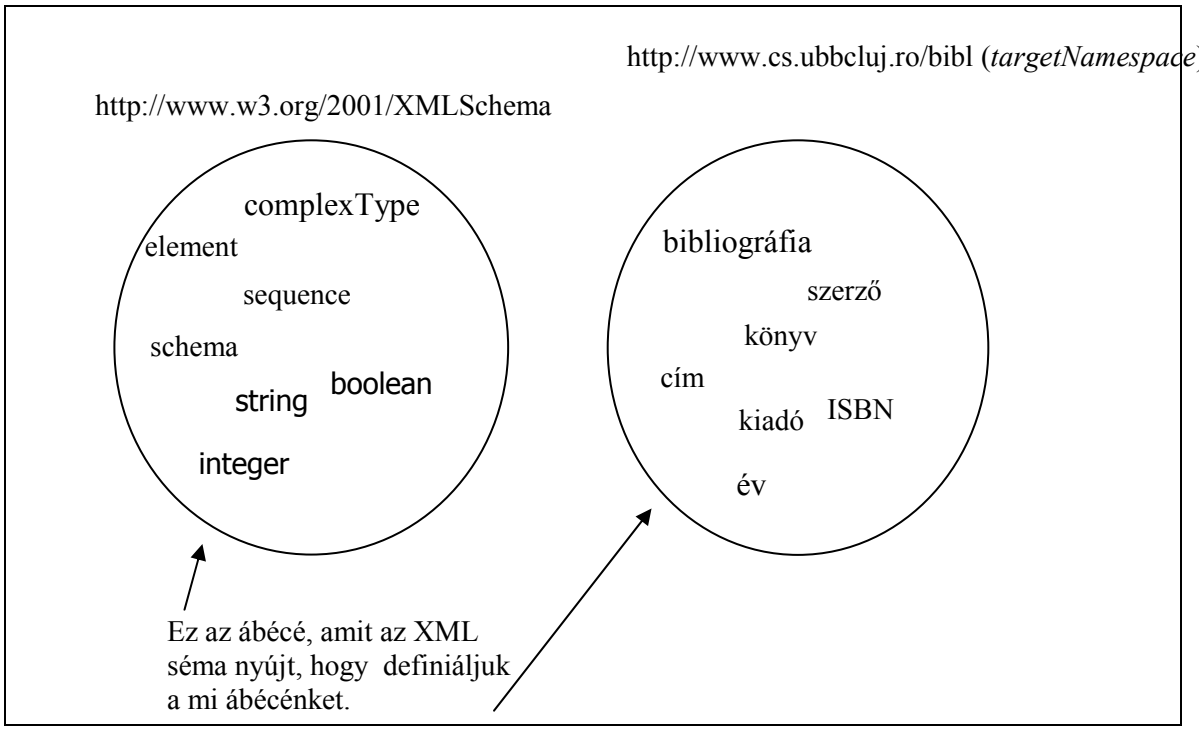
Amint láttuk, a séma definíció a `schema` kulcsszó megadásával kezdődik. A sémához a következő sor segítségével egy séma névteret rendelünk:

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
```

A megadott címen létezik egy XMLSchema.xsd, mely, előre definiált elemek és adattípusok, többek között a `complexType`, `element`, `sequence`, `schema`, `boolean`, `string`, `integer` értelmezését tartalmazza Felhasználva ezen névterületben értelmezett elemeket és típusokat, értelmezhetjük az XML adataink szerkezetét.

A séma névterületéhez egy névterület előtétet rendeltünk. Általában XML sémának az `xsd` előtétet szokták használni. Előtétet használva, a típusokat minősítjük, segítünk az elemzőnek azonosítani a típusokat a különböző névterekben. Mivel névterület előtéttel láttuk el a séma névterületet, névterület előtéttel kell az itt értelmezett típusokat, mint a `complexType`, `element`, `sequence`, `stb.` is ellátnunk.

Előbb definiáljuk a gyökér-elemet, azaz a `bibliográfia` elemet. Ez egy összetett típus, mely legalább egy `könyv` elemet tartalmaz. A `könyv` elem szintén egy összetett típus, gyerek elemei a `cím`, `szerző`, `év`, `ISBN`, `kiadó`, melyek pontosan egyszer kell szerepeljenek.



A sémánk által deklarált elemek: `bibliográfia`, `könyv`, `cím`, `szerző`, `év`, `ISBN`, `kiadó` a `targetNamespace` által megadott névterületbe kerülnek. Mivel ezt a névterületet nem láttuk el előtéttel, ez lesz az alapértelmezett (default), ezért nem kell az ebben értelmezett elemeket, típusokat sem előtéttel ellátnunk.

Megadhatjuk a fenti XML sémát úgy is, hogy az általunk értelmezett séma névteret látjuk el előtéttel, mondjuk bib, és a www.w3.org/2001/XMLSchema által adott séma lesz az alapértelmezett, ekkor a séma a következőképpen alakul:

```
<?xml version="1.0"?>
 <schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.cs.ubbcluj.ro/bibl"
      xmlns:bib="http://www.cs.ubbcluj.ro/bibl"
      elementFormDefault="qualified">
  <element name="bibliográfia">
   <complexType>
    <sequence>
     <element ref="bib:könyv" maxOccurs="unbounded"/>
    </sequence>
   </complexType>
  </element>
  <element name="Book">
   <complexType>
    <sequence>
     <element ref="bib:cím"/>
     <element ref="bib:szerző"/>
     <element ref="bib:év"/>
     <element ref="bib:ISBN"/>
     <element ref="bib:kiadó"/>
    </sequence>
   </complexType>
  </element>
  <element name="cím" type="string"/>
  <element name="szerző" type="string"/>
  <element name="év" type="string"/>
```

```
  <element name="ISBN" type="string"/>
  <element name="kiadó" type="string"/>
 </schema>
```

Amint látjuk nem kell a complexType, element, string, stb. kulcsszavakat előtéttel ellátnunk, viszont az általunk értelmezett elemekre való hivatkozás esetén előtétet kell használnunk.

A ref kulcsszó segítségével hivatkozunk egy elem deklarációra. Az első séma esetén a ref="könyv" nincs előtéttel ellátva, így az alapértelmezett névterületben leírt könyv elemre hivatkozik, mely az első esetben a cél névterület (targetNamespace). A második esetben a cél névterületet előtéttel láttuk el, így szükséges a hivatkozás esetén is az előtét: ref="bib:könyv".

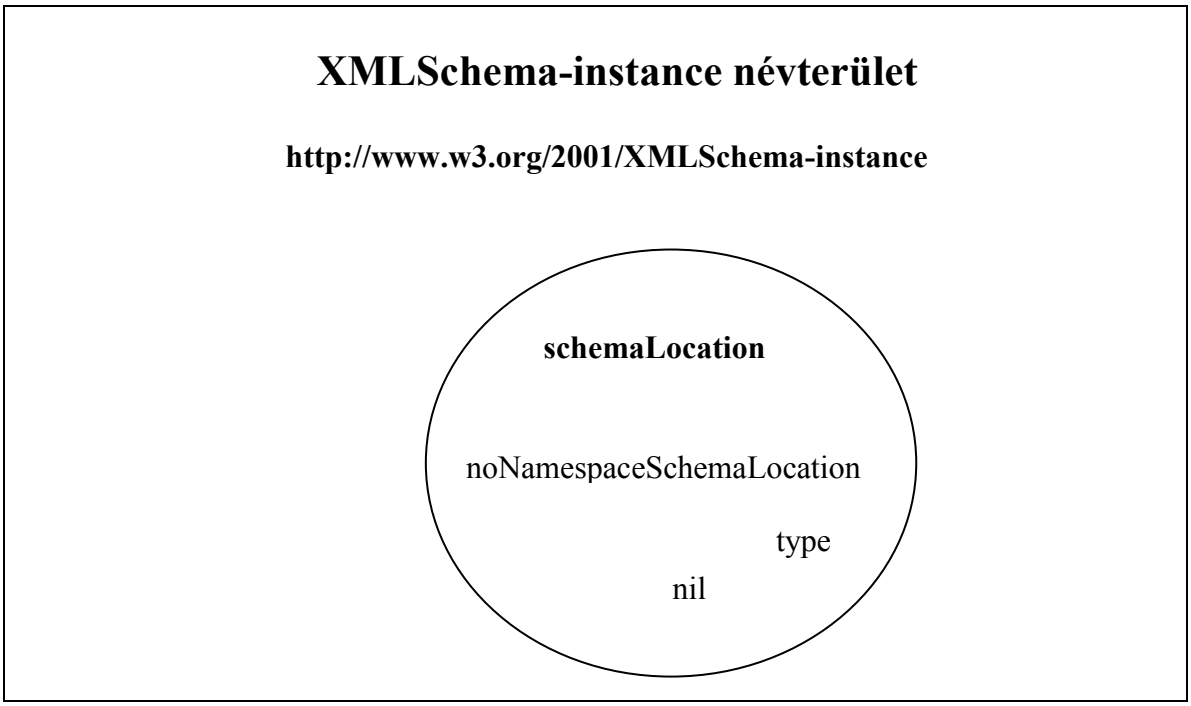
A gyerek elemeket megadhatjuk rögtön a szülő elem után, ún. inline deklarációval.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.cs.ubbcluj.ro/bibl"
            xmlns="http://www.cs.ubbcluj.ro/bibl"
            elementFormDefault="qualified">
 <xsd:element name="bibliográfia">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="könyv"  maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="cím" type="xsd:string"/>
       <xsd:element name="szerző" type="xsd:string"/>
       <xsd:element name="év" type="xsd:string"/>
       <xsd:element name="ISBN" type="xsd:string"/>
       <xsd:element name="kiadó" type="xsd:string"/>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Ha a fenti XML séma a http://www.cs.ubbcluj.ro/bibl címen a bibliografia.xsd állományban található, az XML állományból az XML sémát a következőképpen adhatjuk meg:

```
<?xml version="1.0"?>
 <bibliográfia xmlns="http://www.cs.ubbcluj.ro/bibl"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.cs.ubbcluj.ro/bibl
                            bibliografia.xsd">
  <könyv>
   <cím>An Introduction to Database Systems, 8th Edition</cím>
   <szerző>C. J. Date</szerző>
   <megjév>2004</év>
   <ISBN>973-86190-1-7</ISBN>
   <kiadó>Pearson, Addison Wesley</kiadó>
  </könyv>
        ...
 </bibliográfia>
```

Megfigyelhetjük, hogy a http://www.w3.org/2001/XMLSchema-instance címen egy újabb névterület jelenik meg, mely tartalmazza a schemaLocation definícióját, lásd 14.6. ábrát. A példában ezt a névteret xsi előtéttel láttuk el, ebből a névterületből használjuk a schemaLocation attribútumot, hogy megadjuk az XML állomány sémája hol található. A schemaLocation egy érték pár formájában adja meg a séma ellenőrzőnek, hogy a séma a bibliografia.xsd a http://www.cs.ubbcluj.ro/bibl névterületből származik.



14.6. ábra: Az XMLSchema-instance névterület

Egy XML állomány, melynek szerkezetét egy séma adja meg, helyesnek tekinthető, ha megfelel a megadott sémának, mely viszont akkor helyes, ha megfelel a sémák sémájának (XMLSchema), lásd 14.7 ábra.

Figyeljük meg, hogy a targetNamespace = http://www.cs.ubbcluj.ro/bibl utasítás segítségével a bibliografia.xsd-ben elemeket deklarálunk az adott névterületben, melyet a bibliografia.xml felhasznál, megadva ezt a

```
schemaLocation=http://www.cs.ubbcluj.ro/bibl bibliografia.xsd
```

utasítás segítségével.

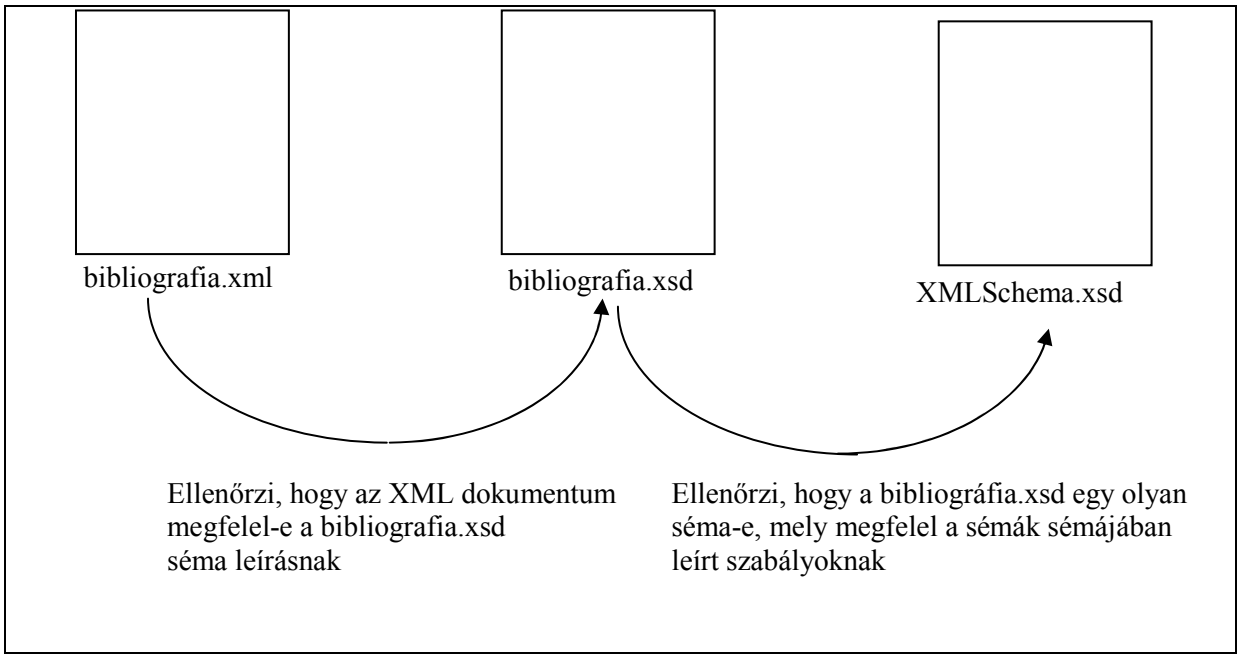
Még azt is észrevehetjük, hogy a cím, szerző, év, ISBN és kiadó elemek deklarálásakor a minOccurs és maxOccurs csak az első példa esetén jelenik meg, utána már nem. Ez azért lehetséges, mert ezek alapértelmezett értéke 1, így elhagyhatjuk.

Típus értelmezése:

Az összetett elem deklarálásának általános formája:

```
<xsd:element name="név" type="típus" minOccurs="int" maxOccurs="int"/>
                        vagy
    <xsd:element name="név" minOccurs="int" maxOccurs="int">
          <xsd:complexType>
                …
          </xsd:complexType>
    </xsd:element>
```

Mi eddig a második formáját használtuk az összetett elem deklarálásnak, például a könyv elem esetén.



14.7. ábra: Két szintű ellenőrzés

Lássunk most példát az összetett elem első típusú deklarálására, ahol nevet adunk a típusnak és az elem deklarálásakor hivatkozunk a megnevezett típusra.

```
<xsd:element name="A" type="ATípusa"/>
 <xsd:complexType name="ATípusa">
  <xsd:sequence>
   <xsd:element name="B" …/>
   <xsd:element name="C" …/>
  </xsd:sequence>
 </xsd:complexType>
```

Ez a deklaráció ekvivalens a következővel, amit inline típusdeklarációnak is nevezhetünk:

```
<xsd:element name="A">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="B" …/>
   <xsd:element name="C" …/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

A fenti két ekvivalens deklarációt együtt nem lehet használni, ily módon nem helyes a következő séma definíció:

```
<xsd:element name="A" type="ATípusa">
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="B" …/>
   <xsd:element name="C" …/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

A bibliográfiával kapcsolatos példa átírva az elem első típusú deklarációjával:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                     targetNamespace="http://www.books.org"
                      xmlns="http://www.books.org"
                      elementFormDefault="qualified">
 <xsd:element name="BookStore">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="könyv" type="kKiadvány"
                maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name="kKiadvány">
  <xsd:sequence>
   <xsd:element name="cím" type="xsd:string"/>
   <xsd:element name="szerző" type="xsd:string"/>
   <xsd:element name="év" type="xsd:string"/>
   <xsd:element name="ISBN" type="xsd:string"/>
   <xsd:element name="kiadó" type="xsd:string"/>
  </xsd:sequence>
 </xsd:complexType>
</xsd:schema>
```

Az XML séma definíció lehetőséget ad új (derivált) típus deklarálására, mely egy alaptípus kibővítésével vagy leszűkítésével jön létre. A következő példa Kiadvány nevű típusa alaptípus, mely három elemet tartalmaz. A KönyvKiadvány típus a Kiadvány típus deriváltja, annak három eleme mellé még két elemet deklarál.

```
<?xml version="1.0"?>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                      targetNamespace="http://www.books.org"
                      xmlns="http://www.books.org"
                     elementFormDefault="qualified">
  <xsd:complexType name="Kiadvány"
   <xsd:sequence>
    <xsd:element name="cím" type="xsd:string"/>
    <xsd:element name="szerző" type="xsd:string"
             maxOccurs="unbounded"/>
    <xsd:element name="év" type="xsd:gYear"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="KönyvKiadvány">
   <xsd:complexContent>
    <xsd:extension base="Kiadvány" >
     <xsd:sequence>
      <xsd:element name="ISBN" type="xsd:string"/>
      <xsd:element name="kiadó" type="xsd:string"/>
     </xsd:sequence>
    </xsd:extension>
   </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="könyvBibliográfia">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="könyv" type="KönyvKiadvány"
            maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:schema>
```

Egyszerű típusok deklarálása:

Az egyszerű típusok nem tartalmazhatnak elemeket, attribútumokat, csak beépített egyszerű típusokat tartalmazhatnak, mint `int, datetime, string, ID, IDREF, IDREFS, language, gYear` (Gregorian év). További beépített típusokat lásd az XMLSéma definíciójánál a http://www.w3.org/TR/xmlschema-2/ (Datatypes) címen.

14.27. példa: Legyen egy egyszerű típus, mely egy PC gyártási éve, a gregorianus évre alapszik és lehetséges értékei 1990 és 2010 között vannak.

```
<xsd:simpleType name="GyartasiEv"
 <xsd:restriction base="xsd:gYear">
  <xsd:minInclusive value="1990/">
  <xsd:maxInclusive value="2010/">
 <xsd:restriction>
<xsd:simpleType> □
```

Egy egyszerű típust a nevével adunk meg, majd a `restriction` elem segítségével megadjuk azt a típust, melyre alapszik az új egyszerű típus és további ún. "facet" adható meg. Általános formája:

```
<xsd:simpleType name= "név">
 <xsd:restriction base= "xsd:egyszerűTípus">
   <xsd:facet value= "érték"/>
   <xsd:facet value= "érték"/>
        …
 </xsd:restriction>
</xsd:simpleType>
```

Az egyszerűTípus amit a `restriction` kulcsszó után megadunk, a legtöbb esetben egy beépített típus, de építhetünk felhasználó által értelmezett egyszerű típusokra is.

A W3C ajánlatban a következő "facet"-ek szerepelnek:

```
length
minLength
maxLength
pattern
enumeration
whitespace
maxInclusive
minInclusive
maxExclusive
minExclusive
totalDigits
fractionDigits
```

Nem lehet mindegyik "facet"-et tetszőleges típussal használni.

A `pattern` kulcsszó után reguláris kifejezéseket használunk a karaktersor típusú adatok szerkezetének a leírására. (lásd a W3C honlapján a használható reguláris kifejezéseket).

14.28. példa: Értelmezzük a telefonszám típust, mely egy országon belüli telefonszámok szerkezetét adja meg :

```
<xsd:simpleType name="TelefonSzám">
 <xsd:restriction base="xsd:string">
  <xsd:length value="10"/>
  <xsd:pattern value="\d{4}-\d{6}"/>
 </xsd:restriction>
</xsd:simpleType>
```

14.29. példa: Egy könyv ISBN száma a következő formájú lehet:

d-ddddd-ddd-d vagy
d-ddd-ddddd-d vagy

d-dd-dddddd-d, ahol d egy számjegyet jelöl és a számjegy csoportok kötőjellel vannak elválasztva. A pattern "facet" segítségével tudjuk megadni a mintát, ahol \d számjegyet jelöl, illetve zárójelben a számjegyek számát. A három különböző lehetséges formáját az ISBN számnak három külön sorban vagy függőleges vonallal elválasztva egymástól adjuk meg. Tehát egy egyszerű típus, mely az ISBN típust adja meg a következő:

```
<xsd:simpleType name="ISBNTípus">
      <xsd:restriction base="xsd:string">
         <xsd:pattern value="\d{1}-\d{5}-\d{3}-\d{1}"/>
    <xsd:pattern value="\d{1}-\d{3}-\d{5}-\d{1}"/>
    <xsd:pattern value="\d{1}-\d{2}-\d{6}-\d{1}"/>
 </xsd:restriction>
</xsd:simpleType>
```

A mintát megadó három sort a következőképpen is megadhatjuk:

```
<xsd:pattern value="\d{1}-\d{5}-\d{3}-\d{1}|\d{1}-\d{3}-\d{5}-\d{1}|
    \d{1}-\d{2}-\d{6}-\d{1}"/> □
```

14.30. példa: Legyen ismét a könyvészetet leíró xml adat szerkezete, felhasználva a fenn értelmezett ISBNTípust és a beépített gYear (Gregorian year) típust.

```
<?xml version="1.0"?>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.cs.ubbcluj.ro/bibl"
            xmlns="http://www.cs.ubbcluj.ro/bibl"
            elementFormDefault="qualified">
    <xsd:simpleType name="ISBNTípus">
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="\d{1}-\d{5}-\d{3}-\d{1}"/>
            <xsd:pattern value="\d{1}-\d{3}-\d{5}-\d{1}"/>
            <xsd:pattern value="\d{1}-\d{2}-\d{6}-\d{1}"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:element name="bibliográfia">
        <xsd:complexType>
            <xsd:sequence>
               <xsd:element name="könyv"  maxOccurs="unbounded">
                 <xsd:complexType>
                    <xsd:sequence>
                      <xsd:element name="cím" type="xsd:string"/>
                      <xsd:element name="szerző" type="xsd:string"/>
                      <xsd:element name="év" type="xsd:gYear"/>
                      <xsd:element name="ISBN"
                                   type="xsd:ISBNTípus"/>
                      <xsd:element name="kiadó" type="xsd:string"/>
                    </xsd:sequence>
                 </xsd:complexType>
               </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema> □
```

Mikor használunk egyszerű, illetve összetett típust?

- Az összetett típus akkor ajánlott, ha egy elemnek gyerek elemet és/vagy attribútumokat akarjuk értelmezni.
- Egyszerű típust definiálunk, ha az egy beépített típusnak a finomítása.

Attribútumok deklarálása:

Egy összetett típusnak lehetnek attribútumai, az egyszerűnek nem. Az attribútumokat a gyerek elemek után adjuk meg és arra az elemre vonatkoznak, amelyikben bennefoglaltatnak.

Általános formája az attribútum deklarálásának XMLSchema-ben:

```
<xsd:attribute name="név" type="egyszerű-típus" use="használat-módja"
      default/fixed="érték"/>
```

Amint látjuk az attribútumnak van egy neve, egy egyszerű típusa. A használat módja lehetséges értékei:

```
required
optional
prohibited
```

Egy attribútum lokális, ha egy elemen belül értelmezzük. Csak a lokális attribútumoknak adhatjuk meg a használat módját. Ezen kívül globális attribútumokat is deklarálhatunk, elemeken kívül, melyekre azután elem hivatkozhat.

14.31. példa: Egy egyszerű példa attribútumokkat:

```
<xsd:element name="foo">
    <xsd:complexType>
        <xsd:sequence>
            …
        </xsd:sequence>
        <xsd:attribute name="bar" …/>
        <xsd:attribute name="boo" …/>
    </xsd:complexType>
</xsd:element> □
```

14.32. példa: Egészítsük ki a példánkat DTD-ben úgy, hogy tartalmazzon attribútumokat is.

```
<!ELEMENT bibliográfia (könyv+)>
<!ELEMENT könyv (cím, szerző, év, ISBN, kiadó)>
<!ATTLIST könyv
   Típus (önéletrajz| tudományos | regény) #REQUIRED
   VanRaktáron (igaz | hamis) "hamis"
   Bíráló CDATA " ">
<!ELEMENT cím (#PCDATA)>
<!ELEMENT szerző (#PCDATA)>
<!ELEMENT év (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT kiadó (#PCDATA)>
```

A megfelelő XML Schema a következő:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.cs.ubbcluj.ro/bibl"
            xmlns="http://www.cs.ubbcluj.ro/bibl"
            elementFormDefault="qualified">
    <xsd:element name="bibliográfia">
       <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="könyv"  maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="cím" type="xsd:string"/>
                  <xsd:element name="szerző" type="xsd:string"/>
                  <xsd:element name="év" type="xsd:string"/>
                  <xsd:element name="ISBN" type="xsd:string"/>
                  <xsd:element name="kiadó" type="xsd:string"/>
                </xsd:sequence>
                <xsd:attribute name="Típus" use="required">
                  <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                     <xsd:enumeration value="önéletrajz"/>
                     <xsd:enumeration value="tudományos"/>
```

```
                    <xsd:enumeration value="regény"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:attribute>
              <xsd:attribute name="VanRaktáron" type="xsd:boolean"
                        default="false"/>
              <xsd:attribute name="Bíráló" type="xsd:string"
                        default=" "/>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Figyeljük meg, hogy a könyv elemen belül adtuk meg az attribútumokat, mégpedig a gyerek elemek után. Tehát a deklarált attribútumok a könyv elem attribútumai, mint ahogy a DTD-ben is megadtuk. Amint látjuk a `Típus` attribútum felsorolás típusú, a másik kettőnek pedig adtunk alapértelmezett értéket. Ez a forma az ún. inline deklaráció. Megadhatjuk hivatkozással is az attribútumokat, attribútum csoportot értelmezve:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.cs.ubbcluj.ro/bibl"
      xmlns="http://www.cs.ubbcluj.ro/bibl"
      elementFormDefault="qualified">
   <xsd:element name="bibliográfia">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element ref="könyv" minOccurs="1" maxO
     curs="unbounded"/>
      </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
   <xsd:element name="könyv">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element ref="cím" minOccurs="1" maxOccurs="1"/>
       <xsd:element ref="szerző" minOccurs="1" maxOccurs="1"/>
       <xsd:element ref="év" minOccurs="1" maxOccurs="1"/>
       <xsd:element ref="ISBN" minOccurs="1" maxOccurs="1"/>
       <xsd:element ref="kiadó" minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attributeGroup ref="BookAttributes"/>
     </xsd:complexType>
   </xsd:element>
   <xsd:element name="cím" type="xsd:string"/>
   <xsd:element name="szerző" type="xsd:string"/>
   <xsd:element name="év" type="xsd:integer"/>
   <xsd:element name="ISBN" type="xsd:string"/>
   <xsd:element name="kiadó" type="xsd:string"/>
   <xsd:attributeGroup name="KönyvTulajdonságok">
     <xsd:attribute name="Típus" use="required">
      <xsd:simpleType>
         <xsd:restriction base="xsd:string">
           <xsd:enumeration value="önéletrajz"/>
           <xsd:enumeration value="tudományos"/>
           <xsd:enumeration value="regény"/>
         </xsd:restriction>
      </xsd:simpleType>
     </xsd:attribute>
     <xsd:attribute name="VanRaktáron" type="xsd:boolean"
```

```
                default="false"/>
      <xsd:attribute name="Bíráló" type="xsd:string" default=" "/>
    </xsd:attributeGroup>
</xsd:schema> □
```

A példában egy másik formáját is láttuk az attribútum deklarálásának:

```
<xsd:attribute name="név" use="használat-módja"
           default/fixed="value">
 <xsd:simpleType>
    <xsd:restriction base="egyszerű-típus">
      <xsd:facet value="érték"/>
    …
    </xsd:restriction>
 </xsd:simpleType>
</xsd:attribute>
```

Az XML Schema nagyon összetett, további részleteket nem mutatunk be ebben a könyvben. (lásd a további részleteket a http://www.w3.org/XML/Schema-ban)

Összefoglalásképpen, lássuk mi is az XML Schema:

- egy adatmodell, melynek segítségével az XML adatok szerkezetét írjuk le.
- egy szerződés: a szervezetek/cégek között, melyek megegyeznek abban, hogy milyen szerkezete legyen az XML adataiknak és ezt egy XML Schema segítségével adják meg, melyet be is kell tartson mindegyik fél.
- metaadat: a séma sok adatot tartalmaz az XML adatokról. Nagyon sok megszorítást lehet a sémában megadni, így megtakaríthatunk nagyon sok adat helyességet ellenőrző programkódot.

14.5. Adatkezelési műveletek az XML nyelvben

Az XQuery XML dokumentumok lekérdezésére szolgáló lekérdezőnyelv, melynek kifejező ereje az OQL nyelvével egyenértékű. Több nyelvet is javasoltak XML dokumentumok lekérdezésére, például XML-QL, XQL, Lorel, de jelenleg valószínű, hogy az XQuery lesz a szabványos, ezért, ezt bővebben, de nem teljes egészében mutatjuk be. Teljes specifikációja a http://www.w3.org/TR/xquery/ címen található.

Az XQuery több korábbi nyelvre alapul, az XPath elérési-út kifejezéseit használja az XML dokumentum részeinek a visszaküldésére. Valójában az XQuery magába foglalja a teljes XPath-t.

Az XQuery csak lekérdezésre ad lehetőséget. Adatmódosításra nincs lehetőség egyelőre csak a DOM (Document Object Model) segítségével, melyet csak a programozók használhatnak, egyszerű felhasználók nem tudják használni. Fejlesztés alatt van egy XUpdate nevű nyelv, mely valószínű szabvány lesz.

Az XQuery nem XML dokumentumokkal dolgozik, mivel az XML dokumentumok karaktersorok, úgy voltak kitalálva, hogy az ember számára olvashatók legyenek. Ennek eredményeként olyan karaktereket tartalmaznak, melyek az ember számára könnyebbé teszi a megértést (új sor, címkék, stb.), viszont a dokumentum reális információ elméleti tartalmát nem támogatják. Egy lekérdezőnyelv viszont az információt az XML dokumentumból kell, hogy feldolgozza. Ìgy az XQuery az XML dokumentumok egy átalakított, absztakt formájával dolgozik. Egy XML dokumentum absztrakt formája az XQuery adatmodell egy példánya.

14.5.1. Az XPath nyelv

Az XPath lehetőséget ad egy XML dokumentumban való navigációra, figyelembe véve a félig-struktúrált adat gráfként való ábrázolását és az útakat a gráfban. Az XQuery nyelv, az XSLT és az XPointer is használja az XPath által nyújtott lehetőségeket. Az XQuery adatmodellben (hasonlóan az XPath esetében) egy XML dokumentum egy fa szerkezettel van ábrázolva. Minden csomópontnak van egy azonósítója, mely megkülönbözteti a többi csomóponttól, még ha teljesen egyformák is. A fa csomópontjainak típusa lehet:

- dokumentum (document)
- csomópont (element)

- szöveges (text)
- tulajdonság (attribute)
- névterület (namespace)
- vezérlési utasítás (processing intruction)
- megjegyzés (comment)

Egy XPath kifejezés eredménye csomópontok egy csoportja, vagy logikai érték, szám vagy szöveg. Egy kifejezés kiértékelésekor figyelembe vevődik az aktuális csomópont.

14.5.2. Függvények

A következő két függvényt említjük: doc() és collection(). A doc() függyvény egy URI (Universal Resource Identifier) által megadott dokumentumot ad vissza, pontosabban egy dokumentum csomópontot. A collection() függvényt akkor használhatjuk, ha egy adatbázist akarunk azonosítani egy lekérdezéshez, mely csomópontok kollekcióját adja meg.

14.33. példa: Legyen a következő könyvészetet leíró példa, mely egy bibl.xml nevű állományban van tárolva.

```
<bibliográfia>
   <könyv év="1994">
      <cím>TCP/IP Illustrated</cím>
      <szerző>
         <vnév>Stevens</vnév>
         <knév>W.</knév>
      </szerző>
      <kiadó>Addison-Wesley</kiadó>
      <ár>65.95</ár>
   </könyv>
   <könyv év="1992">
      <cím>Advanced Programming in the UNIX Environment</cím>
      <szerző>
         <vnév>Stevens</vnév>
         <knév>W.</knév>
      </szerző>
      <kiadó>Addison-Wesley</kiadó>
      <ár>65.95</ár>
   </könyv>
   <könyv év="2000">
      <cím>Data on the Web</cím>
      <szerző>
         <vnév>Abiteboul</vnév>
         <knév>Serge</knév>
      </szerző>
      <szerző>
         <vnév>Buneman</vnév
        <knév>Peter</knév>
      </szerző>
      <szerző>
        <vnév>Suciu</vnév>
        <knév>Dan</knév>
      </szerző>
      <kiadó>Morgan Kaufmann</kiadó>
      <ár>39.95</ár>
   </könyv>
   <könyv év="1999">
      <cím>The Economics of Technology and Content for Digital
         TV</cím>
      <szerkesztő>
```

```
        <vnév>Gerbarg</vnév>
        <knév>Darcy</knév>
      <társulat>CITI</társulat>
      </szerkesztő>
      <kiadó>Kluwer Academic Publishers</kiadó>
      <ár>129.95</ár>
   </könyv>
</bibliográfia>
```

Az XML adat szerkezetét a következő DTD adja meg:

```
<!ELEMENT bibliográfia (könyv* )>
<!ELEMENT könyv (cím, (szerző+ | szerkesztő+ ), kiadó, ár)>
<!ATTLIST könyv év CDATA #REQUIRED >
<!ELEMENT szerző (vnév, knév)>
<!ELEMENT szerkesztő (vnév, knév, társulat)>
<!ELEMENT cím (#PCDATA )>
<!ELEMENT vnév (#PCDATA )>
<!ELEMENT knév (#PCDATA )>
<!ELEMENT társulat (#PCDATA )>
<!ELEMENT kiadó (#PCDATA )>
<!ELEMENT ár (#PCDATA )>
A doc("bibl.xml") függvény visszatéríti az egész XML dokumentumot. □
```

14.5.3. XPath elérési-út kifejezései (path expressions)

Az elérési út egy XML dokumentum adott csomópontjához vezető út megadására szolgál. Minden elérési út lépésekből áll, melyek / vagy // jellel vannak egymástól elválasztva. Minden lépés kiértékelése eredményeként csomópontok sorozatát kapjuk.

14.34. példa: A

```
doc("bibl.xml")/bibliográfia/könyv
```

elérési-út kifejezés esetén a doc függvény megnyitja a bibl.xml állományt, visszaadja a dokumentum csomópontot. A /bibliográfia szelektálja a dokumentum gyökér elemét, a /könyv pedig a könyv elemeket. □

14.35. példa: A

```
doc("bibl.xml")//könyv
```

elérési-út kifejezés eredménye azonos a 14.34. példa által adott eredménnyel. Szintén könyv elemeket ad vissza az adott dokumentumból, arról a szintről, ahol épp könyv elemek találhatók. □
Egy XPath lépés általános formája a következő:

```
irány::csomópont_típus [feltételes_kifejezés]
```

Irányok

Az irány, az aktuális elem és a lépéssel megjelölt csomópontok viszonyát írja le. Lehetséges irányok:

- child:: az aktuális elem gyermek eleme lesz. Ha nincs megadva irány, ez az alapértelmezett. Pl. child::könyv, az aktuális elem könyv gyerek elemeit adja.
- descendant:: az aktuális elem összes leszármazottja (gyermekei és azoknak is a gyermekei rekurzívan). Pl. descendant::szerző azon szerző elemeket adja, melyek az aktuális elem leszármazottai.
- parent:: az aktuális elem szülő elemét jelöli. Pl. parent::könyv kiválasztja az aktuális elem szülő elemét, ha az egy könyv elem.
- self:: az aktuális elemet jelöli. Pl. self::könyv az aktuális elemet adja, ha az egy könyv elem.
- descendant-or-self:: az aktuális elemet és a leszármazottait jelöli.
- ancestor:: az aktuális elem összes ősét jelöli.

- ancestor-or-self:: az aktuális elemet és az összes ősét jelöli.
- following-sibling:: az aktuális elem következő testvérét jelöli.
- preceding-sibling:: az aktuális elem előző testvérét jelöli.
- following:: az aktuális elemet követő minden elem a dokumentum sorrendjében.
- preceding:: az aktuális elemet megelőző összes elem a dokumentum sorrendjében.
- attribute:: az aktuális elem tulajdonságait jelöli. Pl. attribute::könyv, könyv tulajdonságait adja meg az aktuális elemnek.
- namespace:: az aktuális elem névtereit jelöli ki.

Rövidítések

A leggyakrabban használt XPath kifejezéseket rövidített formában is lehet írni, ezek a következők:

(semmi)	child::
@	attributes::
. (pont)	self::node()
.. (két pont)	parent::node()
//	/descendant-or-self::node()

14.36. példa: Az eddigi példákban is a child irány elmaradt. A

```
doc("bibl.xml")/bibliográfia/könyv
```

kifejezés bővebben:

```
doc("bibl.xml")/child::bibliográfia/child::könyv.
```

A következő kifejezés:

```
doc("bibl.xml")/bibliográfia/könyv/attribute::év
```

rövidebben:

```
doc("bibl.xml")/bibliográfia/könyv/@év □
```

Típusok

Amint már láttuk, az XPath 7 féle csomópont típust különböztet meg. Ebből három alaptípus: névterület, tulajdonság, csomópont és a többi négy (szöveges, megjegyzés, gyökér, vezérlési utasítás) a csomópont alaptípus speciális esete. Az XPath-ban minden iránynak van egy elsődleges típusa: ha az aktuális irány tulajdonság, akkor az elsődleges típus is tulajdonság, ha az aktuális irány névterület, akkor az elsődleges típus is névterület, és minden más esetben az elsődleges típus csomópont.

Egy típust kétféleképpen lehet megadni: névterület megadásával vagy csomópont típust kiválasztó utasítások segítségével.

Csomópont típusok:

- "név" – A "név" által megadott nevű tulajdonságot (attribute:: irány esetén) vagy elemeket választja ki. Pl. child::könyv esetén a típus névvel volt megadva és az aktuális elem könyv gyerek elemeit adja meg.
- "*" – az adott iránynak megfelelő összes elemet (bármilyen nevű) kiválasztja.
- comment() – az adott iránynak megfelelő minden megjegyzés elemet kiválaszt.
- text() – az adott iránynak megfelelő minden szöveges elemet kiválaszt.
- processing-instruction() – az adott iránynak megfelelő minden vezérlési utasítást kiválaszt.
- node() – az adott iránynak megfelelő minden elemet, a típusától függetlenül, kiválaszt. Pl. ha az aktuális egy könyv elem, a child::node() kiválasztja annak összes gyermek elemét, a típusától függetlenül.

Feltételes kifejezések

A feltételes kifejezések segítségével az elérési-út kifejezés által szolgáltatott csomópontok egy részhalmazát választjuk ki. A feltételes kifejezést szögletes zárójelben adjuk meg.

14.37. példa: A következő elérési-út kifejezés azon szerző elemeket adja meg, melyeknek vezeték neve Stevens.

```
doc("bibl.xml")/bibliográfia/könyv/szerző[vnév="Stevens"] □
```

Ha a szögletes zárójel numerikus kifejezést tartalmaz, sorszámként lesz kezelve.

14.38. példa: A

```
doc("bibl.xml")/bibliográfia/könyv/szerző[1]
```

kifejezés minden könyv esetén az első szerzőt adja meg. Minden könyv elemre kiértékeli a szerző[1] kifejezést. Ha az egész dokumentum legelső szerzőjét szeretnénk kiválasztani, azt a következő kifejezéssel tehetjük meg:

```
(doc("bibl.xml")/bibliográfia/könyv/szerző)[1] □
```

Tulajdonság értékére is vonatkozhat feltétel. A tulajdonság nevét a @ kell megelőzze.

14.39. példa: A

```
doc("bibl.xml")/bibliográfia/könyv/[@év=2000]
```

kifejezés azon könyv elemeket adja meg, melyek év tulajdonságának értéke 2000. A kifejezést a következő formában is megadhatjuk. A * bármely elem nevét helyettesíti.

```
doc("bibl.xml")/bibliográfia/*/[@év=2000] □
```

Az elérési-út kifejezés kiértékelése a következőképpen történik: minden / bal oldalán levő kifejezést kiértékel és visszaadja a csomópontokat a dokumentumban megjelenő sorrendben. Ha a visszaadott érték más, mint csomópont, típus hibát generál a rendszer. Ezek után a / jobb oldalán levő kifejezés kerül kiértékelésre az összes bal oldali csomópont esetén, összefésülve az eredmény csomópontokat, ahogy a dokumentumban megjelennek. Amikor a jobb oldali kifejezés kerül kiértékelésre, a bal oldali csomópontot tekintjük az aktuális elemnek.

14.40. példa: A következő kifejezés:

```
doc("bibl.xml")/bibliográfia//szerző[1]
```

kiértékelése esetén, az első lépés a dokumentum csomópontot adja vissza, a második lépés pedig a bibliográfia elemet. A harmadik lépés a descendant-or-self::node() lépés segítségével a bibliográfia elemet és az összes leszármazottját szolgáltatja. A negyedik lépés, a harmadik lépés minden csomópontjára az első szerző elemet adja meg. Mivel csak a könyv elemeknek van szerző gyerek elemük, a kifejezés minden könyv esetén az első szerzőt adja meg. □

14.6. Az XQuery lekérdező nyelv

Az XPath nyelv nem elegendő az XML dokumentumok lekérdezésére, csak egy címzési lehetőség. Az XQuery az XPath elérési-út kifejezéseit használja az XML dokumentum részeinek a visszatérítésére. A FLWOR kifejezések az SQL nyelv SELECT parancs helyét veszik át. Dokumentum, elem és tulajdonság konstruktorok segítik új XML adat létrehozását.

14.6.1. Dokumentum, elem és tulajdonság konstruktorok

Az XQuery nyelvben lehetőségünk van XML dokumentum létrehozására a

```
document{}
```

kifejezés segítségével, mely a fenti formában egy üres dokumentumot hoz létre.

Elemeket, tulajdonságokat, megjegyzéseket XML szintaxissal hozhatunk létre.

14.41. példa: A következőkben egy dokumentum csomópontot hozunk létre, mely megjegyzést és elemeket tartalmaz.

```
document {
<?xml version="1.0" encoding="WINDOWS-1250" ?>
<!—Egy jó adatbázis könyv! —>
<könyv év="2000">
 <cím>Database System Implementation</cím>
 <szerző>
    <vnév>Garcia-Molina</vnév>
    <knév>Hector</knév>
 </szerző>
 <szerző>
    <vnév>Ullman</vnév
    <knév>Jeffrey D.</knév>
 </szerző>
 <szerző>
    <vnév>Widom</vnév>
    <knév>Jennifer</knév>
 </szerző>
 <kiadó>Prentice Hall</kiadó>
 <ár>45</ár>
</könyv>
} □
```

Elem konstruktorban kapcsos zárójelek: „{” és „}”, ún. „enclosed” kifejezést határolnak. Ilyen kifejezések egy elem vagy egy tulajdonság értékének a generálására használhatók. A zárójelek közé egy XPath kifejezést, illetve összesítő függvényeket helyezhetünk.

14.42. példa: A következő példa az „enclosed” kifejezések szemléltetésére szolgál:

```
<példa>
  <p> Ez egy lekérdezés. </p>
  <eg> doc("bibl.xml")//könyv[1]/cím </eg>
  <p> Itt a fenti egy lekérdezés eredménye.</p>
  <eg>{ doc("bibl.xml")//könyv[1]/cím }</eg>
</példa>
```

Az eredménye a fenti lekérdezésnek:

```
<példa>
  <p> Ez egy lekérdezés. </p>
  <eg> doc("bibl.xml")//könyv[1]/cím </eg>
  <p> Itt a fenti egy lekérdezés eredménye.</p>
  <eg><cím>TCP/IP Illustrated</cím></eg>
</példa>
```

Amint látjuk, doc("bibl.xml")//könyv[1]/cím XPath kifejezést kiértékeli a rendszer és eredménye, mely egy cím elem, a kapcsos zárójelek között levő kifejezés helyére kerül. □

Az „enclosed” kifejezések segítségével létező XML értékeket új szerkezetben adhatunk meg. Az enclosed kifejezés esetén elemet bevezető és elemet záró tag generálódik.

14.43. példa: A következő lekérdezés eredménye egy címek elemet fog tartalmazni, melynek van egy szám nevű tulajdonsága, mely a bibl.xml állományban létező címek számát tartalmazza és csak cím elemeket ugyanabból az xml állományból. Az XPath elérési-út kifejezés, melyet használunk a doc("bibl.xml")//cím. A szám tulajdonság ezen cím elemek számát tartalmazza, melyet a count függvény segítségével kapunk meg. Vegyük észre, hogy tulajdonság értékét dupla idézőjel közé kell helyezni és ezért az XML állomány nevét egyszeres idézőjel közé tesszük.

```
<címek szám="{ count(doc('bibl.xml')//cím) }">
{
      doc("bibl.xml")//cím
     }
```

```
</címek>
```

A lekérdezés eredménye:

```
<címek szám = "4">
 <cím>TCP/IP Illustrated</cím>
 <cím>Advanced Programming in the Unix Environment</cím>
 <cím>Data on the Web</cím>
 <cím>The Economics of Technology and Content for Digital TV</cím>
</címek> □
```

Az XQuery egy másik lehetőséget is ajánl elemek és tulajdonságok létrehozására, az `element` és `attribute` kulcsszó segítségével.

14.44. példa: A következő példa egy könyv elemet hoz létre az ismert szerkezettel.

```
element könyv
{
     attribute év { 2004 },
     element cím { "An Introduction to Database Systems" },
     element szerző
     {
          element vnév { "Date" },
     element knév { "Chris J." }
     },
element kiadó { "Addison Wesley Higher Education " },
     element ár{ 59.95 }
} □
```

14.6.2. FLWOR kifejezések

Az XQuery egyik legerősebb és legismertebb kifejezései a FLWOR kifejezések (ejtsd flower). Az SQL nyelv SELECT parancsához hasonlít, de nem táblákra, sorokra, oszlopokra vonatkozik, hanem FOR és LET kulcsszó segítségével változót köt értékekhez. Ezek a változók inkább a programozásból ismert változókhoz hasonlítanak, így ezek a kifejezések procedurálisak, ellentétben az SQL SELECT paranccsal, mely egy deklaratív (nem procedurális) nyelv.

A FLWOR kifejezés neve rövidítésekből adódik:

FOR kulcsszó, mely egy vagy több változóhoz kifejezéseket rendel, értékpárokat létrehozva. Mindegyik változó végigfutja a hozzárendelt kifejezés összes lehetséges értékét. Általános formája:

FOR <változó_1> IN <elérési-út kifejezés_1>
⋮
<változó_k> IN <elérési-út kifejezés_2>

A változó rendre felveszi az elérési-út kifejezés minden értékét, melyek a legtöbb esetben csomópontok az XML dokumentumból. Minden ami a FOR után következik a változó összes értékére végrehajtásra kerül.

LET kulcsszó, mely abban különbözik a FOR-tól, hogy a változó nem egyenként veszi fel a hozzárendelt kifejezés összes értékét, hanem a kifejezés összes értékét egyszerre rendeli a változóhoz. Használható a FOR kulcsszóval együtt vagy nélküle is. Ha nincs FOR a kifejezésben, akkor a változó csak egy értéket vesz fel, mely valójában a hozzárendelt kifejezés összes értékének a sorozata;

WHERE kulcsszó, melynek segítségével egy szűrő feltételt tehetünk a FOR és LET által létrehozott értékpárokra;

ORDER BY kulcsszó, mely az értékpárokat adott sorrendbe rendezi;

RETURN kucsszó, melynek segítségével az eredmény értékpárt létre tudjuk hozni.

14.45. példa: A következő bevezető példák nem elérési-út kifejezéseket használnak, hanem számsorozatokat, a szemléletesség érdekében. A gyakorlatban nagyrészt elérési-út kifejezést használunk, de az XPath megengedi ezt a fajta kifejezést is. Az *i* változót használjuk, amint látjuk a

$ jel előzi meg a változókat. Az értékek, amit az $i változót rendre felvesz: 1, 2 és 3. Figyeljük meg, hogyan használunk „enclosed” kifejezést ahhoz, hogy a változó értékét megkapjuk.

```
FOR $i IN (1, 2, 3)
RETURN <eredmény><i>{ $i }</i></eredmény>
```

A lekérdezés eredménye:

```
<eredmény><i>1</i></eredmény>
<eredmény><i>2</i></eredmény>
<eredmény><i>3</i></eredmény> □
```

14.46. példa: A fentihez hasonló példában használjuk a FOR helyett a LET-et. Figyeljük meg, hogy az $i változó egyszerre felveszi az (1, 2, 3) halmaz összes értékét.

```
LET $i:=(1, 2, 3)
RETURN <eredmény><i>{ $i }</i></eredmény>
```

A lekérdezés eredménye:

```
<eredmény><i>1 2 3</i></eredmény>□
```

14.47. példa: Legyen most FOR is LET is a kifejezésben. Amint látjuk, az $i változó minden értékére a $j változó felveszi a hozzárendelt érték sorozatot.

```
FOR $i IN (1, 2, 3)
LET $j:=(1, 2, 3)
RETURN
<eredmény><i>{ $i }</i><j>{ $j }</j></eredmény>
```

A lekérdezés eredménye:

```
<eredmény><i>1</i><j>1 2 3</j></eredmény>
<eredmény><i>2</i><j>1 2 3</j></eredmény>
<eredmény><i>3</i><j>1 2 3</j></eredmény> □
```

14.48. példa: Legyen most két változó a FOR kulcsszó után:

```
FOR $i IN (1, 2, 3),
    $j IN (4, 5, 6)
RETURN
 <eredmény><i>{ $i }</i><j>{ $j }</j></eredmény>
```

Az eredmény a következő:

```
<eredmény><i>1</i><j>4</j></eredmény>
<eredmény><i>1</i><j>5</j></eredmény>
<eredmény><i>1</i><j>6</j></eredmény>
<eredmény><i>2</i><j>4</j></eredmény>
<eredmény><i>2</i><j>5</j></eredmény>
<eredmény><i>2</i><j>6</j></eredmény>
<eredmény><i>3</i><j>4</j></eredmény>
<eredmény><i>3</i><j>5</j></eredmény>
<eredmény><i>3</i><j>6</j></eredmény>
```

A jobb megértés érdekében írjuk át pszeudokódba:

```
FOR i = 1 to 3 do
  FOR j = 4 to 6 do
       eredmény {i, j}
  ENDFOR
ENDFOR □
```

Lássunk olyan lekérdezéseket, ahol a változó egy elérési-út kifejezés különböző értékeit veszi fel.

14.49. példa: Tekintsük ismét a könyvészetet leíró XML dokumentumot, mely a bibl.xml állományban található. A $b változó értékei rendre a dokumentumban található könyv elemek. Egy

adott könyv elem esetén, melyet a $b változó tartalmaz, a WHERE utáni feltétel a könyv elem ár gyerek elemére hivatkozik.

```
FOR $b IN doc("bibl.xml")//könyv
WHERE $b/ár < 50
RETURN $b/cím
```

A lekérdezés eredménye tartalmazza az 50 $-nál olcsóbb könyveket:

```
<cím>Data on the Web</cím> □
```

Az XQuery-ben is használhatunk összesítő függvényeket, mint a count, sum, avg, min és max.

14.50. példa: A könyvészetet leíró XML dokumentum esetén minden könyv esetén az eredmény tartalmazza a könyv címét és szerzőinek a számát. A $b változó értékei rendre a dokumentumban található könyv elemek. Egy adott könyv elem esetén, melyet a $b változó tartalmaz, a $c változó az aktuális könyv elem összes szerző gyerek eleme ($b/szerző), mivel egy könyvnek több szerzője is lehet. A lekérdezés eredménye könyv elemeket fog tartalmazni, melynek cím és szszám nevű gyerek elemei lesznek. A szszám minden könyv esetén a szerzők számát tartalmazza. Figyeljük meg, hogy a count függvény nem elemet ad vissza, mint a $b/cím, ezért „enclosed" kifejezést használunk, mely a változó értékét adja meg, ezért elemet bevezető és elemet záró tagot explicit meg kell adjuk.

```
FOR $b IN doc("bibl.xml")//könyv
LET $c := $b/szerző
RETURN <könyv>{ $b/cím, <szszám>{ count($c) }</szszám>}</könyv>
```

A lekérdezés eredménye:

```
<könyv>
  <cím>TCP/IP Illustrated</cím>
  <szszám>1</szszám>
</könyv>
<könyv>
  <cím>Advanced Programming in the UNIX Environment</cím>
  <szszám>1</szszám>
</könyv>
<könyv>
  <cím>Data on the Web</cím>
  <szszám>3</szszám>
</könyv>
<könyv>
  <cím>The Economics of Technology and Content for Digital
         TV</cím>
  <szszám>0</szszám>
</könyv> □
```

14.51. példa: Kérdezzük továbbra is a könyvészetet tartalmazó XML dokumentumot és keressük a 2000-ben kiadott könyvek címét. A $b változó értékei rendre a dokumentumban található könyv elemek. Egy adott könyv elem esetén, melyet a $b változó tartalmaz a WHERE kulcsszó utáni feltétel segítségével teszteljük, hogy az év tulajdonság egyenlő-e 2000-el. Cím elemeket adunk meg eredményként, és vegyük észre, hogy elemet bevezető és elemet záró tag generálódik.

```
FOR $b IN doc("bibl.xml")//könyv
WHERE $b/@év = "2000"
RETURN $b/cím
```

A lekérdezés eredménye:

```
<cím>Data on the Web</cím> □
```

14.52. példa: Keressük azon könyveket, melyeknek több mint 2 szerzőjük van. Egy adott könyv elem esetén, melyet a $b változó tartalmaz, a $c változó az aktuális könyv elem összes szerző

gyerek elemére hivatkozik. A WHERE kulcsszó utáni feltétel segítségével csak azokat a könyv címeket válogatjuk ki, melyek szerzőinek a száma kettőnél nagyobb.

```
FOR $b IN doc("bibl.xml")//könyv
LET $c := $b//szerző
WHERE count($c) > 2
RETURN $b/cím
```

A lekérdezés eredménye:

```
<cím>Data on the Web</cím> □
```

14.53. példa: Lássunk egy egyszerű példát az eredmény rendezésére. A könyvek cím szerint ábécé sorrendben jelennek meg az eredményben:

```
FOR $c IN doc("bibl.xml")//cím
ORDER BY $c
RETURN $c □
```

14.54. példa: Lássunk egy komplexebb példát az eredmény rendezésére. Rendezzük a könyv elemeket az első szerző neve szerint!

```
FOR $b IN doc("bibl.xml")//könyv
LET $a1 := $b/szerző[1]
ORDER BY $a1/vnév, $a1/knév
RETURN $b/cím □
```

Az eredmény előállításában, a `RETURN` kulcsszó után is használhatunk XQuery kifejezéseket.

14.55. példa: A következő példa árajánlatokat ad meg, elemkonstruktor segítségével.

```
FOR $b IN doc("bibl.xml")//könyv
RETURN <árajánlat> {$b/cím, $b/ár } </árajánlat>
```

A lekérdezés eredménye:

```
<árajánlat>
  <cím>TCP/IP Illustrated</cím> <price>65.95</price>
</árajánlat>
<árajánlat>
  <cím>Advanced Programming in the UNIX Environment</cím>
  <ár>65.95</ár>
</árajánlat>
<árajánlat>
  <cím>Data on the Web</cím>
  <ár>39.95</ár>
</árajánlat>
<árajánlat>
  <cím>The Economics of Technology and Content for Digital
           TV</cím>
  <ár>129.95</ár>
</árajánlat> □
```

A distinct-values() függvény egy csomópont sorozat esetén különböző értékek sorozatát adja meg, és az azonos értékeket kiküszöböli.

14.56. példa: A könyvészetet tartalmazó XML dokumentum esetén, egy szerzőnek két könyve van. Ha a szerzőkre vagyunk kíváncsiak és Stevens-t csak egyszer akarjuk megadni, akkor a következő kifejezést használhatjuk:

```
distinct-values(doc("bibl.xml")//szerző/vnév)
```

A lekérdezés eredménye:

```
Stevens Abiteboul Buneman Suciu
```

Amint látjuk nem elemeket, csak azok értékeit adja meg a distinct-values.

14.57. példa: Abban az esetben, ha az előbbi példa esetén szerző elemeket akarunk megadni, azt a következőképpen tehetjük meg:

```
FOR $n IN distinct-values(doc("bibl.xml")//szerző/vnév)
RETURN <vnév>{ $n }</vnév>
```

A lekérdezés eredménye:

```
<vnév>Stevens</vnév>
<vnév>Abiteboul</vnév>
<vnév>Buneman</vnév>
<vnév>Suciu</vnév> □
```

Az XQuery lehetőséget ad, hogy az IDREF típusú tulajdonságok által adott hivatkozást kövessük. Ha *x* IDREF típusú tulajdonságok halmaza, akkor az *x* => *y* kifejezés megadja azon objektumokat, melyek tag neve *y* és az ID tulajdonsága egyezik valamelyikkel az IDREF-ek közül.

14.58. példa: Tekintsük az első (lásd 14.23) zenés CD-ket leíró DTD és XML adatokat és a következő lekérdezést: Keressük a rock stílusú albumok zeneszámait és előadójuk nevét!

```
FOR $a IN doc("zene.xml")//Album
    $z IN $a/@Zeneszámai=>Zeneszám
LET $e := $a/@Előadója=>Előadó
WHERE $a/Stílus = "rock"
RETURN <rockZene>
{  $z/SzCíme, $e/ENév }
</rockZene>
```

Az `$a` változó az Album elemeken fut végig, a szűrőfeltétel a rock zenét válogatja ki. Az Album elemnek Zeneszámai nevű IDREFS típusú tulajdonsága van és a `$z` változó értéke rendre Albumon belül az összes Zeneszám elem. Ezt úgy tudjuk elérni, ha a => operátorral a hivatkozást követjük. Mivel a Zeneszámai egy tulajdonság, a `@` jel kell megelőzze, majd a hivatkozott elem típusát, vagyis a Zeneszám-ot kell megadjuk. Hasonlóan az Előadója is IDREF típusú tulajdonsága az Album elemnek, a hivatkozást (@Előadója=>) követve és a hivatkozott elem típusát megadva: Előadó, az Album Előadó elemét kapjuk meg.

A lekérdezés eredménye a zene.xml állomány adatait felhasználva:

```
<rockZene>
  <SzCíme>In the Shadows</SzCíme>
  <ENév>The Rasmus</ENév>
</rockZene>
<rockZene>
  <SzCíme>Guilty</SzCíme>
  <ENév>The Rasmus</ENév>
</rockZene> □
```

14.59. példa: Maradjunk az AlbumAB-nál és tekintsük a következő lekérdezést: Adjuk meg zenestíluson belül a rendelkezésünkre álló zene időtartamát, a következő formában:

```
<StílusösszIdő>
  <StílusCsop>
    <Stílus>rock</Stílus>
    <Időtartam>1233</Időtartam>
  </StílusCsop>
  <StílusCsop>
    <Stílus>rap</Stílus>
    <Időtartam>560</Időtartam>
  </StílusCsop>
        ⋮
</StílusösszIdő>
```

A feladatot a következő lekérdezés adja meg. Az $s változó a különböző stílus értékeket veszi fel. Egy adott stílus érték esetén az $a változó azon albumok sorozatát tartalmazza, melyeknek a stílusa az $s változó értékével egyenlő. Ezen albumok összes zeneszáma a $z

változóba kerül, melynek Időtartam nevű elemének értékét összeadjuk a sum($z/Időtartam) függvény segítségével, mely egy értéket és nem elemet ad vissza, ezért szükség volt megadni az elemet bevezető és elemet záró tagot. A distinct-values függvény is értéket és nem elemet ad vissza.

```
<StílusösszIdő>
FOR $s IN distinct-values(doc("zene.xml")//Album/Stílus)
LET $a := doc("zene.xml")//Album[Stílus=$s]
LET $z=$a/@Zeneszámai=>Zeneszám
RETURN
  <StílusCsop>
   <Stílus>{$s}</Stílus>
   <Időtartam>{ sum($z/Időtartam) } </Időtartam>
  </StílusCsop>
</StílusösszIdő> □
```

Ha a zenés albumok XML adatai hierarhikus formában vannak megadva (lásd a 14.24 példát) és a zene2.xml állományban vannak tárolva, a lekérdezéseket is másképpen kell megadjuk.

14.60. példa: Keressük a rock stílusú albumok zeneszámait és előadójuk nevét!

```
FOR $a IN doc("zene2.xml")//Album
    $z IN $a/Zeneszám
LET $e := $a/parent::Előadó
WHERE $a/Stílus = "rock"
RETURN <rockZene>
{  $z/SzCíme, $e/ENév }
</rockZene>
```

Az `$a` változó az Album elemeken fut végig, a szűrőfeltétel a rock zenét válogatja ki. A Zeneszám ebben az esetben gyerekeleme az albumnak, az előadó pedig szülő eleme az albumnak. □

14.61. példa: Adjuk meg zenestíluson belül a rendelkezésünkre álló zene időtartamát:

```
<StílusösszIdő>
FOR $s IN distinct-values(doc("zene2.xml")//Album/Stílus)
LET $a := doc("zene.xml")//Album[Stílus=$s]
LET $z=$a/Zeneszám
RETURN
 <StílusCsop>
   <Stílus>{$s}</Stílus>
   <Időtartam>{ sum($z/Időtartam) } </Időtartam>
 </StílusCsop>
</StílusösszIdő> □
```

A join műveletetre is ad lehetőséget az XQuery. Két xml állomány adatait úgy kapcsolhatjuk össze, hogy mindegyikhez egy-egy változót rendelünk. Ha csak ennyit teszünk, akkor az xml adatok Descartes szorzatát kapjuk. Valószínű erre nagyon ritkán lesz szükségünk, viszont a join műveletre gyakrabban. A joint úgy valósíthatjuk meg, hogy a Descartes szorzatból a where feltétel segítségével kiválogatjuk a megfelelő párokat.

14.62. példa: Legyen ismét a könyvészetet leíró adat a bibl.xml állományban és feltételezzük, hogy ezen könyvekről bírálatokat tartalmazó adatok egy review.xml állományban vannak tárolva. Legyen a reviews.xml állomány tartalma:

```
<bírálatok>
  <bírálat>
     <cím>TCP/IP Illustrated</cím>
     <osztályzás>5</osztályzás>
     <megjegyzés>Excellent technical content. Not much plot.
     </megjegyzés>
  </bírálat>
  <bírálat>
     <cím>Data on the Web</cím>
```

```
     <osztályzás>4</osztályzás>
     <megjegyzés>A well founded introduction to semistructured
                      data. </megjegyzés>
  </bírálat>
</bírálatok>
```

A következő lekérdezés a bibl.xml állományban található bibliográfiai elemekhez a róluk szóló bírálatokat társítja, ha van róluk bírálat a reviews.xml állományban. A könyv vagy cikk *címe* az, ami alapján összetársítjuk a könyvet, illetve a cikket, a róla szóló bírálattal. Az eredmény cím és bírálat párosokat fog tartalmazni.

```
FOR $c IN distinct-values(doc("bibl.xml")//cím)
      $b in doc("reviews.xml")//bírálat
WHERE $c = $b/cím
RETURN <bírálat>{ $c, $b/megjegyzés }</bírálat>
```

A lekérdezés eredménye:

```
<bírálat>
     <cím>TCP/IP Illustrated</cím>
     <megjegyzés>Excellent technical content. Not much plot.
     </megjegyzés>
</bírálat>
<bírálat>
     <cím>Data on the Web</cím>
     <megjegyzés>A well founded introduction to semistructured
          data. </megjegyzés>
</bírálat>
```

Amint látjuk, csak azok a könyvek jelennek meg, melyekről van bírálat, tehát ez egy belső (inner) join. □
Ha azokat a könyveket is az eredménybe szeretnénk látni, melyekről nem létezik bírálat, a következőképpen tehetjük meg:

14.63. példa: Ez a példa egy külső baloldali összekapcsolás (left outer join) a könyvek (bibl.xml) és bírálatok (reviews.xml) között.

```
FOR $c IN doc("bibl.xml")//cím
RETURN
<bírálat>
{ $c }
   {
     FOR $b in doc("reviews.xml")//bírálat
     WHERE $b/cím = $c
     RETURN $b/megjegyzés
 }
</bírálat>
```

14.64. példa: Vegyük ismét a zenés CD-ket tartalmazó példát, melyet még úgy is megtervezhetjük, hogy három különböző állományba tároljuk az előadókat, albumokat és zeneszámokat.
Vegyük az Előadók.xml-t, melyben az előadókat tároljuk és szerkezete a következő:

```
<!DOCTYPE Előadók[
     <!ELEMENT Előadók (Előadó*)
     <!ELEMENT Előadó (ENév, Nemzetiség) >
     <!ELEMENT ENév (#PCDATA)>
     <!ELEMENT Nemzetiség (#PCDATA)>
]>
```

Tekintsük az Albumok.xml-t, melyben az albumokat tároljuk a következő szerkezettel:

```
<!DOCTYPE Albumok[
     <!ELEMENT Albumok (Album*)
     <!ELEMENT Album (ACíme, ElőadóNév, Stílus, MegjelenEve) >
     <!ELEMENT ACíme (#PCDATA)>
```

```
        <!ELEMENT ElőadóNév (#PCDATA)>
        <!ELEMENT Stílus (#PCDATA)>
        <!ELEMENT MegjelenEve (#PCDATA)>
]>
```

A Zeneszamok.xml, a zeneszámokat tartalmazza és szerkezete a következő:

```
<!DOCTYPE Zeneszámok [
        <!ELEMENT Zeneszámok (Zeneszám*)
        <!ELEMENT Zeneszám (SzCíme, AlbumCíme, Időtartam) >
        <!ELEMENT SzCíme (#PCDATA)>
        <!ELEMENT AlbumCíme (#PCDATA)>
        <!ELEMENT Időtartam (#PCDATA)>
]>
```

Amint látjuk az albumnál hivatkozunk az előadóra a nevével, illetve a zeneszámtól az albumra.

Ebben az esetben, ha kíváncsiak vagyunk a zeneszámok előadóira, illetve stílusára, használhatjuk a join műveletet.

```
FOR $z IN doc("Zeneszamok.xml")//Zeneszám
    $a in doc("Albumok.xml")//Album
WHERE $z/AlbumCíme = $a/ACíme
RETURN <szám>
{ $z/SzCíme, $z/AlbumCíme, $a/ElőadóNév, $a/Stílus}
       </szám> □
```

15. Könyvészet

[AbCo01] M. Abbey, M. J. Corey, I. Abramson: *Oracle8i, Kézikönyv kezdőknek*, Panem Kiadó, 2001.

[Ab97] S. Abiteboul: *Quering Semi-Structured Data*, ICDT '97, Lecture Notes in Computer Science, Springer-Verlag, 1997.

[AbBuSu00] S. Abiteboul, P. Buneman, D. Suciu: *Data on the Web*, *From Relations to Semistructured Data and XML*, Morgan Kaufmann Publishers, SanFransisco, California, 2000.

[AbHuVi95] S. Abiteboul, R. Hull, V. Vianu: *Foundations of Databases*, Addison-Wesley Publishing Company, 1995.

[AQMW97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener: *The Lorel query language for semistructured data*, International Journal on Digital Libraries, 1(1):pp.68-88, April 1997.

[AtBaDe90] M. Atkinson, F. Bancilhon, D. DeWitt: *The Object-Oriented Database System Manifesto*, Deductive and Object-Oriented Databases, 1st Intern. Conf. Kyoto, Japan, pp. 223-239, North Holland, W. Kim, J. M. Nicolas, S. Nishio ed, 1990.

[BaWi] B. Bakker, I. Widarto: *An Introduction to XQuery*,

http://www.perfectxml.com.

[BeMa93] E. Bertino, L. Martino: *Object-Oriented Database Systems*, Concept and Arhitectures, Addison-Wesley, 1993.

[BeNePe92] E. Bertino, M. Negri, G. Pelagatti: *Object-Oriented Query Languages: The Notion and the Issues*, IEEE Trans. on Knowledge and Data Engin. Vol. 4 No. 3, pp. 223-237, 1992.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu: *A Query Language and Optimization Techniques for Unstructured Data*, SIGMOND Proc. 1996.

[ChSt93] D. N. Chorafas, H. Steinmann: *Object-Oriented Databases*, PTR Prentice-Hall, 1993.

[Co70] E. Codd: *A Relational Model of Data for Large Shared Data Banks*, Communications of ACM, 13(6), pp: 377-387, 1970.

[Co82] E. F. Codd: *Relational Databases: A Practical Foundation for Productivity*, Communication ACM, pp: 109-117, 1982.

[Da04] C. J. Date: *An Introduction to Database Systems*, 8th Edition, Pearson Education, Inc. Addison-Wesley Higher Education, 2004.

[DFFLS99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu: *A query language for XML*, International World Wide Web Conference, 1999.

[GaUlWi00] H. Garcia-Molina, J. D. Ullman, J. Widom: *Database System Implementation*, Prentice Hall Upper Saddle River, New Jersey, 2000.

[Gr93] G. Graefe: *Query Evaluation Techniques for Large Databases*, ACM Computing Surveys, 25(2), 1993.

[Ha94] B. Halassy: *Az adatbázis tervezés alapjai és titkai,* IDG Hungary, 1994.

[He94] P. Helman: *The science of Database Management*, Irwin, 1994.

[KoSh98] D. Konopnicki, O. Shmueli: *Information Gathering in the World Wide Web: The W3QL Query Language and the W3QS System*, ACM Trans. on Database Systems, Vol. 23, No. 4, pp. 369-410, 1998.

[LaStWa97] J. B. Lagorce, A. Stockus, E. Waller: *Object-Oriented Database Evolution*, Database Theory ICDT 6-th Intern. Conf. Delphi, pp. 379- 393, Springer-Verlag, 1997.

[LeRiVe88] C. Lecluse, P. Richard, F. Velez: *O_2, an Object-Oriented Data Model*, Conf. on Object-Oriented Database Systems, pp. 227-236, 1988.

[LiMo01] Q. Li, B. Moon: *Indexing and Quering XML Data for Regular Path Expresions*, Proc. Of 27th VLDB Conf. Roma, 2001.

[LBBI95] I. Lungu, C. Bodea, G. Bădescu, C. Ioniţă: *Baze de date*, *Organizare, proiectare şi implementare*, ALL Educational, 1995.

[Lu95] T. Luers: *Bazele Oracle 7*, Teora, 1995.

[McAbGoWi97] J. McHugh, S. Abiteboul, R. Goldman, J. Widom: *Lore: A database management system for semistructured data*, SIGMOND Record, 26(3), pp.54-66, September, 1997.

[McWi99] J. McHugh, J. Widom: *Query Optimization for Semistructured Data*, Proc. of VLDB, Edingurgh, England, 1999, pp. 315-326

[ÖzVa91] M. T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*, Prentice-Hall, 1991.

[Ra98] R. Ramakrishnan: *Database Management Systems*, WCB McGraw-Hill, Boston, 1998.

[RaMo03] S. Raghavan, H. Garcia-Molina: *Representing Web graphs*. In Proc. of IEEE Intl. Conf. on Data Enginnering, March 2003.

[St96] M. Stonebraker: *Object-Relational DBMSs*, Morgan Kaufmann Publishers, San Fransisco, 1996.

[StOz95] D. D. Straube, M. T. Ozsu: *Query Optimization and Execution Plan Generation in Object-Oriented Data Management Systems*, IEEE Trans.on Knowledge and Data Engin. Vol.7 No.2, pp. 210-227, 1995.

[Šu03] Šunderic: *SQL Server 2000, Stored Procedure & XML Programming*, Second Edition, McGraw-Hill/Osborne, 2003.

[Ul88] J. D. Ullman: *Principles of Databases and Knowledge-Base Systems, Vol I: Classical Database Systems*, Computer Science Press, New York, 1988.

[Ul89] J. D. Ullman: *Principles of Databases and Knowledge-Base Systems, Vol II: The New Technologies*, Computer Science Press, New York, 1989.

[Ul92] J. D. Ullman: *A Comparison between Deductive and Object-Oriented Database Systems*, Deductive and Object-Oriented Databases, 2nd Intern. Conf. Munich, pp. 263-277, Springer-Verlag A. Pirotte, C. Delobel, Gottlob ed., 1992.

[UlWi97] J. D. Ullman, J. Widom: *A First Course in Database Systems*, Prentice Hall Upper Saddle River, New Jersey, 1997.

[VaKa95] Z. Kása, V. Varga: *A Practical Approach to Security in Databases*, Research Seminar on Computer Science, Universitatea "Babeş-Bolyai" Cluj, pp. 15-18, 1995.

[VaKa96] Z. Kása, V. Varga: *A Graph Model for Distributed Database Systems*, Studia Univ. "Babeş-Bolyai" Cluj-Napoca, Informatica, Vol. I, Nr. 2, pp. 3-8, 1996.

[Va97] V. Varga: *An Algorithm for Data Localization in Distributed Database Systems*, Research Seminar on Computer Science, Universitatea "Babeş-Bolyai" Cluj-Napoca, Preprint Nr. 2, pp. 61-68, 1997.

[Va98] V. Varga: *Stochastic Optimization for the Join of three Relations in Distributed Databases I. The Theory and one Application*, Studia Univ. "Babeş-Bolyai" Cluj-Napoca, Informatica, vol. XLIII, No. 2, 1998, pp. 37-46.

[Va99] V. Varga: *Stochastic Optimization for the Join of three Relations in Distributed Databases II. Generalization and more Applications*, Studia Univ. "Babeş-Bolyai" Cluj-Napoca, Informatica, vol. XLIV, No. 1, 1999, pp. 55-62.

[VaMo99] V. Varga, C. Moroşanu: *Stochastic Optimization for the Join of four Relations in Distributed Databases*, BAC -XC -B /1999 Proceedings of the PAMM conf. PC-126/ God, Budapesta, pp. 85-102.

[VaMaMo01] T. Márkus, C. Moroşanu, V. Varga: *Stochastic Query Optimization in Distributed Databases using Semijoins*, Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae Sectio Computatorica 20(2001), pp. 107-131.

[VaDuGr04] V. Varga, D. Dumitrescu, C. Grosan: *Solving Stochastic Optimization in Distributed Databases using Genetic Algorithms*, Advances in Databases and Information Systems, LNCS3255, 8th East-European Conference, ADBIS 2004, Budapest, Hungary, September 2004, Springer, pp. 259-274.