Memóriaadatbázisok

áttekintő előadás az

Adatbázisok haladóknak

c. tárgy keretében, 2012. szeptember 18.

Szerző: *Marton József Ernő* BME-VIK TMIT

Tartalom

Bevezetés

Memóriaadatbázison (IMDB[1]) olyan adatbázis rendszert értünk, ahol a tárolt és kezelt adatok *elsődleges*[2] példánya a fizikai memóriában található [GaK92], szemben a lemez-alapú[3] adatbázisokkal (DRBD[4]), ahol az a lemezes-alapú háttértáron található. Az esetleges biztonsági másodpéldányok persze lehetnek lemezes vagy egyéb nem felejtő tárolón.

Ahogy növekszik az elérhető memória-kapacitás, egyre több alkalmazás által kezelt összes adat fér el a fizikai memóriában. A fizikai memóriában tárolt adatok hozzáférési ideje nagyságrendekkel kisebb, mint a lemezről beolvasandó adaté. Ebből következően a feldolgozási idők lerövidülhetnek, ami idő-kritikus, akár soft vagy hard real-time feladatok elvégzését teszi lehetővé. Persze mindig lesz olyan adattömeg, ami meghaladja az elérhető fizikai memória-kapacitást. Az ilyen adattömegeknek is lehet olyan része, amelyet a feldolgozási műveletek az átlagosnál jóval gyakrabban használnak. Ezen tulajdonság alapján érdemes lehet az adatbázist egy IMDB és egy DRDB részre osztani (partícionált adatbázis).

Bizonyos alkalmazások nem igénylik az adataik tartósságát: főként beágyazott rendszerekben jellemző, hogy a funkcionalitásához szükséges adatbázist működés közben építi fel (pl. IP routerek forgalomirányítási táblája) vagy induláskor letölti egy fejállomásról (pl. programajánló információs rendszerek). Az utóbbiak számára nem is a memóriában tárolt adatok gyorsabb elérése miatt csábító választás egy memóriaadatbázis-rendszer[5]. Sokkal fontosabb, hogy bonyolult, a DRDB-hez kapcsolódó funkciók, mint például a lemezes IO-alrendszer, vagy a tranzakció-naplók vezetéséhez szükséges alkalmazás-logika és memória-szükséglet eliminálható. Ezáltal kisebb lesz az eszköz/alkalmazás memória-lenyomata[6] és számítási kapacitás-szükséglete is, így az előállítási költsége[7] [Ste02].

A telekommunikációs rendszerek esetén akár a CAC[8], akár a számlázáshoz szükséges forgalmi adatgyűjtés területén szükséges magas átbocsátó képesség miatt jó választás lehet IMDB alkalmazása.

---

1 IMDB – In-Memory DataBase, vagy MMDB – Main-Memory DataBase: memória-alapú adatbázis, vagy memóriaadatbázis

2 Elsődleges vagy munkapéldány: a logikai adatelem azon példánya vagy példányainak összessége, amin a tranzakciók műveleteket végeznek.

3 Az SSD-k térnyerésének idejét éljük, de jelenleg azokat még tipikusan a diszkekhez hasonló elérést és szervezést lehetővé tevő interfésszel találjuk meg a piacon. Ekkor a „lemez-alapú" helyett pontosabban úgy fogalmazhatunk, hogy a blokkos elérést lehetővé tevő tároló fölött működő adatbázis-kezelő rendszer.

4 DRDB – Disk-Resident DataBase: lemezes tároló alapú adatbázis

5 IMDBMS – In-Memory Database Management System: memória-alapú adatbázis-kezelő rendszer

6 memory footprint

7 Nem jellemző beágyazott rendszerekben DRDBMS alkalmazása. Az említett okokból a saját fejlesztésű adatszerkezetek alkalmazását válthatja ki egy IMDBMS rendszerrel a rendszerfejlesztő. Az erőforrás-igényt tehát ebben a környezetben kell vizsgálni, és dönteni, hogy a saját fejlesztésű megoldás vagy egy esetleges speciális rendszer alkalmazása éri meg jobban.

8 CAC – Call Admission Control: hívásengedélyezés

Időrendi áttekintés

Az 1980-as évek első felében a high-end rendszerekben néhány megabyte és néhányszor tíz megabyte közötti fizikai memória volt elérhető. Ezzel összhangban eleinte a kutatások a megnövekedett puffer-kapacitás kihasználása irányában összpontosultak megmaradván a DRDB séma mellett. Az évtized közepére már 128 MB is elérhető volt [LeC86] a high-end rendszerekben. Ekkor került az érdeklődés középpontjába a relációs adatbázisok memória-rezidens tárolása. Különböző, a fizikai memória tulajdonságait kihasználó adatszerkezeteket alkottak a relációk és az indexek tárolására valamint a lekérdezések optimális végrehajtására.

Az évtized vége felé a tranzakciókezelés és a zavartalan működés mellett készített backup-ok és visszaállítási pontok kérdése volt a fő kutatási irány. Ekkor a piacon már kereskedelmi termékként is elérhetőek voltak IMDB rendszerek, például az IBM *IMS/VS Fast Path* vagy *Office by example* [GaK92] rendszerei.

Az 1990-es évek stagnálását követően 2001-ben újabb, már a processzor-cache adta lehetőségeket is jobban kihasználó módszereket, index-adatszerkezeteket [BMR01] publikálnak.

Növekszik a kereskedelemben elérhető eszközök száma. A napjainkban elérhető rendszerek közül kettőt említenénk: beágyazott rendszerekben az *eXtremeDB*[9], míg enterprise rendszerekben a *TimesTen*[10] egy-egy képviselője az IMDB elvnek.

Mára[11] 1GB RAM költsége 20USD alatt van, és néhány TB-os memória-kapacitású szerver is könnyedén elérhető a piacon.

Legfontosabb kérdések

Vajon jogos-e feltételeznünk, hogy az egész adatbázis a memóriában van? Mi történik a memóriában tárolt adatokkal a tápfeszültség kimaradása esetén? Van-e lényegi különbség az IMDB és a nagy cache-el felszerelt DRDB rendszerek között? Vizsgáljuk meg ezeket a kérdéseket közelebbről!

Elférhet-e teljes adatbázis a memóriában?

Érezzük, hogy a kérdés nincs pontosan megfogalmazva. Az egyetlen korrekt válasz csak az “attól függ” lehet [GaK92]. Vannak olyan alkalmazások, ahol a kezelt adat mérete alapján a válaszunk egyértelműen igen. Ilyenek például a korábban említett telekommunikációs routerek vagy kisebb raktárkészleteket kezelő adatbázisok.

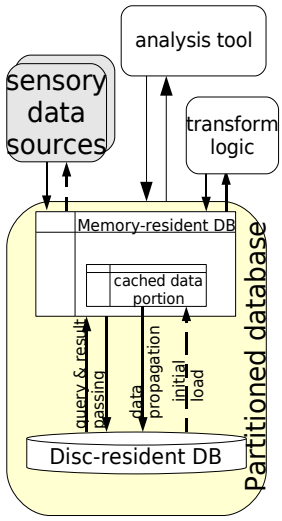
Bizonyos esetekben – ha szükséges – elérhető minden adat memória-rezidens tárolása. Tekintsünk például egy néhány százezer kötetes könyvtárat, amelynek néhány tízezer olvasója van.

9 McObject's eXtremeDB – http://www.mcobject.com/extremedbfamily.shtml
10 Oracle TimesTen – http://www.oracle.com/timesten/index.html
11 2012. ősz

Egy ezt kezelő adatbázis mérete néhányszor 5GB nagyságú lehet. Ma könnyűszerrel lehetséges ennyi memóriával felszerelni egy adatbázisszervert. Egy könyvtári adatbázis azonban mégsem az az eset, ahol ez ma megéri: a napi feladatok ellátására tökéletesen megfelel egy DRDB.

A másik véglet lehet a SETI projekt által begyűjtött gigantikus mennyiségű adat, vagy időjárási műholdképek. Ezek memória-rezidens tárolására valószínűleg sosem lesz elég a rendelkezésre álló operatív memória.

Az utóbbi két esetben érdemes lehet megfontolni az adatok egy részének memória-rezidens tárolását, esetleg a hozzáférési gyakoriság függvényében az adatok migrálását a DRDB és IMDB rendszerek között. A migráció nem csupán gyorsítótárazás! Az adatokat olyan reprezentációra (adatszerkezetek) kell hozni, ami kihasználja a tároló médium adottságait. A "nyerő stratégia" talán a partícionált adatbázis lehet: az adatbázis-kezelő rendszer automatikusan felismeri, hogy a rendelkezésre álló fizikai memória és az adatok hozzáférési-statisztikái alapján az adatbázis melyik részét kezeli IMDB-hozzáállással, és melyik részét DRDB-elven [GaK92]. Az automatikus felismerés helyett a felosztást a rendszer tervezőjének kezébe is adhatjuk.



1. ábra: Az elemzőeszközök a partícionált adatbázishoz csatlakoznak, amíg az adatforrások friss adati közvetlenül az IMDB részbe érkeznek, hogy minél hatékonyabb lehessen az előfeldolgozásuk.

Az 1. ábrán szenzor-adatfolyamok fogadására és analízisére kialakított adatbázis látható partícionált adatbázis-kezelő rendszerben. Az adatbázis-kezelővel kapcsolatban álló komponensek egy része kapcsolódhat közvetlenül az IMDB vagy DRDB részhez: ekkor szükséges figyelembe vennie, hogy az adatbázisnak nem az egészét látja. Kapcsolódhat a partícionált adatbázis-kezelőhöz, mint egészhez, amikor a rendszer belső mechanizmusai biztosítják, hogy a műveletek (lekérdezések) az adatbázis adatainak teljességével dolgozzanak függetlenül attól, hogy egyes adatelemek az IMDB, DRDB, vagy éppen mindkét részben megtalálhatóak.

Nem-felejtő memória kérdése

A lemez passzív adattároló eszköz: az adatok a tárolási elv sajátosságai miatt minden erőfeszítés nélkül megmaradnak a tápfeszültség kimaradása esetén is, ha egyszer felírásra kerültek. A fizikai memória ezzel szemben elveszti az adatait, ha a tápfeszültség kiesik. Ennek a valószínűsége csökkenthető UPS vagy telep alkalmazásával. De továbbra is aktív adattároló[12] eszközként viselkedik: ha az UPS-ből kifogy az energia, az adatok ugyanúgy elvesznek [GaK92]. A memória bithiba-valószínűsége hardveres támogatással csökkenthető éppúgy, mint a diszkek esetén: hibajelző ill. javító kódolással vagy modulháromszorozással (vö. RAID megoldások).

12 aktív adattároló – az adatok tárolásához a tápfeszültség folyamatos rendelkezésre állása szükséges.

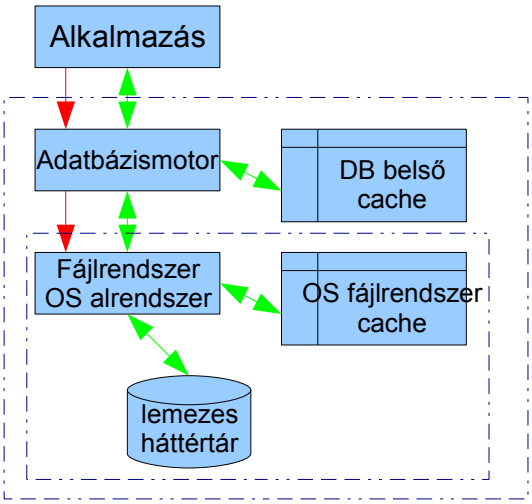
Összefoglalva nem mondható, hogy létezik nem-felejtő memória, de az sem, hogy nem létezik. A lényeg az, hogy bizonyos technikákkal fokozható a megbízhatósága, de ezzel nem szűnik meg a backup-ok létjogosultsága.

IMDBMS a nagy gyorsítótárral szerelt DRDBMS-sel szemben

Felvetődik a kérdés, mi a különbség a nagy memória-gyorsítótáras DRDB rendszerek és az IMDB között. A gyorsítótár lehet az operációs rendszer illetőleg a DBMS szintjén. Tekintsük előbb az első esetet: az operációs rendszer a lemezes háttértároló tartalmának egy részét cache-seli. Ekkor az adatbáziskezelő rendszer a szokásos rendszerhívással kéri a megfelelő adat beolvasást. A lemezkezelő alrendszer kiszámítja az adat fizikai címét a lemezen, majd ellenőrzi, hogy benn van-e a cache-ben. Találat esetén visszatér, egyébként beolvasás után tér csak vissza. Ebben az esetben tehát a rendszerhívások és a kapcsolódó környezetváltások nem spórolhatók meg, csupán a blokk-beolvasás [Ste02]. Szembetűnő tehát, hogy az adatok lemezen való elrendezésére különös figyelmet kell fordítani annak érdekében, hogy a cache-találatok aránya nagy legyen.

A 2. ábrán az az eset látható, amikor a DBMS saját gyorsítótárat tart fenn. Ekkor cache-találat esetén az IO-alrendszerhez fordulás összes többletmunkája megspórolható. A lényegi különbség az, hogy a memóriában más adatszerkezetek használata célszerű, mint a lemezes tároláskor. Egyszerű cache-selés esetén bár bekerül az adat a memóriába, de a reprezentációja diszkre optimalizált: például egy index B-fában a memóriában tárolva nem optimális, hiszen mások a költségfüggvények, amit korábban bemutatott T-fa adatstruktúra [LeC86b] hatékonyabban használ ki.

További különbség, hogy bizonyos bonyolult alkalmazáslogikai elemek (pl. a diszken az adatok elrendezését optimalizáló, esetleg klaszterező megoldások) implementálása szükségtelen, mert az egyes feladatokra a memória sajátosságait jobban kihasználó, egyszerűbb módszerek létezhetnek. Például klaszterezés helyett az előreszámított illesztések mutatókkal implementálhatók [LeC86].



2. ábra: adathozzáférés komponensei DRDB esetén
a piros nyilak üzenetátadást, míg a zöldek adatáramlást jelölnek
a szaggatott keret átlépése a futási környezetváltásokat mutatja

Kommunikációs alternatíva

A hagyományos adatbáziskezelő-rendszerek esetében megszokott kliens-szerver modell az IMDB rendszerenél is elérhető. Amikor azonban a 2. ábrán látható környezetváltások egy részét megspóroltuk, felmerül az igény: ha egyazon gépen fut az adatbázis-kezelő és az adatokat használó alkalmazás, akkor miért ne spórolhatnánk meg az alkalmazás és

adatbázis-kezelő rendszer közötti kommunikáció közben esedékes környezetváltásokat. Ezt megtehetjük, ha az alkalmazás címterébe kerül az adatbázis-kezelő egy osztott memóriaterületen keresztül. Ez persze sok esetben biztonsági kockázatot is jelent, hiszen az alkalmazás a saját címterében szinte bármit megtehet, ha van közvetlen memória-manipulálási lehetősége. Másfelől a környezetváltáson túl is nyerünk, hiszen ha egyazon címtérben van az alkalmazás és az adatbázis, akkor a lekérdezések eredményének a visszaadása csupán egy iterátor átadását jelentheti, nincs szükség az adatok másolására.

Fontosabb részterületek

Adatszerkezetek

Egy adatbázis-rendszer adatok tárolására és kezelésére létrehozott eszköz. Természeténél fontos kérdés tehát, hogy az adatokat milyen strukturában tárolja. A választott struktúrának illeszkednie kell a tároló médium és a tárolt adat sajátosságaihoz egyaránt. Adatmodellben a jól bevált relációs megfelelő választás, a legtöbb IMDB rendszer ezt implementálja. Az implementációban azonban az IMDB rendszerek új megoldásokat igényelnek. Alább áttekintjük az indexek és a relációk tárolására alkalmazható legfontosabb adatszerkezeteket.

Index-adatszerkezetek

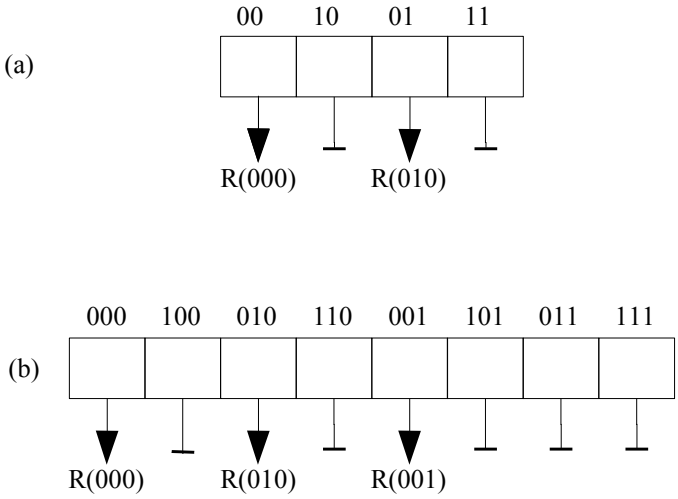
A tároló médium sajátosságait az egyes adatkezelő műveletek költségével vehetjük figyelembe. Lemezes médium esetében ez a költség lényegileg a blokk-műveletek számával arányos. Persze nem független az egyes blokkok elérésének sorrendjétől, de nagyságrendekkel kisebb mértékben szerepel abban. Az adathozzáférés költségét a fizikai memória esetében elsősorban a hozzáférendő adat mennyisége határozza meg. Bár közvetlen („random”) hozzáférésű tárról van szó, a szekvenciális hozzáférés ebben az esetben is kifizetődő, igaz, csupán csekély mértékben változtatja a hozzáférési költséget[13].

Hash[14]-alapú módszerrel két megközelítésben is találkozhatunk az IMDB terén: az indexek és a zárak implementálásához egyaránt hasznos. Ez a két eset a módszereket tekintve nem különbözik jelentős mértékben. Sokkal lényegesebbek a felvetődő skálázási, illetőleg méretezési kérdések, amelyek meghaladják e dokumentum kereteit.

13 Bizonyos körülmények között a processzor-cache hatékony alkalmazása módosíthatja ezt az állítást. Hatását később tárgyaljuk.

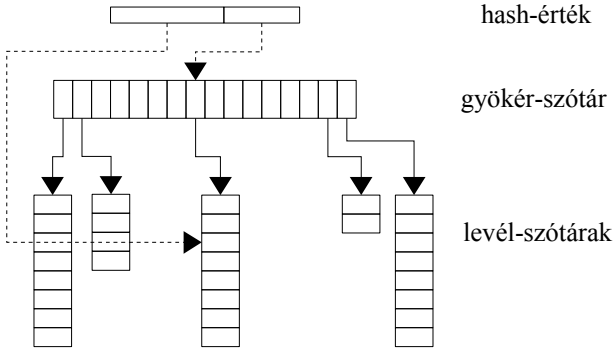
14 A K halmazon értelmezett *h: K → {0, ..., N-1}* függvényt hash-függvénynek nevezzük. Általában a képhalmaz elemszáma (számossága) sokkal kisebb, mint a *K* halmazé. A hash-alapú indexelés/elérés ezt kihasználva és el jó kompromisszumot a tárkihasználás és az elemek elérési ideje között.

A kiterjeszthető vödrös hash[15] a hagyományos vödrös hash egy adaptív változata. A kezelendő kulcsok hashelt értékéből mindig csak annyi bitet vesz figyelembe, hogy a meghatározott vödrökbe eső kulcsok száma ne haladja meg a vödör-kapacitást. Szükség esetén a releváns bitek számát növeli a módszer. Ebben az esetben egy keresett kulcs megtalálásához szükséges idő független a tárolt kulcsok számától: csupán a vödörkapacitástól függ. Amennyiben a *b* vödörkapaciás 1-nél nagyobb, úgy az átlagos tárigénye *m* rekord tárolása esetén $O\left(m^{1+\frac{1}{b}}\right)$ [AnP92]. Az 3. ábrán a hash-függvény 3 bites értékeket vesz fel, és *R(nnn)* jelöl egy megfelelő hash-értékű rekordot.



3. ábra: Kiterjeszthető vödrös hash, b=1
(a) a hash-függvény 2 bitje releváns, azaz 4 a szótár mérete
(b) vödörtúlcsordulás miatt szótárduplázás történt

Az egyszerű, többes besorolású hash[16] az előbbinek egy kiterjesztése: az első szinten egy fix szótárméretű adatstruktúra, amelynek minden bejegyzése alá egy kiterjeszthető vödrös hash kapcsolódik. Láthatóan (4. ábra) nem sokkal bonyolultabb a kapott struktúra, de a tárkihasználása az előbbinél nagyságrendileg jobb: megfelelő méretezés esetén a felhasznált tárkapacitás közel lineáris a tárolt rekordok *m* számában [AnP92]. Mivel a gyökér-szótár fix méretű, csak elegendően sok információ birtokában méretezhető úgy a struktúra, hogy az megfeleljen az aktuális igényeknek: tehát egy nem adaptív struktúráról van szó.



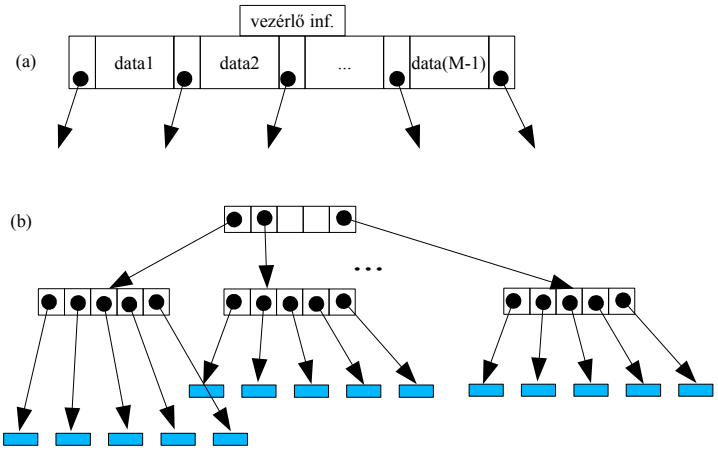
4. ábra: Egyszerű többes-besorolású hash

Ha a fa-szerkezetnek nem korlátozzuk strukturálisan a mélységét, és az egyes csomópontokban elhelyezett mini hash-szótárakat a beillesztett rekordok függvényében adaptívan növeljük illetve vonjuk össze a gyerekcsomópontokkal, akkor a Fast Search Multi-Directory Hash[17] szerkezethez juthatunk, amelyről bővebben az [AnP92] irodalomban olvashatunk. Előnye ezen adaptív struktúrának, hogy mérete – várható értékben – lineáris az *m* tárolt rekordok számában. Hátránya,

15 Extendible hashing
16 Simple multi-directory hashing
17 FSMH – Gyorsan kereshető több-szótáras hash

hogy az előbbiekhez képest bonyolultabb az implementációja, és a fenntartása több adminisztrációt igényel.

Az index adatstruktúrák másik családja a keresőfák, amelyről bővebben az [RIS98] irodalom nyújt tájékoztatást. Ebben a dokumentumban megvizsgáljuk, hogy a blokk-alapú tárolók esetén alkalmazott B-fa miért nem a legjobb választás a fizikai memóriában, és a legelterjedtebb alternatívájához, a T-fához jutunk el.



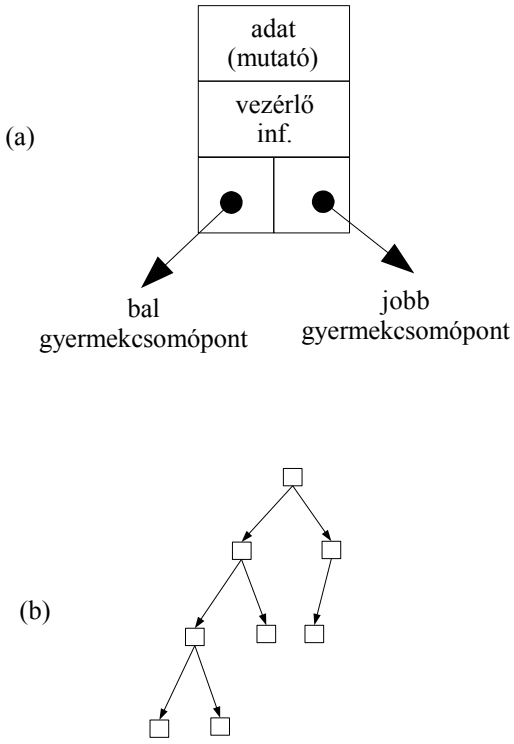
5. ábra: (a) egy B-fa csomópont
(b) egy B-fa
(a kék csomópontokban egyenként 4 adatelem foglalhat helyet)

A B-fák (5. ábra) [RIS98] belső csomópontjainak több, nevezetesen B/2..B gyermeke lehet, és minden csomópont részfái egyazon magasságúak[18]. Ha egy csomópontot egy blokkon kívánunk tárolni, akkor ezzel a feltétellel a blokk "kihasználtságát" írhatjuk elő. Így a fa mélysége a tárolt kulcsok számának logaritmusával arányos. Azonban a bejáráskor minden csomópont esetén bináris kereséssel kell eldöntenünk, hogy melyik részfában folytatjuk a keresést. Ez diszk-alapú rendszernél nem megy a hatékonyság rovására, hiszen ott a diszk-IO műveletek száma az, ami meghatározza a költséget. A memóriában gazdaságtalan lehet, mert sokáig tart.

A bináris keresőfák[19] családja ebből a szempontból jobb választás: egyetlen összehasonlítással el lehet dönteni, hogy melyik részfában kell a keresést folytatni. Vegyük azonban figyelembe, hogy a memóriában az indexelt értékeket nem kell duplikálnunk: elegendő egy mutató elhelyezése a hivatkozott n-esre[20]. Így azonban az adminisztrációhoz (a struktúra leírásához) felhasznált tárterület a hasznos területnek legkevesebb duplája, ami túlságosan sok! Ugyanannyi tárolt elemet tekintve a bináris keresőfa mélysége azonban nagyobb lehet. Ha a fa építése során megköveteljük az AVL tulajdonság[21] teljesülését, a magassága továbbra is a tárolt elemek logaritmusával lesz arányos. Egy elem keresése során tehát (konstansszor) több csomópont meglátogatására lesz szükségünk, mint a B-fák esetében, de ez a memóriában nem probléma, hiszen a mutatók követése gyors művelet.

18 Ezt az adatstruktúrához tartozó műveletek algoritmusai garantálják, melyről szintén a hivatkozott irodalomban olvashatunk.
19 Egy bináris keresőfa egy olyan fa, amelynek minden csomópontjának legfeljebb 2 gyermeke van, továbbá minden csomópontjában 1 kulcsot tárolunk. Igaz továbbá, hogy valamely csomópont bal részfájában csak az ott tárolt értéknél kisebb, míg a jobb oldali részfájában az ott tárolt értéknél nem kisebb kulcsú elemek találhatóak (ezt nevezzük keresőfa-tulajdonságnak).
20 n-touple, n-es – a rekord szinonímájának tekinthető ebben a környezetben
21 AVL tulajdonság: bináris fák esetében valamely csomóponthoz tartozó bal- és jobb-oldali részfa magasságának a különbsége legfeljebb 1 lehet. AZ AVL tulajdonságot teljesítő bináris fák az AVL fák [RIS98].

Ha a B-fa jobb tárkihasználását egyesítjük az AVL-fánál látott egyszerű követőrészfa-kiválaszthatósági tulajdonsággal, a T-fához [LeC86b] jutunk: az AVL-fa minden egyes csomópontjában több kulcs címét is tároljuk. Egy csomópontban az alkalmazott rendezés szerint szomszédos elemek foglalnak helyet, így 2 összehasonlítással eldönthetjük, hogy a keresett értéket tartalmazó csúcsot megtaláltuk (ez az ún. *bennfoglaló csúcs*), vagy kiválaszthatjuk azt a részfát, ahol a keresést folytatni kell. Ha megtaláltuk a bennfoglaló csúcsot, akkor egyetlen bináris kereséssel eldönthető, hogy valóban létezik-e a keresett érték (ha létezik, meg is található).



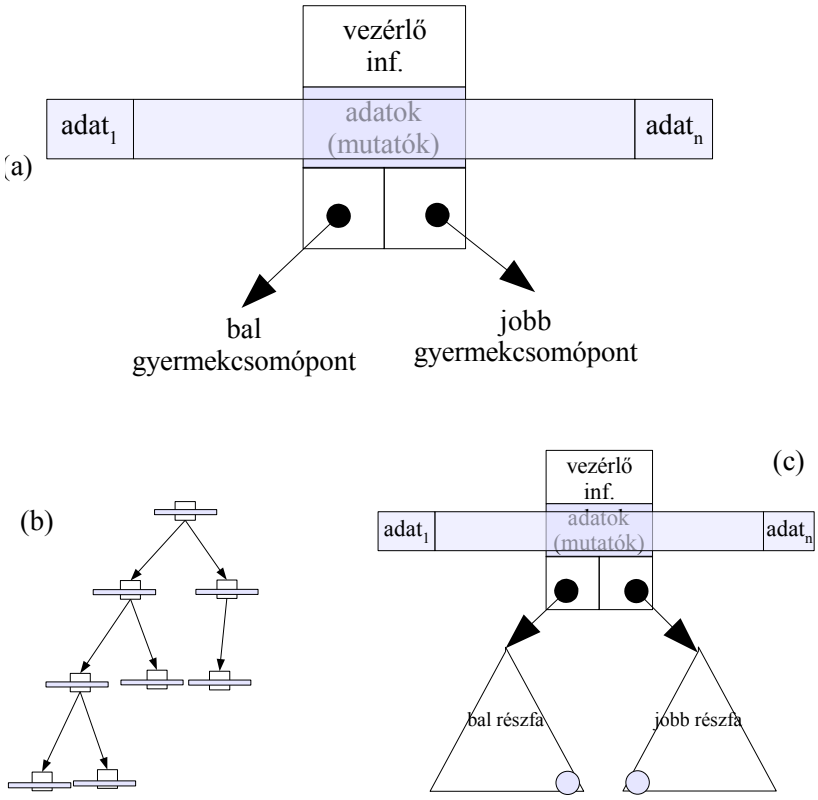
6. ábra: (a) egy AVL-fa csomópont
(b) egy AVL fa

A T-fa kezelésére használt algoritmusok [LeC86b] (elem beszúrása, törlése) lényegében megegyeznek az AVL-fákhoz tartozó eljárásokkal (lásd: [RIS98]), itt csak a fő különbségekre térünk ki.

Eddig nem említett követelmény, hogy minden belső csúcs esetén az ott tárolt adatok száma egy maximális és egy minimális érték közé esik, amelyek rendszerint csak egy kis additív konstansban térnek el egymástól. Az egy csúcsban tárolt elemek (rendezés szerinti) szomszédosságából, valamint az AVL-fáktól kölcsönzött keresőfa-tulajdonságból adódik, hogy az egy csúcsban tárolt elemek legnagyobb alsó szomszédja és a legkisebb felső szomszédja az 7/(c) ábrán kék körrel jelölt helyen van a fában.

A fába való beszúrás során előbb megkeressük a beszúrandó elem bennfoglaló csúcsát. Ha van még ott hely, akkor egyszerűen berakjuk, ellenkező esetben az ott tárolt legkisebb elemet ($adat_1$) kivesszük, és a beillesztendő adatot elhelyezzük a csúcsban annak rendezés szerinti helyére. A kivett elemet a bal részfába próbáljuk az előbbiek szerint beszúrni. Ha nincs bal részfa, akkor egy új csúcs beszúrása válik szükségessé. Ez elronthatja az AVL tulajdonságot, amelyet egyetlen (esetleg dupla) forgatással helyre lehet hozni (8/a,b ábra). A forgatás következtében elromolhat a T-fákra megkövetelt kitöltöttségi tulajdonság, amennyiben levélcsomópont is közvetlenül részt vesz a forgatásban. Ezt csupán a forgatásban résztvevő csomópontok között az elemek áthelyezésével orvosolhatjuk (8/c ábra).

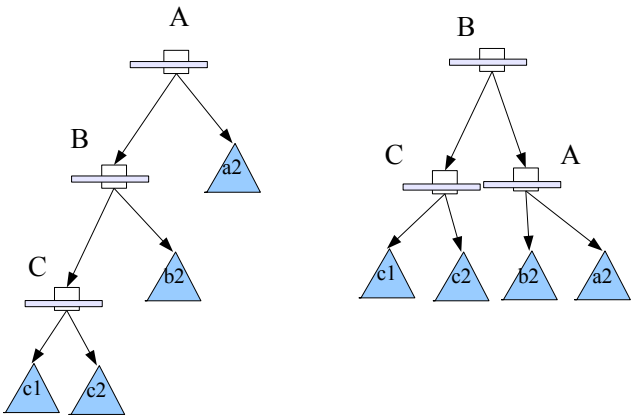
Elem törlésekor a beszúráshoz hasonlóan megkeressük a törlendő elemet befoglaló csúcsot, ahonnan egyszerűen kitöröljük azt. Ha a kitöltöttségi követelmény sérül, akkor a csúcshoz tartozó legnagyobb alsó szomszéd-elemet beemeljük. Ha újra sérülne a kitöltöttségi követelmény, akkor a sérült csúcsba emeljük át annak legnagyobb alsó szomszéd-elemét, és így tovább. A törlés során legutoljára meglátogatott csomópontban, és sorban annak őseiben ellenőrizni kell az AVL tulajdonságot, és szükség esetén forgatásokkal helyre kell hozni. Ha a forgatásban levél-csomópont is részt vesz, akkor a belső csúcsokra vonatkozó kitöltöttségi tulajdonság sérülhet, amelyet a forgatásban részt vevő csúcsok közötti elem-áthelyezéssel a beszúráshoz hasonlóan orvosolni kell.



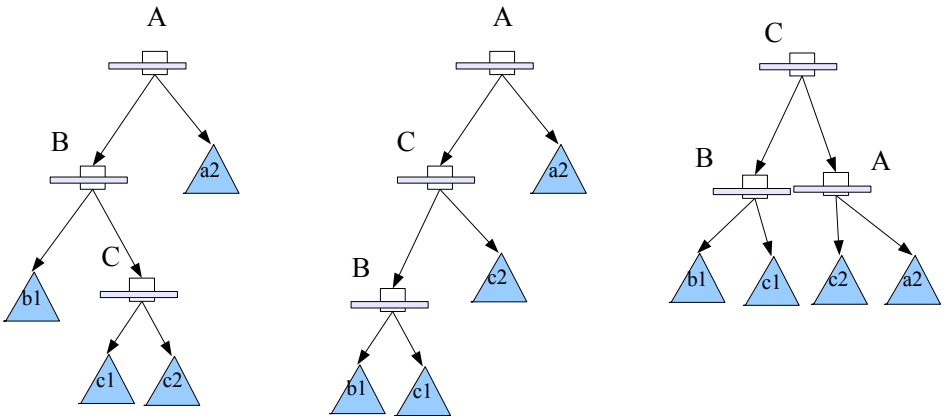
7. ábra: a T-fa struktúra elemei
(a) egy csomópont; (b) egy T-fa
(c) a tárolt elemek elhelyezkedése a T-fában

A memóriakapacitás-igény csökkentésére a T-fában csak mutatókat tároltunk a tényleges kulcsra. A keresés meggyorsítására azonban érdemes lehet a tényleges kulcs valamely részét szintén eltárolnunk. A gyorsítás ekkor nem közvetlenül a mutató-indirekciók eliminációjából adódik. Sokkal fontosabb, hogy a fa bejárása során a processzor-cache-ben a találatok arányát növeli, ha nem hivatkozunk az algoritmusok során gyakran egymástól távoli memóriacímekre. A részleges-kulcsú T-fák[22] segítségével éppen ez a viselkedés érhető el, ahol az egyes csomópontokban a hivatkozott rekord címe mellett a kulcsból eltárolva néhány bitet sok esetben elérhető, hogy a továbblépési döntést a tényleges kulcs-értékek összehasonlítása nélkül meghozzuk. A nyereség nyilvánvaló, hiszen ahogy a fizikai memória hozzáférési ideje nagyságrendben kisebb, mint a lemezeké, úgy a processzor-cacheben lévő adatok hozzáférési ideje újabb nagyságrendi lépés. A módszer részletei és figyelemreméltó teljesítőképessége a [BMR01] irodalomban olvashatók.

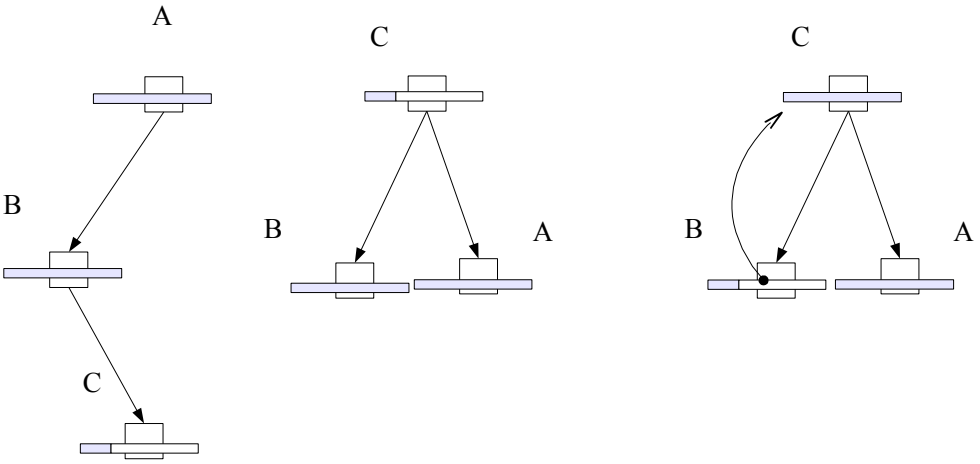
22 pkT-fa, partial key T-Tree



(a)



(b)



(c)

8. ábra: T-fa forgatások (az A jelű csúcsnál sérül az AVL tulajdonság)
(a) egyszerű bal-forgatás; (b) bal-jobb dupla forgatás
(c) bal-jobb dupla forgatás a kitöltöttségi követelmény helyreállításával

Relációk

A relációk tárolása nagymértékben egyszerűbb az indexeknél. Egy *leíró*ban feljegyezzük a reláció alaptulajdonságait, mezőit, azok típusát, és egyéb adminisztratív információkat. A tényleges adattárolás következik ezután. Fix hosszú rekordok esetén egyszerűen egy tömbben tárolhatjuk a rekordokat. Rekord törlésekor csak megjelöljük a megüresedő helyet, valamint nyilvántartásba vesszük a megüresedő helyet. Szükség esetén tömörítjük a tárat, ez azonban költséges művelet. Új elem beszúrásakor azt egy üres helyre írjuk, amit esetleg a tömbhöz további tárterületet foglalva tudunk csak megtenni, ami szintén meglehetősen drága művelet lehet. Ha nem ragaszkodunk a rekordok szekvenciális eléréséhez, sokkal kényelmesebb dolgunk van: ha létezik legalább egy index a relációhoz, akkor az egyes rekordok a memóriában akárhol elhelyezkedhetnek, így egyszerűen a memória-menedzserre bízhatjuk az összes adminisztratív feladatot, és megspórolhatjuk az említett drága műveleteket.

Adódik a kérdés, hogy mi a teendő a rekordok változó hosszúságú mezőivel? Ésszerű keretek között korlátos mezőméret esetén fix hosszú mezőként is kezelhetőek az adattárolás szintjén. Ha ez nem vezetne gazdaságos tároláshoz, a rekordokban csupán egy mutatót tárolunk, amely egy elkülönített halomterületre mutat, ahol azok tárolhatók. Ezen a halomterületen újra memóriamenedzselési kérdések vetődnek fel, de sokkal kisebb léptékben. Ezekkel itt nem foglalkozunk.

A változó hosszúságú rekord-mezők mellett érdemes megvizsgálni a távoli kulcsokat tartalmazó mezőket is! Ha a mező értéke helyett csupán egy mutatót tárolunk a hivatkozott reláció megfelelő rekordjára, akkor kétszeresen is nyerünk: egyfelől tárterület és változó hosszúságú mezők esetén adminisztrációs komplexitás terén spórolunk, másfelől ezen távoli kulcsok mentén előreszámított (fél)illesztések hozhatóak létre [LeC86], ami a lekérdezés-feldolgozás során térül meg. Az előreszámított félillesztések megvalósítása nem új találmány: DRDB rendszerekben is előszeretettel alkalmazzák. Azonban ezen a téren is egyszerűsítési lehetőség adódik a fizikai memória természetéből: míg a diszkes tároláskor erre a különböző relációkan összetartozó rekordok egy blokkra rendezése, az ún. *klaszterezés*e jelent megoldást, addig a memóriaadatbázisok területén a sokkal egyszerűbb, mutatós megoldás kézenfekvő.

Lekérdezés-feldolgozás

A lekérdezések feldolgozását két pontban tekintjük át. Előbb megvizsgáljuk a relációs algebra alapműveleteinek egy megvalósítását, majd az eredménylisták előállítását tekintjük az azokat kiszolgáló adatszerkezetekre is utalva. Nem elhanyagolható kérdés a lekérdezések optimalizációja, ám az meghaladja e dokumentum kereteit.

A relációs algebra alapműveletei

Vetítés ($\pi_{(A, B)}$ R)

A vetítés az eredményreláció leírójának módosításával kezdődik. Bizonyos esetekben nem érdemes az így keletkező duplumokat ténylegesen kiválogatni [LeC86]: elegendő a következő feldolgozási fázisban vagy az eredmények visszaadásakor menet közben átugorni. Különösen fontos észrevétel ez akkor, ha a részeredmény listát egy létező relációra alapozzuk, amelyhez létezik megfelelő összetett index, vagy a megmaradó mezők szerint rendezett bejárásra más módon van lehetőségünk. Ha a duplumok eltávolítása szükségessé válik, akkor a korábbiakban említett hash-alapú módszerek valamelyike jó választásnak tűnik: a megfelelő méretezés ebben az esetben nem jelent akkora problémát, mint az indexek esetében, hiszen lényegében statikus struktúráról van szó.

A részeredmény-listákat, ha különválasztottuk a létező relációktól – erre komplexebb lekérdezésekkor mindenképpen sor kerül előbb-utóbb –, akkor a relációk tárolásánál megismert tömb-alapú módszer a legpraktikusabb tárolnunk.

Szűrés ($\sigma_{(Gy='alma')}$ R)

A szűrés szintén olyan művelet, amely az eredményreláció leírójának módosításával megoldható, és csak az eredmény visszaadásakor vagy a további feldolgozáskor elegendő eldobni a mintára nem illeszkedő rekordokat. Duplumok kiszűrése szükség esetén hash-alapú módszerekkel szintén hatékonyan megoldható.

Illesztések

A távoli kulcsok mentén történő fél-illesztések egyszerűségét a relációkhoz kapcsolódó adatszerkezetek részben már tárgyaltuk. Az illesztések további típusaira a diszk-alapú adatbázis-rendszerekben használt hash-alapú módszerek használhatók, melyekről a [DeG85] irodalom részletes leírással szolgál.

Tranzakciókezelés, az ACID tulajdonság

A tranzakciókezelés összetartozó műveletsorozatok végrehajtására fogalmaz meg négy követelményt: az *atomicitás*t[23], azaz valamely műveletsor vagy teljes egészében, vagy egyáltalán nem hajtódik végre, a *konzisztenciá*t[24], amely a tárolt adatokra megfogalmazott kényszerek meg nem sértését követeli meg, a konkurens műveletek interferenciáját tiltó *elkülönítés*t[25], valamint a sikeresen végrehajtott módosítások eredményeinek *tartósság*át[26].

Az atomicitás zárak implementálásával illetőleg a módosítás előtt a rekord duplikálásával oldható meg [LeC86]. A zárak alkalmazása egyszerű művelet, és a költsége kicsi, hiszen a gyors

23 A - atomicity
24 C - consistency
25 I - isolation
26 D - durability

adat-elérési és feldolgozási képesség miatt a tranzakciók várható értékben rövid ideig tartanak. Így lehetőség nyílik zárakkal nagyobb adategységek [GaK92], akár teljes relációk kizárólagos hozzáférését biztosítani.

A konzisztencia követelményét szintén a rekordok módosítás előtti duplázásával érhetjük el: ekkor hiba esetén rendelkezésre áll a korábbi verzió, míg a tranzakció sikeres lefutása esetén gyorsan véglegesíthető a változás.

A tranzakciók hatásainak elkülönítését a műveletek megfelelő ütemezésével érhetjük el. A DRDB rendszerekhez kialakított sorosítható ütemezések, és zár-kezelési protokollok itt is alkalmazhatók, de soros ütemezés megvalósítására is adódik lehetőség a rövidebb tranzakciók miatt. A soros ütemezés előnye a tranzakciók közötti környezetváltások megspórolása: tágabb értelemben a processzorcache tartalma is beletartozhat a környezetbe, ami a cache-találatok esetén szignifikáns sebességnövekedést eredményez.

Az adatok tartósságát tranzakció-naplók és backup-ok készítésével biztosíthatjuk. Ezek tárolása általában valamilyen mágneses tárolási elvű adathordozón (diszken vagy mágnesszalagon) oldható meg. Azonban a tartósság követelményét kielégítendő minden sikeres tranzakció befejeződése előtt ki kellene írni a tranzakció-naplót a lemezre. Ezzel persze lelassítanánk az írást is végző tranzakciók végrehajtását. Egy megoldás lehet ún. *stable memory*[27]-val felszerelt architektúra alkalmazása, ahol néhány lemez-blokknyi tranzakció-napló elfér [GaK92], és azt egy külön processzor vagy folyamat periodikusan, csoportosítva írja ki a lemezre. Ez tehát aszinkron művelet a tranzakció befejeződéséhez képest, így biztosítható a rendszer magasabb áteresztő képessége.

Hardvertámogatás

A fizikai memória hibavalószínűségének csökkentésére, illetőleg a tápellátás folyamatos biztosítására különböző megoldásokat alkalmazhatunk, amelyeket a nem felejtő memória kérdését tárgyaló részben tekintettünk át.

Szintén a nem-felejtő memória került előtérbe a tranzakciók tartóssági követelményének kielégítésekor. Ekkor speciális architektúra (ún. *stable memory*) és külön napló-mentő processzor használata segíthet.

Tágabb értelemben hardvertámogatásnak tekinthető egy többprocesszoros architektúra alkalmazása is. Eddig nem tárgyalt témakör, az ütemezések fair-tulajdonsága. A soros ütemezés esetében sokkal könnyebb azt biztosítani, legalábbis statisztikai alapon [GaK92]: sokkal kisebb valószínűséggel van jelen egyszerre *N* db hosszú tranzakció, mint egyetlen.

27 Olyan fizikai-memória megoldás, amelynek a megbízhatósága vetekszik a lemezes tárolókéval. Részletesen a [CKK89] irodalom foglalkozik a témával.

Összefoglalás

A memóriaadatbázisok alkalmazásának láttuk előnyeit és hátrányait is. Bizonyos feladatok elvégzéséhez egyszerűbb algoritmusokat alkalmazhatunk, mint a diszk-alapú rendszerekben, míg másokhoz bonyolultabb, újszerű megoldásokra van szükség.

A konkrét feladat és követelményei ismeretében IMDB alkalmazása mellett vagy éppen ellen dönteni: ez a rendszert tervező mérnök feladata. Ma már rendelkezésre állnak hibrid rendszerek is, amelyek az adatbázis egy részét IMDB, másik részét DRDB elven kezelik, és teszik ezt a programozó számára teljesen transzparens módon. Ezen rendszerek bizonyos korlátok között képesek az egyes relációk hozzáférési statisztikái alapján migrálni az adatok megfelelő részét az IMDB és a DRDB alrendszer között. Az adatbázis tervezője azonban nagyobb rálátással rendelkezik az elkészítendő rendszerre, így – a rendszer indulásakor legalábbis – jobb konfigurációt definiálhat, mint az automatizmus.

Irodalomjegyzék

[GaK92]: Hector Garcia-Molina, Kenneth Salem: Main Memory Database Systems: An Overview, 1992 (: IEEE Transactions on knowledge and data engineering - Vol 4, No.6, December)

[Ste02]: Steve Graves: In-Memory Database Systems, 2002 (SSC Publications, Inc.: Linux Journal - Issue 101, September) http://www.linuxjournal.com/article/6133

[LeC86]: Tobin J Lehman, Michael J Carey: Query Processing in Main Memory Database Management Systems, 1986 (ACM Press - New York, NY, USA: ACM SIGMOD Record , Proceedings of the 1992 ACM SIGMOD international conference on Management of data - Volume 15 Issue 2, June) http://portal.acm.org/citation.cfm?id=16878

[BMR01]: Philip Bohannon, Peter McIlroy, Rajeev Rastogi: Main-Memory Index Structures with Fixed-Size Partial Keys, 2001 (ACM Press - New York, NY, USA: ACM SIGMOD Record , Proceedings of the 2001 ACM SIGMOD international conference on Management of data SIGMOD '01 - Volume 30 Issue 2, May) http://portal.acm.org/citation.cfm?id=375681

[LeC86b]: Tobin J Lehman, Michael J Carey: A Study of Index Structures for Main Memory Database Management Systems, 1986 (Proceedings of the Twelfth International Conference on Very Large Data Bases - August) http://www.vldb.org/conf/1986/P294.PDF

[AnP92]: Anastasia Analyti, Sakti Pramanik: Fast search in main memory databases, 1992 (ACM Press - New York, NY, USA: ACM SIGMOD Record , Proceedings of the 1992 ACM SIGMOD international conference on Management of data - Volume 21 Issue 2, June) http://portal.acm.org/citation.cfm?id=130317

[RIS98]: Rónyai Lajos, Ivanyos Gábor, Szabó Réka: Algoritmusok, Typotex - Budapest 1999 http://www.typotex.hu/book/m_0020.htm

[DeG85]: David J. DeWitt, Robert Gerber: Multiprocessor Hash-Based Join Algorithms, 1985 (Morgan Kaufman pubs. (Los Altos CA), Stockholm -) http://citeseer.ist.psu.edu/dewitt85multiprocessor.html

[CKK89]: G. Copeland, T. Keller, R. Krishnamurthy, M. Smith: The case for safe RAM, 1989 (Proceedings of the 15th international conference on Very large data bases -) http://portal.acm.org/citation.cfm?id=88887 ; http://www.vldb.org/conf/1989/P327.PDF