15. tétel
Relációs adatbázisok

Bulatovic Nikola

2019. június 22.

Kivonat

A relációs adatmodell, logikai és fizikai operátorok (seek, scan és a joinok változatai, aggregálás), adattárolási modellek (row store, column store), indexek (klaszterezett index, nem klaszterezett index), a B-fa, kulcsok és kényszerek, tranzakciók. Lekérdezésoptimalizálás. Az adatbetöltés menete.

Ez a tétel az *Adatmodellek és adatbázisok a tudományban* kurzus alapján készült [1]. Megjegyzés: az utolsó fejezetek témakörei (Tranzakció kezelése c. fejezettől) nem szerepeltek az előadáson, így ezek nem szerepelnek részleteiben a tételben.

Tartalomjegyzék

1. Bevezetés - adatbázis szerverek

Az adatbázis szerver általában olyan dedikált szerver, amely különféle adatbázis szolgáltatásokat nyújt más gépeknek vagy programoknak a kliens-szerver modell alapján leírt módokon. Az ilyen szervereken a háttértár, az I/O és a hálózat konfigurációja erősen optimalizált. Az adatbázis szerver fő feladatai:

- rendezett adattárolás a megmaradó (non-volatile) memóriában vagyis a háttértárolón
- lekérdezések (query) gyors végrehajtása
- adatmódosítás
- tranzakciók kezelése (hosszú, konkurens műveletek atomizált végrehajtása)
- adat konzisztencia fenntartása
- adatbiztonság (rendszerhiba esetén is megmaradó adatok)

Többféle adatbázis kezelő rendszer (database management system - DBMS) létezik, melyek ilyen adatbázis szerver funkcionalitással rendelkeznek. Az egyik legismertebb és legtöbbet használt a relációs adatbázis kezelő rendszerek (RDBMS). Ezeknek többféle implementációja létezik: Oracle, Microsoft SQL Server, vagy a nyílt forráskódú Postgres és MySQL. Ezek közös tulajdonsága, hogy mindegyik relációs adatmodellt alkalmaz, a lekérdezésekhez pedig egy adat-orientált deklaratív nyelvet az SQL-t (Structured Query Language) használ.

2. Relációs adatmodell

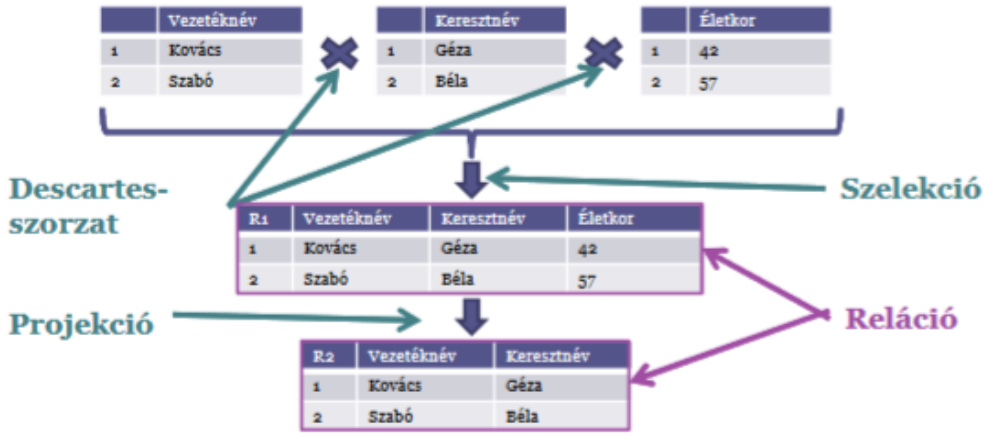
Az adatbázisok logikai alapeleme a tábla (table). A táblák oszlopait a séma jellemzi: az adattípus és méret előre definiált. Az adatot magát a tábla sorai jelentik, amelyek száma tetszőleges és azonos formátumúak. A táblákon belül különböző megkötések lehetnek:

- elsődleges kulcs (primary key), ami a táblán belül egyedi és egy vagy több oszlop kombinációja, mellyel a rendezés jól definiált (kulcsok összehasonlíthatóak $<$,$>$,$=$)
- másodlagos kulcs (foreign key), ami egy másik tábla elsődleges kulcsára mutat

Az adatbázisok működési logikáját a relációs adatmodell írja le, melynek matematikai alapja a relációs algebra. A relációs adatszerkezet alapfogalmai a következők:

- **halmaz**: egy táblázat oszlopa, vagy oszlopok rendezett listája (tuple)
- **műveletek**: Descartes-szorzat, szelekció, projekció, unió, különbség
- **séma**: megkötés, hogy milyen halmazok, milyen sorrendben szorozhatóak össze
- **reláció**: Descartes-szorzat részhalmaza.

A logikai működést - matematikai precizitás mellőzve- úgy lehet összefoglalni, hogy a lekérdezések a táblázatok Descartes-szorzatán végzett műveletek utáni részhalmaz képzést jelent (1. ábra).



1. ábra. **A relációs algebra sematikus ábrázolása.** A lekérdezés végeredménye (reláció) a halmazok Descartes-szorzatának egy részhalmaza.

3. Adatok fizikai tárolása az adatbázis szerveren

Az adatbázisok alapproblémai a következők:

- a háttértár mindig jóval lassabb, mint a központi memória és a CPU (megoldás: indexelés)
- a háttértár szekvenciálisan sokkal gyorsabban olvasható, mint random módon (megoldás: sorok lapokba (page) szervezése, lapok megfelelő sorrendben tárolása)
- a memória mérete mindig sokkal kisebb, mint az adatbázis mérete (megoldás: ügyes algoritmusok, hogy ne kelljen transzformálni a memória és a diszken levő formátum között).

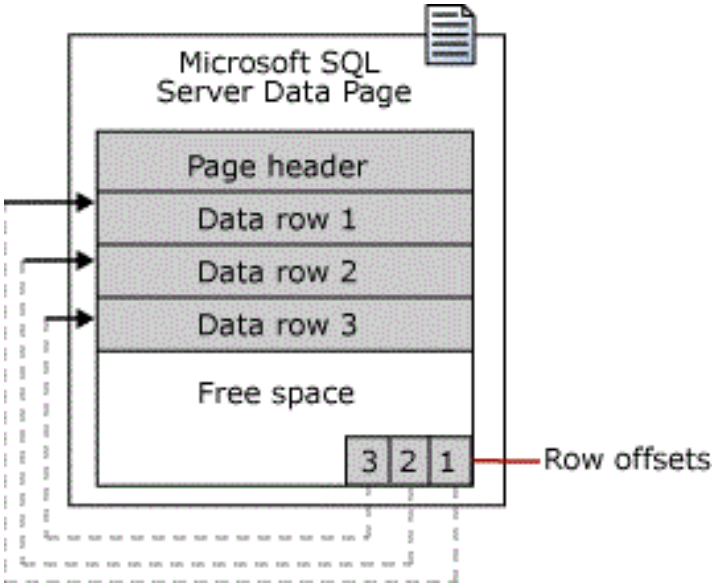
Az adatok fizikai szervezésének megértése elengedhetetlen ahhoz, hogy lássuk hogyan kell a háttértárat és az adatbázist optimálisan konfigurálni. Sokszoros sebességnövekedés érhető el a naiv adatbázis szervezéshez képest. A továbbiakban sor alapú adattárolás bemutatása következik.

3.1. Lapok

A fizikai egységek a következők: adatbázis > file group > file > extent > page, illetve a tranzakciós napló (lásd Tranzakciók fejezet). A tárolás alapegysége tehát a lap (2. ábra):

- mérete fix 8kB
- 8 lap alkot egy extent-et (64 kB)

- csak teljes extent írható/olvasható a lemezre/ről
- formátum a memóriában és a lemezen azonos
- több fajtája van (tábla adat, index stb.)

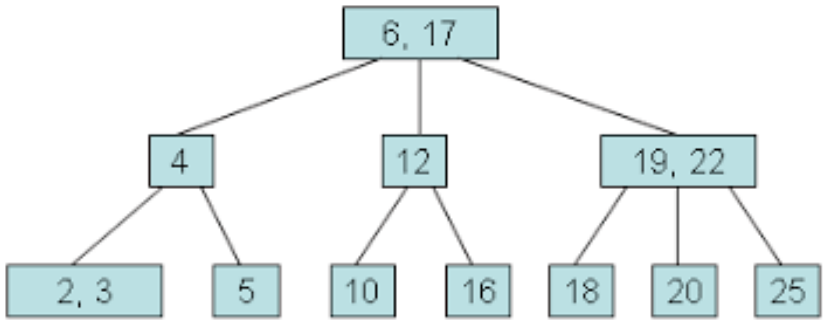


2. ábra. **A tárolás alapegysége, a lap (page).** Az adatbázisban tárolt fájlok által allokált tárhely logikailag ilyen lapokra vannak osztva. Minden lap rendelkezik egy headerrel, amely tartalmazza a lap számát, típusát, a maradék üres hely méretét. A táblázat sorai a lapokon tárolódnak sorfolytonosan. Minden lap rendelkezik egy row offset táblázattal, ami megadja hogy a lapon szereplő sorok hányadik bájtnál kezdődnek a lap elejéhez képest.

3.2. Adatstruktúrák

Ahogy korábban láttuk az adattárolás logikai egysége a tábla, azon belül pedig a sor. Ha egy tábla sorai nem meghatározott sorrendben tárolódnak (heap table), akkor egy új sor beírása könnyű (az utolsó lap végére kell írni), azonban kereséskor az egész táblát végig kell olvasni. Mint láttuk, az elsődleges kulcs használata egy lehetséges rendezést definiál a sorok között. Ha létezik egy ilyen rendezés, akkor táblát klaszterezett indexszel tárolhatjuk fizikailag.

Sorba rendezett adatok tárolására egy lehetséges adatstruktúra a B-fa, amely a hagyományos bináris-fa általánosítása. A bináris fa esetében egy új adatpont beírásakor gyakran az egész fát újra kell építeni, ami használhatatlanná teszi adatok diszken történő tárolására. A B-fa is csomópontokból áll, de az d számú adatsort és $d+1$ számú pointert tartalmaz (bináris esetben $d = 1$). A pointerek további csomópontokra mutatnak, amelyekben található érték a pointer csomópontjában lévő adatok értékei közé esik (3. ábra).



3. ábra. **A B-fa legegyszerűbb sematikus ábrázolása.**

A B-fa építésekor ha egy csomópont betelik, akkor kettéosztjuk. Az ilyen módon konstruált fa egy ún. *self balancing* kereső fa, ami azt jelenti hogy egy elem megtalálása (a fa mélysége) $\sim o(log_d n)$. A B-fa ezzel megoldja a bináris fa problémáit.

Ennek ellenére néhány művelet a B-fa esetében is költséges lehet, a keresett csomópont mélységének függvényében. Ezt a problémát oldja fel a B-fa egy optimalizálás, a B+-fa. Ahelyett, hogy minden csomópontot egyformán kezelne, két típusba sorolja őket: a közbenső szinteken csak a kulcsokat és a pointereket tárolja, a tényleges adatokat csak a legalsó szinten. Ezek alapján az adatbázis tábláit B+-fa struktúrában tárolhatjuk. A fa szintje egy lapnak felel meg. Mivel a B+-fa struktúrában kétféle csomópont van ehhez két fajta lapot igényel: index és sorokat tároló lap. A lapokon a pointerek a soron következő/előző lapokra mutat (4. ábra). Ez a következő előnyökkel jár:

- a tábla szekvenciálisan, a kulcs szerinti sorrendben (vagy visszafele) olvasható
  SELECT * FROM t WHERE id BETWEEN 12 AND 66

- egy adott kulcsú sor gyorsan megtalálható
  SELECT * FROM t WHERE id = 12

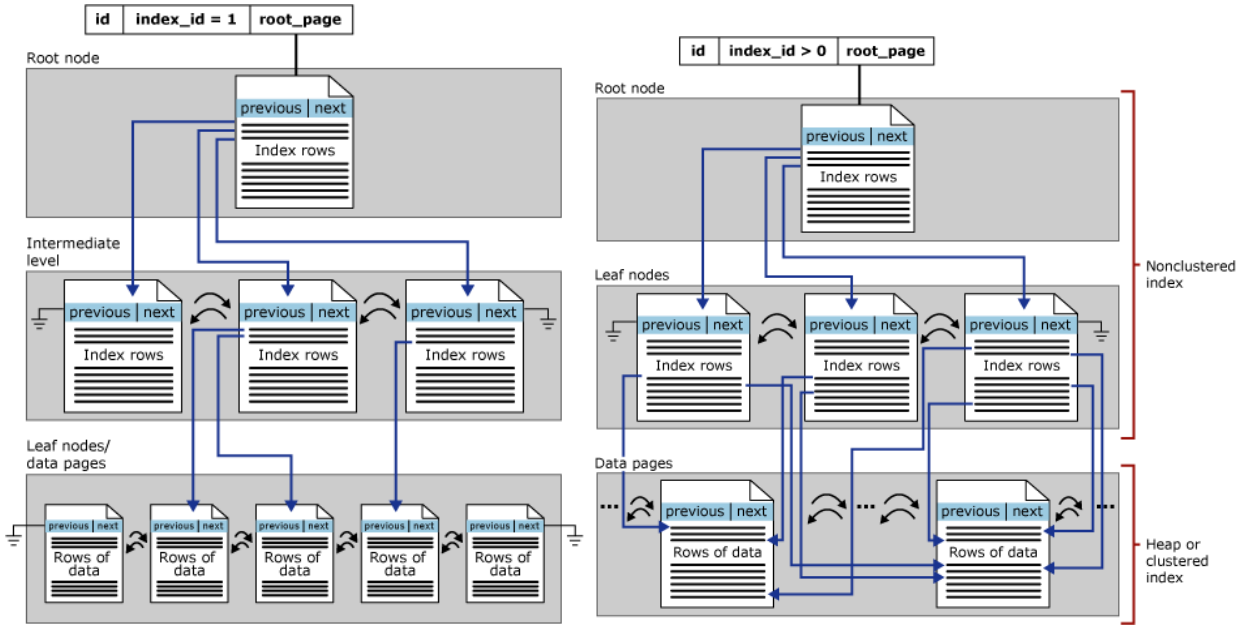
- egy adott kulcs tartomány gyorsan megtalálható, nem kell rendezni
  SELECT * FROM t ORDER BY id (ORDER BY elhagyható)

4. Indexek

4.1. Nem klaszterezett indexek

Ahogy az előzőekben láttuk egy táblához klaszterezett index építhető, amely jelentősen gyorsíthatja a lekérdezéseket. Egy táblán csak egy ilyen index lehet, amely meghatározza a tárolás sorrendjét. Az optimalizált lekérdezés csak az indexhez tartozó elsődleges kulcs alapján lehetséges, Ha más oszlop szerint is hasonló gyorsasággal szeretnénk keresni, akkor nem klaszterezett indexeket kell használunk.

A nem klaszterezett index a tábla soraitól eltérő struktúrával rendelkezik, mivel az indexek sorrendje nem tükrözi az adatok fizikailag tárolt sorrendjét. Ehelyett az elsődleges kulcstól eltérő oszlop vagy oszlopokra épül és az ezek szerinti sorrendet indexeli. Ezek tárolási struktúrája hasonlóan B+-tree, azonban a fa utolsó (leaf) csomópontjai nem a tényleges adatot tárolják, hanem csak egy arra mutató pointert (4. ábra).



4. ábra. **Klaszterezett (bal) és nem klaszterezett index (jobb).** A klaszterezett index esetében a fa levelei a tényleges adatot tárolják. A nem klaszterezett index esetében pedig egy pointer található arra az adat page-re, amelyen a tényleges adat található. Ha a tábla nem rendelkezik klaszterezett indexszel, akkor a pointer heap page-re mutat, ahol az adat van.

4.2. Beágyazott oszlopok

Ha olyan lekérdezést írunk, amihez csak az indexelt oszlopok kellenek, akkor elég csak az index struktúrát végigolvasni, az eredmény ezekből egyértelmű. Ha más oszlop is szerepel a lekérdezésben, akkor ezeket külön be kell tölteni a táblából.

Nem klaszterezett indexek létrehozásakor hozzáadhatunk beágyazott oszlopokat (included columns) az indexhez olyan formában, hogy az index pointerei mellé bekerülnek ezek az oszlopok is. Ekkor nem kell az adatsort külön betölteni kereséskor.

4.3. Indexek tulajdonságai

Indexek használatának előnyei:

- adott oszlopok szerinti keresés gyors, de csak ha a legelső indexelt oszlopra erős a megszorítás
- az indexelt adatok az előírt sorrendben, szekvenciálisan olvashatóak.

Hátrányai:

- több tárhelyet igényel(az indexelt oszlopok még egyszer tárolódnak
- beszúráskor, módosításkor, törléskor frissíteni kell

5. Logikai és fizikai operátorok

6. Tranzakciók kezelése

7. Lekérdezésoptimalizálás

8. Az adatbetöltés menete

9. Oszlop alapú adattárolás

130 old en

Hivatkozások

[1] Dobos László - Design and implementation of scientific databases http://www.vo.elte.hu/~dobos/teaching/scidb2016/scidb_en.pdf.