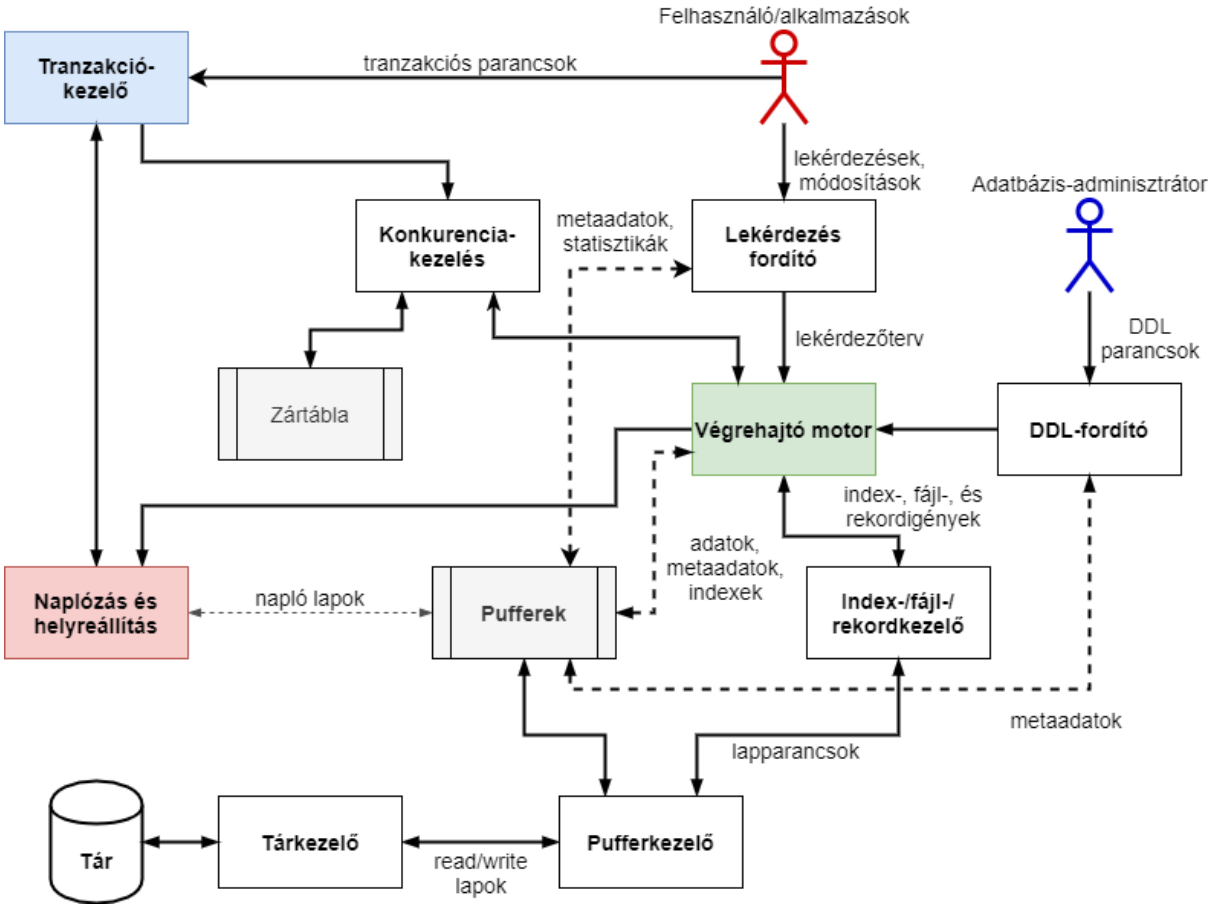
19. Adatbázisok optimalizálása és konkurencia kezelése

Az adatbázis-kezelő rendszerek feladata, részei

Adatbázis-kezelő rendszer alatt olyan számítógépprogramot értünk, mely megvalósítja nagy tömegű adat biztonságos tárolását, gyors lekérdezhetőségét és módosíthatóságát, tipikusan egyszerre több felhasználó számára.

Az adatbázis-kezelési tevékenységeket két csoportra szokás osztani:

- Adatdefiníciós műveletek.
  - A definíciós eszközökkel rendelkező nyelveket összefoglalóan *Data Definition Language (DDL)* nevezzük.
- Adatmanipulációs műveletek.
  - Az adatok manipulációjára szolgáló nyelveket összefoglalóan *Data Manipulation Language-nek (DML)* nevezzük.



1. ábra. Adatbázis-kezelő rendszer részei.

Az egyes alkotórészek rövid jellemzése

- **Lekérdezésfordító**: A lekérdezésfordító elemzi és optimalizálja a lekérdezést, ami alapján elkészíti a lekérdezés-végrehajtási tervet (lekérdezéstervet).
- **Végrehajtómotor**: A végrehajtómotor a lekérdezésfordítótól megkapja a lekérdezéstervet, majd kisebb adatdarabokra (tipikusan rekordokra, egy reláció soraira) vonatkozó kérések sorozatát adja át az erőforrás-kezelőnek.
- **Erőforrás-kezelő**: Az erőforrás-kezelő ismeri a relációkat tartalmazó adatfájlokat, a fájlok rekordjainak formátumát, méretét, valamint az indexfájlokat. Az adatkéréseket az erőforrás-kezelő lefordítja lapokra, amit átad a pufferkezelőnek.
- **Pufferkezelő**: Feladata, hogy a másodlagos adattárolóról (lemez, stb.) az adatok megfelelő részét olvassa be a központi memória puffereibe. A pufferkezelő információkat cserél a tárkezelővel, hogy megkapja az adatokat a lemezről.
- **Tárkezelő**: Adatokat ír-olvas a másodlagos adattárolóról. Előfordulhat, hogy igénybe veszi az operációs rendszer parancsait is, de sokszor közvetlenül a lemezkezelőhöz intézi a parancsait.
- **Tranzakciókezelő**: A lekérdezéseket és más tevékenységeket tranzakciókba szervezzük. A tranzakciók olyan egységek, amelyeket atomosan és elkülöníthetően kell végrehajtani, valamint a végrehajtásnak tartósnak kell lennie, illetve a tranzakció végrehajtása nem állíthat elő érvénytelen adatbázis-állapotot (azaz konzisztens). A tranzakciókezelő hajtatja végre a tranzakciókat és gondoskodik a naplózásról és helyreállításról, valamint a konkurenciakezelésről.

Fizikai fájlszervezés

Az adatbázisban lévő adatokat a memóriában és a háttértárolón tároljuk. A memória és a háttértároló közötti adatávitelért a pufferkezelő modul a felelős. A memória és a háttértároló között átvihető legkisebb egységet ***blokk***nak nevezzük.

Egy blokk tartalmaz:

- blokk fejlécet
- rekordokat
- szabad helyet (opcionálisan)

A blokk-fejlécben általában a blokk sorszámát, a rekordok méretét és azok számát vagy szabad hely kezdetét szokták tárolni.

Egy rekord mezőkből áll. A relációs adatmodellben például egy adattábla egy sorának egy rekord felel meg, és minden egyes attribútumnak egy mező felel meg. A blokkokhoz hasonlóan a rekordnak is lehet fejléce. Ebben lehet például a
- *rekord sorszáma*, a
- *töröltség jelző flag*, vagy a
- *rekord mérete*.

Jelölések:

$\boldsymbol{B}$	blokk mérete	$\boldsymbol{R}$	rekord mérete
$\boldsymbol{b}$	blokkok száma	$\boldsymbol{r}$	rekordok száma
$\boldsymbol{bf}$	*blokkolási faktor*	$bf = \lfloor \frac{B}{R} \rfloor$	

Adatfájlok felépítése

A legegyszerűbb adatfájl a rendezetlen fájl, más néven a kupac (heap). Ilyenkor a rekordokat egyszerűen egymás után beírjuk a blokkokba.

Tárigény	$b = \lceil \frac{r}{bf} \rceil$
Keresés	átlagosan $\frac{b}{2}$, legrosszabb esetben b
Beszúrás *(a fájl végére)*	1 olvasás + 1 írás
Törlés	keresés + 1 írás (*törölt flag beállítás*)
Módosítás	keresés + 1 írás

Ha egy mező szerint rendezett fájlt használunk, akkor arra a mezőre vonatkozó keresés meggyorsítható, de cserében a frissítések bonyolultabbak lesznek.

Tárigény	$b = \lceil \frac{r}{bf} \rceil$
Keresés	$\log_2 b$, *logaritmikus keresés*
Beszúrás *(a fájl végére)*	léptetni kell, átalagosan $\frac{b}{2}$ írás
Törlés	keresés + 1 írás (*törölt flag beállítás*)
Módosítás	keresés + 1 írás

Az előző nem jó megoldás. A beszúráson javíthatunk úgy, hogy az új rekordokat a fájl végére, egy rendezetlen területre írjuk be. Ekkor a beszúrás műveletigénye csökken. Azonban a fájl végén lévő rendezetlen blokkokat időnként (üresjárati időben, kötegelt feldolgozásban) bele kell illeszteni a rendezett fájlba.

Keresés *($A = a$ alakú)*	$\log_2 b + k$, ahol k a rendezetlen blokkok száma
Beszúrás	1 olvasás + 1 írás
Törlés	keresés + 1 írás (*törölt flag beállítás*)
Módosítás	keresés + 1 írás
Karbantartás	$b \log_2 b$ (*adatbázis újrarendezés*)

Másik lehetőség a beszúrás javítására, ha üres helyeket hagyunk a blokkokban. Például lehet minden blokk kezdetben csak félig telítve. Ilyenkor szükség van az adatbázis karbantartására. A túltelített blokkokat szétvágni, nehogy teljesen beteljenek.

Tárigény	$2b$
Keresés *($A = a$ alakú)*	$\log_2 b + 1$, *logaritmikus keresés*
Beszúrás	keresés + 1 írás
Törlés	keresés + 1 írás (*törölt flag beállítás*)
Módosítás	keresés + 1 írás
Karbantartás	költséges

Indexstruktúrák

Ha tudjuk, hogy egy mező szerint sokat fogunk keresni, akkor az adatbázist a szerint a mező szerint indexeljük. Ha *az adatfájl rendezett*, ***elsődleges index****ről beszélünk.* Ha az adatfájl *nem rendezett*, vagy egy *másik mezőre készítünk indexet*, akkor azt ***másodlagos index***nek nevezzük. Egy index lehet ritka és sűrű index. A ritka indexben csak az egyes blokkok elején lévő kulcsot tüntetjük fel, a sűrű indexben minden értéket. Ritka indexet csak rendezett adatfájlra lehet használni.

Elsődleges index

Az elsődleges index olyan mezőre vonatkozik, amelyik szerint az adatfájl rendezett, így csak egy elsődleges index adható meg. Elsődleges indexnek használhatunk ritka indexet. A ritka indexben csak az adatfájl blokkjainak elején lévő kulcs-értékeket tüntetjük fel. Az indexben csak a kulcs-blokkszám párokat kell tárolni, ezért az index rekordmérete is kicsi. Az index rekordjainak száma megegyezik az adatfájl blokkjainak számával.

Tárigény	$b_I = \lceil \frac{b}{bf_I} \rceil \ll b$
Keresés *($A = a$ alakú)*	$\log_2 b_I + 1$ (beolvasás) $\ll \log_2 b$, *logaritmikus keresés*
Frissítések	bonyolult, ld. rendezett adatfájl

Másodlagos index

A másodlagos index olyan mezőre vonatkozik, amelyik szerint nincs rendezve az adatfájl. Lehet rendezetlen az adatfájl, vagy lehet, hogy másik mező szerint rendeztük. A másodlagos index egy sűrű index, az adatfájl egyes rekordjához egy index-rekord tartozik. Az index egy rekordjában kulcs-rekord mutató párok vannak.

Tárigény	$b_I = \lceil \frac{r}{bf_I} \rceil$
Keresés *($A = a$ alakú)*	$\log_2 b_I$, *logaritmikus keresés*
Frissítések	bonyolult, ld. rendezett adatfájl

Az indexek frissítésekor ugyanazok a problémák merülhetnek fel, mint a rendezett adatfájloknál. A beszúrás gyorsítására itt is használható részben kitöltött blokk, egy így tovább.

Bitmap index

A bitmap indexeket az oszlopok adott értékeihez szokták hozzárendelni, az alábbi módon:

- Ha az oszlopban az i. sor értéke megegyezik az adott értékkel, akkor a bitmap index i. tagja egy 1-es.
- Ha az oszlopban az i. sor értéke viszont nem egyezik meg az adott értékkel, akkor a bitmap index i. tagja egy 0.

Így egy lekérdezésnél csak megfelelően össze kell AND-elni, illetve OR-olni a bitmap indexeket, és az így kapott számsorozatban megkeresni, hol van 1-es. A bináris értékeket szokás szakaszhossz kódolással tömöríteni a hatékonyabb tárolás érdekében.

Többszintű indexek, keresőfák, B-fák

Az index olyan, mint egy rendezett adatfájl, ezért szintén indexelhető elsődleges indexel. Ha az indexet is indexeljük, akkor ***többszintű indexelés***ről beszélünk. Mivel az index már rendezett, azt ritka indexel lehet indexelni. Ráadásul az index rekordmérete is kisebb, mint az adattábláé, ezért nagyobb a blokkolási faktora. Ilyenkor az indexek tárigénye növekszik, de a keresés sokkal gyorsabb lesz.

A t. szintű index: az indexszinteket is indexeljük, összesen t szintig.
A t. szinten ($I(t)$) bináris kereséssel keressük meg a fedő indexrekordot.

Követjük a mutatót, minden szinten, és végül a főfájlban: $\log_2\Big(B\big(I(t)\big)\Big) + t$ blokkolvasás. Ha a legfelső szint 1 blokkból áll, akkor $t+1$ blokkolvasást jelent. Minden szint blokkolási faktora megegyezik, mert egyforma hosszúak az indexrekordok.

A t. szinten 1 blokk: $1 = \frac{B}{bf(I)^t}$. Azaz $t = \log_{bf(I)}(B) < \log_2(B)$, tehát jobb a rendezett fájlszervezésnél.

A $log_{bf(I)}(B) < \log_2\big(B(I)\big)$ is teljesül általában, így az egyszintű indexeknél is gyorsabb.

Tárigény	$b_1 + b_2 + \ldots + b_t$
Keresés *($A = a$ alakú)*	$t + \log_2 b_t$, ahol t a szintek száma és a legfelső szinten b_t blokk van

Logikailag az index egy rendezett lista. Fizikailag a rendezett sorrendet táblába rendezett mutatók biztosítják.

A fa struktúrájú indexek B-fákkal ábrázolhatók. A B-fák megoldják a bináris fák kiegyenlítetlenségi problémáját, mivel "alulról" töltjük fel őket. A B-fa egy csomópontjához több kulcsérték tartozhat. A mutatók más csomópontokra mutatnak, és így az összes kulcsértékre az adott csomóponton. Mivel a B-fák kiegyenlítettek (minden ág egyenlő hosszú, vagyis ugyanazon a szinten fejeződik be), kiküszöbölik a változó elérési időket, amik a bináris fákban megfigyelhetők.

Bár a kulcsértékek és a hozzájuk kapcsolódó címek még mindig a fa minden szintjén megtalálhatók, és ennek eredménye: egyenlőtlen elérési utak, és egyenlőtlen elérési idő, valamint komplex fakeresési algoritmus az adatfájl logikailag soros olvasására.
Ez kiküszöbölhető, ha nem engedjük meg az adatfájl címek tárolását levélszint felett. Ebből következően: minden elérés ugyanolyan hosszú utat vesz igénybe, aminek egyenlő elérési idő az eredménye, és egy logikailag soros olvasása az adatfájlnak a levélszint elérésével megoldható. Nincs szükség komplex fakeresési algoritmusra.

A B-fa egy csomópontjához több kulcsérték tartozhat. A mutatók más csomópontokra mutatnak, és így az összes kulcsértékre az adott csomóponton. Ilyenkor a keresés műveletigénye tovább csökken, bár most is logaritmikus, de a logaritmus alapszáma bf_I ritka indexre vonatkozó blokkolási faktor. A módosíthatóság javítására pedig a blokkokban üres helyeket hagyunk. Ezt minden szinten elvégezve a beszúrás, törlés műveletigénye is logaritmikus lesz.

Szintek száma	$t = \log_{bf_I} b$
Tárigény	nagyságrendben lineáris
Keresés *($A = a$ alakú)*	$t + \log_{bf_I} b$
Frissítési műveletek	$t + \log_{bf_I} b$

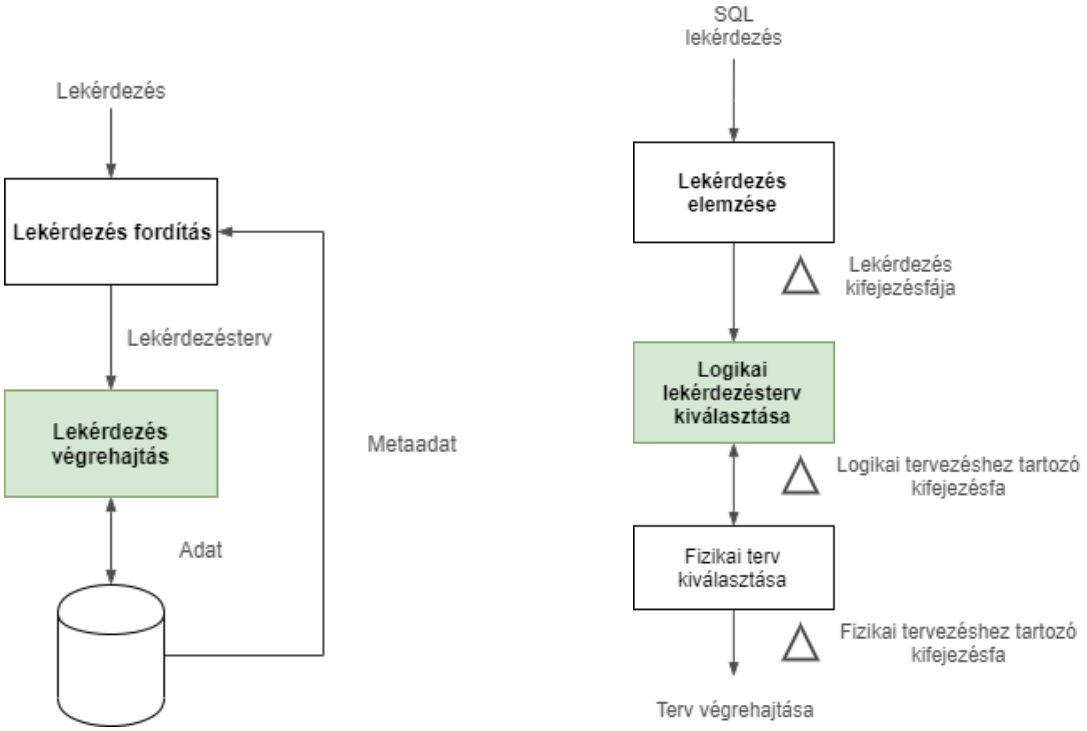
Hasító index

- A hasító függvényekkel az A=a alakú kereséseket tudjuk felgyorsítani. A rekordok edényekbe (bucket) particionálva helyezkednek el, melyekre a h hasítómezőn értelmezett függvény osztja szét őket.
- Hatékony egyenlőségi keresés, beszúrás, és törlés jellemzi, viszont nem támogatja az intervallumos keresést ($a < A < b$).
- A rekordokat blokkláncokba soroljuk, és a blokklánc utolsó blokkjának első üres helyére tesszük a rekordokat a beérkezés sorrendjében.
- A blokkláncok száma lehet
  - előre adott (*statikus hasítás*, ekkor a számot K-val jelöljük), vagy
  - a tárolt adatok alapján változhat (*dinamikus hasítás*).
- A besorolás az indexmező értékei alapján történik. Egy $h(x) \in \{1, .., K\}$ hasító függvény értéke mondja meg, hogy melyik kosárba tartozik a rekord, ha x volt az indexmező értéke a rekordban.
- A hasító függvény általában maradékos osztáson alapul (például $mod(K)$).
- Akkor jó egy hasító függvény, ha nagyjából egyforma hosszú blokkláncok keletkeznek, azaz egyenletesen sorolja be a rekordokat. Ekkor a blokklánc $\frac{B}{K}$ blokkból áll.

- A keresés költsége (A=a alakú):
  - Ha az indexmező és keresési mező eltér, akkor kupac szervezést jelent.
  - Ha az indexmező és keresési mező megegyezik, akkor csak elég a $h(a)$ sorszámú kosarat végignézni, amely $\frac{B}{K}$ blokkból álló kupacnak felel meg, azaz $\frac{B}{K}$ legrosszabb esetben. A keresés így K-szorosára gyorsul.
- Miért nem érdemes nagy K-t választani? A tárméret B, ha minden blokk nagyjából tele van.
  - Nagy K esetén sok olyan blokklánc lehet, amely egy blokkból fog állni, és a blokkban is csak 1 rekord lesz. Ekkor a keresési idő: 1 blokkbeolvasás, de B helyett T számú blokkban tároljuk az adatokat.
  - Módosításnál $\frac{B}{K}$ blokkból álló kupac szervezésű kosarat kell módosítani.

Lekérdezések végrehajtása, optimalizálása

A lekérdezésfeldolgozó egy relációs adatbázis-kezelő komponenseinek azon csoportja, amelyik a felhasználó lekérdezéseit, valamint adatmódosító utasításait lefordítja adatbázis-műveletekre, amiket végre is hajt.



2. ábra. A lekérdezésfeldolgozás és lekérdezésfordítás lépései.

A lekérdezés végrehajtása tulajdonképpen az adatbázist manipuláló algoritmusok összessége, azonban a végrehajtás előtt szükség van a lekérdezések fordítására.

A lekérdezés-fordítás lépései

1. Egy elemző fát építünk fel, amely a lekérdezést és annak szerkezetét jellemzi
2. Az elemző fából egy kezdeti lekérdezés-tervet készítünk, amelyet átalakítunk egy ezzel ekvivalens tervvé, aminek végrehajtási ideje várhatóan kisebb lesz. Elkészül a logikai lekérdezésterv, amely relációs algebrai kifejezéseket tartalmaz.
3. A logikai tervet átalakítjuk fizikai tervvé úgy, hogy a logikai terv operátoraihoz kiválasztunk egy algoritmust, valamint meghatározzuk az operátorok végrehajtási sorrendjét. A fizikai terv olyan részleteket is tartalmaz, hogy pl. kell-e rendezni, vagy hogyan férünk hozzá az adatokhoz.

Algebrai optimalizálás

A relációs algebrai kifejezéseket minél gyorsabban akarjuk kiszámolni. A kiszámítás költsége arányos a relációs algebrai kifejezés részkifejezéseinek megfelelő relációk tárolási méreteinek összegével. A módszer az, hogy műveleti tulajdonságokon alapuló ekvivalens átalakításokat alkalmazunk, azért, hogy várhatóan kisebb méretű relációk keletkezzenek.

Az eljárás heurisztikus, tehát nem az argumentum relációk valódi méretével számol. Az eredmény nem egyértelmű, ugyanis az átalakítások sorrendje nem determinisztikus, így más sorrendben végrehajtva az átalakításokat más végeredményt kaphatunk, de mindegyik általában jobb költségű, mint amiből kiindultunk.

Az optimalizáló algoritmus a következő heurisztikus elveken alapul:

- Minél hamarabb szelektáljunk, hogy a részkifejezések várhatóan kisebb relációk legyenek.
- A szorzás utáni kiválasztásokból próbáljunk természetes összekapcsolásokat képezni, mert az összekapcsolás hatékonyabban kiszámolható, mint a szorzatból történő kiválasztás.
- Vonjuk össze az egymás utáni unáris műveleteket (kiválasztásokat és vetítéseket), és ezekből lehetőleg egy kiválasztást, vagy vetítést, vagy kiválasztás utáni vetítést képezzünk. Így csökken a műveletek száma, és általában a kiválasztás kisebb relációt eredményez, mint a vetítés.
- Keressünk közös részkifejezéseket, amiket így elég csak egyszer kiszámolni a kifejezés kiértékelése során.

Relációs algebrai műveletek megvalósítása

Jelölések: σ (*szigma*), θ (*théta*), π (*pi*)

Kiválasztás

A kiválasztás (σ) lehetséges megvalósításai:

- Lineáris keresés: Olvassunk be minden lapot és keressük az egyezéseket (egyenlőség vizsgálat esetén). Az átlagos költség a lapok száma, ha a mező nem kulcs, illetve a lapok számának fele, ha a mező kulcs.
- Bináris (logaritmikus) keresés: Csak rendezett mező esetén használható.
- Elsődleges index használata.
- Másodlagos index használata.

Összetett kiválasztás Előfordulhat, hogy több feltétel van, amelyek és/vagy kapcsolatban vannak egymással. Ennek lehetséges megvalósításai a következők.

- Konjunkciós kiválasztás esetén ($\sigma_{\theta_1 \wedge \ldots \wedge \theta_n}$):
  - Válasszuk ki a legkisebb költségű σ_{θ_i}-t, és azt végezzük el (lásd fent), majd az eredményt szűrjük a többi feltételre.
    Költség: Az egyszerű kiválasztás költsége lesz a kiválasztott σ_{θ_i}-re.
  - Ha mindegyik θ_i mezőjére van indexünk, akkor keressük az indexekben és adjuk vissza a megfelelő sorok rowid-jeit. Végül vegyük ezek ***metszetét***.
    Költség: az indexekben való keresés összköltsége + a rekordok beolvasása.
- Diszjunkciós kiválasztás esetén ($\sigma_{\theta_1 \vee \ldots \vee \theta_n}$):
  - Lineáris keresés.
  - Ha mindegyik θ_i mezőjére van indexünk, akkor keressük az indexekben és adjuk vissza a megfelelő sorok rowid-jeit. Végül vegyük ezek ***unióját***.
    Költség: az indexekben való keresés összköltsége + a rekordok beolvasása.

Vetítés és halmazműveletek

Vetítésnél és halmazműveleteknél a duplikátumokat ki kell szűrni.

Vetítés (π) megvalósítása:

1. Kezdeti átnézés: eldobjuk a felesleges mezőket.
2. Duplikátumok törlése: ehhez az eredményt rendezzük az összes mező szerint, így a duplikáltak szomszédosak lesznek. Ezeket kell eldobni.

A költség a kezdeti átnézés, a rendezés és a duplikátumok törlésének összköltsége lesz.

Összekapcsolások

Beágyazott ciklusú összekapcsolás (nested loop join) ($R \bowtie_Z S$)

Bármekkora méretű relációra használható, nem szükséges, hogy az egyik reláció elférjen a memóriában. Két fajtája van, a *sor* és a *blokk* alapú.

- **Sor alapú beágyazott ciklusú összekapcsolás**
  - Legyen R és S a két összekapcsolandó reláció
    - S belső reláció (kisebb méretű)
    - R külső reláció
  - Az algoritmus a következő:

R minden t_R rekordján
    S minden t_S rekordján
        ha (t_R egyezik t_S $Z - n$) $t_R.t_S$ kiírása
    vége
vége

  - Költség:
    - Jó esetben S belefér a memóriába, ekkor csak egyszer kell beolvasni S-t, majd mindvégig a memóriában tarthatjuk.
      Ebben az esetben mindkét reláció lapjait egyszer kell beolvasni: $B_R + B_S$
    - Legrosszabb esetben mindkét relációból csak egy-egy lap fér bele a memóriába.
      Ekkor R *minden egyes soránál* végig kell olvasni S-t: $N_R * B_S + B_R$.

- **Blokk alapú beágyazott ciklusú összekapcsolás**
  - Az algoritmus a következő:

R minden X_R lapján
    S minden X_S lapján
        R minden t_R rekordján
            S minden t_S rekordján
                ha (t_R egyezik t_S $Z - n$) $t_R.t_S$ kiírása
            vége
        vége
    vége
vége

  - Költség:
    - Jó esetben S belefér a memóriába, ekkor csak egyszer kell beolvasni S-t, majd mindvégig a memóriában tarthatjuk.
      Ebben az esetben mindkét reláció lapjait egyszer kell beolvasni: $B_R + B_S$
    - Legrosszabb esetben mindkét relációból csak egy-egy lap fér bele a memóriába.
      Ekkor R *minden egyes lapjánál* végig kell olvasni S-t: $B_R * B_S + B_R$.

Összefésüléses rendező összekapcsolás (merge join)

- A relációk az összekapcsolási mezők szerint rendezettek.
- Egyesítjük a rendezett relációkat:
  - mutatók az első rekordra mindkét relációban
  - beolvasunk S-ből egy rekordcsoportot, ahol az összekapcsolási attribútum értéke megegyezik
  - beolvasunk rekordokat R-ből és feldolgozzuk.
- A rendezett relációkat csak egyszer kell végigolvasni
  - Összekapcsolás költsége:
    rendezés költsége + $B_s + B_R$ (a két reláció lapjainak száma).

Hasításos összekapcsolás (hash join)

- Az összekapcsolási attribútumot használjunk hasítókulcsként, és felosztjuk a rekordokat a memóriába elférő részekre
  - R rekordjainak felosztása $R_0 \dots R_n - 1$
  - S rekordjainak felosztása $S_0 \dots S_n - 1$
- Ez után az egymáshoz rendelt kosárpárokat összekapcsoljuk blokk alapú beágyazott ciklusú összekapcsolással, hasítófüggvény alapú indexet használva.
  - Összekapcsolás költsége: $2 * (B_R + B_S) + (B_R + B_S)$

Több tábla összekapcsolása

Az összekapcsolások kommutatívak és asszociatívak, ezért az eredmény szempontjából mindegy, hogy milyen sorrendben kapcsoljuk össze őket. A sorrend viszont befolyásolhatja a hatékonyságot, ugyanis rossz választás esetén a köztes eredmények nagy méretűek lesznek.

A legjobb összekapcsolási fa megtalálása n reláció egy halmazához:

- Hogy megtaláljuk a legjobb összekapcsolási fát n reláció egy S halmazához, vegyük az összes lehetséges tervet mely így néz ki: $S_1 \bowtie (S - S_1)$, ahol S_1 az S tetszőleges nem üres részhalmaza.
- Rekurzívan számítsuk ki S részhalmazainak összekapcsolásának költségeit, hogy meghatározzuk minden egyes terv költségét. Válasszuk a legolcsóbbat.
- Mikor bármely részhalmaz terve kiszámításra került, az újbóli kiszámítás helyett tároljuk el és hasznosítsuk újra amikor ismét szükség lesz rá.

Tranzakciókezelés

Konzisztens adatbázis: Az adatbázisokra különböző megszorítások adhatók meg. Az adatbázis konzisztens állapotban van, ha kielégíti az összes ilyen megszorítást. Konzisztens adatbázis egy olyan adatbázis, amely konzisztens állapotban van.

A konzisztencia sérülhet a következő esetekben:

- Tranzakcióhiba: hibásan megírt, rosszul ütemezett, félbehagyott tranzakciók.
- Adatbázis–kezelési hiba: az adatbázis-kezelő valamelyik komponense nem, vagy rosszul hajtja végre a feladatát.
- Hardverhiba: elvész egy adat, vagy megváltozik az értéke.
- Adatmegosztásból származó hiba.

Tranzakció: Konzisztenciát tartó adatbázis-műveletek sorozata.
Ezek után mindig feltesszük, hogy ha a T tranzakció indulásakor az adatbázis konzisztens állapotban van, akkor ha T egyedül fut le, az adatbázis konzisztens állapotban lesz a futás végén (közben kialakulhat inkonzisztens állapot).

Helyesség feltétele:

- Ha leáll egy vagy több tranzakció (abort, vagy hiba miatt), akkor is konzisztens adatbázist kapunk.
- Minden egyes tranzakció induláskor konzisztens adatbázist lát.

A tranzakcióktól a következő tulajdonságokat szoktuk elvárni (ACID):

- Atomosság (**A**, azaz *Atomicity*): a tranzakció "mindent vagy semmit" jellegű végrehajtása (vagy teljesen végrehajtjuk, vagy egyáltalán nem hajtjuk végre).
- Konzisztencia (**C**, azaz *Consistency*): az a feltétel, hogy a tranzakció megőrizze az adatbázis konzisztenciáját, azaz a tranzakció végrehajtása után is teljesüljenek az adatbázisban előírt konzisztenciamegszorítások.
- Elkülönítés (**I**, azaz *Isolation*): az a tény, hogy minden tranzakciónak látszólag úgy kell lefutnia, mintha ez alatt az idő alatt semmilyen másik tranzakciót sem hajtanánk végre.
- Tartósság (**D**, azaz *Durability*): az a feltétel, hogy ha egyszer egy tranzakció befejeződött, akkor már soha többé nem veszhet el a tranzakciónak az adatbázison kifejtett hatása.

A konzisztenciát mindig adottnak tekintjük. A másik három tulajdonságot viszont az adatbázis-kezelő rendszernek kell biztosítania, de ettől időnként eltekintünk. Feltesszük, hogy az adatbázis adategységekből, elemekből áll. Az adatbáziselem a fizikai adatbázisban tárolt adatok egyfajta funkcionális egysége, amelynek értékét tranzakciókkal lehet elérni (kiolvasni) vagy módosítani (kiírni). Adatbáziselem pl. a reláció, relációsor, lap.

A tranzakciók alaptevékenységei

A tranzakció és az adatbázis kölcsönhatásának három fontos helyszíne van:

- Az adatbázis elemeit tartalmazó lemezblokkok területe.
- A pufferkezelő által használt virtuális vagy valós memóriaterület.
- A tranzakció memóriaterülete.

Ahhoz, hogy a tranzakció egy adatbáziselemet beolvashasson, azt előbb memóriapuffer(ek)be kell hozni, ha még nincs ott. Ezt követően tudja a puffer(ek) tartalmát a tranzakció saját memóriaterületére beolvasni.

Az adatbáziselem új értékének kiírás fordítva történik: előbb a tranzakció kialakítja az új értéket saját memóriaterületén, majd ez az érték másolódik át a megfelelő puffer(ek)be.

A pufferek tartalmának lemezre írásáról a pufferkezelő dönt. Vagy azonnal lemezre írja a változásokat, vagy nem.

A naplózási algoritmusok és más tranzakciókezelő algoritmusok tanulmányozása során különböző jelölésekre lesz szükség, melyekkel a különböző területek közötti adatmozgásokat írhatjuk le.

A következő alapműveletek használjuk:

- $INPUT(X)$:
  - Az X adatbáziselemet tartalmazó lemezblokk másolása a pufferbe.
- $READ(X, t)$:
  - Az X adatbáziselem bemásolása a tranzakció t lokális változójába.
  - Ha az X-et tartalmazó blokk még nincs a memóriában, akkor $INPUT(X)$-et is beleértjük.
- $WRITE(X, t)$:
  - A t lokális változó tartalmaz az X adatbáziselem memóriapufferbeli tartalmába másolódik.
  - Ha az X-et tartalmazó blokk még nincs a pufferben, akkor előbb $INPUT(X)$ is végrehajtódik.
- $OUTPUT(X)$:
  - Az X adatbáziselemet tartalmazó blokk kiírása lemezre.

A továbbiakban feltételezzük, hogy egy adatbáziselem nem nagyobb egy blokknál.

Naplózás és helyreállítás

Az adatokat meg kell védeni a rendszerhibáktól, ezért szükség van az adatok helyreállíthatóságára. Erre az elsődleges technika a naplózás, amely valamilyen biztonságos módszerrel rögzíti az adatbázisban végrehajtott módosítások történetét.

A napló (log) nem más, mint naplóbejegyzések (log records) sorozata, melyek arról tartalmaznak információt, hogy mit tett egy tranzakció. Rendszerhiba esetén a napló segítségével rekonstruálható, hogy mit tett a tranzakció a hiba fellépéséig.

Naplóbejegyzések

Úgy kell tekintenünk, hogy *a napló*, mint fájl *kizárólag bővítésre van megnyitva.*
Tranzakció végrehajtásakor a naplókezelőé a feladat, hogy minden fontos eseményt rögzítsen a naplóban.

Az összes naplózási módszer által használt naplóbejegyzések:

- $\langle START\ T \rangle$: Ez a bejegyzés jelzi a T tranzakció végrehajtásának kezdetét.
- $\langle COMMIT\ T \rangle$: A T tranzakció rendben befejeződött, már nem akar további módosításokat végrehajtani.
- $\langle ABORT\ T \rangle$: A T tranzakció abortált, nem tudott sikeresen befejeződni. Az általa tett változtatásokat nem kell a lemezre másolni, vagy ha a lemezre másolódtak, akkor vissza kell állítani.

Semmisségi (undo) naplózás

A semmisségi naplózás lényege, hogy ha nem biztos, hogy egy tranzakció műveletei rendben befejeződtek és minden változtatás lemezre íródott, akkor a tranzakció hatását vissza kell vonni, azaz az adatbázist olyan állapotba kell visszaállítani, mintha a tranzakció el se kezdődött volna.

A semmisségi naplózásnál szükség van még egy fajta naplóbejegyzésre, a módosítási bejegyzésre, amely egy $\langle T, X, v \rangle$ hármas:

- T tranzakció az
- X adatbáziselemet módosította, és
- X-nek a módosítás előtti értéke v volt.

A semmisségi naplózás szabályai:

- Ha a T tranzakció módosítja az X adatbáziselemet, akkor a $\langle T, X, v \rangle$ naplóbejegyzést az előtt kell a lemezre írni, hogy az új értéket a lemezre írná a rendszer.
- Ha a tranzakció hibamentesen befejeződött, akkor a $COMMIT$ bejegyzést csak azután szabad lemezre írni, hogy a tranzakció által végrehajtott összes módosítás lemezre íródott.

Helyreállítás a semmisségi naplózás használatával

Tegyük fel, hogy rendszerhiba történt. Ekkor előfordulhat, hogy egy tranzakció nem atomosan hajtódott végre, azaz bizonyos módosításai már lemezre íródtak, de mások még nem. Ekkor az adatbázis inkonzisztens állapotba kerülhet. Ezért rendszerhiba esetén gondoskodni kell az adatbázis konzisztenciájának visszaállításáról. Semmisségi naplózás esetén ez a be nem fejeződött tranzakciók által végrehajtott módosítások semmissé tételét jelenti.

Visszaállítás ellenőrzőpont nélkül

Ez a legegyszerűbb módszer. A teljes naplót látjuk.

- Az első feladat a tranzakciók felosztása sikeresen befejezett és befejezetlen tranzakciókra.
    - Egy T tranzakció sikeresen befejeződött, ha van a naplóban $\langle COMMIT\ T\rangle$ bejegyzés. Ekkor T önmagában nem hagyhatta inkonzisztens állapotban az adatbázist.
- Amennyiben találunk a naplóban $\langle START\ T\rangle$ bejegyzést, de $\langle COMMIT\ T\rangle$ bejegyzést nem, akkor feltételezhetjük, hogy T végrehajtott olyan módosítást az adatbázisban, amely még nem íródott ki lemezre.
    - Ekkor T nem befejezett tranzakció, hatását semmissé kell tenni.
- Az algoritmus a következő:
    - A helyreállítás-kezelő elkezdi vizsgálni a naplóbejegyzéseket az utolsótól kezdve, visszafelé haladva, közben feljegyzi azokat a T tranzakciókat, melyre $\langle COMMIT\ T\rangle$ vagy $\langle ABORT\ T\rangle$ bejegyzést talált.
    - Visszafelé haladva, amikor $\langle T, X, v\rangle$ naplóbejegyzést lát:
        1. Ha T-re találkozott már $COMMIT$ bejegyzéssel, akkor nem tesz semmit.
        2. Más esetben T nem teljes vagy abortált. Ekkor a helyreállítás-kezelő az X adatbáziselem értékét v-re változtatja.

A fenti változtatások végrehajtás után minden nem teljes T tranzakcióra $\langle ABORT\ T\rangle$-t ír a napló végére és kiváltja a napló lemezre írását. Ezt követően az adatbázis normál használata folytatódhat.

Ellenőrzőpont-képzés

A helyreállítás elvben a teljes napló átvizsgálását igényelné. Ha undo naplózást használunk, akkor ha egy T tranzakcióra van $COMMIT$ bejegyzés a naplóban, akkor a T tranzakcióra vonatkozó bejegyzések nem szükségesek a helyreállításhoz, viszont nem feltétlenül igaz az, hogy törölhetjük a T tranzakcióra vonatkozó $COMMIT$ előtti bejegyzéseket. A legegyszerűbb megoldás időnként ellenőrzőpontokat készíteni.

Az egyszerű ellenőrzőpont képzése

1. Új tranzakció indítására vonatkozó kérések leállítása.
2. A még aktív tranzakciók befejeződésének és a $COMMIT/ABORT$ bejegyzés naplóba írásának kivárása.
3. A napló lemezre írása.
4. A $\langle CKPT \rangle$ naplóbejegyzés képzése, naplóba írása, majd a napló lemezre írása.
5. Tranzakcióindítási kérések kiszolgálásának újraindítása.

Az ellenőrzőpont előtt végrehajtott tranzakciók befejeződtek, módosításaik lemezre kerültek. Ezért elég az utolsó ellenőrzőpont utáni részét elemezni a naplónak helyreállításnál.

Ellenőrzőpont létrehozása a rendszer működése közben

Az egyszerű ellenőrzőpont-képzéssel az a probléma, hogy nem engedi új tranzakciók elindítását, amíg az aktív tranzakciók be nem fejeződnek. Ez viszont még sok időt igénybe vehet, a felhasználó számára pedig leállítottnak tűnik a rendszer, hiszen nem tud új tranzakciót indítani. Ezt nem engedhetjük meg. Egy bonyolultabb módszer azonban lehetővé teszi ellenőrzőpont képzését anélkül, hogy az új tranzakciók indítását fel kellene függeszteni.

E módszer lépései:

1. $\langle START\ CKPT(T_1, T_2, ...T_k) \rangle$ bejegyzés készítése és a napló lemezre írása.
   $T_1, ..T_k$ az éppen aktív, befejezetlen tranzakciók.
2. Meg kell várni a $T_1, ..T_k$ tranzakciók befejeződését.
   Eközben indíthatók új tranzakciók.
3. Ha az ellenőrzőpont-képzés kezdetén még aktív $T_1, ..T_k$ tranzakciók mindegyike befejeződött, akkor $\langle END\ CKPT \rangle$ naplóbejegyzés készítése és lemezre írása.

Helyreállítás: Visszafelé elemezve megtaláljuk a be nem fejezett tranzakciókat, az ezen tranzakciók által módosított adatbáziselemek tartalmát a régi értékre állítjuk vissza.
Két eset fordulhat elő, vagy $\langle END\ CKPT \rangle$, vagy $\langle START\ CKPT(T_1, T_2, ...T_k) \rangle$ naplóbejegyzéssel találkozunk előbb.

- Ha előbb $\langle END\ CKPT \rangle$ bejegyzéssel találkozunk, akkor az összes be nem fejezett tranzakcióra vonatkozó bejegyzés megtalálható a legközelebbi $\langle START\ CKPT(T_1, T_2, ...T_k) \rangle$ bejegyzésig. Az ennél korábbiakkal nem kell foglalkoznunk.
- Ha $\langle START\ CKPT(T_1, T_2, ...T_k) \rangle$ bejegyzéssel találkozunk előbb, akkor a hiba ellenőrzőpont-képzés közben történt. Ekkor a $T_1, ..T_k$ tranzakciók közül a legkorábban elindítottnak a START bejegyzéséig kell visszamenni, ami viszont biztosan az ezt megelőző START CKPT bejegyzés után található.

Általános szabályként, ha END CKPT-ot írunk a lemezre, akkor az azt megelőző START CKPT bejegyzést megelőző bejegyzésekre nincs szükség a helyreállítás szempontjából.

Helyrehozó (redo) naplózás

Redo vs. undo naplózás:

- A helyrehozó naplózás a semmisségi naplózással szemben helyreállításnál figyelmen kívül hagyja a befejezetlen tranzakciókat és befejezi a normálisan befejezettek által végrehajtott változtatásokat.
- Undo naplózás esetén a COMMIT naplóba írása előtt megköveteljük a módosítások lemezre írását. Ezzel szemben redo naplózás esetén csak akkor írjuk lemezre a tranzakció által végrehajtott módosításokat, ha a COMMIT bejegyzés a naplóba íródott és lemezre került.
- Undo naplózásnál a módosított elemek régi értékére van szükség helyreállításnál, redo-nál pedig az újra.

Helyrehozó naplózás szabályai: Itt a $\langle T, X, v\rangle$ bejegyzés jelentse azt, hogy a T tranzakció az X adatbáziselemet értékét v-re változtatta. Annak sorrendjét, hogy az adat- és naplóbejegyzések hogyan kell, hogy lemezre kerüljenek, az alábbi, ún. "írj korábban" naplózási szabály határozza meg: Mielőtt az adatbázis bármely X elemét a lemezen módosítanánk, szükséges, hogy a $\langle T, X, v\rangle$ és $\langle COMMIT\ T\rangle$ naplóbejegyzések lemezre kerüljenek.

Helyreállítás helyrehozó naplózás használatával:

- Ha egy T tranzakció esetén nincs $\langle COMMIT\ T\rangle$ bejegyzés a naplóban, akkor tudjuk, hogy T módosításai nem kerültek lemezre, így ezekkel nem kell foglalkozni.
- Ha viszont T befejeződött, azaz van $\langle COMMIT\ T\rangle$ bejegyzés, akkor vagy lemezre kerültek a módosításai, vagy nem. Ezért meg kell ismételni T módosításait.

Helyreállítás:

1. Meghatározni azon tranzakciókat, amelyre van COMMIT bejegyzés a naplóban.
2. Elemezni a naplót az elejéről kezdve. Ha $\langle T, X, v\rangle$ bejegyzést találunk, akkor ha T befejezett tranzakció, akkor v értékét kell X-be írni. Ha T befejezetlen, nem teszünk semmit.
3. Ha végig értünk a naplón, akkor minden be nem fejezett T tranzakcióra $\langle ABORT\ T\rangle$ naplóbejegyzést írunk a naplóba és a naplót lemezre írjuk.

Helyrehozó naplózás ellenőrzőpont-képzéssel

Helyrehozó naplózásnál a működés közbeni ellenőrzőpont-képzés lépései:

1. $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ naplóbejegyzés készítése és lemezre írása, ahol $T_1, ...T_k$ az aktív tranzakciók.
2. Az összes olyan adatbáziselem kiírása lemezre, amelyeket olyan tranzakciók írtak pufferbe, amelyek $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ előtt befejeződtek (COMMIT), de puffereik még nem kerültek lemezre.
3. $\langle END\ CKPT\rangle$ naplóbejegyzés készítése és lemezre írása.

Visszaállítás ellenőrzőponttal kiegészített redo naplózásnál:
Két eset fordulhat elő: az utolsó ellenőrzőponttal kapcsolatos naplóbejegyzés vagy START CKPT, vagy END CKPT.

- Tegyük fel, hogy az utolsó ellenőrzőpont-bejegyzés a naplóban END CKPT. Ekkor a $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ előtt befejeződött tranzakciók módosításai már biztosan lemezre kerültek, ezekkel nem kell foglalkoznunk, viszont a T_i-kel és a START CKPT bejegyzés után indított tranzakciókkal foglalkoznunk kell. Ekkor olyan visszaállítás kell tennünk, mint a sima helyrehozó naplózásnál, annyi különbséggel, hogy csak a T_i-ket és a START CKPT után indított tranzakciókat kell figyelembe venni. A keresésnél a legkorábbi $\langle START\ T_i\rangle$ bejegyzésig kell visszamenni, ahol T_i a $T_1, ...T_k$ valamelyike.
- Tegyük fel, hogy az utolsó ellenőrzőpont-bejegyzés a naplóban $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$. Nem lehetünk biztosak benne, hogy az ez előtt befejeződött tranzakciók módosításai lemezre íródtak, ezért az ezt megelőző END CKPT előtti $\langle START\ CKPT(S_1, S_2, \ldots, S_m)\rangle$ kell visszamenni. Vissza kell állítani azoknak a tranzakcióknak a módosításait, amik S_i-k közül valók, vagy $\langle START\ CKPT(S_1, S_2, \ldots, S_m)\rangle$ után indultak és befejeződtek.

Semmisségi/helyrehozó (undo/redo) naplózás

Undo/redo naplózásnál a módosítást jelző naplóbejegyzések $\langle T, X, v, w\rangle$ alakúak, ami azt jelenti, hogy a T tranzakció az X adatbáziselem értékét v-ről w-re változtatta.

Undo/redo naplózás esetén a következő előírást kell betartani: Mielőtt az adatbázis bármely X elemének értékét módosítanánk a lemezen, a $\langle T, X, v, w\rangle$ naplóbejegyzésnek a lemezre kell kerülnie.

Speciálisan a $\langle COMMIT\ T\rangle$ bejegyzés megelőzheti és követheti is a módosítások lemezre írását.

Helyreállítás undo/redo naplózásnál

Az alapelvek:

1. A legkorábbitól kezdve állítsuk helyre minden befejezett tranzakció hatását.
2. A legutolsótól kezdve tegyük semmissé a be nem fejezett tranzakciók hatását.

Undo/redo naplózás ellenőrzőpont-képzéssel:

1. Írjunk a naplóba $\langle START\ CKPT(T_1, T_2, ...T_k)\rangle$ bejegyzést, ahol $T_1, ..T_k$ az aktív tranzakciók, majd írjuk lemezre a naplót.
2. Írjuk lemezre azokat a puffereket, amelyek módosított adatbáziselemeket tartalmaznak (piszkos pufferek). Redo naplózással ellentétben itt minden piszkos puffert lemezre írunk, nem csak a befejezettekét.
3. Írjunk $\langle END\ CKPT\rangle$ naplóbejegyzést a naplóba és írjuk ki lemezre.

Konkurenciakezelés

A tranzakciók közötti egymásra hatás az adatbázis inkonzisztenssé válását okozhatja, még akkor is, amikor a tranzakciók külön-külön megőrzik a konzisztenciát és rendszerhiba sem történt. Ezért valamiképpen szabályoznunk kell, hogy a különböző tranzakciók egyes lépései milyen sorrendben következzenek egymás után.

A tranzakciós lépések szabályozásának feladatát az adatbázis-kezelő rendszer ***ütemező*** (scheduler) része végzi. Azt az általános folyamatot, amely biztosítja, hogy a tranzakciók egyidejű végrehajtása során megőrizzék a konzisztenciát, ***konkurenciavezérlés***nek nevezzük.

Amint a tranzakciók az adatbáziselemek olvasását és írását kérik, ezek a kérések az ütemezőhöz kerülnek, amely legtöbbször közvetlenül végrehajtja azokat. Amennyiben a szükséges adatbáziselem nincs a pufferben, először a pufferkezelőt hívja meg.
Bizonyos esetekben azonban nem biztonságos azonnal végrehajtani a kéréseket. Az ütemezőnek ekkor késleltetnie kell a kérést, sőt bizonyos esetben abortálnia kell a kérést kiadó tranzakciót.

Az alapfeltevésünk (helyességi elv), hogy ha minden egyes tranzakciót külön hajtunk végre, akkor azok megőrzik a konzisztens adatbázis-állapotot. A gyakorlatban viszont a tranzakciók általában konkurensen futnak, ezért a helyességi elv nem alkalmazható közvetlenül. Így olyan ütemezéseket kell tekintenünk, amelyek biztosítják, hogy ugyanazt az eredményt állítják elő, mintha a tranzakciókat egymás után, egyesével hajtottuk volna végre.

Ütemezések

Az ***ütemezés*** egy vagy több tranzakció által végrehajtott műveletek időrendben vett sorozata, amelyben az egy tranzakcióhoz tartozó műveletek sorrendje megegyezik a tranzakcióban megadott sorrenddel. Az ütemezéseknél csak az olvasási és írási műveletekkel foglalkozunk.

Soros ütemezés: Egy ütemezés soros, ha bármely T és T' tranzakcióra, ha T-nek van olyan művelete, amely megelőzi T' valamely műveletét, akkor T minden művelete megelőzi T' minden műveletét. A soros ütemezést a tranzakciók felsorolásával adjuk meg, pl. (T_1, T_2).

Sorbarendezhetőség: Egy ütemezés sorba rendezhető, ha ugyanolyan hatással van az adatbázis állapotára, mint valamelyik soros ütemezés, függetlenül az adatbázis kezdeti állapotától.

Jelölések:

- $w_i(x)$ azt jelenti, hogy a T_i tranzakció írja az x adatbáziselemet.
- $r_i(x)$ azt jelenti, hogy a T_i tranzakció olvassa az x adatbáziselemet.

Konfliktus: Konfliktus akkor van, ha van olyan egymást követő műveletpár az ütemezésben, amelynek ha a sorrendjét felcseréljük, akkor legalább az egyik tranzakció viselkedése megváltozik. Tegyük fel, hogy T_i és T_j különböző tranzakciók.
Ekkor nincs konfliktus, ha a pár:

- $r_i(X)$ és $r_j(Y)$, még akkor sem, ha $X = Y$. Azaz két különböző tranzakció által végrehajtott olvasási művelet sosem áll konfliktusban egymással, még akkor sem, ha ugyanarra az adatbázis-elemre vonatkoznak.
- $r_i(X)$ és $w_j(Y)$, ha $X \neq Y$
- $w_i(X)$ és $r_j(Y)$, ha $X \neq Y$
- $w_i(X)$ és $w_j(Y)$, ha $X \neq Y$

Konfliktus van, ha:

- Ugyanannak a tranzakciónak bármely két művelete konfliktusban van, hiszen ezek nem cserélhetők fel.
- Ugyanazt az adatbáziselemet két különböző tranzakció éri el, és ezek közül legalább az egyik írási művelet.

Konfliktusekvivalens ütemezések: Két ütemezés konfliktusekvivalens, ha szomszédos műveletek nem konfliktusos cseréjével egymásba vihetők.

Konfliktus-sorbarendezhető ütemezések: Egy ütemezés konfliktus-sorbarendezhető, ha konfliktusekvivalens valamely soros ütemezéssel. A konfliktus-sorbarendezhetőség elégséges, de nem szükséges feltétele a sorbarendezhetőségnek. Piaci rendszerekben a konfliktus-sorbarendezhetőséget ellenőrzik.

Megelőzési gráf: Adott a T_1 és T_2 tranzakcióknak, esetleg további tranzakcióknak is, egy S ütemezése. T_1 megelőzi T_2-t S-ben, ha van a T_1-ben olyan A_1 művelet és T_2-ben olyan A_2 művelet, melyekre:

1. A_1 megelőzi A_2-t S-ben,
2. A_1 és A_2 ugyanarra az adatbáziselemre vonatkoznak, és
3. legalább az egyik írási művelet

Ezek pont azok a feltételek, amikor A_1 és A_2 konfliktusban vannak, nem cserélhetők fel. Ezeket a megelőzéseket megelőzési gráffal szemléltethetjük. A megelőzési gráf csomópontjai S-beli tranzakciók. Ha a tranzakciókat T_i-vel jelöljük, legyen i a T_i-hez tartozó csomópont a gráfban. Az i csomópontból j csomópontba megy irányított él, ha T_i megelőzi T_j-t.

Megelőzési gráf és a konfliktus-sorbarendezhetőség kapcsolata: Egy S ütemezés konfliktus-sorbarendezhető akkor és csak akkor, ha megelőzési gráfja körmentes. Ekkor a megelőzési gráf csúcsainak bármely topologikus rendezése megad egy konfliktusekvivalens soros ütemezést.

Zárak

Zárak használatával is elérhető a konfliktus-sorbarendezhetőség. Ha az ütemező zárakat használ, akkor a tranzakcióknak zárakat kell kérniük és feloldaniuk az adatbáziselemek olvasásán és írásán felül. A zárak használatának két értelemben is helyesnek kell lennie:

- Tranzakciók konzisztenciája: A műveletek és a zárak az alábbi elvárások szerint kapcsolódnak egymáshoz:
  1. A tranzakció csak akkor olvashat vagy írhat egy elemet, ha már korábban zárolta azt, és még nem oldotta fel a zárat.
  2. Ha egy tranzakció zárol egy elemet, akkor azt később fel kell szabadítania.
- Az ütemezések jogszerűsége: A zárak értelme feleljen meg a szándék szerinti elvárásnak: nem zárolhatja két tranzakció ugyanazt az elemet, csak úgy, ha az egyik előbb már feloldotta a zárat.

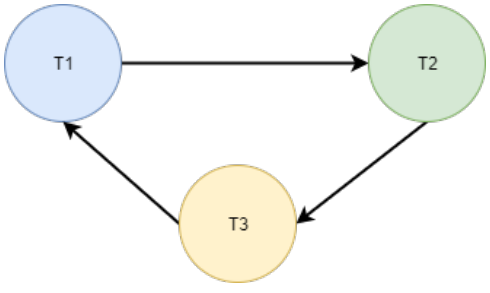
Zárolási ütemező: A zárolási ütemező feladata, hogy akkor és csak akkor engedélyezze a kéréseket, ha az jogszerű ütemezést eredményez. Ebben segít a zártábla, amely minden adatbáziselemhez megadja azt a tranzakciót, feltéve, hogy van ilyen, amelyik éppen zárolja az adott elemet.

Kétfázisú zárolás: Kétfázisú zárolásról (2FZ) beszélünk, ha minden tranzakcióban minden zárolási művelet megelőz minden feloldási műveletet. Azok a tranzakciók, amelyek eleget tesznek 2FZ-nek, 2FZ tranzakcióknak nevezzük. Konzisztens, 2FZ tranzakciók jogszerű ütemezése konfliktus-sorbarendezhető.

$$T_i = \ldots\ldots l_i(A) \ldots\ldots\ldots u_i(A) \ldots\ldots$$
$$\rightarrow \quad \nexists l_j(X) \quad \leftarrow$$

Holtpont: Holtpontról beszélünk, ha az ütemező arra kényszerít egy tranzakciót, hogy örökké várjon egy olyan zárra, amelyet egy másik tranzakció tart zárolva. Tipikus példa holtpont kialakulására, ha 2 tranzakció egymás által zárolt elemeket akar zárolni.
A kétfázisú zárolás nem tudja megakadályozni holtpontok kialakulását.

A felismerésben segít a zárkérések sorozatához tartozó ***várakozási gráf***: csúcsai a tranzakciók és akkor van él T_i-ből T_j-be, ha T_i vár egy olyan zár elengedésére, amit T_j tart éppen. A várakozási gráf változik az ütemezés során, ahogy újabb zárkérések érkeznek vagy zárelengedések történnek, vagy az ütemező abortáltat egy tranzakciót.



3. ábra. Példa holtpontra: $l_1(A); l_2(B); l_3(C); l_1(B); l_2(C); l_3(A)$

Különböző zármódú zárolási rendszerek

Probléma: A T tranzakciónak akkor is zárolnia kell az X adatbáziselemet, ha csak olvasni akarja X-et, írni nem.

- Ha nem zárolnánk, akkor esetleg egy másik tranzakció azalatt írna X-be új értéket, mialatt T aktív, ami nem sorba rendezhető viselkedést okoz.
- Másrészről pedig miért is ne olvashatná több tranzakció egyidejűleg X értékét mindaddig, amíg egyiknek sincs engedélyezve, hogy írja.

Osztott és kizárólagos zárak

Mivel ugyanannak az adatbáziselemnek két olvasási művelete nem eredményez konfliktust, így ahhoz, hogy az olvasási műveleteket egy bizonyos sorrendbe soroljuk, nincs szükség zárolásra.
Viszont szükséges azt az elemet is zárolni, amelyet olvasunk, mert ennek az elemnek az írását nem szabad közben megengednünk.
Az íráshoz szükséges zár viszont "erősebb", mint az olvasáshoz szükséges zár, mivel ennek mind az olvasásokat, mind az írásokat meg kell akadályoznia.

A legelterjedtebb zárolási séma két különböző zárat alkalmaz: az ***osztott zár***akat vagy olvasási zárakat, és a ***kizárólagos zár***akat vagy írási zárakat.

Tetszőleges X adatbáziselemet vagy egyszer lehet zárolni kizárólagosan, vagy akárhányszor lehet zárolni osztottan, ha még nincs kizárólagosan zárolva.
Amikor írni akarjuk X-et, akkor X-en kizárólagos zárral kell rendelkeznünk, de ha csak olvasni akarjuk, akkor X-en akár osztott, akár kizárólagos zár megfelelő.

Feltételezzük, hogy ha olvasni akarjuk X-et, de írni nem, akkor előnyben részesítjük az osztott zárolást.

1. Tranzakciók konzisztenciája: Nem írhatunk kizárólagos zár fenntartása nélkül, és nem olvashatunk valamilyen zár fenntartása nélkül.
   Minden zárolást követnie kell egy ugyanannak az elemnek a zárolását feloldó műveletnek.
2. Tranzakciók kétfázisú zárolása: A zárolásoknak meg kell előzniük a zárak feloldását.
3. Az ütemezések jogszerűsége: Egy elemet vagy egyetlen tranzakció zárol kizárólagosan, vagy több is zárolhatja osztottan, de a kettőegyszerre nem lehet.

Kompatibilitási mátrix

A kompatibilitási mátrix minden egyes zármódhoz rendelkezik egy sorral és egy oszloppal.

A sorok egy másik tranzakció által az X elemre már érvényes záraknak felelnek meg, az oszlopok pedig az X-re kért zármódoknak felelnek meg.

A kompatiblitási mátrix használatának szabálya: Egy A adatbáziselemre $C(= S \vee X)$ módú zárat akkor és csak akkor engedélyezhetünk, ha a táblázat minden olyan R sorára, amelyre más tranzakció már zárolta A-t R módban, a C oszlopban „*Igen*" szerepel.

Ha ilyen zár már van kiadva	Megkaphatjuk-e ezt a típusú zárat? S	X
S	Igen	Nem
X	Nem	Nem

4. ábra. Az osztott (S) és kizárólagos (X) zárak kompatibilitási mátrixa.

Kompatibilitási mátrixok használata

1. A mátrix alapján dönti el az ütemező, hogy egy ütemezés/zárkérés legális-e, illetve ez alapján várakoztatja a tranzakciókat. Minél több az „*Igen*" a mátrixban, annál kevesebb lesz a várakoztatás.

2. A mátrix alapján keletkező várakozásokhoz elkészített várakozási gráf segítségével az ütemező kezeli a holtpontot.

3. A mátrix alapján készíti el az ütemező a megelőzési gráfot egy zárkérés-sorozathoz:

   - a megelőzési gráf csúcsai trakzakciók és akkor van és T_i-ből T_j-be, ha van olyan A adategység, amelyre az ütemezés során Z_k zárat kért és kapott T_i, ezt engedte, majd
   - ezután A-ra legközelebb T_j kért és kapott olyan Z_s zárat, hogy a mátrixban a Z_k sor Z_s oszlopában „*Nem*" áll.
     Vagyis olyankor lesz él, ha a két zár nem kompatibilis egymással; nem mindegy a két művelet sorrendje

$$\begin{array}{c} \ldots T_i : Z_k(A) \ldots T_i : U Z_k(A) \ldots T_j : Z_s(A) \\ \uparrow \quad \text{nem kompatibilis} \quad \uparrow \end{array}$$

A sorbarendezhetőséget a megelőzési gráf segítségével lehet eldönteni.

T: Egy csak zárkéréseket és zárelengedéseket tartalmazó jogszerű ütemezés sorbarendezhető akkor és csak akkor, ha a kompatibilitási mátrix alapján felrajzolt megelőzési gráf nem tartalmaz irányított kört.