Adatbázisrendszerek megvalósítása 1

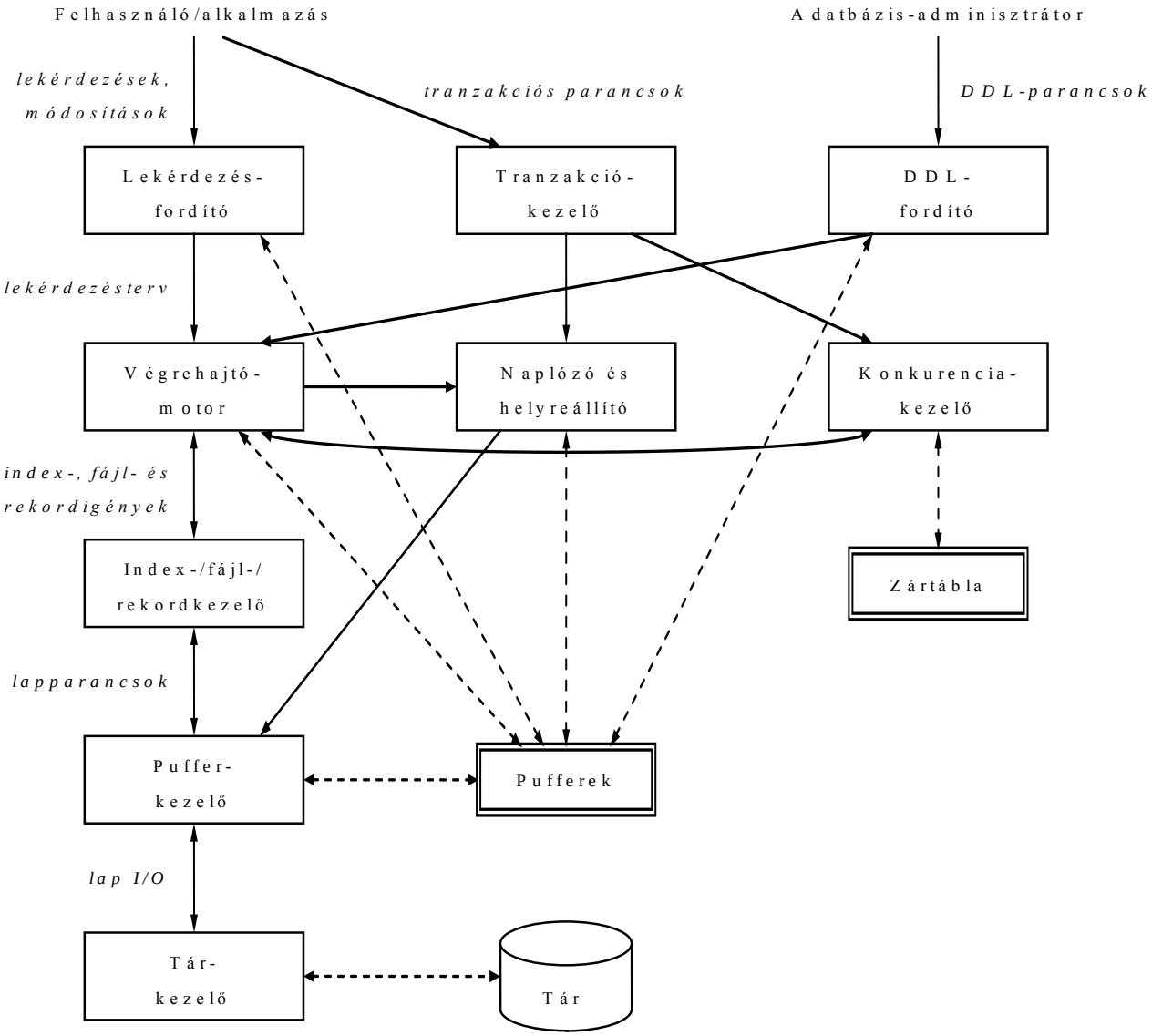
Irodalom: Hector Garcia-Molina – Jeffrey D. Ullman – Jennifer Widom: Adatbázisrendszerek megvalósítása, 8. és 9. fejezet

Előfeltétel: Adatbázisrendszerek tárgy.

Tartalom: Rendszerhibák és a kivédésükre szolgáló naplózási technikák; konkurenciakezelés.

Bevezetés

Az adatbázis-kezelő rendszer alkotórészei



Az ábrán egy teljes adatbázis-kezelő rendszer vázát láthatjuk. Az egyvonalas dobozok a rendszer alkotórészeit jelentik, míg a dupla dobozok memóriabeli adatszerkezeteket reprezentálnak. A folytonos vonalak jelölik az olyan vezérlésátadást, ahol adatok is áramlanak, a szaggatott vonalak pedig csak az adatmozgást jelölik. Az adatbázis-kezelő rendszerrel kapcsolatos kölcsönhatások döntő többsége az ábra

bal oldalán lévő útvonalat követi. A felhasználó vagy az alkalmazói program olyan működést indít el, amelynek nincs hatása az adatbázissémára, viszont hatással lehet az adatbázis tartalmára (módosító utasítás esetén), illetve adatokat gyűjthet ki az adatbázisból (lekérdezés esetén). Két olyan útvonal van, amely mentén a felhasználó cselekménye hatást gyakorol az adatbázisra:

1. *A lekérdezés megválaszolása.* A lekérdezésfordító elemzi és optimalizálja a lekérdezést. Az eredményül kapott lekérdezés-végrehajtási tervet (röviden lekérdezéstervet) vagy a lekérdezés megválaszolásához szükséges tevékenységek sorozatát továbbítja a végrehajtómotornak. A végrehajtómotor kisebb adatdarabokra (tipikusan rekordokra) vonatkozó kérések sorozatát adja át az erőforrás-kezelőnek. Az erőforrás-kezelő ismeri a relációkat tartalmazó adatfájlokat, a fájlok rekordjainak formátumát, méretét és az indexfájlokat is. Az indexfájlok segítenek abban, hogy az adatfájlok elemeit gyorsan meg lehessen találni. Az adatkéréseket az erőforrás-kezelő lefordítja lapokra, és ezeket a kéréseket továbbítja a pufferkezelőnek. A pufferkezelő feladata, hogy a másodlagos adattárolón (általában lemezen) tárolt adatok megfelelő részét hozza be a központi memória puffereibe. A pufferek és a lemez közti adatátvitel egysége általában egy lap vagy egy lemezblokk. A pufferkezelő információt cserél a tárkezelővel, hogy megkapja az adatokat a lemezről. Megtörténhet, hogy a tárkezelő az operációs rendszer parancsait is igénybe veszi, de tipikusabb, hogy az adatbázis-kezelő a parancsait közvetlenül a lemezvezérlőhöz intézi.
2. *A tranzakció feldolgozása.* A lekérdezéseket és más tevékenységeket tranzakciókba csoportosíthatjuk. A tranzakciók olyan munkaegységek, amelyeket atomosan és más tranzakcióktól látszólag elkülönítve kell végrehajtani. Gyakran minden egyes lekérdezés vagy módosítás önmagában is egy tranzakció. Ezenkívül a tranzakció végrehajtásának tartósnak kell lennie, ami azt jelenti, hogy bármelyik befejezett tranzakció hatását még akkor is meg kell tudni őrizni, ha a rendszer összeomlik a tranzakció befejezése utáni pillanatban. A tranzakciófeldolgozót két fő részre osztjuk:
   a) *Konkurenciavezérlés-kezelő* vagy *ütemező* (scheduler): a tranzakciók elkülönítésének és atomosságának biztosításáért felelős.
   b) *Naplózás- és helyreállítás-kezelő*: a tranzakciók atomosságáért és tartósságáért felelős.

A tranzakció

A *tranzakció* (transaction) az adatbázis-műveletek végrehajtási egysége, amely DML-beli utasításokból áll, és a következő tulajdonságokkal rendelkezik:

- *Atomosság* (**a**tomicity): a tranzakció „mindent vagy semmit” jellegű végrehajtása (vagy teljesen végrehajtjuk, vagy egyáltalán nem hajtjuk végre).
- *Konzisztenciamegőrzés* (**c**onsistency preservation): az a feltétel, hogy a tranzakció megőrizze az adatbázis konzisztenciáját, azaz a tranzakció végrehajtása után is teljesüljenek az adatbázisban előírt konzisztenciamegszorítások (integritási megszorítások), azaz az adatelemekre és a közöttük lévő kapcsolatokra vonatkozó elvárások.
- *Elkülönítés* (**i**solation): az a tény, hogy minden tranzakciónak látszólag úgy kell lefutnia, mintha ez alatt az idő alatt semmilyen másik tranzakciót sem hajtanánk végre.
- *Tartósság* (**d**urability): az a feltétel, hogy ha egyszer egy tranzakció befejeződött, akkor már soha többé nem veszhet el a tranzakciónak az adatbázison kifejtett hatása.

Ezek a tranzakció *ACID-tulajdonságai* (ACID properties). A konzisztenciamegőrzést mindig adottnak tekintjük (lásd később: korrektség alapelve), a másik három tulajdonságot viszont az adatbázis-kezelő rendszernek kell biztosítania, bár ettől időnként eltekintünk. Ha egy ad hoc utasítást adunk az SQL-rendszernek, akkor minden lekérdezés vagy adatbázis-módosító utasítás egy tranzakció. Amennyiben beágyazott SQL-interfészt használva a programozó készíti el a tranzakciót, akkor egy tranzakcióban több SQL-lekérdezés és -módosítás szerepelhet. A tranzakció ilyenkor általában egy DML-utasítással kezdődik,

és egy `COMMIT` vagy `ROLLBACK` utasítással végződik. Ha a tranzakció valamely utasítása egy triggert aktivizál, akkor az is a tranzakció részének tekintendő, akárcsak a trigger által kiváltott további triggerek. (A trigger olyan programrész, amely bizonyos események bekövetkeztekor automatikusan lefut.)

A tranzakció feldolgozása

A tranzakciófeldolgozó biztosítja az adatok konkurens elérését és a helyreállíthatóságot (resiliency) a tranzakciók korrekt végrehajtásával. A tranzakciókezelő fogadja az alkalmazás tranzakciós utasításait. Az alkalmazás azt is megmondja a tranzakciókezelőnek, hogy mikor kezdődnek és végződnek a tranzakciók, és még egyéb információt is ad az alkalmazás elvárásairól (például lehet, hogy nem akarja megkövetelni az atomosságot). A tranzakciófeldolgozó a következő feladatokat hajtja végre:

1. *Naplózás*: Annak érdekében, hogy a tartósságot biztosítani lehessen, az adatbázis minden változását külön feljegyezzük (naplózzuk) lemezen. A naplókezelő (log manager) többféle eljárásmód közül választja ki azt, amelyiket követni fog. Ezek az eljárásmódok biztosítják azt, hogy teljesen mindegy, mikor történik a rendszerhiba vagy a rendszer összeomlása, a helyreállítás-kezelő meg fogja tudni vizsgálni a változások naplóját, és ez alapján vissza tudja állítani az adatbázist valamilyen konzisztens állapotába. A naplókezelő először a pufferekbe írja a naplót, és egyeztet a pufferkezelővel, hogy a pufferek alkalmas időpillanatokban garantáltan íródjanak ki lemezre, ahol már az adatok túlélhetik a rendszer összeomlását.
2. *Konkurenciavezérlés*: A tranzakcióknak úgy kell látszódniuk, mintha egymástól függetlenül, elkülönítve végeznénk el őket. A legtöbb rendszerben igazából sok tranzakciót kell egyszerre végrehajtani. Így aztán az ütemező (konkurenciavezérlés-kezelő) feladata, hogy meghatározza az összetett tranzakciók résztevékenységeinek egy olyan sorrendjét, amely biztosítja azt, hogy ha ebben a sorrendben hajtjuk végre a tranzakciók elemi tevékenységeit, akkor az összhatás megegyezik azzal, mintha a tranzakciókat tulajdonképpen egyenként és egységes egészként hajtottuk volna végre. A tipikus ütemező ezt a munkát azáltal látja el, hogy az adatbázis bizonyos részeire elhelyezett *zárakat* (lock) karbantartja. Ezek a zárak megakadályoznak két tranzakciót abban, hogy rossz kölcsönhatással használják ugyanazt az adatrészt. A zárakat rendszerint a központi memória *zártáblájában* (lock table) tárolja a rendszer (lásd ábra). Az ütemező azzal befolyásolja a lekérdezések és más adatbázis-műveletek végrehajtását, hogy megtiltja a végrehajtómotornak, hogy hozzányúljon az adatbázis zár alá helyezett részeihez.
3. *Holtpont feloldása*: A tranzakciók az ütemező által engedélyezett zárak alapján versenyeznek az erőforrásokért. Így előfordulhat, hogy olyan helyzetbe kerülnek, amelyben egyiküket sem lehet folytatni, mert mindegyiknek szüksége lenne valamire, amit egy másik tranzakció birtokol. A tranzakciókezelő feladata, hogy ilyenkor közbeavatkozzon, és töröljön (abortáljon) egy vagy több tranzakciót úgy, hogy a többit már folytatni lehessen.

A rendszerhibák kezelése

A kérdés az, milyen technikákkal lehet biztosítani a helyreállíthatóságot, azaz hogyan tudjuk megőrizni az adatok integritását rendszerhibák előfordulásakor. Az adatoknak nem szabad sérülniük több hibamentes lekérdezés vagy adatbázis-módosítás egyszerre történő végrehajtásakor sem, ezzel a konkurenciavezérlés foglalkozik.

A helyreállíthatóság biztosítására az elsődleges technika a *naplózás* (logging), amely valamilyen biztonságos módszerrel rögzíti az adatbázisban végrehajtott módosítások történetét. Három különböző módszert tanulmányozunk: a semmisségi (undo), a helyrehozó (redo) és a semmisségi/helyrehozó (undo/redo) naplózást. Foglalkozunk továbbá a *helyreállítással* (recovery), azzal az eljárással, amikor a naplót felhasználva az adatbázist konzisztens állapotba hozzuk, valamint az *archiválással* (dump, backup),

mellyel biztosíthatjuk, hogy az adatbázis nemcsak az ideiglenes rendszerhibákat, de a teljes adatbázis elveszítését is túlélje.

A hibák fajtái

Az adatbázis lekérdezése vagy módosítása során számos dolog hibát okozhat a billentyűzeten történt adatbeviteli hibáktól kezdve az adatbázist tároló lemez elhelyezésére szolgáló helyiségben történő robbanásig.

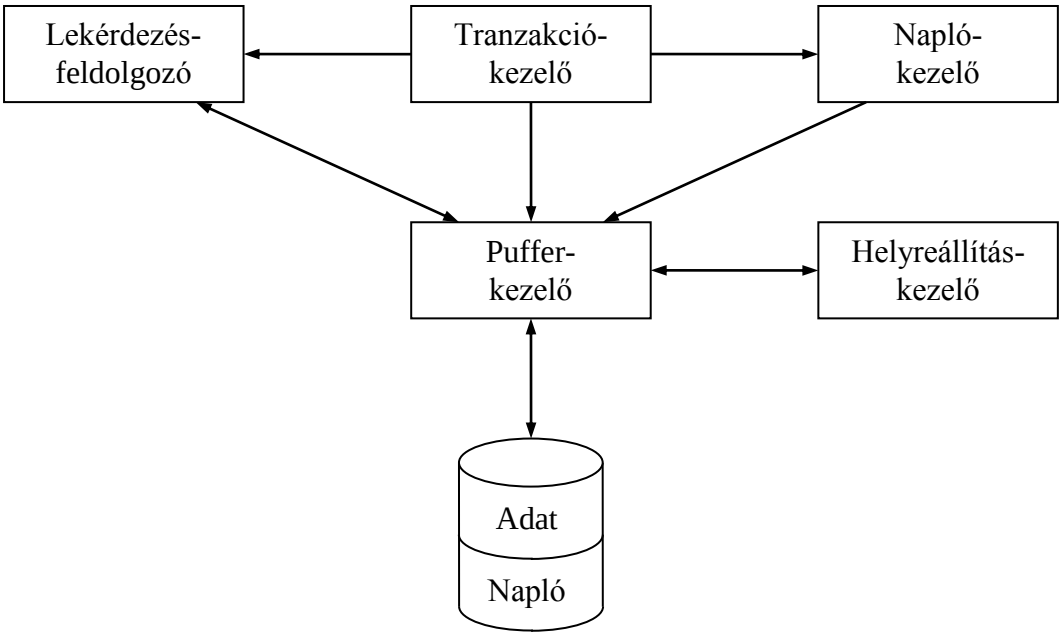
- *Hibás adatbevitel*: Ezek a hibák gyakran nem észrevehetők. Ha például a felhasználó elüt egy számot egy telefonszámban, akkor az adat még úgy néz ki, mint egy telefonszám, csak éppen tartalmilag hibás lesz. Ha viszont kihagy egy számjegyet a telefonszámból, akkor már formailag is hibás (ha megkövetelünk egy rögzített formátumot). A modern adatbázis-kezelő rendszerek számos szoftverelemet biztosítanak a fentiekhez hasonló adatbeviteli hibák felismerésére. Például az SQL2 és az SQL3 szabványokban az adatbázis tervezője megadhat előírásokat, mint például kulcsra, külső kulcsra vagy értékekre vonatkozó megszorításokat (hogy például a telefonszámnak 10 jegyből kell állnia). A triggerek azok a programok, amelyek bizonyos típusú módosítások (például egy relációba történő beszúrás) esetén hajtódnak végre, annak ellenőrzésére, hogy a frissen bevitt adatok megfelelnek-e az adatbázis-tervező által megszabott előírásoknak.
- *Készülékhibák*: A lemezegységek olyan helyi hibái, melyek egy vagy több bit megváltozását okozzák, a lemez szektoraihoz rendelt paritás-ellenőrzéssel megbízhatóan felismerhetők. A lemezegységek jelentős sérülése, elsősorban az író-olvasó fejek katasztrófái, az egész lemez olvashatatlanná válását okozhatják. Az ilyen hibákat általában az alábbi megoldások segítségével kezelik:
  1. A *RAID-módszerek* (Redundant Array of Independent Disks) valamelyikének használatával az elveszett lemez tartalma visszatölthető.
  2. Az *archiválás* használatával az adatbázisról másolatot készítünk valamilyen eszközre (például szalagra vagy optikai lemezre). A mentést rendszeresen kell végezni vagy teljes, vagy növekményes (csak az előző mentés óta történt változásokat archiváljuk) mentést használva. A mentett anyagot az adatbázistól biztonságos távolságban kell tárolnunk.
  3. Az adatbázisról fenntarthatunk *elosztott, on-line másolatokat***.** Ebben az esetben biztosítanunk kell a másolatok konzisztenciáját.
- *Katasztrofális hibák*: Ebbe a kategóriába soroljuk azokat a helyzeteket, amikor az adatbázist tartalmazó eszköz teljesen tönkremegy robbanás, tűz, vandalizmus vagy akár vírusok következtében. A RAID ekkor nem segít, mert az összes lemez és a paritás-ellenőrző lemezeik is egyszerre használhatatlanná válnak. A másik két biztonsági megoldás viszont alkalmazható katasztrofális hibák esetén is.
- *Rendszerhibák*: Minden tranzakciónak van *állapota*, mely azt képviseli, hogy mi történt eddig a tranzakcióban. Az állapot tartalmazza a tranzakció kódjában a végrehajtás pillanatnyi helyét és a tranzakció összes lokális változójának értékét. A rendszerhibák azok a problémák, melyek a tranzakció állapotának elvesztését okozzák. Tipikus rendszerhibák az áramkimaradásból és a szoftverhibákból eredők, hiszen ezek a memória tartalmának felülírásával járhatnak. Ha egy rendszerhiba bekövetkezik, onnantól kezdve nem tudjuk, hogy a tranzakció mely részei kerültek már végrehajtásra, beleértve az adatbázis-módosításokat is. A tranzakció ismételt futtatásával nem biztos, hogy a problémát korrigálni tudjuk (például egy mezőnek eggyel való növelése esetén). Az ilyen jellegű problémák legfontosabb ellenszere minden adatbázis-változtatás naplózása egy elkülönült, nem illékony naplófájlban, lehetővé téve ezzel a visszaállítást, ha az szükséges. Ehhez hibavédett naplózási mechanizmusra van szükség.

A naplókezelő és a tranzakciókezelő

A tranzakciók korrekt végrehajtásának biztosítása a tranzakciókezelő feladata. A tranzakciókezelő részrendszer egy sor feladatot lát el, többek között

- jelzéseket ad át a naplókezelőnek úgy, hogy a szükséges információ naplóbejegyzés formában a naplóban tárolható legyen;
- biztosítja, hogy a párhuzamosan végrehajtott tranzakciók ne zavarhassák egymás működését (ütemezés).

A tranzakciókezelőt és kapcsolatait az alábbi ábra mutatja:



A tranzakciókezelő a tranzakció tevékenységeiről üzeneteket küld a naplókezelőnek, üzen a pufferkezelőnek arra vonatkozóan, hogy a pufferek tartalmát szabad-e vagy kell-e lemezre másolni, és üzen a lekérdezésfeldolgozónak arról, hogy a tranzakcióban előírt lekérdezéseket vagy más adatbázis-műveleteket kell végrehajtania.

A naplókezelő a naplót tartja karban. Együtt kell működnie a pufferkezelővel, hiszen a naplózandó információ elsődlegesen a memóriapufferekben jelenik meg, és bizonyos időnként a pufferek tartalmát lemezre kell másolni. A napló (adat lévén) a lemezen területet foglal el, ahogy ez az ábrán is látszik.

Ha baj van, akkor a helyreállítás-kezelő aktivizálódik. Megvizsgálja a naplót, és ha szükséges, a naplót használva helyreállítja az adatokat. A lemez elérése most is a pufferkezelőn át történik.

A tranzakciók korrekt végrehajtása

Definiálnunk kell, mit értünk korrekt végrehajtás alatt. Feltesszük, hogy az adatbázis elemekből áll. Az *adatbáziselem* (database element) a fizikai adatbázisban tártolt adatok egyfajta funkcionális egysége, amelynek értékét tranzakciókkal lehet elérni (kiolvasni) vagy módosítani (kiírni). Az elemek alatt érthetünk relációt (vagy OO megfelelőjét, az osztálykiterjedést), relációsort (vagy OO megfelelőjét, az objektumot) vagy lemezblokkot, illetve -lapot. Ez utóbbi a legjobb választás a naplózás szempontjából, mivel ekkor a puffer egyszerű elemekből fog állni, és ezzel elkerülhető néhány súlyos probléma, például amikor az adatbázis valamely elemének egy része van csak a nem illékony memóriában (a lemezen).

Az adatbázis összes elemének pillanatnyi értékét az adatbázis *állapotának* (database state) nevezzük. Bizonyos adatbázis-állapotokat *konzisztensnek* (consistent) tekintünk, míg a többi adatbázis-állapotot *inkonzisztensnek* (inconsistent) minősítjük. A konzisztens állapotok kielégítik az adatbázissémára vonatkozó összes *explicit megszorítást* (explicit constraint) és *implicit megszorítást* (implicit constraint), melyek az adatbázis tervezőjének elgondolásaiban szerepelnek. Az explicit megszorítások betartását az adatbázis-kezelő rendszer kényszeríteni tudja azzal, hogy az olyan tranzakciókat, melyek megsértik az előírt összefüggéseket, a rendszer visszautasítja, így az adatbázisban semmilyen változtatás nem történik. Az implicit megszorítások azok, amelyeket nem tudunk egzakt módon jellemezni. Az egyetlen lehetőségünk az ilyen megszorítások betartásának biztosítására annak feltételezése, hogy ha valaki jogot kap az adatbázis módosítására, akkor neki legyen joga annak eldöntésére is, hogy melyek az elvárt implicit megszorítások.

A tranzakciókra vonatkozó alapvető feltevésünk *a korrektség alapelve* (correctness principle): Ha a tranzakciót minden más tranzakciótól függetlenül (egyedül) és rendszerhiba nélkül végrehajtjuk, és ha indulásakor az adatbázis konzisztens állapotban volt, akkor a tranzakció befejezése után is konzisztens állapotban lesz (elkülönítés + atomosság –> konzisztenciamegőrzés). A korrektség alapelvéhez kapcsolódik a naplózás technikája és a konkurenciavezérlési mechanizmus. Két lehetőség inkonzisztens állapot előidézésére:

- Nem teljesül a tranzakció atomosság tulajdonsága: ha a tranzakciónak csak egy részét sikerült végrehajtani, akkor nagy esélyünk van arra, hogy az általa előállított adatbázis-állapot nem lesz konzisztens.
- A párhuzamosan végrehajtott tranzakciók jó eséllyel inkonzisztens állapothoz vezetnek, hacsak meg nem teszünk bizonyos megelőző lépéseket.

A tranzakciók alaptevékenységei

A tranzakció és az adatbázis kölcsönhatásának három fontos helyszíne van:

1. az adatbázis elemeit tartalmazó lemezblokkok területe;
2. a pufferkezelő által használt virtuális vagy valós memóriaterület;
3. a tranzakció memóriaterülete.

Ahhoz, hogy a tranzakció egy adatbáziselemet beolvashasson, azt előbb memóriapuffer(ek)be kell behozni, ha még nincs ott. Ezt követően tudja a puffer(ek) tartalmát a tranzakció a saját memóriaterületére beolvasni. Az adatbáziselem új értékének kiírása fordított sorrendben történik: az új értéket a tranzakció alakítja ki a saját memóriaterületén, majd ez az új érték másolódik át a megfelelő puffer(ek)be. Fontos, hogy egy tranzakció sohasem módosíthatja egy adatbáziselem értékét közvetlenül a lemezen!

A pufferek tartalmát vagy azonnal lemezre lehet írni, vagy nem; az erre vonatkozó döntés általában a pufferkezelő joga. A naplózó rendszer használatának egyik lefőbb lépése a rendszerhibákból való helyreállíthatóság biztosítása érdekében a pufferkezelő ösztönzése a pufferbeli blokkok megfelelő időpontban történő lemezre írására. Ugyanakkor a lemez I/O-műveletek számának csökkentésére az adatbázis-kezelő rendszerek megengedhetik a módosításoknak csak az illékony memóriában történő végrehajtását, legalábbis bizonyos ideig és bizonyos feltételek teljesülése esetén.

A különböző területek közötti adatmozgásokat megvalósító alapműveletek leírására a következő jelölésrendszert vezetjük be:

1. `INPUT(X)`: Az `X` adatbáziselemet tartalmazó lemezblokk másolása a memóriapufferbe.
2. `READ(X,t)`: Az `X` adatbáziselem bemásolása a tranzakció `t` lokális változójába. Részletesebben: ha az `X` adatbáziselemet tartalmazó blokk nincs a memóriapufferben, akkor előbb végrehajtódik `INPUT(X)`. Ezután kapja meg a `t` lokális változó `X` értékét.

3. `WRITE(X,t)`: A `t` lokális változó tartalma az `X` adatbáziselem memóriapufferbeli tartalmába másolódik. Részletesebben: ha az `X` adatbáziselemet tartalmazó blokk nincs a memóriapufferben, akkor előbb végrehajtódik `INPUT(X)`. Ezután másolódik át a `t` lokális változó értéke a pufferbeli `X`-be.
4. `OUTPUT(X)`: Az `X` adatbáziselemet tartalmazó puffer kimásolása lemezre.

A fenti műveleteknek akkor van értelmük, ha feltételezzük, hogy az adatbáziselemek elférnek egy-egy lemezblokkban és így egy-egy pufferben is, azaz feltételezhetjük, hogy az adatbáziselemek pontosan a blokkok. Adatbáziselem lehet egy relációsor is, ha a relációs séma nem engedi meg nagyobb sorok előfordulását, mint amennyi hely egy blokkban rendelkezésre áll. Ha az adatbáziselem több blokkot foglal el, akkor úgy is tekinthetjük, hogy az adatbáziselem minden blokkméretű része önmagában egy adatbáziselem. A naplózási mechanizmus, amelyet arra használunk, hogy a tranzakció ne fejeződhessen be az `X` kiírása nélkül, atomos; azaz vagy lemezre írja `X` összes blokkját, vagy semmit sem ír ki. A továbbiakban feltételezzük, hogy az adatbáziselem nem nagyobb egy blokknál.

A `READ` és a `WRITE` műveleteket a tranzakciók használják, az `INPUT` és `OUTPUT` műveleteket a pufferkezelő alkalmazza, illetve bizonyos feltételek mellett az `OUTPUT` műveletet a naplózási rendszer is használja.

Példa. Annak bemutatására, hogy a tranzakció mikor és hogyan használja a fenti alapműveleteket, tegyük fel, hogy az adatbázis két, `A` és `B` eleme tartalmának az adatbázis minden konzisztens állapotában meg kell egyeznie. A `T` tranzakció tartalmazza a következő két lépést:

```
A := A*2;
B := B*2;
```

Ha a tranzakcióra az egyetlen konzisztenciaelvárás az `A` = `B`, továbbá ha `T` konzisztens adatbázis-állapotban indul, és tevékenységét rendszerhiba, valamint a párhuzamosan működő tranzakciókkal való kölcsönhatás nélkül be tudja fejezni, akkor az adatbázis befejezéskori állapotának is konzisztensnek kell lennie. Ekkor `T` megduplázva két azonos tartalmú elem értékét, kap két új, azonos értékű elemet.

`T` végrehajtása maga után vonja `A` és `B` lemezről való beolvasását, az aritmetikai műveletek a `T` lokális változóiban kerülnek végrehajtásra, végül `A` és `B` új értékei visszaírásra kerülnek a puffereikbe. `T`-t hat lényeges lépésből állónak tekinthetjük:

```
READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);
```

Ehhez még hozzáadódik az, hogy a pufferkezelő önállóan végrehajt `OUTPUT` lépéseket a pufferek tartalmának lemezre történő visszaírása végett. Legyen kezdetben `A` = `B` = 8. Az `A` és `B` pufferbeli és lemezen tárolt értékei és a `T` tranzakció `t` lokális változójának értékei lépésenként a következők:

Tevékenység	*t*	*M*–A	*M*–B	*D*–A	*D*–B
`READ(A,t)`	8	8		8	8
`t := t*2`	16	8		8	8
`WRITE(A,t)`	16	16		8	8
`READ(B,t)`	8	16	8	8	8
`t := t*2`	16	16	8	8	8
`WRITE(B,t)`	16	16	16	8	8
`OUTPUT(A)`		16	16	16	8
`OUTPUT(B)`		16	16	16	16

`T` első lépésben beolvassa `A`-t, mely igény a pufferkezelőben kiváltja az `INPUT(A)` műveletet, ha `A` még nincs a pufferben. `A` értéke a `READ` utasítás hatására a `T` tranzakció memóriaterületére, a `t` változóba is bemásolódik. A következő lépés megduplázza `t` tartalmát, ennek nincs hatása sem `A` pufferbeli, sem `A` lemezen tárolt értékére. A harmadik lépés írja `t`-t `A` pufferébe, ennek nincs hatása `A` lemezen tárolt értékére.

A következő három lépés ugyanez, csak B-re vonatkozóan. Végül az utolsó két lépésben másolódik A és B lemezre.

Figyeljük meg, hogy ezen lépések összességének végrehajtása alatt az adatbázis konzisztenciája megőrződik. Ha az OUTPUT(A) végrehajtása előtt rendszerhiba fordul elő, akkor ennek nincs hatása a lemezen tárolt adatbázisra, az még olyan, mintha T egyáltalán nem is működött volna, így a konzisztencia megőrződött. Ha rendszerhiba áll elő az OUTPUT(A) végrehajtása után, de még az OUTPUT(B) végrehajtása előtt, akkor az adatbázis inkonzisztens állapotban marad. Azt nem tudjuk biztosítani, hogy ilyen szituáció soha elő ne forduljon, de lépéseket tehetünk azért, hogy amikor mégis bekövetkezik, akkor a problémát elháríthassuk: vagy A és B értékének 8-ra való visszállításával, vagy mindkettő 16-ra növelésével.

Példa. Tegyük fel, hogy az adatbázisra vonatkozó konzisztenciamegszorítás: $0 \leq A \leq B$. Állapítsuk meg, hogy a következő tranzakciók megőrzik-e az adatbázis konzisztenciáját!

a) `A := A + B; B := A + B;`
b) `B := A + B; A := A + B;`
c) `A := B + 1; B := A + 1;`

Semmisségi (undo) naplózás

Az első kérdés, hogy milyen úton biztosítható a tranzakciók atomossága. A *napló* (log) nem más, mint *naplóbejegyzések* (log records) sorozata, melyek mindegyike arról tartalmaz valami információt, hogy mit tett egy tranzakció. A tranzakció tevékenysége nyomon követhető azáltal, hogy a tranzakció működésének hatása lépésenként naplózódik, és ez minden tranzakcióra érvényes.

Ha rendszerhiba fordul elő, akkor a napló segítségével rekonstruálható, hogy a tranzakció mit tett a hiba fellépéséig. A naplót (az archívmentéssel együtt) használhatjuk akkor is, amikor eszközhiba keletkezik a naplót nem tároló lemezen. Általánosságban a katasztrófák hatásának kijavítását követően a tranzakciók hatását meg kell ismételni, és az általuk adatbázisba írt új értékeket ismételten ki kell írni. Egyes tranzakciók hatását viszont vissza kívánjuk vonni, azaz kérjük az adatbázis visszaállítását olyan állapotba, mintha a tekintett tranzakció nem is működött volna.

Az első naplózási stílus, melyet *semmisségi* (undo, visszavonási) *naplózásnak* neveznek, csak az utóbbi típusú helyreállításra alkalmas. Ha nem teljesen biztos, hogy a tranzakció hatásai teljesen befejeződtek és lemezen tárolódtak, akkor minden olyan változtatást, melyet a tranzakció tehetett az adatbázisban, semmissé kell tenni, azaz az adatbázist vissza kell állítani a tranzakció működése előtti állapotába.

Naplóbejegyzések

Úgy kell tekintenünk, hogy a napló mint fájl kizárólag bővítésre van megnyitva. Tranzakció végrehajtásakor a naplókezelő feladata, hogy minden fontos eseményt a naplóban rögzítsen. A napló blokkjai mindenkor naplóbejegyzésekkel vannak feltöltve, mindegyik bejegyzés egy-egy naplózandó eseményre vonatkozik. A naplóblokkokat elsődlegesen a memóriában hozza létre a rendszer, és a pufferkezelő az adatbázis többi blokkjához hasonlóan kezeli őket. A naplóblokkokat, amint csak lehetséges, a nem illékony tárolóra írja a rendszer.

A naplózás minden típusa a naplóbejegyzésnek számos formáját használja. Egyelőre a következőkkel foglalkozunk:

1. <START T>: Ez a bejegyzés jelzi a T tranzakció (végrehajtásának) elkezdődését.

2. <COMMIT T>: A T tranzakció rendben befejeződött, az adatbázis elemein már semmi további módosítást nem kíván végrehajtani. Minthogy azt nem tudjuk felügyelni, hogy a pufferkezelő mikor dönt a memóriablokkok lemezre másolásáról, így általában nem lehetünk biztosak abban, hogy ha meglátjuk a <COMMIT T> naplóbejegyzést, akkor a változtatások a lemezen már megtörténtek. Ha ragaszkodunk ahhoz, hogy a módosítások már a lemezen is megtörténjenek, ezt az igényt a naplókezelőnek kell kikényszerítenie (mint például a semmisségi naplózás esetén).
3. <ABORT T>: A T tranzakció nem tudott sikeresen befejeződni. Ha a T tranzakció abortált (a normálisnál korábban befejeződött), az általa tett változtatásokat nem kell a lemezre másolni. A tranzakciókezelő feladata annak biztosítása, hogy az ilyen változtatások ne jelenjenek meg a lemezen, vagy ha volt valami hatásuk a lemezen, akkor az törlődjön. Az abortált tranzakció hatásainak helyreállításával később foglalkozunk. Az abortálás oka lehet egy hiba a tranzakció kódjában (például 0-val való osztás), melyet a tranzakció kilövésével kezel a rendszer; de az adatbázis-kezelő rendszer is abortálhat egy tranzakciót például holtponti helyzetben (lásd később).
4. <T,X,v>: Ez a *módosítási bejegyzés*. Jelentése: A T tranzakció módosította az X adatbáziselemet, melynek módosítás előtti értéke v volt. A módosítási bejegyzés által leírt változtatás rendesen csak a memóriában történt meg, a lemezen nem; azaz a naplóbejegyzés a WRITE tevékenységre vonatkozik, nem pedig az OUTPUT-ra! A semmisségi naplózás nem rögzíti az adatbáziselem új értékét, csak a módosítás előtti értéket. A semmisségi naplózást alkalmazó rendszerekben a helyreállítás-kezelő feladata a tranzakció lehetséges hatásainak semmissé tétele, amelyhez elegendő csak a régi érték tárolása.

Felmerülhet a kérdés, hogy milyen nagy a módosítást leíró naplóbejegyzés. Ha az adatbáziselemek lemezblokkok, és a módosítást leíró naplóbejegyzés tartalmazza az adatbáziselem régi (módosítás előtti) értékét (vagy mind a régi, mind az új értékét, amint azt az undo/redo naplózásnál látni fogjuk), akkor előfordulhat, hogy a naplóbejegyzés a blokknál nagyobb méretű lesz. Ez nem feltétlen probléma, mert – minden hagyományos fájlhoz hasonlóan – a naplót lemezblokkok sorozatának tekinthetjük, mely bájtsorozatot tartalmaz, a (technikai) blokkhatároktól függetlenül. Ezáltal mód nyílik a napló tömörítésére is. Például bizonyos körülmények között csak a módosításokat kell naplóznunk, azaz csak a tranzakció által módosított sor érintett attribútumainak neveit és azok régi értékeit.

A semmisségi naplózás szabályai

Ahhoz, hogy a rendszerhibák utáni helyreállításra a semmisségi naplózást használhassuk, a tranzakcióknak két előírást kell kielégíteniük:

U_1: Ha a T tranzakció módosítja az X adatbáziselemet, akkor a <T,X,v> típusú naplóbejegyzést *azt megelőzően* kell lemezre írni, hogy X új értékét lemezre írná a rendszer (write-ahead logging; WAL).

U_2: Ha a tranzakció hibamentesen teljesen befejeződött, akkor a COMMIT naplóbejegyzést csak *azt követően* szabad lemezre írni, hogy a tranzakció által módosított összes adatbáziselem már lemezre íródott, ezután viszont a lehető leggyorsabban.

Összefoglalva: az egy tranzakcióhoz tartozó lemezre írásokat a következő sorrendben kell megtenni:

1. az adatbáziselemek módosítására vonatkozó naplóbejegyzések;
2. maguk a módosított adatbáziselemek;
3. a COMMIT naplóbejegyzés.

Az első két lépés az egyes módosításokra vagy módosítások csoportjaira vonatkozóan önmagában, külön-külön is végrehajtható (nem kell a tranzakció összes módosítására csoportosan megtenni).

A naplóbejegyzések lemezre írásának kikényszerítésére a naplókezelőnek szüksége van a FLUSH LOG műveletre, mely felszólítja a pufferkezelőt az összes korábban még ki nem írt naplóblokk lemezre való kiírására. A FLUSH LOG műveletet a tevékenységek közé fogjuk iktatni.

Példa. A semmisségi naplózás fényében vizsgáljuk meg újra a korábbi példában megismert tranzakciót. Kibővítjük a korábbi táblázatot, bemutatván a naplóbejegyzéseket is és a naplókiírási tevékenységet is a T tranzakció végrehajtása során:

Lépés	*Tevékenység*	*t*	*M*–A	*M*–B	*D*–A	*D*–B	*Napló*
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A,8>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B,8>
8)	FLUSH LOG						
9)	OUTPUT(A)		16	16	16	8	
10)	OUTPUT(B)		16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

Az első, ami történik, az a <START T> bejegyzés naplóba írása. Utána jön A beolvasása, majd t módosítása, melynek nincs semmilyen hatása sem a lemezen tárolt adatbázisra, sem annak memóriapufferben található egyetlen részére sem. A 2) és 3) lépés nem igényel naplóbejegyzést, mert nincs hatásuk az adatbázisra.

A 4) lépés A új értékének pufferbe írása. Ezen módosításra vonatkozik a <T,A,8> naplóbejegyzés, mely azt rögzíti, hogy A korábbi értékét (8) T megváltoztatta. Látható, hogy az új érték (16) nincs megemlítve a semmisségi naplózás naplójában.

A következő három lépésben ugyanaz hajtódik végre B-re vonatkozóan, mint korábban A-ra. E ponton a T tranzakció rendben befejeződött, tevékenységét véglegesíteni kell. A és B értékét lemezre kell másolni, betartva a semmisségi naplózás két szabályát, a következő lépéseknek kötött sorrendben kell megtörténnie.

Első, hogy A és B addig nem másolható lemezre, amíg a módosítást leíró naplóbejegyzések lemezre nem kerülnek. Ezt a 8) lépéssel biztosítjuk: a FLUSH LOG hatására az eddigi összes naplóbejegyzés lemezre íródik. E kiírást követően a 9) és 10) lépések A-t és B-t lemezre másolják. Ezeket a lépéseket a pufferkezelő valósítja meg.

Ekkor lehetséges a T tranzakció teljes és sikeres befejezése, ezt jelzendő a 11) lépésben a <COMMIT T> bejegyzés a naplóba íródik. Végül a 12) lépésben ismét ki kell adni a FLUSH LOG utasítást azért, hogy biztosítsuk a <COMMIT T> naplóbejegyzés lemezre való kiírását. Enélkül, bár olyan helyzetben vagyunk, hogy a tranzakció teljesen és hibamentesen befejeződött, ennek a napló későbbi elemzésekor esetleg nem fogjuk nyomát találni. Az ilyen szituációk olyan furcsa viselkedést eredményezhetnek, hogy hiba esetén a felhasználó azt tapasztalja, hogy a tranzakció hibamentesen befejeződött, a lemezre kiírt módosítások mégis semmissé váltak, a tranzakciót ugyanis abortáltnak tekinti a rendszer.

A fentiek alapján azt gondolhatnánk, hogy az egyes tranzakciókhoz tartozó tevékenységek elkülönülten hajtódnak végre. Az adatbázis-kezelő rendszernek viszont számos tranzakció szimultán kezelését kell megoldania. Így egy T tranzakció naplóbejegyzései a naplóban más tranzakciók bejegyzéseivel keveredhetnek. Ha a többi tranzakció valamelyike is a napló lemezre írását kezdeményezi, akkor a T-re vonatkozó naplóbejegyzések esetleg már korábban lemezre kerülnek, mint ahogy azt a tranzakció előírja. Ebből persze nem származik probléma. A <COMMIT T> naplóbejegyzést úgysem fogjuk a T OUTPUT utasításai végrehajtásának befejezésénél korábban kiírni, ezzel biztosítani tudjuk, hogy a módosított adatbázisértékek előbb megjelenjenek a lemezen, mint a COMMIT naplóbejegyzés.

Kényes helyzet áll viszont elő, ha két adatbáziselem közös blokkban található. Ilyenkor az egyik lemezre írása maga után vonja a másik kiírását is. Legrosszabb esetben az egyik adatbáziselem túl korai kiírásával megsértjük az U_1 szabályt. Ez szükségessé tehet további előírásokat a tranzakcióra nézve azért, hogy a semmisségi naplózási módszer használható legyen. Például valamilyen zárolási módszert kell alkalmaznunk annak megelőzésére, hogy két tranzakció egyszerre ugyanazon blokkot használja. Ilyen problémák akkor jelentkeznek, amikor az adatbáziselemek blokkok részei. Emiatt célszerű adatbáziselemnek a blokkot tekinteni.

Helyreállítás a semmisségi naplózás használatával

Tételezzük fel, hogy rendszerhiba történt. Előfordulhat, hogy egy adott tranzakció által végzett adatbázis-módosítások közül egyesek már lemezre íródtak, míg mások (melyeket ugyanez a tranzakció hajtott végre) még nem. Ha így történt, a tranzakció nem atomosan hajtódott végre, így az adatbázis inkonzisztens állapotba kerülhetett. A helyreállítás-kezelő feladata a napló használatával az adatbázist konzisztens állapotba visszaállítani.

Először a legegyszerűbb helyreállítás-kezelő módszerrel foglalkozunk, mely a teljes naplót látja, függetlenül annak méretétől. Később egy sokkal finomabb megközelítést nézünk meg, amikor ellenőrzőpont periodikus készítésével a rendszer korlátozza azt a távolságot, ameddig a helyreállítás-kezlőnek a korábbi történéseket vizsgálnia kell.

A helyreállítás-kezelő első feladata a tranzakciók felosztása sikeresen befejezett és nem befejezett tranzakciókra. Ha található `<COMMIT T>` naplóbejegyzés, akkor az U_2 szabálynak megfelelően a `T` tranzakció által végrehajtott összes adatbázis-módosítás már korábban lemezre íródott. Így a `T` tranzakció önmagában a hiba fellépésekor nem hagyhatta az adatbázist inkonzisztens állapotban.

Ha a naplóban találunk `<START T>` bejegyzést, de nem találunk sem `<COMMIT T>`, sem `<ABORT T>` bejegyzést, akkor a `T` tranzakció végrehajthatott az adatbázisban olyan módosításokat, melyek még a hiba fellépése előtt lemezre íródtak, míg más változtatások még a memóriapufferekben sem történtek meg, vagy ott megtörténtek ugyan, de a lemezre már nem íródtak ki. Ilyen esetben a `T` *nem komplett tranzakció*, és hatását semmissé kell tenni, azaz a `T` által módosított adatbáziselemek értékeit vissza kell állítani a korábbi értékeikre. Az U_1 szabály betartása biztosítja, hogy ha `T` a hiba jelentkezése előtt `X` értékét módosította, akkor még ez előtt a lemezen lévő naplóba kellett kiíródni egy `<T,X,v>` bejegyzésnek. Így a helyreállítás során módunkban áll a `v` értéket az `X` adatbáziselembe visszaírni. Felmerülhet a kérdés, hogy `X` értéke nem `v`-e amúgy is az adatbázisban, de nem is érdemes ezt ellenőrizni.

Minthogy a naplóban számos rendesen befejezett és teljesen be nem fejezett tranzakció nyomát találhatjuk, és ezek közül több tranzakció módosíthatta az `X` adatbáziselemet is, nagyon ügyelnünk kell arra, hogy milyen sorrendben állítjuk vissza `X` korábbi tartalmát. Ezért a helyreállítás-kezelő a naplót a végéről kezdi átvizsgálni. Amint halad a napló átvizsgálásával, megjegyzi mindazon `T` tranzakciókat, melyekre vonatkozóan a naplóban `<COMMIT T>` vagy `<ABORT T>` bejegyzést talált. Ahogy halad visszafelé, és elér egy `<T,X,v>` bejegyzésig, akkor a következő lehetőségek vannak:

- ha ugyanerre a `T` tranzakcióra vonatkozó `COMMIT` bejegyzéssel már találkozott, akkor nincs teendője, hiszen `T` rendesen és teljesen befejeződött, hatásait tehát nem kell semmissé tenni;
- ha `ABORT` bejegyzéssel találkozott a `T` tranzakcióra vonatkozóan, akkor sincs teendője, ebben az esetben ugyanis `T`-t egyszer már helyreállítottuk;
- minden más esetben `T` nem komplett tranzakció, ekkor a helyreállítás-kezelő `X` értékét `v`-re cseréli.

Miután a helyreállítás-kezelő végrehajtotta a fenti változtatásokat, minden nem komplett T tranzakcióra vonatkozóan <ABORT T> bejegyzést ír a naplóba, és kiváltja annak naplófájlba való kiírását is (FLUSH LOG). Ekkor folytatódhat az adatbázis normál használata, új tranzakciók indulhatnak.

Példa. Tekintsük át, hogy mi történik, ha a fenti példában különböző időpontokban rendszerhiba következik be:

1. A hiba a 12) lépést követően jelentkezett. Tudjuk, hogy ekkor a <COMMIT T> bejegyzést már lemezre írta a rendszer. A hiba kezelése során a T tranzakció hatásait nem kell visszállítani, a T-re vonatkozó összes naplóbejegyzést a helyreállítás-kezelő figyelmen kívül hagyhatja.
2. A hiba a 11) és 12) lépések között jelentkezett. Ekkor előfordulhat, hogy a COMMIT bejegyzést tartalmazó naplóbejegyzés már lemezre íródott, például ha a naplóbejegyzés kiírását egy másik tranzakció már kérte a pufferkezelőtől. Ha így történt, akkor T-re vonatkozólag a hiba kezelése az 1. esethez hasonló. Ha azonban a COMMIT bejegyzés a lemezen nem található, akkor a helyreállítás-kezelő a T tranzakciót befejezetlennek tekinti. Ahogy olvassa a naplót visszafelé, először a <T,B,8> bejegyzést fogja megtalálni (a T tranzakcióra vonatkozólag). Ennek megfelelően a lemezen B tartalmába 8-at ír vissza. Majd a <T,A,8> naplóbejegyzés miatt A tartalmába is 8 kerül. Végezetül <ABORT T> bejegyzést ír a naplóba és a lemezre.
3. Ha a hiba a 10) és 11) lépések között lépett fel, akkor a COMMIT bejegyzés még biztosan nem történt meg, tehát T befejezetlen, hatásainak semmissé tétele a 2. esetnek megfelelően történik.
4. A 8) és 10) lépések között bekövetkező hiba fellépésekor az előző esethez hasonlóan T hatásait semmissé kell tenni. Az egyetlen különbség, hogy az A és/vagy B módosítása még nem jelent meg a lemezen. Ettől függetlenül mindkét adatbáziselem korábbi értékét (8) állítja vissza a rendszer.
5. Amennyiben a hiba a 8) lépésnél korábban jelentkezik, akkor még az sem biztos, hogy a T tranzakcióra vonatkozó naplóbejegyzések közül bármi is lemezre került. Az U_1 szabály miatt azonban tudjuk, hogy mielőtt az A és/vagy B adatbáziselemek a lemezen módosulnának, a megfelelő módosítási naplóbejegyzésnek a naplóban meg kell jelennie. Így ha T módosította is a lemezen A és/vagy B értékét, a megfelelő naplóbejegyzés hatására a helyreállítás-kezelő semmissé teszi ezeket a módosításokat.

Tegyük fel, hogy egy korábbi hiba utáni helyreállítás közben ismét rendszerhiba lép fel. A semmisségi (és a másik kettő) naplózás oly módon van megtervezve, hogy a korábbi érték változtatás előtti tárolása következtében a helyreállító lépések idempotensek, ami azt jelenti, hogy a helyreállító tevékenység többszöri végrehajtása pontosan ugyanolyan hatású, mint egyszeri végrehajtása. Ha találunk egy <T,X,v> naplóbejegyzést, akkor nem számít, hogy X értéke már v, X értékét (esetleg ismételten) v-re állíthatjuk. Hasonlóan semmi problémát nem okoz, ha a helyreállítási folyamat egészét (vagy félbemaradt részét) többször megismételjük.

Az ellenőrzőpont-képzés

Mint láttuk, a helyreállítás elvben a teljes napló átvizsgálását igényli. Semmisségi naplózás esetén ha egy tranzakció a COMMIT naplóbejegyzést már kiírta a naplóba, akkor az ezen tranzakcióra vonatkozó naplóbejegyzésekre a helyreállítás során nincs már szükség (hacsak nem kívánjuk később elemezni a tranzakciókat). Gondolhatnánk arra, hogy a tranzakcióra vonatkozó, a COMMIT-ot megelőző naplóbejegyzéseket törölhetnénk a naplóból, de ezt nem mindig tehetjük meg. Ennek oka az, hogy gyakran sok tranzakció működik egyszerre, és ha a naplót egy tranzakció befejezése után csonkítanánk, esetleg elveszítenénk más, még aktív tranzakciókra vonatkozó bejegyzéseket, így nem tudnánk a naplót a helyreállításra használni.

E probléma megoldására a legegyszerűbb mód, ha a naplóra vonatkozóan ismétlődően *ellenőrzőpontokat* (checkpoint) képezünk. Kétféle ellenőrzőpont-képzés létezik: egyszerű és a rendszer működése közbeni. Az egyszerű ellenőrzőpont-képzés a következőképpen történik:

1. Új tranzakcióindítási kérések kiszolgálásának letiltása.
2. A még aktív tranzakciók helyes és teljes befejezésének vagy abortálásának és a `COMMIT` vagy `ABORT` bejegyzés naplóba írásának kivárása.
3. A napló lemezre való kiírása (`FLUSH LOG`).
4. `<CKPT>` naplóbejegyzés képzése és kiírása a naplóba, majd újra `FLUSH LOG`.
5. Tranzakcióindítási kérések kiszolgálása.

Az ellenőrzőpont kiírását megelőzően végrehajtott tranzakciók mind befejeződtek, és az U_2 szabálynak megfelelően módosításaik lemezre kerültek. Ennek megfelelően ezen tranzakciók tevékenységére nézve egy esetleges későbbi hiba elhárításakor már nem igényel a rendszer helyreállítást. A helyreállítás során a naplót a végétől visszafelé csak a `<CKPT>` bejegyzésig kell elemezni azért, hogy a nem befejezett tranzakciókat azonosítsuk. Amikor a `<CKPT>` bejegyzést megtaláljuk, ebből tudjuk, hogy már láttuk az összes befejezetlen tranzakciót. Mivel az ellenőrzőpont-képzés alatt újabb tranzakció nem indulhatott, látnunk kellett a befejezetlen tranzakciókhoz tartozó összes naplóbejegyzést. Ezért nem szükséges a `<CKPT>` bejegyzésnél korábbi naplórészt elemeznünk, és – hacsak más okból nincs szükségünk rá – biztonsággal törölhetjük vagy felülírhatjuk.

Példa. Tekintsük az alábbi naplórészletet:

```
<START T1>
<T1,A,5>
<START T2>
<T2,B,10>
<T2,C,15>
<T1,D,20>
<COMMIT T1>
<COMMIT T2>
<CKPT>
<START T3>
<T3,E,25>
<T3,F,30>
```

Tegyük fel, hogy a 4. bejegyzés után úgy döntünk, hogy ellenőrzőpontot hozunk létre. Minthogy T_1 és T_2 aktív tranzakciók, meg kell várnunk befejeződésüket, mielőtt a `<CKPT>` bejegyzést a naplóba írnánk. Tegyük fel, hogy a naplórészlet végén rendszerhiba lép fel. A naplót a végétől visszafelé elemezve T_3-at fogjuk az egyetlen be nem fejezett tranzakciónak találni, így `E` és `F` korábbi értékeit kell csak visszaállítanunk. Amikor megtaláljuk a `<CKPT>` bejegyzést, tudjuk, hogy nem kell a korábbi naplóbejegyzéseket elemeznünk, végeztünk az adatbázis állapotának helyrehozásával.

Felmerülhet a kérdés, hogy hogyan találjuk meg az utolsó naplóbejegyzést. A napló lényegében egy fájl, melynek blokkjai tartalmazzák a naplóbejegyzéseket. A blokk még ki nem töltött részeit üresként jelöljük. Ha a bejegyzéseket soha nem írjuk felül, akkor a helyreállítás-kezelő az utolsó bejegyzést úgy keresi meg, hogy megkeresi az első üres bejegyzést, és az ezt megelőző bejegyzés a fájl utolsó érvényes bejegyzése. Ha viszont a régi naplóbejegyzéseket felülírjuk, akkor a naplóbejegyzéseket egyedi, növekvő sorszámmal kell ellátnunk:

~~1~~ 9	~~2~~ 10	~~3~~ 11	4	5	6	7	8

Ekkor azt a bejegyzést kell megtalálnunk, melynek nagyobb a sorszáma, mint a következőé: ez a bejegyzés a napló pillanatnyi vége. A gyakorlatban a nagyméretű napló több fájl egyesítése is lehet. Logikailag ekkor is egy fájlnak tekintjük, és a végét a megfelelő részfájlban keressük.

Ellenőrzőpont-képzés a rendszer működése közben

Az egyszerű ellenőrzőpont-képzési technika problémája, hogy gyakorlatilag le kell állítani a rendszer működését az ellenőrzőpont elkészültéig. Minthogy az aktív tranzakciók még hosszabb időt igényelhetnek a normális vagy abnormális befejeződésükig, a felhasználó számára a rendszer leállítottnak tűnhet. Egy jóval bonyolultabb módszerrel, a *működés közbeni ellenőrzőpont-képzéssel* (nonquiescent checkpointing) elérjük, hogy az ellenőrzőpont-képzés alatt új tranzakciók indulását ne kelljen szüneteltetni. E módszer lépései:

1. <START CKPT(T_1,…,T_k)> naplóbejegyzés készítése és lemezre írása (FLUSH LOG). T_1,…,T_k az éppen aktív tranzakciók nevei.
2. Meg kell várni a T_1,…,T_k tranzakciók mindegyikének normális vagy abnormális befejeződését, nem tiltva közben újabb tranzakciók indítását.
3. Ha a T_1,…,T_k tranzakciók mindegyike befejeződött, akkor <END CKPT> naplóbejegyzés elkészítése és lemezre írása (FLUSH LOG).

Az ilyen típusú napló felhasználásával a következőképpen tudunk rendszerhiba után helyreállítani: a naplót a végétől visszafelé elemezve megtaláljuk az összes nem befejezett tranzakciót, régi értékére visszaállítjuk az ezen tranzakciók által megváltoztatott adatbáziselemek tartalmát. Két eset fordulhat elő aszerint, hogy visszafelé olvasván a naplót az <END CKPT> vagy a <START CKPT(T_1,…,T_k)> naplóbejegyzést találjuk előbb:

- Ha előbb az <END CKPT> naplóbejegyzéssel találkozunk, akkor tudjuk, hogy az összes még be nem fejezett tranzakcióra vonatkozó naplóbejegyzést a legközelebbi korábbi <START CKPT(T_1,…,T_k)> naplóbejegyzésig megtaláljuk. Ott viszont megállhatunk, az annál korábbiakat akár el is dobhatjuk.
- Amennyiben a <START CKPT(T_1,…,T_k)> naplóbejegyzéssel találkozunk előbb, az azt jelenti, hogy a katasztrófa az ellenőrzőpont-képzés közben fordult elő. Ennek következtében a T_1,…,T_k tranzakciók nem fejeződtek be (legalábbis nem tudtuk a befejeződést regisztrálni) a hiba fellépéséig. Ekkor a be nem fejezett tranzakciók közül a legkorábban kezdődött tranzakció indulásáig kell a naplóban visszakeresnünk, annál korábbra nem. Az ezt megelőző olyan START CKPT bejegyzés, amelyhez tartozik END CKPT, biztosan megelőzi a keresett összes tranzakció indítását leíró bejegyzéseket. Ha a START CKPT előtt olyan START CKPT bejegyzést találunk, amelyhez nem tartozik END CKPT, akkor ez azt jelenti, hogy korábban is ellenőrzőpont-képzés közben történt rendszerhiba. Az ilyen „ellenőrzőpont-kezdeményeket” figyelmen kívül kell hagyni. Ezenfelül, ha az ugyanazon tranzakcióra vonatkozó naplóbejegyzéseket összeláncoljuk, akkor nem kell a napló minden bejegyzését átnéznünk ahhoz, hogy megtaláljuk a még be nem fejezett tranzakciókra vonatkozó bejegyzéseket, elegendő csak az adott tranzakció bejegyzéseinek láncán visszafelé haladnunk.

Általános szabályként elmondható, hogy ha egy <END CKPT> naplóbejegyzést kiírunk lemezre, akkor az azt megelőző START CKPT bejegyzésnél korábbi naplóbejegyzéseket törölhetjük.

Példa. Tekintsük az alábbi naplórészletet:

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<START CKPT(T_1,T_2)>
<T_2,C,15>
<START T_3>
<T_1,D,20>
<COMMIT T_1>
<T_3,E,25>

<COMMIT T_2>
<END CKPT>
<T_3,F,30>

Most úgy döntünk, hogy működés közbeni ellenőrzőpontot hozunk létre a 4. bejegyzés után. Minthogy e pillanatban T_1 és T_2 aktív tranzakciók, ezért kell az 5. bejegyzést felírnunk. Tegyük fel, hogy amíg T_1 és T_2 befejeződésére várunk, azalatt egy harmadik tranzakció (T_3) is elkezdődik.

Tegyük fel, hogy a naplórészlet végén rendszerhiba lép fel. A naplót a végétől visszafelé vizsgálva úgy fogjuk találni, hogy T_3 egy be nem fejezett tranzakció, ezért hatásait semmissé kell tenni. Az utolsó naplóbejegyzés arról informál bennünket, hogy az F adatbáziselembe a 30 értéket kell visszaállítani. Amikor az <END CKPT> naplóbejegyzést találjuk, tudjuk, hogy az összes be nem fejezett tranzakció a megelőző START CKPT naplóbejegyzés után indulhatott csak el. Tovább haladva visszafelé, megtaláljuk a <T_3,E,25> bejegyzést, mely megmondja nekünk, hogy az E adatbáziselem értékét 25-re kell visszaállítani. Ezen bejegyzés és a START CKPT naplóbejegyzés között további elindult, de be nem fejeződött tranzakcióra vonatkozó bejegyzést nem találunk, így az adatbázison mást már nem kell megváltoztatnunk.

Tegyük fel most, hogy az ellenőrzőpont-képzés közben történt katasztrófa, a <T_3,E,25> bejegyzés után. Visszafelé elemezve a naplót, azonosítjuk a T_3, majd a T_2 tranzakciókat, melyek nincsenek befejezve, tehát helyreállító módosításokat kell tennünk. Amikor megtaláljuk a <START CKPT(T_1,T_2)> naplóbejegyzést, megtudjuk, hogy az egyetlen további olyan tranzakció, mely lehetséges, hogy nincs befejezve, a T_1. Minthogy azonban a <COMMIT T_1> bejegyzést már láttuk, ebből tudjuk, hogy T_1 nem be nem fejezett tranzakció. Láttuk már továbbá a <START T_3> bejegyzést is, így már tudjuk, hogy csak addig kell folytatnunk a napló elemzését, amíg a T_2 START bejegyzését meg nem találjuk. Eközben még a B adatbáziselem értékét is visszaállítjuk 10-re.

Helyrehozó (redo) naplózás

A semmisségi naplózás természetes és egyszerű stratégiát valósít meg a napló kezelésére és rendszerhibák esetén a visszaállításra, de a probléma megoldásának nem ez az egyetlen lehetséges megközelítése. A semmisségi naplózás potenciális problémája az, hogy csak azután tudjuk befejezni a tranzakciót, hogy az összes adatbázis-módosítása lemezre íródott. Olykor a lemezműveletekkel tudnánk takarékoskodni, ha megengednénk, hogy az adatbázis-módosításokat csak a memóriában végezzék a tranzakciók, miközben a napló az eseményeket rögzíti, azért, hogy katasztrófa esetén is biztonságban legyen az adatbázis.

Az adatbáziselemek lemezre való azonnali visszaírásának kényszerét elkerülhetjük, ha a *helyrehozó naplózás* (redo logging) módszerét választjuk. Az alapvető különbségek a semmisségi és a helyrehozó naplózás között az alábbiak:

- Amíg a semmisségi naplózás a helyreállítás során a be nem fejezett tranzakciók hatásait semmissé teszi, a befejezett tranzakciók hatásait pedig nem módosítja, addig a helyrehozó naplózás figyelmen kívül hagyja a be nem fejezett tranzakciókat, és megismétli a normálisan befejezettek által végrehajtott változtatásokat.
- A semmisségi naplózás megkívánja az adatbáziselemek lemezen való módosítását a COMMIT naplóbejegyzés lemezre írása előtt, a helyrehozó naplózás viszont a COMMIT naplóbejegyzés lemezre írását várja el, mielőtt bármit is változtatna a lemezen lévő adatbázisban.
- A semmisségi naplózás U_1 és U_2 szabályainak betartása mellett csak a módosított adatbáziselemek régi tartalmát kell megőriznünk az esetleges visszaállítás biztosításához, a helyrehozó naplózással történő helyreállításhoz a módosított elemek új értékére van szükség. Emiatt a helyrehozó naplózás

naplóbejegyzései ugyanolyan formájúak, de más a jelentésük, mint a semmisségi naplózásnál alkalmazottaké.

A helyrehozó naplózás szabályai

A helyrehozó naplózás az adatbáziselemek módosítását a naplóbejegyzésben az új értékkel képviseli (nem pedig a régivel, mint a semmisségi naplózásnál). Ez a bejegyzés ugyanúgy néz ki, mint a semmisségi naplózásnál használt: <T,X,v>, a jelentése azonban más: a T tranzakció az X adatbáziselemnek a v értéket adta. E bejegyzésben az X régi értékét nem jelzi semmi. Ha egy T tranzakció módosítja egy X adatbáziselem értékét, akkor egy <T,X,v> bejegyzést kell a naplóba írni.

Az adatbáziselemek és a naplóbejegyzések lemezre kerülésének sorrendjét az alábbi egyszerű szabály írja le:

R_1: Mielőtt az adatbázis bármely X elemét a lemezen módosítanánk, szükséges, hogy az X ezen módosítására vonatkozó összes naplóbejegyzés, azaz <T,X,v> és <COMMIT T>, lemezre kerüljön.

Minthogy a COMMIT bejegyzést csak akkor írhatjuk a naplóba, ha a tranzakció teljesen és hibamentesen befejeződött, így az csak a módosításokat leíró bejegyzések után állhat, ezért úgy is összegezhetjük az R_1 szabályt, hogy ha helyrehozó naplózást használunk, akkor az egy tranzakcióra vonatkozó lemezre írásoknak a következő sorrendben kell megtörténniük:

1. az adatbáziselemek módosítását leíró naplóbejegyzések lemezre írása;
2. a COMMIT naplóbejegyzés lemezre írása;
3. az adatbáziselemek értékének tényleges cseréje a lemezen.

Példa. Nézzük meg a korábban megismert tranzakciót helyrehozó naplózás használatával:

Lépés	*Tevékenység*	*t*	*M*–A	*M*–B	*D*–A	*D*–B	*Napló*
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A,16>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B,16>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)		16	16	16	8	
11)	OUTPUT(B)		16	16	16	16	

A főbb különbségek a semmisségi és a helyrehozó naplózás használata között tehát a következők: A módosítási bejegyzésekben A és B új értéke szerepel, nem a régi. A COMMIT bejegyzés korábbra került, a 8) lépésbe. Ezt követően a napló lemezre írását kiváltó FLUSH LOG következik, így a T tranzakció által végrehajtott módosításokat leíró összes naplóbejegyzés lemezre íródik. Csak ezt követően kerül lemezre A és B új értéke. Az ábrán ezen új értékek kiírását a közvetlenül következő 10) és 11) lépésekben láthatjuk, bár a gyakorlatban ezekre esetleg csak később kerül sor.

Helyreállítás a helyrehozó naplózás használatával

A helyrehozó naplózás R_1 szabályának fontos következménye, hogy ha a naplóban nincs <COMMIT T> bejegyzés, akkor tudjuk, hogy a T tranzakció nem hajtott végre az adatbázisban módosítást a lemezen. Így a be nem fejezett (nem teljes) tranzakciók a helyreállítás során úgy tekinthetők, mintha meg sem történtek volna. Problémát a befejezett tranzakciók jelenthetnek, mert nem tudjuk, hogy az általuk elvégzett

adatbázis-változtatások közül melyek íródtak már lemezre. Szerencsére a helyrehozó naplózás naplója éppen azon információkat (az új értékeket) tartalmazza, melyekre szükségünk van a helyreállításhoz. Ezen új értékeket kell lemezre írnunk, attól függetlenül, hogy esetleg már korábban is kiíródtak. A rendszerkatasztrófa bekövetkezése után a helyrehozó naplózással történő helyreállításhoz a következőket kell tennünk:

1. Meghatározzuk a befejezett tranzakciókat (`COMMIT`).
2. Elemezzük a naplót az elejétől kezdve. Minden `<T,X,v>` naplóbejegyzés esetén:
   a) Ha `T` nem befejezett tranzakció, akkor nem kell tenni semmit.
   b) Ha `T` befejezett tranzakció, akkor `v` értéket kell írni az `X` adatbáziselembe.
3. Minden `T` be nem fejezett tranzakcióra vonatkozóan `<ABORT T>` naplóbejegyzést kell a naplóba írni, és a naplót ki kell írni lemezre (`FLUSH LOG`).

Példa. Nézzük meg, hogyan lehet a fenti példában a helyreállítást elvégezni a különböző pillanatokban bekövetkező katasztrófák esetén:

1. Ha a katasztrófa a 9) lépés után bármikor következik be, akkor a `<COMMIT T>` bejegyzés már lemezen van. A helyreállító rendszer `T`-t befejezett tranzakcióként azonosítja. Amikor a naplót az elejétől kezdve elemzi, a `<T,A,16>` és a `<T,B,16>` bejegyzések hatására a helyreállítás-kezelő az `A` és `B` adatbázis-elemekbe a 16 értéket írja. Ha a katasztrófa a 10) és 11) lépések között következett be, akkor `A` újraírása redundáns ugyan, de `B` írása lényeges lépés az adatbázis konzisztens állapotának eléréséhez. Amennyiben a hiba a 11) lépést követően keletkezett, akkor mindkét adatbáziselem új értékének lemezre írása redundáns, de semmi gondot nem okoz.
2. Ha a hiba a 8) és 9) lépések között jelentkezik, akkor bár a `<COMMIT T>` bejegyzés már a naplóba került, de nem biztos, hogy lemezre íródott. Ha lemezre került, akkor a helyreállítási eljárás az 1. esetnek megfelelően történik, ha nem, akkor pedig a 3. esetnek megfelelően.
3. Ha a katasztrófa a 8) lépést megelőzően keletkezik, akkor `<COMMIT T>` naplóbejegyzés még biztosan nem került lemezre, így `T` be nem fejezett tranzakciónak tekintendő. Ennek megfelelően `A` és `B` értékeit a lemezen még nem változtatta meg a `T` tranzakció, tehát nincs mit helyreállítani. Végül egy `<ABORT T>` bejegyzést írunk a naplóba.

Mivel sok befejezett tranzakció is adhatott új értéket ugyanazon `X` adatbáziselemnek, ezért a helyrehozó naplózás alkalmazásakor a naplót a korábbi bejegyzésektől a későbbiek felé (időrendben) haladva kell elemeznünk. Így érhető el, hogy `X` adatbázisbeli végső értéke a normálisan befejeződött tranzakciók által utoljára adott legyen. Ugyanazt az állapotot érjük el tehát, mint ami a semmisségi naplózásnál a napló visszafelé elemzésével volt elérhető.

Helyrehozó naplózás ellenőrzőpont-képzés használatával

A semmisségi naplózásnál látottakhoz hasonlóan a helyrehozó naplózás naplójába is illeszthetünk ellenőrzőpontokat. A helyrehozó naplózásnál azonban új probléma jelentkezik: minthogy a befejeződött tranzakciók módosításainak lemezre írása a befejeződés után sokkal később is történhet, így az e vonatkozásban ugyanazon pillanatban aktív tranzakciók számát nincs értelme korlátozni, tehát nincs értelme az egyszerű ellenőrzőpont-képzésnek. Tekintet nélkül arra, hogy az ellenőrzőpont-képzés alatt tranzakciók indulását megengedjük vagy sem, a kulcsfeladat – amit meg kell tennünk az ellenőrzőpont-készítés kezdete és befejezése közötti időben – az összes olyan adatbáziselem lemezre való kiírása, melyeket befejezett tranzakciók módosítottak, és még nem voltak lemezre kiírva. Ennek megvalósításához a pufferkezelőnek nyilván kell tartania a *piszkos puffereket* (dirty buffers), melyekben már végrehajtott, de lemezre még ki nem írt módosításokat tárol. Azt is tudnunk kell, hogy mely tranzakciók mely puffereket módosították.

Másrészről viszont be tudjuk fejezni az ellenőrzőpont-képzést az aktív tranzakciók (normális vagy abnormális) befejezésének kivárása nélkül, mert ők ekkor még amúgy sem engedélyezik lapjaik lemezre írását. A helyrehozó naplózásban a működés közbeni ellenőrzőpont-képzés a következőkből áll:

1. <START CKPT(T_1,…,T_k)> naplóbejegyzés elkészítése és lemezre írása, ahol T_1,…,T_k az összes éppen aktív tranzakció.
2. Az összes olyan adatbáziselem kiírása lemezre, melyeket olyan tranzakciók írtak pufferekbe, melyek a START CKPT naplóba írásakor már befejeződtek, de puffereik lemezre még nem kerültek.
3. <END CKPT> bejegyzés naplóba írása, és a napló lemezre írása.

Példa. Tekintsük az alábbi naplórészletet:

<START T_1>
<T_1,A,5>
<START T_2>
<COMMIT T_1>
<T_2,B,10>
<START CKPT(T_2)>
<T_2,C,15>
<START T_3>
<T_3,D,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

Amikor az ellenőrzőpont-képzés elkezdődött, csak T_2 volt aktív, de a T_1 által A-ba írt érték még nem biztos, hogy lemezre került. Ha még nem, akkor A-t lemezre kell másolnunk, mielőtt az ellenőrzőpont-képzést befejezhetnénk. A napló érzékelteti, hogy az ellenőrzőpont-képzés befejezéséig más események is bekövetkezhetnek: T_2 a C adatbáziselem tartalmát módosítja, elindul egy új tranzakció (T_3), és módosítja D értékét. Az ellenőrzőpont-képzés befejezése után már csak a T_2 és T_3 tranzakciók befejeződése történt meg.

Visszaállítás az ellenőrzőpont-képzéssel kiegészített helyrehozó típusú naplózással

Mint a semmisségi naplózásnál, most is az ellenőrzőpontok naplóba illesztése segít a napló átvizsgálásának korlátozásában. Most is két eset fordulhat elő:

- Tegyük fel, hogy a katasztrófa előtt a naplóba feljegyzett utolsó ellenőrzőpont-bejegyzés <END CKPT>. Ekkor tudjuk, hogy az olyan értékek, melyeket olyan tranzakciók írtak, melyek a <START CKPT(T_1,…,T_k)> naplóbejegyzés megtétele előtt befejeződtek, már biztosan lemezre kerültek, így nem kell velük foglalkoznunk. Foglalkoznunk kell viszont a T_i-kkel és az ellenőrzőpont kialakításának megkezdése után induló tranzakciókkal: ezeknek lehetnek olyan adatbázis-módosításaik, melyek még nem kerültek lemezre, pedig a tranzakció már befejeződött. Ekkor olyan visszaállítást kell tennünk, amilyenről már szó volt, azzal a különbséggel, hogy figyelmünket azon tranzakciókra korlátozhatjuk, melyek az utolsó <START CKPT(T_1,…,T_k)> naplóbejegyzésben a T_i-k között szerepelnek, vagy ezen naplóbejegyzést követően indultak el. A naplóban való keresés során a legkorábbi <START T_i> naplóbejegyzésig kell visszamennünk, annál korábbra nem. Ezek a START naplóbejegyzések akárhány korábbi ellenőrzőpontnál előbb is felbukkanhatnak. Ahogy a semmisségi naplózásnál, az adott tranzakciókra vonatkozó naplóbejegyzések visszafelé keresése segít megtalálni a számunkra éppen fontos bejegyzéseket.
- Tegyük fel, hogy a naplóba feljegyzett utolsó ellenőrzőpont-bejegyzés a <START CKPT(T_1,…,T_k)>. Nem lehetünk abban biztosak, hogy az ezt megelőzően befejezett tranzakciók által módosított

adatbáziselemek már lemezre íródtak. Ezért az előző <END CKPT> bejegyzéshez tartozó <START CKPT(S_1,…,S_m)> bejegyzésig vissza kell keresnünk, és helyre kell állítanunk az olyan befejeződött tranzakciók tevékenységének eredményeit, melyek ez utóbbi <START CKPT(S_1,…,S_m)> bejegyzés után indultak, vagy az S_i-k közül valók.

Példa. Tekintsük ismét az előbbi naplórészletet. Ha a katasztrófa a végén lép fel, akkor visszafelé keresve megtaláljuk az <END CKPT> bejegyzést. Ekkor tudjuk, hogy a helyreállítás szempontjából elegendő csak azon tranzakciókat figyelembe venni, melyek vagy a <START CKPT(T_2)> bejegyzés felírását követően indultak, vagy szerepelnek e bejegyzés listájában (most csak T_2). Így a vizsgálandó tranzakcióhalmazunk a (T_2, T_3). <COMMIT T_2> és <COMMIT T_3> bejegyzéseket találunk, ebből tudjuk, hogy mindkét tranzakció hatását helyre kell állítanunk. A naplóban visszafelé meg kell keresnünk a <START T_2> bejegyzést, és innen már időrendben haladva a naplóban a következő módosítási bejegyzéseket találjuk a T_2 és T_3 befejezett tranzakciókra vonatkozóan: <T_2,B,10>, <T_2,C,15> és <T_3,D,20>. Mivel azt nem tudjuk, hogy ezen változtatások a lemezen már megtörténtek-e, ezért most a lemezre újraírjuk a B, a C és a D tartalmát, megfelelően 10, 15 és 20 értékeket adva nekik.

Tegyük fel, hogy a katasztrófa a <COMMIT T_2> és a <COMMIT T_3> bejegyzések között történt. A helyreállítás az előbbi esethez hasonló, azzal a különbséggel, hogy T_3 nem befejezett tranzakció, ennek megfelelően a <T_3,D,20> helyreállítást nem kell végrehajtani. D értékét tehát a helyreállítás során nem változtatjuk meg, hacsak a vizsgált naplórészben található, másik tranzakcióra vonatkozó bejegyzés miatt meg nem kell változtatnunk. A helyreállítást követően egy <ABORT T_3> bejegyzést írunk a naplóba.

Ha a hiba az <END CKPT> bejegyzést megelőzően lépett fel, akkor az utolsó előtti START CKPT bejegyzést kell megkeresnünk (melynek már van <END CKPT> párja), és annak listájából tudjuk meg, melyek az aktív tranzakciók. Ha nem találunk korábbi ellenőrzőpont-bejegyzést, akkor mindenképpen a napló elejére kell mennünk. Így esetünkben az egyedüli befejezett tranzakciónak T_1-et fogjuk találni, ezért a <T_1,A,5> tevékenységet helyreállítjuk. A helyreállítást követően <ABORT T_2> és <ABORT T_3> bejegyzést írunk a naplóba.

Minthogy a tranzakciók több ellenőrzőpont készítésekor is aktívak lehetnek, célszerű lehet, hogy a <START CKPT(T_1,…,T_k)> naplóbejegyzésbe nemcsak az aktív tranzakciók neveit, hanem olyan mutatókat is elhelyezzünk, melyek az aktív tranzakciók indulását leíró bejegyzések naplóbeli helyét adják meg. Így eljárva biztonsággal meg tudjuk állapítani, hogy a napló mely korábbi részeit törölhetjük. Amikor <END CKPT> bejegyzést írunk a naplóba, akkor tudjuk, hogy a naplóban már sosem kell korábbra visszatekintenünk, mint ahol a T_i aktív tranzakcióra vonatkozó legkorábbi <START T_i> bejegyzést találjuk. Következésképpen az ezen START bejegyzést megelőző bejegyzések törölhetők.

Semmisségi/helyrehozó (undo/redo) naplózás

Láthattuk, hogy a naplózás két különböző megközelítése abban mutat eltérést, hogy a napló az adatbáziselemek értékének módosítása esetén a régi vagy az új értéket tartalmazza. Mindkét módszernek vannak bizonyos hátrányai is:

- A semmisségi naplózás alkalmazása megköveteli, hogy az adatokat a tranzakció befejezésekor nyomban lemezre írjuk, ezzel (esetleg jelentősen) növeljük a végrehajtandó lemezműveletek számát.
- A helyrehozó naplózás minden módosított adatbázisblokk pufferben tartását igényli egészen a tranzakció rendes és teljes befejezéséig, így a napló kezelésével együtt (esetleg jelentősen) növeljük a tranzakciók átlagos pufferigényét.
- Mindkét naplózási módszer az ellenőrzőpont képzése közben ellentétes igényeket támaszt a pufferek lemezre írása szempontjából, kivéve, ha az adatbáziselemek teljes blokkok vagy blokkok sokasága.

Például ha a puffer tartalmaz egy A adatbáziselemet, melyet egy rendesen és teljesen befejezett tranzakció módosított, és tartalmaz egy B adatbáziselemet is, melyet olyan tranzakció módosított, melyre vonatkozóan a COMMIT bejegyzés még nem került lemezre, akkor – az R_1 szabálynak megfelelően – a puffer lemezre másolását igényeljük A miatt, viszont tiltjuk ennek megtételét B miatt.

A *semmisségi/helyrehozó* (undo/redo) *naplózás* a tevékenységek elvégzési sorrendjének rugalmasságát növeli azáltal, hogy bővíti a naplózott információk körét.

A semmisségi/helyrehozó naplózás szabályai

A semmisségi/helyrehozó naplózás – egyetlen különbséggel – ugyanolyan típusú naplóbejegyzéseket használ, mint a naplózás másik két módszere. E módszerben az adatbáziselem értékének módosítását leíró naplóbejegyzés négykomponensű: a <T,X,v,w> naplóbejegyzés azt jelenti, hogy a T tranzakció az adatbázis X elemének korábbi v értékét w-re módosította. A semmisségi/helyrehozó naplózást alkalmazó rendszernek a következő előírást kell betartania:

UR_1: Mielőtt az adatbázis bármely X elemének értékét – valamely T tranzakció által végzett módosítás miatt – a lemezen módosítanánk, ezt megelőzően a <T,X,v,w> módosítást leíró naplóbejegyzésnek lemezre kell kerülnie.

A semmisségi/helyrehozó naplózás UR_1 szabálya csak azokat a feltételeket kényszeríti, amelyek a semmisségi és a helyrehozó naplózási szabályok mindegyikében szerepelnek. Ezáltal a <COMMIT T> bejegyzés megelőzheti, de követheti is az adatbáziselemek lemezen történő bármilyen megváltoztatását.

Példa. Nézzük meg a szokásos példánkat semmisségi/helyrehozó naplózás használatával:

Lépés	*Tevékenység*	*t*	*M*–A	*M*–B	*D*–A	*D*–B	*Napló*
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A,8,16>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B,8,16>
8)	FLUSH LOG						
9)	OUTPUT(A)		16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)		16	16	16	16	

A módosítást leíró naplóbejegyzések az A és B adatbáziselemeknek mind a régi, mind az új értékét tartalmazzák. Ebben a sorozatban a <COMMIT T> naplóbejegyzés kiírását az A és B adatbáziselemek lemezre való írása közé tettük. A 10) lépés kerülhetett volna a 8) vagy 9) lépés elé, vagy a 11) lépés mögé is.

Helyreállítás a semmisségi/helyrehozó naplózás használatával

Amikor a semmisségi/helyrehozó naplózást használjuk, és helyreállításra kényszerülünk, akkor a módosítást leíró naplóbejegyzésben megtaláljuk mind a T tranzakció hatásainak semmissé tételéhez szükséges régi, mind a T tranzakció hatásainak helyreállításához szükséges új adatbáziselem-értékeket. A semmisségi/helyrehozó módszer alapelvei a következők:

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

Nem elég a kettő közül az egyik, mindkét eljárásra szükségünk van. A rugalmasság lehetővé teszi, hogy a COMMIT bejegyzés és a lemezen végrehajtott adatbázis-módosítások egymáshoz viszonyított sorrendje kötetlen legyen, így előfordulhat az is, hogy egy befejezett tranzakció néhány vagy összes változtatása még nem került lemezre, és az is, hogy egy be nem fejezett tranzakció néhány vagy összes változtatása már lemezen is megtörtént.

Példa. A hiba fellépésének időpontja függvényében különböző helyreállítási lehetőségeink vannak:

1. Ha a katasztrófa a <COMMIT T> naplóbejegyzés lemezre írását követően fordul elő, akkor T-t befejezett tranzakciónak tekintjük. 16-ot írunk mind A-ba, mind B-be. Az események jelenlegi sorrendjében A-nak már 16 a tartalma, de B-nek lehet, hogy nem, aszerint, hogy a hiba a 11) lépés előtt vagy után következett be.
2. Ha a katasztrófa a <COMMIT T> naplóbejegyzés lemezre írását megelőzően következett be, akkor T befejezetlen tranzakciónak számít. Ez esetben A és B korábbi értéke, 8 íródik lemezre. Ha a hiba a 9) és 10) lépések között következett be, akkor A értéke már 16 volt a lemezen, és emiatt a 8-ra való visszaállítás feltétlenül szükséges. Ebben a konkrét példában a B értéke nem igényelne visszaállítást (mert még meg sem változott), ha pedig a hiba a 9) lépés előtt következik be, akkor A sem igényelné a visszaállítást. Mivel általában nem lehetünk biztosak abban, hogy a visszaállítás szükséges-e vagy sem, így azt (a biztonság kedvéért) mindig végre kell hajtanunk.

A semmisségi naplózáshoz hasonlóan a semmisségi/helyrehozó naplózás is olyan viselkedést mutat, hogy a tranzakció a felhasználó számára korrekten befejezettnek tűnik, de még a <COMMIT T> naplóbejegyzés lemezre kerülése előtt fellépett hiba utáni helyreállítás során a rendszer a tranzakció hatásait semmissé teszi ahelyett, hogy helyreállította volna. Amennyiben ez a lehetőség problémát jelent, akkor a semmisségi/helyrehozó naplózás során egy további szabályt célszerű bevezetni:

UR_2: A <COMMIT T> naplóbejegyzést nyomban lemezre kell írni, amint megjelenik a naplóban.

Ennek teljesítéséért a fenti példában a 10) lépés után egy FLUSH LOG lépést kell beiktatnunk.

Nem adtuk meg azt, hogy a semmisségi vagy a helyrehozó lépést tesszük meg előbb. Előfordulhat, hogy a T tranzakció rendben és teljesen befejeződött, és emiatt helyreállítása során egy X adatbáziselem T által kialakított értékét rekonstruáljuk, melyet viszont egy be nem fejezett, és ezért visszaállítandó U tranzakció korábban módosított. A probléma nem az, hogy először helyreállítjuk X értékét, és aztán állítjuk vissza U előttire, vagy fordítva. E szituációban egyik út sem helyes, mert a végső adatbázis-állapot nagy valószínűséggel így is, úgy is inkonzisztens lesz.

A gyakorlatban az adatbázis-kezelő rendszereknek a módosítások naplózásánál többet kell tenniük. Biztosítaniuk kell, hogy ilyen szituációk ne fordulhassanak elő. Ezzel a konkurenciakezelés foglalkozik. Később megnézzük, hogyan biztosítható T és U elkülönítése, amivel az ugyanazon X adatbáziselemen való kölcsönhatásuk elkerülhető.

Semmisségi/helyrehozó naplózás ellenőrzőpont-képzéssel

A működés közbeni ellenőrzőpont-képzés valamivel egyszerűbb a semmisségi/helyrehozó naplózás alkalmazásakor, mint a másik két naplózási módszernél. Csak a következőket kell tennünk:

1. Írjunk a naplóba <START CKPT(T_1,…,T_k)> naplóbejegyzést, ahol T_1,…,T_k az éppen aktív tranzakciók, majd írjuk a naplót lemezre.
2. Írjuk lemezre az összes piszkos puffert, tehát azokat, melyek egy vagy több módosított adatbáziselemet tartalmaznak. A helyrehozó naplózástól eltérően itt az összes piszkos puffert lemezre írjuk, nem csak a már befejezett tranzakciók által módosítottakat.
3. Írjunk <END CKPT> naplóbejegyzést a naplóba, majd írjuk a naplót lemezre.

A semmisségi/helyrehozó naplózás által a lemezre írások sorrendjére vonatkozóan biztosított rugalmasság miatt megengedhetjük a be nem fejezett tranzakciók változtatásainak lemezre való kiírását. Így megengedhetjük a teljes blokknál kisebb adatbáziselemek használatát is, melyek közös pufferbe kerülnek.

Példa. Tekintsük az alábbi naplórészletet:

<START T_1>
<T_1,A,4,5>
<START T_2>
<COMMIT T_1>
<T_2,B,9,10>
<START CKPT(T_2)>
<T_2,C,14,15>
<START T_3>
<T_3,D,19,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

A példa megegyezik a helyrehozó naplózásnál felírt példával, csak a módosítási bejegyzések változtak, hogy megfeleljenek a semmisségi/helyrehozó naplózás szabályainak. Az egyszerűség kedvéért minden régi érték eggyel kisebb az új értéknél.

Az ellenőrzőpont képzésének kezdetekor T_2 az egyetlen aktív tranzakció. Minthogy ez a napló semmisségi/helyrehozó napló, így lehetséges, hogy a T_2 által B-nek adott új érték (10) lemezre íródik, ami nem volt megengedett a helyrehozó naplózásban. Most lényegtelen, hogy ez a lemezre írás mikor történik meg. Az ellenőrzőpont képzése alatt biztosan lemezre írjuk B-t (ha még nem került oda), mivel minden piszkos (változásban érintett) puffert kiírunk lemezre. Hasonlóan A-t – melyet a befejezett T_1 tranzakció alakított ki – is lemezre fogjuk írni, ha még nem került oda.

- Ha a katasztrófa ezen eseménysorozat végén jelentkezik, akkor T_2-t és T_3-at teljesen és rendesen befejezett tranzakciónak tekintjük. A T_1 tranzakció az ellenőrzőpontnál korábbi. Minthogy <END CKPT> bejegyzést találunk a naplóban, így T_1-ről biztosan tudjuk, hogy teljesen és rendesen befejeződött, valamint az általa elvégzett módosítások lemezre íródtak. Ezért a T_2 és T_3 által végrehajtott módosítások helyreállítandók, T_1 pedig figyelmen kívül hagyható. Amikor olyan tranzakció hatásait állítjuk helyre, mint amilyen a T_2 is, akkor a naplóban nem kell a <START CKPT(T_2)> bejegyzésnél korábbra visszatekinteni, mert tudjuk, hogy a T_2 által az ellenőrzőpont-képzést megelőzően elvégzett módosítások az ellenőrzőpont képzése alatt lemezre íródtak.
- Másik példaként tegyük fel, hogy a katasztrófa éppen a <COMMIT T_3> bejegyzés lemezre írása előtt fordult elő. Ekkor T_2-t befejezett, T_3-at pedig befejezetlen tranzakciónak kell tekintenünk. T_2 tevékenységét helyreállítandó C értékét a lemezen 15-re írjuk; B-t már nem kell 10-re írnunk a lemezen, mert tudjuk, hogy ez már lemezre került az <END CKPT> előtt. A helyreállító naplózástól eltérően T_3 hatásait semmissé tesszük, azaz a lemezen D tartalmát 19-re írjuk. Ha T_3 az ellenőrzőpont-képzés előtt már aktív tranzakció lett volna, akkor a naplóban a START CKPT bejegyzésben szereplő befejezetlen tranzakciók közül a legkorábban elindult T_i tranzakció <START T_i> bejegyzéséig kellene visszakeresnünk, hogy megtaláljuk a T_i (most T_2 vagy T_3) semmissé teendő tevékenységeit leíró naplóbejegyzéseket. A helyrehozó lépést viszont most is elég a START CKPT bejegyzéstől végrehajtani.
- Ha a katasztrófa az <END CKPT> bejegyzés előtt lép fel, akkor figyelmen kívül hagyjuk az utolsó START CKPT bejegyzést, és a fentieknek megfelelően járunk el.

Az eszközök meghibásodásának kezelése

A naplózással a rendszerhibák ellen védekezhetünk, amelyek következtében legfeljebb a memóriában tárolt ideiglenes adatok vesznek el, de a lemezről semmi nem veszhet el. Ugyanakkor komoly hibát okoz egy vagy több lemez elvesztése. Az adatbázist a naplóból elméletileg akkor tudjuk rekonstruálni, ha:

- a naplót tároló lemez különbözik az adatbázist tartalmazó lemez(ek)től;
- a naplót sosem dobjuk el az ellenőrzőpont-képzést követően;
- a napló helyrehozó vagy semmisségi/helyrehozó típusú, így az új értékeket (is) tárolja.

Ugyanakkor a napló esetleg az adatbázisnál is gyorsabban növekedhet, így nem praktikus a naplót örökre megőrizni.

Az archívmentés

Az eszközök meghibásodásának kezelésére az egyik módszer az *archiválás* (archiving), az adatbázis másolatának elkészítése egy vagy több, az adatbázisétól különböző adathordozón. Ha lehetséges, lezárjuk az adatbázist addig, amíg elkészítjük a biztonsági másolatot (backup) valamely tárolóeszközön (például optikai lemezen vagy mágnesszalagon), majd a biztonsági másolatot az adatbázistól távol, biztonságos helyen tároljuk. A biztonsági másolat megőrzi az adatbázis mentéskori állapotát, és ha eszközhiba lép fel, akkor a mentésből az adatbázis ezen állapotát vissza tudjuk állítani.

A napló használatával sokkal frissebb állapotot tudunk rekonstruálni. Ha a biztonsági másolat készítése után történt adatbázis-változásokról keletkező naplót megőrizzük, és az túlélte az eszköz meghibásodását, akkor a hiba után (esetleg másik lemezen) visszaállítva a biztonsági másolatot, a napló felhasználásával a mentés óta történt adatbázis-változásokat is át tudjuk vezetni az adatbázison. A naplóról, amilyen gyorsan csak lehet, távoli másolatot készítünk. Ezzel a napló elvesztése ellen védekezhetünk. Így ha a napló az adatokkal együtt elveszik is, akkor még mindig használhatjuk az adatbázis mentését és a napló távoli másolatát az adatbázis visszaállítására, egészen addig a pillanatig, amikor a napló utolsó átvitele történt a távoli másolatára.

Ha az adatbázis nagy, akkor a biztonsági mentés elkészítése (kiírása) hosszas folyamat. Általánosan bevált, hogy nem mentik a teljes adatbázist minden archiváló alkalommal. Ezért a mentésnek két szintjét különböztetjük meg:

- *teljes mentés* (full dump), amikor az egész adatbázisról másolat készül;
- *növekményes mentés* (incremental dump), amikor az adatbázisnak csak azon elemeiről készítünk másolatot, melyek az utolsó teljes vagy növekményes mentés óta megváltoztak.

Lehetséges a mentésnek több szintjét is használni: a teljes mentést 0-dik szintűnek tekintjük, az *i*-edik szintű mentésen pedig azt a mentést értjük, amely az utolsó *i*-edik, illetve annál alacsonyabb szintű mentés óta megváltozott elemek másolatát tartalmazza. Egy új *i*-edik szintű mentés készítésekor az *i*-nél magasabb szintű mentéseket töröljük, vagy visszaállításkor figyelmen kívül hagyjuk.

Az adatbázist a teljes mentésből és a megfelelő növekményes mentésekből tudjuk rekonstruálni, a helyrehozó vagy a semmisségi/helyrehozó naplózás rendszerhiba utáni visszaállítási folyamatához hasonló módszerrel. Visszamásoljuk a teljes mentést, majd az ezt követő legkorábbi növekményes mentéstől kezdve végrehajtjuk a növekményes mentésekben tárolt változtatásokat. Többszintű mentés esetén a 0-nál magasabb szintű mentéseket szintszám szerint növekvőleg, az azonos szintszámú mentéseket pedig időrendi sorrendben vesszük figyelembe. A növekményes mentések az adatoknak csak azt a kis részét érintik, amely az utolsó mentés óta változott meg, így ezek kevesebb helyet igényelnek, és gyorsabban menthetők, mint a teljes mentés.

Felmerülhet a kérdés, hogy miért nem csak a naplót mentjük, hiszen a napló használatával egy régi mentésből is helyreállíthatnánk az adatbázist. Nem nyilvánvaló, hogy milyen gyakran célszerű biztonsági mentést készíteni, ez az adatbázis méretén és tipikus módosítási fokán múlik. Amíg az adatbázisnak naponta esetleg csak kis része változik, addig a naplózandó módosítások tömege egy egész év folyamán sokkal nagyobb lehet, mint maga az adatbázis. Ha soha nem archiválunk, akkor a napló soha nem csonkolható, és a napló tárolási/kezelési költsége hamar túllépheti az adatbázis másolatának tárolási költségét.

Archiválás működés közben

Az előbbiekben bemutatott, egyszerűnek látszó archiválással az a probléma, hogy sok adatbázist nem lehet lezárni arra az időre, amíg a biztonsági mentést elkészítjük. Így – a működés közbeni ellenőrzőpont-képzéshez hasonlóan – meg kell oldanunk a *működés közbeni archiválást* (nonquiescent archiving) is. A működés közbeni ellenőrzőpont-képzés megkísérli az indulásakori adatbázis-állapotot lemezre írni. Az ellenőrzőpont létrehozásának környékén keletkezett kis naplórészletre támaszkodva az adatbázis állapotában történt minden olyan eltérést rendbe tudunk hozni, melyet az okozott, hogy az ellenőrzőpont képzése alatt új tranzakciók indulhattak és lemezírások történhettek.

Ehhez hasonlóan a működés közbeni archiválás megbízhatóan tud az adatbázisról olyan másolatot készíteni, ami az archiválás kezdetének többé-kevésbé megfelelő adatbázis-állapotot rögzíti. Ugyanakkor a mentés alatti percekben vagy órákban az adatbázis működése sok adatbáziselemet cserélhet. Ha az adatbázis mentésből való visszaállítása szükéges, akkor a mentés alatt keletkezett naplóbejegyzések felhasználásával az adatbázis konzisztens állapota állítható elő. Az ellenőrzőpont képzésekor tehát az adatokat a memóriából a lemezre visszük, és a napló lehetővé teszi a rendszerhibák utáni helyreállítást, míg archiváláskor az adatokat a lemezről másodlagos háttértárolóra visszük, és az archívmentés a naplóval lehetővé teszi az eszközhibák utáni helyreállítást.

A működés közbeni archiválás az adatbázis elemeit valamely fix sorrendben másolja, mialatt megeshet, hogy ezen elemeket az éppen végrehajtott tranzakciók módosítják. Ennek eredményeként megtörténhet, hogy a biztonsági mentésbe másolt adatbáziselem értéke nem ugyanaz, mint a mentés megkezdésekor volt. Amíg a mentés alatt keletkezett naplót megőrizzük, addig az eltérések a napló felhasználásával korrigálhatók.

Példa. Tegyük fel, hogy adatbázisunk 4 elemből áll: `A`, `B`, `C` és `D`. Ezek értéke az archiválás kezdetekor rendre 1, 2, 3, 4. A mentés közben `A` értéke 5-re, `C` értéke 6-ra, `B` értéke 7-re módosul. Az adatbáziselemeket a mentéskor sorban másoljuk az archívumba, az események sorrendje pedig legyen a következő:

Lemez	*Mentés*
	`A`
`A := 5`	
	`B`
`C := 6`	
	`C`
`B := 7`	
	`D`

Ekkor noha az adatbázis tartalma a mentés kezdetekor 1, 2, 3, 4 volt, a mentés végére pedig 5, 7, 6, 4 lett, a mentett archívumba 1, 2, 6, 4 került, jóllehet ilyen adatbázis-állapot a mentés ideje alatt nem is fordult elő.

Részletesebben a biztonsági mentés (archívum) elkészítése a következő lépésekből áll. Feltételezzük, hogy az alkalmazott naplózási módszer a helyrehozó vagy a semmisségi/helyrehozó módszerek valamelyike; a semmisségi naplózás nem alkalmas a működés közbeni archiválással való használatra (bővebben lásd a példa után).

1. A <START DUMP> bejegyzés naplóba írása.
2. Az alkalmazott naplózási módnak megfelelő ellenőrzőpont kialakítása.
3. A menteni kívánt adatlemez(ek) teljes vagy növekményes mentésének végrehajtása, ügyelve arra, hogy az adatok másolata (a mentés) biztonságos, távoli helyre kerüljön.
4. Gondoskodjunk arról is, hogy a napló szükséges részéről is másolat készüljön, és az is biztonságos, távoli helyre kerüljön. A mentett naplórész tartalmazza legalább a 2. pontbeli ellenőrzőpont-képzés közben keletkezett naplóbejegyzéseket, melyeknek túl kell élniük az adatbázist hordozó eszköz meghibásodását.
5. <END DUMP> bejegyzés naplóba írása.

A mentés befejezésekor biztonsággal eldobhatjuk a naplónak azt a részét, amelyre nincs szükség a 2. pontban végrehajtott ellenőrzőpont-képzéshez tartozó helyreállítási folyamat szabályai szerint.

Példa. Tegyük fel, hogy a fenti adatbázis mentés közbeni módosításait két tranzakció, T_1 (mely A-t és B-t módosította) és T_2 (mely C-t módosította) végezte, melyek a mentés kezdetekor aktívak voltak. Semmisségi/helyrehozó naplózási módszert alkalmazva a mentés alatti események lehetséges naplóbejegyzései a következők:

<START DUMP>
<START CKPT(T_1,T_2)>
<T_1,A,1,5>
<T_2,C,3,6>
<COMMIT T_2>
<T_1,B,2,7>
<END CKPT>
a mentés befejezése
<END DUMP>

Látható, hogy T_1 nem fejeződött be a mentés befejezéséig. Az eléggé valószínűtlen, hogy egy tranzakció a teljes mentés egész ideje alatt aktív maradjon, de ez a lehetőség nem befolyásolja a bemutatandó helyreállítási módszer helyességét.

Most már az is látható, hogy miért nem használható a semmisségi naplózás a működés közbeni archiválással. Tegyük fel, hogy a <START CKPT(T_1,T_2)> bejegyzés után elindul egy T_3 tranzakció, amely módosítja A értékét, majd B értékét, aztán rendesen befejeződik, tehát egy <COMMIT T_3> is a naplóba kerül, de csak az <END CKPT> bejegyzés naplóba kerülése után (azaz a mentés közben). Mivel semmisségi naplózás esetén az OUTPUT műveletek a módosítási bejegyzés naplóba írását követően bármikor lefuthatnak, ezért előfordulhat, hogy A értékét annak módosítása után, de B értékét annak módosítása előtt archiváljuk. A helyreállítás folyamán a T_3 tranzakcióval nem foglalkozunk, mert megtaláltuk a naplóban a <COMMIT T_3> bejegyzést, így olyan eredményt kapunk, mintha T_3 nem atomosan hajtódott volna végre. Helyrehozó naplózást használva ilyen eset nem fordulhat elő, mert akkor OUTPUT művelet csak a COMMIT után futhat le, és így vagy nem történik változtatás a lemezen (ha nincs COMMIT), vagy „újra lejátsszuk” a tranzakciót (ha van COMMIT). Semmisségi/helyrehozó naplózás esetén pedig minden tranzakciót vagy semmissé teszünk (ha nincs COMMIT), vagy helyreállítunk (ha van COMMIT), tehát szintén nem fordulhat elő nem atomos viselkedés.

Helyreállítás az archívmentés és a napló használatával

Tegyük fel, hogy készülékhiba lépett fel, és az adatbázist rekonstruálnunk kell. A helyreállítást a legutolsó biztonsági mentés és a napló távoli mentése felhasználásával végezzük. A következő lépéseket hajtjuk végre:

1. Az adatbázis visszaállítása a biztonsági mentésből:
   a) Meg kell keresni a legutolsó teljes mentést, belőle rekonstruálni az adatbázist.
   b) Ha vannak későbbi növekményes mentések, akkor ezeket időrendi sorrendben használva módosítjuk az adatbázist. Többszintű mentés esetén az 1. szinttől kezdve sorban az összes szint összes mentését alkalmazni kell (szintenként, azon belül időrendben).
2. Módosítjuk az adatbázist a napló katasztrófát túlélt részével, a naplózási módszernek megfelelő helyreállítási eljárást használva.

Példa. Tegyük fel, hogy a fenti példában szereplő biztonsági mentés elkészítését követően történik eszközmeghibásodás, és a napló ezt túlélte. Azért, hogy az eljárást érdekesebbé tegyük, tekintsük úgy, hogy a napló katasztrófát túlélt részében nincs <COMMIT T_1> bejegyzés, van viszont <COMMIT T_2>. Az adatbázist először a biztonsági mentésből visszatöltjük, így A, B, C, D értékei rendre 1, 2, 6, 4 lesznek.

Ezután a naplót vesszük elő. Minthogy T_2 befejezett tranzakció, helyreállítjuk azon lépés hatását, amely C értékét 6-ra módosította. Példánkban C értéke már 6, de előfordulhatna, hogy

- C mentése azt megelőzően történt, hogy C értékét a T_2 tranzakció módosította volna;
- a mentésben C-nek később kapott értéke van, mely értéket olyan tranzakció állított be, melyre vonatkozó COMMIT bejegyzést a naplóban vagy találunk, vagy nem. C értékét a mentésben talált értékre akkor állítjuk, ha az ezt beállító tranzakció COMMIT bejegyzését megtaláljuk.

Minthogy T_1 gyaníthatóan nem befejezett tranzakció (mert COMMIT bejegyzését nem találjuk), így T_1 hatásait semmissé kell tennünk. A T_1-re vonatkozó naplóbejegyzések használatával meg tudjuk állapítani, hogy A értékét 1-re, B értékét 2-re kell visszaállítanunk. Előfordulhat persze, hogy a mentésen ez az értékük, de ettől eltérő értékeik is lehetnek, ha A és/vagy B módosított értéke archiválódott. (Ez a módosításnak és a mentésnek az időbeli sorrendjétől függ.)

Az Oracle naplózási és archiválási rendszere

Az alábbi információk forrása az Oracle Database Administrator's Guide és az Oracle Database Backup and Recovery User's Guide.

A napló

Egy szerverpéldány rendszerhibája esetén az Oracle az online naplófájlokat használja az adatbázis automatikus helyreállításához. A *példány-helyreállítás* (instance recovery) azonnal megtörténik, amint a példány újraindul a rendszerhiba után. A helyreállítási műveletek alapja a *napló* (redo log), amely az adatbázis változásait tárolja, amint azok bekövetkeznek. Minden Oracle szerverpéldány rendelkezik egy naplóval, amellyel védekezhetünk a rendszerhibák ellen. Két részből áll: az online és az archivált naplóból.

Az *online napló* két vagy több online naplófájlból áll, amelyek *naplóbejegyzésekkel* (redo record vagy redo entry) vannak feltöltve, ezeket pedig *változásvektorok* (change vector) alkotják. A változásvektorok az adatbázis egy blokkjának a változásáról tartalmaznak információkat. Ha például megváltoztatunk egy fizetési értéket egy alkalmazottakra vonatkozó adatokat tároló táblában, egy új naplóbejegyzés jön létre egy-egy változásvektorral a táblát tartalmazó adatfájl blokkjának, az undo szegmens blokkjának és az undo szegmens tranzakciós táblájának a változásáról (lásd később). A naplóbejegyzések ideiglenesen az SGA (System Global Area) memóriapuffereiben tárolódnak, amelyeket a Log Writer (LGWR) háttérfolyamat folyamatosan ír ki valamelyik naplófájlba. (Az SGA tartalmazza az adatbáziselemeket tároló puffereket is, amelyeket pedig a Database Writer háttérfolyamat ír lemezre.) Ha egy felhasználói folyamat befejezte egy tranzakció végrehajtását, akkor a LGWR a tranzakcióhoz tartozó naplóbejegyzéseket az SGA

memóriapuffereiből az egyik naplófájlba írja, és hozzájuk rendel egy időbélyeget (*system change number*, SCN), amellyel a befejezett tranzakció naplóbejegyzéseit azonosíthatjuk. A rendszer csak azután értesíti a felhasználói folyamatot, hogy a tranzakció véglegesítődött, miután az adott tranzakcióra vonatkozó összes naplóbejegyzés lemezre került. A naplóbejegyzések azelőtt is lemezre íródhatnak, mielőtt a megfelelő tranzakció véglegesítődne. Ha a napló memóriapufferei megtelnek, vagy egy másik tranzakció véglegesítődik, a LGWR az összes naplóbejegyzést lemezre írja, még akkor is, ha ezek egy része nincs véglegesítve. Ha szükséges, ezek a változások semmissé tehetők.

Ahogy említettük, az online napló két vagy több naplófájlból áll. Az Oracle egyszerre csak egy naplófájlt használ a naplóbejegyzések kiírására. Azt a naplófájlt, amelyikbe a LGWR éppen ír, *aktuális* naplófájlnak nevezzük. Azokat a naplófájlokat, amelyek szükségesek egy példány-helyreállításhoz (azaz a benne tárolt változások még nem mind íródtak lemezre), *aktív* naplófájloknak, amelyekre pedig nincs szükség (azaz a benne tárolt változások már mind lemezre íródtak), *inaktív* naplófájloknak nevezzük. Azért van szükség legalább két állományra, hogy az egyik akkor is elérhető legyen a naplóbejegyzések írására, mialatt a másik épp archiválás alatt áll (ha az adatbázis ARCHIVELOG módban van). Az online naplófájlok ciklikusan töltődnek föl. Amikor az aktuális naplófájl megtelt, a LGWR a következő elérhető naplófájlt kezdi el feltölteni. Amikor az utolsó elérhető naplófájl is megtelt, akkor újra az elsőt kezdi el írni, újrakezdve a kört. A megtelt naplófájlok attól függően lesznek újra elérhetők a LGWR számára, hogy a napló archiválása be van-e kapcsolva. Ha nem (az adatbázis NOARCHIVELOG módban van), akkor egy megtelt naplófájl akkor lesz elérhető, ha már inaktívvá vált. Ha az archiválás be van kapcsolva (az adatbázis ARCHIVELOG módban van), akkor egy megtelt naplófájl akkor lesz elérhető, ha már inaktívvá vált, *és* a naplófájlt már archiválta az egyik archiváló háttérfolyamat (ARC*n*).

Naplóváltásnak (log switch) nevezzük azt a pillanatot, amikor a rendszer befejezi az egyik naplófájl írását, és elkezdi egy másikét. Naplóváltás általában akkor történik, amikor az aktuális naplófájl teljesen megtelt, és az írást a következő naplófájlban kell folytatni. Beállíthatjuk azonban, hogy szabályos időközönként is történjen naplóváltás, függetlenül attól, hogy az aktuális naplófájl megtelt-e már. Ezenkívül manuálisan is kérhetünk naplóváltást. Valahányszor naplóváltás történik, az Oracle egy új sorszámot (log sequence number) rendel ahhoz a naplófájlhoz, amibe a LGWR megkezdi az írást. Amikor a rendszer archiválja a naplófájlokat, az archivált napló megőrzi a sorszámát. Az a naplófájl, amit újra elkezdünk használni, a soron következő sorszámot kapja meg. Így tehát minden online vagy archivált naplófájl egyedileg azonosítható a sorszámával. Helyreállítás során az Oracle a szükséges archivált vagy online naplófájlokat a sorszámaik szerinti növekvő sorrendben alkalmazza.

Magának a naplónak a meghibásodása ellen védekezhetünk a *multiplexelt online napló* (multiplexed redo log) segítségével, ami azt jelenti, hogy a napló kettő vagy több egyenértékű másolata kezelhető automatikusan. Ha multiplexeljük a naplófájlokat, a LGWR párhuzamosan ugyanazokat az információkat írja a különböző egyenértékű naplófájlokba, ezáltal kiküszöbölve az egyikük megsérüléséből eredő adatvesztést. Legjobb, ha a másolatok különböző lemezeken vannak, mert ha az egyik lemez megsérül, akkor a napló többi másolata még mindig rendelkezésre áll a helyreállításhoz. Azonban még ha a másolatok ugyanazon a lemezen vannak is, a redundáns tárolás segíthet kivédeni a szektorhibákat, állományszerkezeti hibákat stb.

Lehetőség van tehát arra, hogy a megtelt online naplófájlokat archiváljuk, mielőtt újra felhasználnánk őket. Az *archivált (offline) napló* (archived redo log) az ilyen archivált naplófájlokból tevődik össze. A naplófájlok archiválása csak akkor lehetséges, ha az adatbázis ARCHIVELOG módban fut. Az archiválás lehet automatikus vagy manuális.

NOARCHIVELOG módban az online naplófájlok archiválása nem lehetséges. Az adatbázis vezérlőfájlja jelzi, hogy a megtelt naplófájlokat nem szükséges archiválni. Így amikor egy megtelt naplófájl inaktívvá válik egy naplóváltást követően, azt a fájlt a LGWR újra felhasználhatja. NOARCHIVELOG módban az adatbázis csak rendszerhiba után állítható helyre, eszközhiba esetén nem. A helyreállításhoz csak az online naplófájlokban tárolt legfrissebb adatbázis-módosításokat használhatjuk fel. Ha NOARCHIVELOG módban eszközhiba következik be, akkor csak a legfrissebb teljes mentés időpontjáig állíthatjuk vissza az

adatbázist, az azt követő tranzakciók hatása elvész. NOARCHIVELOG módban nem végezhetünk online táblaterület-mentést, és nem is használhatjuk fel a korábban, ARCHIVELOG módban készült online táblaterület-mentéseket. Egy NOARCHIVELOG módban működő adatbázist csak teljes mentésből állíthatunk vissza, amely az adatbázis zárt állapotában készült. Emiatt NOARCHIVELOG módban célszerű az adatbázisról rendszeresen teljes mentést készíteni.

ARCHIVELOG módban a napló archiválása be van kapcsolva. Az adatbázis vezérlőfájlja jelzi, hogy a megtelt naplófájlokat nem használhatja fel újra a LGWR, amíg azok nincsenek archiválva. A megtelt naplófájlok egy naplóváltást követően archiválhatók. A naplófájlok archiválásának az alábbi előnyei vannak:

- Az adatbázis mentése az online és archivált naplófájlokkal együtt garantálja, hogy minden véglegesített tranzakció helyreállítható az operációs rendszer vagy a lemez meghibásodása esetén.
- Ha elérhetőek az archivált naplófájlok, akkor egy működés közben készített mentést is felhasználhatunk a helyreállításhoz.
- Fenntarthatunk az adatbázisunkról egy másolatot, amelyet az eredeti adatbázis archivált naplójának a másolatra történő folyamatos alkalmazásával tarthatunk naprakészen.

Az Oracle az online naplót kizárólag helyreállításra használja. Az adminisztrátorok azonban egy SQL interfészen keresztül lekérdezéseket hajthatnak végre rajta a *LogMiner* naplóelemző eszköz segítségével. A naplófájlok ugyanis hasznos információkat szolgáltathatnak a korábbi adatbázis-tevékenységekről.

Minden Oracle adatbázis rendelkezik egy *vezérlőfájllal* (control file), amely egy kisméretű bináris állomány, és az adatbázis fizikai szerkezetéről tárol információkat. A vezérlőfájl tartalmazza

- az adatbázis nevét,
- az adatbázishoz tartozó adat- és naplófájlok nevét és helyét,
- az adatbázis létrehozásának idejét,
- az aktuális naplósorszámot,
- ellenőrzőpont-információkat.

Az adatbázis normális működéséhez az Oracle szervernek írási módban el kell tudnia érni a vezérlőfájlt. Nélküle nem lehet csatlakozni az adatbázishoz, és nehézkes a helyreállítás. A vezérlőfájl az adatbázissal egy időben jön létre. Alapértelmezésben az adatbázis létrehozásakor a vezérlőfájlnak legalább egy példánya (néhány operációs rendszer esetén eleve több példánya) is létrejön. A legjobb, ha minden Oracle adatbázis legalább két vezérlőfájllal rendelkezik, mindegyik különböző fizikai adathordozón: ez a *multiplexelt vezérlőfájl* (multiplexed control file). Ha egy vezérlőfájl lemezhiba miatt megsérül, a hozzá tartozó szerverpéldányt le kell állítani. A lemezhiba elhárítása után a sérült vezérlőfájl helyreállítható a másik lemezen tárolt ép példányának felhasználásával, és a szerverpéldány újraindítható. Ebben az esetben nincs szükség eszközhiba utáni helyreállításra.

Az undo információk

Az Oracle a semmisségi és a helyrehozó naplózás egy speciális kombinációját valósítja meg. Ahogy láttuk, a tranzakciók helyrehozásához szükséges információkat (az adatbázisblokkok módosított értékeit) az online napló tartalmazza. A tranzakciók hatásainak semmissé tételéhez szükséges információk pedig alapesetben egy vagy több *undo táblaterületen* (undo tablespace) tárolódnak (vagy más táblaterületen elhelyezkedő rollback szegmensekben – lásd később). Ez azt jelenti, hogy az Oracle az undo adatokat az adatbázisban tárolja, nem külső naplófájlokban. Az undo adatok tehát ugyanolyan blokkokban helyezkednek el, mint az adatbázis más adatai, és ezen blokkok változásai ugyanúgy naplózásra kerülnek. Az Oracle így hatékonyan, külső naplófájlok olvasása nélkül tud hozzáférni az undo adatokhoz. Az undo táblaterület a tranzakciók által módosított adatok régi értékeit tárolja attól függetlenül, hogy ezek a tranzakciók véglegesítettek-e vagy

sem. Az undo információkat használjuk egy aktív tranzakció visszagörgetésére, egy megszakadt tranzakció helyreállítására, az olvasási konzisztencia biztosítására és flashback műveletek végrehajtására is.

Az undo táblaterület *undo szegmensekből* (undo segment), azok pedig *undo bejegyzésekből* (undo record vagy undo entry) állnak. Egy undo bejegyzés többek között a megváltozott attribútum(ok) azonosítóját (címét), a módosítást végző tranzakciós műveletet, az annak hatását semmissé tevő utasítást és az attribútum(ok) régi értékét tárolja. Az undo bejegyzés mindig előbb kerül lemezre, mint ahogy az adatbázisban megtörténik a megfelelő attribútumok módosítása. Az ugyanazon tranzakcióhoz tartozó bejegyzések össze vannak láncolva, így könnyen visszakereshetők, ha az adott tranzakciót vissza kell görgetni.

Az undo táblaterületen minden undo szegmenshez tartozik egy *tranzakciós tábla* (transaction table), amely az adott undo szegmenst használó tranzakciók azonosítóit tartalmazza. Minden tranzakciós tábla fix számú bejegyzésből (slotból) áll. Ez a szám az adatblokk méretétől függ, amit viszont az operációs rendszer határoz meg. Minden bejegyzéshez egy tranzakció tartozik. Az Oracle sorban rendeli hozzá a tranzakciókat a tranzakciós tábla szabad elemeihez. Ha a tábla betelik, elölről kezdi felhasználni a szabad elemeket. Egy elem akkor válik szabaddá, ha az általa képviselt tranzakció véglegesítődött. Ha minden elem aktív tranzakcióhoz tartozik, akkor egy újabb tranzakciónak várakoznia kell, amíg valamelyik elem fel nem szabadul.

Ha egy tranzakció befejeződött, akkor a rá vonatkozó undo bejegyzésekre visszagörgetési vagy tranzakció-helyreállítási célból ugyan nincs többé szükség, azonban mégsem törölhetők, mert elképzelhető, hogy még a tranzakció befejeződése előtt elindult egy olyan lekérdezés, amelyhez szükség van a módosított adatok régi értékeire (ezt nevezzük olvasási konzisztenciának – lásd *Az Oracle konkurenciavezérlési technikája* című részt). Ezenkívül a flashback műveletek sikeressége is a régebbi undo adatok elérhetőségén múlhat. Ezen okok miatt a régi undo információkat a lehető legtovább célszerű megőrizni.

Ha egy adatbázist a Database Configuration Assistant (DBCA) segédprogrammal hozunk létre, automatikusan létrejön egy UNDOTBS1 nevű undo táblaterület is. Saját undo táblaterület is készíthető a CREATE DATABASE vagy a CREATE UNDO TABLESPACE utasítás segítségével. Amikor a szerverpéldány elindul, automatikusan kiválasztja az első elérhető undo táblaterületet. Ha nincs ilyen, akkor a rendszer undo táblaterület nélkül indul el, és a SYSTEM táblaterületet használja az undo bejegyzések tárolására, ez azonban nem ajánlott. Ha az adatbázis több undo táblaterülettel rendelkezik, a használni kívánt undo táblaterületet magunk is megadhatjuk az UNDO_TABLESPACE paraméter segítségével.

A példány-helyreállítás lépései

Amikor egy undo bejegyzés az undo szegmensbe kerül, a naplóban erről is készül egy naplóbejegyzés, hiszen az undo táblaterületek – más táblaterületekhez hasonlóan – az adatbázis részét képezik. Ez azt eredményezi, hogy az online napló a permanens objektumokra vonatkozó undo információkat is tárolja. Az adatbázisban bekövetkező minden egyes változtatás hatására tehát létrejön egy undo bejegyzés a módosított attribútum(ok) régi értékével, egy naplóbejegyzés a módosított adatokat tartalmazó adatblokkok új értékével, valamint egy másik naplóbejegyzés az undo bejegyzést tartalmazó adatblokk új értékével.

A példány-helyreállítás első lépése a *rolling forward* (vagy *cache recovery*), amelynek során az online naplóban feljegyzett változásokat átvezetjük az adatbázisra. A naplót elegendő az utolsó ellenőrzőponttól kezdődően átvizsgálni. Az ellenőrzőpont garantálja, hogy minden olyan véglegesített módosítás, amelynek az SCN értéke kisebb az ellenőrzőponténál, lemezre került. Az ellenőrzőpont pozíciója (SCN értéke) számos esetben módosulhat, például amikor a Database Writer háttérfolyamat lemezre írja a piszkos puffereket.

A rolling forward lépés után kapott adatbázis nagy valószínűséggel inkonzisztens lesz. Ezután minden olyan módosítást, amely nem volt véglegesítve, semmissé kell tenni. Mivel az online naplóban az undo adatok is feljegyzésre kerültek, a rolling forward lépés a megfelelő undo szegmenseket is helyreállítja. Az

Oracle ezek alapján semmissé tesz az adatbázisban minden olyan nem véglegesített módosítást, amely a rendszerhiba bekövetkezése előtt vagy a rolling forward lépés alatt keletkezett. Ez a lépés a *rolling back* (vagy *transaction recovery*).

Az undo információk kezelésének módjai

Az undo információk menedzselése két módon történhet: *automatikus* (automatic undo management) és *manuális* (manual undo management) módban. Automatikus módban az Oracle automatikusan kezeli az undo szegmenseket az undo táblaterületeken, nincs szükség felhasználói beavatkozásra. Ez az alapértelmezett mód egy újonnan telepített adatbázis esetén. Manuális módban nem használunk undo táblaterületet, az undo információk *rollback szegmensekben* (rollback segment), azaz felhasználó által kezelt undo szegmensekben tárolódnak. A rollback szegmensek által elfoglalt tárterület kezelése összetett feladat, és nagy terhet ró a DBA-ra.

Mentés és visszaállítás

A mentés és visszaállítás középpontjában az adatbázist alkotó adatállományok fizikai mentése áll, amely lehetővé teszi az adatbázis későbbi rekonstrukcióját. Az Oracle az RMAN nevű parancssori eszközt ajánlja az adatbázis hatékony mentésére és visszaállítására. Az RMAN védelme az adatfájlokra, a vezérlőfájlokra, a szerverparamétereket tartalmazó fájlokra és az archivált naplófájlokra terjed ki. Ezek az állományok szükségesek az adatbázis rekonstruálásához. Az RMAN-t úgy tervezték, hogy szorosan együttműködjön az adatbázisszerverrel, blokk szintű hibafelismerést biztosítva a mentés és a visszaállítás során. A mentés során optimalizálja a tárhelyfoglalást az állományok multiplexelésével és tömörítéssel, valamint támogatja a vezető szalagos és egyéb tárolóeszközöket. A mentési eljárás fizikai szinten zajlik, így véd az állományok sérülései (pl. egy adatfájl véletlen letörlése vagy egy lemezmeghajtó meghibásodása) ellen. Az RMAN pillanat-visszaállításra is alkalmas, kiküszöbölve bizonyos logikai hibákat, amikor más technikák (pl. a flashback műveletek) már nem használhatók.

NOARCHIVELOG módban a megtelt inaktív naplófájlok felülírhatók. Ilyenkor az adatbázis védve van a rendszerhibák ellen, de nincs védve a készülékhibák ellen. ARCHIVELOG módban a megtelt naplófájlok archiválásra kerülnek. Ekkor az adatbázis mind a rendszerhibák, mind a készülékhibák ellen védve van, viszont további hardveres erőforrásokra lehet szükség.

Egy adatfájl *teljes mentése* (full backup) magában foglalja az állomány összes blokkját. A *növekményes mentés* (incremental backup) csak azokat a blokkokat másolja, amelyek módosulnak a mentések között. A *nulladik szintű növekményes mentés* – amely az adatfájl összes blokkját másolja – használható egy növekményes mentési stratégia kiindulópontjaként. Az *első szintű növekményes mentés* csak azokat a blokkokat másolja, amelyek az utolsó nulladik vagy első szintű mentés óta megváltoztak. Egy első szintű mentés lehet *kumulatív* (cumulative), ha tartalmazza az összes megváltozott blokkot az utolsó nulladik szintű mentés óta, vagy *differenciális* (differential), ha csak az utolsó nulladik vagy első szintű mentés óta történt változásokat tartalmazza. A tipikus növekményes mentési stratégiák szabályos időközönként (pl. naponta) készítenek első szintű mentéseket. Visszaállítás során az RMAN automatikusan alkalmazza mind a növekményes mentéseket, mind a naplót, hogy rekonstruálja az adatbázis egy kívánt időpontbeli állapotát.

A mentés lehet *konzisztens* vagy *inkonzisztens*. A konzisztens mentés az adatbázis konzisztens állapotában készül. Az adatbázis konzisztens lesz, miután leállítottuk a SHUTDOWN NORMAL, a SHUTDOWN IMMEDIATE vagy a SHUTDOWN TRANSACTIONAL paranccsal. A konzisztens leállítás garantálja, hogy minden naplózott módosítás lemezre íródik. Ha ezután mountoljuk az adatbázist, és készítünk egy mentést, akkor később eszköz-helyreállítás nélkül visszaállíthatjuk és megnyithatjuk az adatbázist. Természetesen azonban elveszítjük a mentés készítése után futott tranzakciók hatását.

Minden olyan mentést, amely nem konzisztens, inkonzisztensnek nevezünk. Egy nyitott adatbázisról készült mentés mindig inkonzisztens, mint ahogy egy rendszerhiba utáni vagy egy SHUTDOWN ABORT paranccsal leállított adatbázisról készült mentés is. Ha az adatbázist inkonzisztens mentésből állítjuk vissza, először *eszköz-helyreállítást* (media recovery) kell végeznünk, mielőtt megnyithatnánk az adatbázist. Ennek során a naplóban jelen lévő, a mentés elkészítését követően bekövetkezett változásokat alkalmazzuk az adatfájlokra. Az RMAN nem engedi meg inkonzisztens mentések készítését, ha az adatbázis NOARCHIVELOG módban van. Ha azonban az adatbázis ARCHIVELOG módban van, és mentjük az archivált naplót és az adatfájlokat, az inkonzisztens mentések egy jól működő mentési és helyreállítási stratégia alapját képezhetik. Az inkonzisztens mentések nagyobb rendelkezésre állást kínálnak, mert nem kell leállítanunk az adatbázist ahhoz, hogy teljes védelmet biztosító mentéseket készíthessünk.

Az eszköz-helyreállításhoz szükség van egy vezérlőfájlra, az adatfájlokra (amelyeket tipikusan mentésből állítunk vissza), valamint az online és archivált naplófájlokra, amelyek az adatfájlok mentése óta történt változásokat tartalmazzák. Az eszköz-helyreállítást leggyakrabban készülékhibák (pl. egy állomány vagy egy lemez elvesztése) vagy felhasználói hibák (pl. egy tábla tartalmának a letörlése) utáni helyreállításra használjuk.

Az eszköz-helyreállítás lehet *teljes visszaállítás* (complete recovery) vagy *pillanat-visszaállítás* (point-in-time recovery). A teljes visszaállítás vonatkozhat külön az egyes adatfájlokra, táblaterületekre vagy az egész adatbázisra. A pillanat-visszaállítás rendszerint a teljes adatbázisra vonatkozik (vagy az RMAN segítségével néha csak egyes táblaterületekre). Teljes visszaállítás esetén visszamásoljuk a mentett adatfájlokat, majd alkalmazzuk rájuk az archivált és online naplófájlokban leírt módosításokat. Az adatbázis a hiba időpontjában fennálló állapotába kerül vissza, és adatvesztés nélkül megnyitható. Pillanat-visszaállítás esetén az adatbázist egy felhasználó által választott múltbeli időpillanatban fennálló állapotába állítjuk vissza. Először visszamásoljuk az adott időpillanat előtt készített mentésből az adatfájlokat, valamint az archivált naplófájlok teljes halmazát a mentés készítésének idejétől a kiválasztott időpontig. Ezután átvezetjük a mentéstől az adott időpontig végrehajtott módosításokat az adatfájlokra. A kiválasztott időpont utáni módosításokat figyelmen kívül hagyjuk.

Konkurenciavezérlés

A tranzakciók közötti egymásra hatás az adatbázis-állapot 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 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, *konkurenciavezérlésnek* (concurrency control) 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.

Soros és sorba rendezhető ütemezések

A konkurenciavezérlés tanulmányozását azzal kezdjük, hogy megvizsgáljuk, a konkurensen végrehajtott tranzakciók milyen feltételekkel tudják megőrizni az adatbázis-állapot konzisztenciáját. Az alapfeltevésünk az volt, hogy ha minden egyes tranzakciót elkülönítve hajtunk végre (anélkül, hogy más tranzakció konkurensen futna), akkor az adatbázist konzisztens állapotból konzisztens állapotba alakítjuk (korrektség alapelve). A gyakorlatban azonban a tranzakciók általában más tranzakciókkal egyidejűleg futnak, emiatt ez az elv közvetlenül nem használható. Olyan ütemezéseket kell alkalmaznunk, amelyek biztosítják, hogy ugyanazt az eredményt állítják elő, mintha a tranzakciókat egyesével hajtottuk volna végre.

Ütemezések

Az *ütemezés* (schedule) egy vagy több tranzakció által végrehajtott lényeges műveletek időrendben vett sorozata, amelyben az egy tranzakcióhoz tartozó műveletek sorrendje megegyezik a tranzakcióban megadott sorrenddel. A konkurenciakezelés szempontjából a lényeges olvasási és írási műveletek a központi memória puffereiben történnek, nem pedig a lemezen. Tehát csak a `READ` és `WRITE` műveletek sorrendje számít, amikor a konkurenciával foglalkozunk, az `INPUT` és `OUTPUT` műveleteket figyelmen kívül hagyjuk.

Példa. Tekintsünk két tranzakciót és az adatbázison kifejtett hatásukat, amikor egy meghatározott sorrendben hajtjuk végre a műveleteiket:

T_1	T_2
`READ(A,t)`	`READ(A,s)`
`t := t+100`	`s := s*2`
`WRITE(A,t)`	`WRITE(A,s)`
`READ(B,t)`	`READ(B,s)`
`t := t+100`	`s := s*2`
`WRITE(B,t)`	`WRITE(B,s)`

`t` és `s` T_1-nek és T_2-nek lokális változói, nem adatbáziselemek. Tételezzük fel, hogy az egyetlen konzisztenciamegszorítás az `A = B`. Mivel T_1 `A`-hoz és `B`-hez is hozzáad 100-at, és T_2 `A`-t és `B`-t is megszorozza 2-vel, tudjuk, hogy az egyes tranzakciók egymástól elkülönítve futva megőrzik a konzisztenciát.

Soros ütemezések

Azt mondjuk, hogy egy ütemezés *soros* (serial schedule), ha benne bármely két T és T′ tranzakcióra teljesül, hogy ha T-nek van olyan művelete, amely megelőzi T′ valamelyik műveletét, akkor T összes művelete megelőzi T′ valamennyi műveletét. Másképpen fogalmazva az ütemezés úgy épül fel a tranzakciós műveletekből, hogy először az egyik tranzakció összes műveletét tartalmazza, azután egy másik tranzakció összes műveletét stb., miközben nem cseréli fel a műveleteket.

Példa. A fenti tranzakcióknak két soros ütemezése van, az egyikben T_1 megelőzi T_2-t, a másikban T_2 előzi meg T_1-et. Legyen a kezdeti állapot A = B = 25. Ekkor a két ütemezés a következőképpen alakul:

T_1	T_2	A	B
READ(A,t)		25	
t := t+100			
WRITE(A,t)		125	
READ(B,t)			25
t := t+100			
WRITE(B,t)			125
	READ(A,s)	125	
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		125
	s := s*2		
	WRITE(B,s)		250

T_1	T_2	A	B
	READ(A,s)	25	
	s := s*2		
	WRITE(A,s)	50	
	READ(B,s)		25
	s := s*2		
	WRITE(B,s)		50
READ(A,t)		50	
t := t+100			
WRITE(A,t)		150	
READ(B,t)			50
t := t+100			
WRITE(B,t)			150

Láthatjuk, hogy A és B végső értéke különböző a két ütemezésben, de nem is a végeredmény a központi kérdés addig, amíg a konzisztenciát megőrizzük. Általában nem várjuk el, hogy az adatbázis végső állapota független legyen a tranzakciók végrehajtásának sorrendjétől.

A soros ütemezést úgy ábrázolhatjuk, hogy a műveleteket a végrehajtásuk sorrendjében felsoroljuk. Mivel a soros ütemezésben a műveletek sorrendje csak magától a tranzakciók sorrendjétől függ, ezért a soros ütemezést elegendő a tranzakciók felsorolásával megadni, például: (T_1, T_2), illetve (T_2, T_1).

Sorba rendezhető ütemezések

A tranzakciókra vonatkozó korrektségi elv szerint minden soros ütemezés megőrzi az adatbázis konzisztenciáját. Kérdés, hogy van-e más ütemezés is, amely szintén biztosítja a konzisztencia megmaradását. A válasz igen, ahogy azt a következő példa mutatja. Általában azt mondjuk, hogy egy ütemezés *sorba rendezhető* (serializable schedule), ha ugyanolyan hatással van az adatbázis állapotára, mint ugyanazon tranzakciók valamelyik soros ütemezése, függetlenül az adatbázis kezdeti állapotától.

Példa. Tekintsük a fenti két tranzakció következő két ütemezését:

T_1	T_2	A	B
READ(A,t)		25	
t := t+100			
WRITE(A,t)		125	
	READ(A,s)	125	
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			25
t := t+100			
WRITE(B,t)			125
	READ(B,s)		125
	s := s*2		
	WRITE(B,s)		250

T_1	T_2	A	B
READ(A,t)		25	
t := t+100			
WRITE(A,t)		125	
	READ(A,s)	125	
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		25
	s := s*2		
	WRITE(B,s)		50
READ(B,t)			50
t := t+100			
WRITE(B,t)			150

Az első példa egy sorba rendezhető, de nem soros ütemezést ad meg. Ebben az ütemezésben T_2 azután van hatással A-ra, miután T_1 volt, de mielőtt T_1 hatással lenne B-re. Mégis azt látjuk, hogy a két tranzakció hatása megegyezik a (T_1, T_2) soros ütemezés hatásával. Ezt könnyű belátni tetszőleges konzisztens kiindulási állapotra: A = B = c-ből kiindulva A-nak is és B-nek is 2(c + 100) lesz az értéke, tehát a konzisztenciát mindig megőrizzük.

A második példában szereplő ütemezés viszont nem sorba rendezhető. Itt T_1 dolgozik előbb A-val, viszont T_2 dolgozik előbb B-vel, ennek hatásaként másképpen kell kiszámolnunk A-t és B-t: A := 2(A + 100), B := 2B + 100. Az ilyen viselkedést a különböző konkurenciavezérlési technikákkal el kell kerülnünk.

A tranzakció szemantikájának hatása

A sorbarendezhetőség eldöntéséhez eddig a tranzakciók műveleteinek a sorrendjét néztük meg. Azonban a tranzakciók részletei is számítanak, ahogyan ezt a következő példából láthatjuk:

Példa. Tekintsük az alábbi ütemezést, amely csak a T_2 által végrehajtott számításokban különbözik a legutolsó példánktól, mégpedig abban, hogy nem 2-vel szorozza meg A-t és B-t, hanem 1-gyel:

T_1	T_2	A	B
READ(A,t)		25	
t := t+100			
WRITE(A,t)		125	
	READ(A,s)	125	
	s := s*1		
	WRITE(A,s)	125	
	READ(B,s)		25
	s := s*1		
	WRITE(B,s)		25
READ(B,t)			25
t := t+100			
WRITE(B,t)			125

A és B értéke az ütemezés végén megegyezik, és könnyen ellenőrizhetjük, hogy a konzisztens kezdeti állapottól függetlenül a végállapot is konzisztens lesz. Valójában az egyetlen végállapot az, amelyet vagy a (T_1, T_2) vagy a (T_2, T_1) soros ütemezés eredményez.

Felmerülhet a kérdés, hogy mi értelme van a T_2 tranzakciónak. Valójában több elfogadható tranzakciót helyettesíthetnénk a helyére, amely A-t és B-t változatlanul hagyná. T_2 például lehetne olyan tranzakció, amely csak kiíratja A-t és B-t. Vagy a felhasználótól kérhet be adatokat, hogy kiszámoljon egy F tényezőt, amivel beszorozza A-t és B-t, és előfordulhat olyan felhasználói input, amelyre F = 1.

Sajnos az ütemező számára nem reális a tranzakciós számítások részleteinek figyelembevétele. Mivel a tranzakciók gyakran tartalmaznak általános célú programozási nyelven írt kódokat éppúgy, mint SQL nyelvű utasításokat, néha nagyon nehéz megválaszolni azokat a kérdéseket, mint például „ez a tranzakció A-t egy 1-től különböző értékkel szorozta-e meg”. Az ütemezőnek azonban látnia kell a tranzakciók olvasási és írási kéréseit, így tudhatja, hogy az egyes tranzakciók mely adatbáziselemeket olvasták be, és mely elemek változhattak meg. Az ütemező feladatának egyszerűsítésére megszokott a következő feltétel:

- Bármely A adatbáziselemnek egy T tranzakció olyan értéket ír be, amely az adatbázis-állapottól függ oly módon, hogy ne forduljon elő aritmetikai egybeesés.

Más szóval: ha T tudna A-ra olyan hatással lenni, hogy az adatbázis-állapot inkonzisztenssé váljon, akkor T ezt meg is teszi. Ezt a feltevést később pontosítjuk, amikor a sorbarendezhetőség biztosítására adunk meg feltételeket.

A tranzakciók és az ütemezések jelölése

Ha elfogadjuk, hogy egy tranzakció által végrehajtott pontos számítások tetszőlegesek lehetnek, akkor nem szükséges a helyi számítási lépések részleteit néznünk. Csak a tranzakciók által végrehajtott olvasások és írások számítanak, így a tranzakciókat és az ütemezéseket rövidebben jelölhetjük. Ekkor $r_T(X)$ és $w_T(X)$ tranzakcióműveletek, és azt jelentik, hogy a T tranzakció olvassa, illetve írja az X adatbáziselemet. Továbbá, mivel a tranzakcióinkat általában T_1, T_2, …-vel fogjuk jelölni, ezért megállapodunk abban, hogy $r_i(X)$ és $w_i(X)$ ugyanazt jelöli, mint $r_{Ti}(X)$, illetve $w_{Ti}(X)$.

Példa. A fenti példákban szereplő tranzakciók az alábbi módon írhatók fel:

T_1: $r_1(A); w_1(A); r_1(B); w_1(B);$
T_2: $r_2(A); w_2(A); r_2(B); w_2(B);$

Nem említettük sehol a t és s lokális változókat, és nem jelöltük azt sem, hogy mi történt a beolvasás után A-val és B-vel. Mindezt úgy értelmezzük, hogy az adatbáziselemek megváltozásában a „legrosszabbat fogjuk feltételezni”.

Másik példaként nézzük meg a T_1 és T_2 tranzakciók korábban felírt sorba rendezhető ütemezését:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

Pontosítva a jelölést:

1. Egy tranzakció *műveletét* $r_i(X)$ vagy $w_i(X)$ formában fejezzük ki, amely azt jelenti, hogy a T_i tranzakció olvassa, illetve írja az X adatbáziselemet.
2. Egy T_i *tranzakció* az i indexű műveletekből álló sorozat.
3. A T tranzakcióhalmaz egy S *ütemezése* olyan műveletek sorozata, amelyben minden T halmazbeli T_i tranzakcióra teljesül, hogy T_i műveletei ugyanabban a sorrendben fordulnak elő S-ben, mint ahogy magában a T_i definíciójában szerepeltek. Azt mondjuk, hogy S az őt alkotó tranzakciók műveleteinek *átlapolása* (interleaving).

Konfliktus-sorbarendezhetőség

Most egy olyan elégséges feltételt adunk meg, amely biztosítja egy ütemezés sorbarendezhetőségét. A forgalomban lévő rendszerek ütemezői a tranzakciók sorbarendezhetőségére általában ezt az erősebb feltételt biztosítják, amelyet *konfliktus-sorbarendezhetőségnek* nevezünk. Ez a *konfliktus* fogalmon alapul: A *konfliktus* (conflict) vagy *konfliktuspár* két olyan egymást követő művelet az ütemezésben, amelyeknek ha a sorrendjét felcseréljük, akkor legalább az egyik tranzakció viselkedése megváltozhat. Egy tranzakció két szomszédos műveletét mindig konfliktusnak tekintjük.

Konfliktusok

Vegyük észre, hogy a legtöbb műveletpár nincs konfliktusban a fenti értelemben. Legyen T_i és T_j két különböző tranzakció ($i \neq j$).

1. $r_i(X); r_j(Y);$ sohasem konfliktus, még akkor sem, ha $X = Y$, mivel egyik lépés sem változtatja meg az értékeket.
2. $r_i(X); w_j(Y);$ nincs konfliktusban, feltéve, hogy $X \neq Y$, mivel T_j írhatja Y-t, mielőtt T_i beolvasta X-et, X értéke ettől ugyanis nem változik. Annak sincs hatása T_j-re, hogy T_i olvassa X-et, ugyanis ez nincs hatással arra, hogy milyen értéket ír T_j Y-ba.
3. $w_i(X); r_j(Y);$ nincs konfliktusban, ha $X \neq Y$, ugyanazért, amiért a 2. pontban.

4. $w_i(X); w_j(Y);$ sincs konfliktusban, ha $X \neq Y$.

Másrészt három esetben nem cserélhetjük fel a műveletek sorrendjét:

a) Ugyanannak a tranzakciónak két művelete, például $r_i(X); w_i(Y);$ konfliktus, mivel egyetlen tranzakción belül a műveletek sorrendje rögzített, és az adatbázis-kezelő rendszer ezt a sorrendet nem rendezheti át.
b) Különböző tranzakciók ugyanarra az adatbáziselemre vonatkozó írása, például $w_i(X); w_j(X);$ konfliktus, mivel X értéke az marad, amit T_j számolt ki. Ha felcseréljük a sorrendjüket, akkor pedig X-nek a T_i által kiszámolt értéke marad meg. Az a feltevésünk, hogy „nincs egybeesés”, azt adja, hogy a T_i és a T_j által kiírt értékek lehetnek különbözőek, és ezért az adatbázis valamelyik kezdeti állapotára különbözni fognak.
c) Különböző tranzakciók által ugyanazon adatbáziselem olvasása és írása is konfliktus, azaz $r_i(X); w_j(X);$ és $w_i(X); r_j(X);$ is konfliktus. Ha átvisszük $w_j(X)$-et $r_i(X)$ elé, akkor a T_i által olvasott X-beli érték az lesz, amit a T_j kiírt, amiről pedig feltételeztük, hogy nem szükségképpen egyezik meg X korábbi értékével. Tehát $r_i(X)$ és $w_j(X)$ sorrendjének cseréje befolyásolja, hogy T_i milyen értéket olvas X-ből, ez pedig befolyásolja T_i működését.

Levonhatjuk a következtetést, hogy különböző tranzakciók bármely két műveletének sorrendje felcserélhető, hacsak nem:

1. ugyanarra az adatbáziselemre vonatkoznak, és
2. legalább az egyik művelet írás.

Ezt az elvet kiterjesztve tetszőleges ütemezést véve annyi nem konfliktusos cserét készíthetünk, amennyit csak kívánunk, abból a célból, hogy az ütemezést soros ütemezéssé alakítsuk át. Ha ezt meg tudjuk tenni, akkor az eredeti ütemezés sorba rendezhető volt, ugyanis az adatbázis állapotára való hatása változatlan marad minden nemkonfliktusos cserével.

Azt mondjuk, hogy két ütemezés *konfliktusekvivalens* (conflict-equivalent), ha szomszédos műveletek nemkonfliktusos cseréinek sorozatával az egyiket átalakíthatjuk a másikká. Azt mondjuk, hogy egy ütemezés *konfliktus-sorbarendezhető* (conflict-serializable schedule), ha konfliktusekvivalens ugyanazon tranzakciók valamely soros ütemezésével. A konfliktus-sorbarendezhetőség elégséges feltétele a sorbarendezhetőségnek, vagyis egy konfliktus-sorbarendezhető ütemezés sorba rendezhető ütemezés is egyben. Azonban a konfliktus-sorbarendezhetőség nem szükséges ahhoz, hogy egy ütemezés sorba rendezhető legyen, mégis általában ezt a feltételt ellenőrzik a forgalomban lévő rendszerek ütemezői, amikor a sorbarendezhetőséget kell biztosítaniuk.

Példa. Legyen az ütemezés a következő:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

Azt állítjuk, hogy ez az ütemezés konfliktus-sorbarendezhető. A következő cserékkel ez az ütemezés átalakítható a (T_1, T_2) soros ütemezéssé, ahol az összes T_1-beli művelet megelőzi az összes T_2-beli műveletet:

$r_1(A); w_1(A); r_2(A); \underline{w_2(A); r_1(B);} w_1(B); r_2(B); w_2(B);$
$r_1(A); w_1(A); \underline{r_2(A); r_1(B);} w_2(A); w_1(B); r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); r_2(A); \underline{w_2(A); w_1(B);} r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); \underline{r_2(A); w_1(B);} w_2(A); r_2(B); w_2(B);$
$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

Felmerül a kérdés, hogy miért nem szükséges a konfliktus-sorbarendezhetőség a sorbarendezhetőséghez. Korábban már láttunk erre egy példát, amikor a tranzakció szemantikáját figyelembe véve állapíthattuk csak meg a sorbarendezhetőséget. Akkor megnéztük, hogy a T_2 által végrehajtott speciális számítások miatt miért volt az ütemezés sorba rendezhető. Pedig az az ütemezés nem konfliktus-sorbarendezhető, ugyanis

A-t T_1 írja előbb, B-t pedig T_2. Mivel sem A írását, sem B írását nem lehet átrendezni, semmilyen módon nem kerülhet T_1 összes művelete T_2 összes művelete elé, sem fordítva.

Vannak olyan sorba rendezhető, de nem konfliktus-sorbarendezhető ütemezések is, amelyek nem függnek a tranzakciók által végrehajtott számításoktól. Tekintsük például a T_1, T_2 és T_3 tranzakciókat, amelyek mindegyike X értékét írja. T_1 és T_2 Y-nak is ír értéket, mielőtt X-nek írnának értéket. Az egyik lehetséges ütemezés, amely éppen soros is, a következő:

S_1: $w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$

Az S_1 ütemezés X értékének a T_3 által írt értéket, Y értékének pedig a T_2 által írt értéket adja. Ugyanezt végzi a következő ütemezés is:

S_2: $w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$

Intuíció alapján átgondolva annak, hogy T_1 és T_2 milyen értéket ír X-be, nincs hatása, ugyanis T_3 felülírja X értékét. Emiatt S_1 és S_2 X-nek is és Y-nak is ugyanazt az értéket adja. Mivel S_1 soros ütemezés, és S_2-nek bármely adatbázis-állapotra ugyanaz a hatása, mint S_1-nek, ezért S_2 sorba rendezhető. Ugyanakkor mivel nem tudjuk felcserélni $w_1(X)$-et $w_2(X)$-szel, így cseréken keresztül nem lehet S_2-t valamelyik soros ütemezéssé átalakítani. Tehát S_2 sorba rendezhető, de nem konfliktus-sorbarendezhető.

Megelőzési gráfok és teszt a konfliktus-sorbarendezhetőségre

Viszonylag könnyű megvizsgálnunk egy S ütemezést, és eldöntenünk, hogy konfliktus-sorbarendezhető-e vagy nem. Az az alapötlet, hogy ha valahol konfliktusban álló műveletek szerepelnek S-ben, akkor az ezeket a műveleteket végrehajtó tranzakcióknak ugyanabban a sorrendben kell előfordulniuk a konfliktusekvivalens soros ütemezésekben, mint ahogyan az S-ben voltak. Tehát a konfliktusban álló műveletpárok megszorítást adnak a feltételezett konfliktusekvivalens soros ütemezésben a tranzakciók sorrendjére. Ha ezek a megszorítások nem mondanak ellent egymásnak, akkor találhatunk konfliktusekvivalens soros ütemezést. Ha pedig ellentmondanak egymásnak, akkor tudjuk, hogy nincs ilyen soros ütemezés.

Adott a T_1 és T_2 ($T_1 \neq T_2$), esetleg további tranzakcióknak egy S ütemezése. Azt mondjuk, hogy T_1 *megelőzi* T_2-t, ha van a T_1-ben olyan A_1 művelet és a T_2-ben olyan A_2 művelet, hogy

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

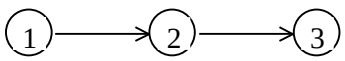
Másképpen fogalmazva: A_1 és A_2 konfliktuspárt alkotna, ha szomszédos műveletek lennének. Jelölése: $T_1 <_S T_2$. Látható, hogy ezek pontosan azok a feltételek, amikor nem lehet felcserélni A_1 és A_2 sorrendjét. Tehát A_1 az A_2 előtt szerepel bármely S-sel konfliktusekvivalens ütemezésben. Ebből az következik, hogy ha ezek közül az ütemezések közül az egyik soros ütemezés, akkor abban T_1-nek meg kell előznie T_2-t.

Ezeket a megelőzéseket a *megelőzési gráfban* (precedence graph) összegezhetjük. A megelőzési gráf csomópontjai az S ütemezés tranzakciói. Ha a tranzakciókat T_i-vel jelöljük, akkor a T_i-nek megfelelő csomópontot az i egésszel. Az i csomópontból a j csomópontba vezet irányított él, ha $T_i <_S T_j$.

Példa. A következő S ütemezés a T_1, T_2 és T_3 tranzakciókat tartalmazza:

S: $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

Az S ütemezéshez tartozó megelőzési gráf a következő:



Ha az A-val kapcsolatos műveleteket nézzük meg, akkor több okot találunk, hogy miért igaz a $T_2 <_S T_3$. Például $r_2(A)$ az S-ben $w_3(A)$ előtt áll, és $w_2(A)$ az $r_3(A)$ és a $w_3(A)$ előtt is áll. A három észrevételünk közül bármelyik elegendő, hogy igazoljuk, valóban vezet él 2-ből 3-ba a megelőzési gráfban. Hasonló módon ha megnézzük a B-vel kapcsolatos műveleteket, akkor szintén több okot találunk, hogy miért igaz a $T_1 <_S T_2$. Például az $r_1(B)$ művelet a $w_2(B)$ művelet előtt áll. Tehát az S megelőzési gráfban 1-ből 2-be szintén vezet él. Ez a két él és csak ez a két él az, amelyeket az S ütemezésben szereplő műveletek sorrendjéből tudunk ellenőrizni.

Van egy egyszerű szabály, amivel megmondhatjuk, hogy egy S ütemezés konfliktus-sorbarendezhető-e:

- Rajzoljuk fel S megelőzési gráfját, és nézzük meg, tartalmaz-e kört! Ha igen, akkor S nem konfliktus-sorbarendezhető, ha nem, akkor az, és ekkor a csomópontok bármelyik topologikus sorrendje megadja a konfliktusekvivalens soros sorrendet.

Egy körmentes gráf csomópontjainak *topologikus sorrendje* a csomópontok bármely olyan rendezése, amelyben minden $a \to b$ élre az a csomópont megelőzi a b csomópontot a topologikus rendezésben.

Példa. A fenti megelőzési gráf körmentes, így az S ütemezés konfliktus-sorbarendezhető. A csomópontoknak, azaz a tranzakcióknak csak egyetlen sorrendje van, amely konzisztens a gráf éleivel, ez pedig a (T_1, T_2, T_3). S-et tehát át lehet alakítani olyan ütemezéssé, amelyben a három tranzakció mindegyikének az összes művelete ugyanebben a sorrendben van, és ez a soros ütemezés:

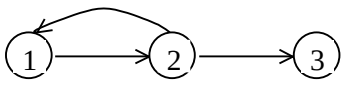
S': $r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B); r_3(A); w_3(A);$

Ahhoz, hogy belássuk, megkaphatjuk S-ből S'-t szomszédos elemek cseréjével, az első észrevételünk, hogy az $r_1(B)$-t konfliktus nélkül az $r_2(A)$ elé hozhatjuk. Ezután három cserével a $w_1(B)$-t közvetlenül az $r_1(B)$ utánra tudjuk vinni, ugyanis mindegyik közbeeső művelet az A-ra vonatkozik. Ezután az $r_2(B)$-t és a $w_2(B)$-t csak az A-ra vonatkozó műveleteken keresztül át tudjuk vinni pontosan a $w_2(A)$ utáni helyzetbe, amivel megkapjuk S'-t.

Példa. Tekintsük az alábbi ütemezést:

S_1: $r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

S_1 csak abban különbözik S-től, hogy az $r_2(B)$ művelet három hellyel előbb szerepel. Az A-ra vonatkozó műveleteket megvizsgálva most is csak a $T_2 <_{S1} T_3$ megelőzési kapcsolathoz jutunk. De ha B-t vizsgáljuk, akkor nemcsak $T_1 <_{S1} T_2$ teljesül (ugyanis $r_1(B)$ és $w_1(B)$ a $w_2(B)$ előtt szerepel), hanem $T_2 <_{S1} T_1$ is (ugyanis $r_2(B)$ a $w_1(B)$ előtt fordul elő). Emiatt az S_1 ütemezéshez tartozó megelőzési gráf a következő:



Ez a gráf nyilvánvalóan tartalmaz kört (ciklikus), ezért arra következtethetünk, hogy S_1 nem konfliktus-sorbarendezhető, ugyanis intuíció alapján láthatjuk, hogy bármely konfliktusekvivalens soros ütemezésben T_1-nek T_2 előtt is és után is kellene állnia, tehát nem létezik ilyen ütemezés.

Miért működik a megelőzési gráfon alapuló tesztelés?

Láttuk, hogy a megelőzési gráfban a kör túl sok megszorítást jelent a feltételezett konfliktusekvivalens soros ütemezésre nézve. Azaz ha létezik a $T_1 \to T_2 \to \ldots \to T_n \to T_1$ n darab tranzakcióból álló kör, akkor a feltételezett soros sorrendben T_1 műveleteinek meg kell előzniük a T_2-ben szereplő műveleteket, amelyeknek meg kell előzniük a T_3-belieket és így tovább egészen T_n-ig. De T_n műveletei emiatt a T_1-

beliek mögött vannak, ugyanakkor meg is kellene előzniük a T_1-belieket a $T_n \rightarrow T_1$ él miatt. Ebből következik, hogy ha a megelőzési gráf tartalmaz kört, akkor az ütemezés nem konfliktus-sorbarendezhető.

A másik irányt egy kicsit nehezebb belátnunk. Azt kell megmutatnunk, hogy amikor a megelőzési gráf körmentes, akkor az ütemezés műveletei átrendezhetők szomszédos műveletek szabályos cseréivel úgy, hogy az ütemezés egy soros ütemezéssé váljon. Ha ezt meg tudjuk tenni, akkor bebizonyítottuk, hogy minden körmentes megelőzési gráffal rendelkező ütemezés konfliktus-sorbarendezhető. A bizonyítás az ütemezésben részt vevő tranzakciók száma szerinti indukcióval történik:

Alapeset: Ha $n = 1$, vagyis csak egyetlen tranzakcióból áll az ütemezés, akkor az már önmagában soros, tehát biztosan konfliktus-sorbarendezhető.

Indukció: Legyen S a $T_1, T_2, \ldots, T_n$ n darab tranzakció műveleteiből álló ütemezés. Tételezzük fel, hogy S-nek körmentes megelőzési gráfja van. Ha egy véges gráf körmentes, akkor van legalább egy olyan csomópontja, amelybe nem vezet él. Legyen a T_i tranzakciónak megfelelő i csomópont egy ilyen csomópont. Mivel az i csomópontba nem vezet él, nincs S-ben olyan A művelet, amely

1. valamelyik T_j ($i \neq j$) tranzakcióra vonatkozik,
2. T_i valamely műveletét megelőzi, és
3. ezzel a művelettel konfliktusban van.

Ha lenne ilyen, akkor a megelőzési gráfban lennie kellene egy élnek a j csomópontból az i csomópontba (hiszen ekkor T_j megelőzi T_i-t), márpedig az i csomópontba nem vezet él.

Így lehetséges, hogy T_i minden műveletét S legelejére mozgatjuk át, miközben megtartjuk a sorrendjüket. Az ütemezés most a következő alakú:

(T_i műveletei) (a többi $n-1$ tranzakció műveletei)

Most tekintsük S második részét, vagyis a T_i-től különböző összes tranzakciónak a műveleteit. Mivel ezek a műveletek egymáshoz viszonyítva ugyanabban a sorrendben vannak, mint ahogyan S-ben voltak, ennek a második résznek a megelőzési gráfját megkapjuk S megelőzési gráfjából, ha elhagyjuk belőle az i csomópontot és az ebből a csomópontból kimenő éleket.

Mivel az eredeti megelőzési gráf körmentes volt, és csomópontok, illetve élek törlésével nem válhatott ciklikussá, ezért a második rész megelőzési gráfja is körmentes. Továbbá, mivel a második része $n-1$ tranzakciót tartalmaz, alkalmazzuk rá az indukciós feltevést. Így tudjuk, hogy a második rész műveletei szomszédos műveletek szabályos cseréivel átrendezhetők soros ütemezéssé. Ily módon magát S-et alakítottuk át olyan soros ütemezéssé, amelyben T_i műveletei állnak legelöl, és a többi tranzakció műveletei ezután következnek valamilyen soros sorrendben. Az indukciót beláttuk, és így következik, hogy minden olyan ütemezés, amelynek körmentes a megelőzési gráfja, konfliktus-sorbarendezhető.

A sorbarendezhetőség biztosítása zárakkal

Képzeljünk el egy olyan tranzakcióhalmazt, amely megszorítások nélkül hajtja végre a műveleteit. Ezek a műveletek is egy ütemezést alkotnak, de nem valószínű, hogy ez az ütemezés sorba rendezhető lenne. Az ütemező feladata az, hogy megakadályozza az olyan műveleti sorrendeket, amelyek nem sorba rendezhető ütemezésekhez vezetnek. Először az ütemező legáltalánosabb szerkezetét tekintjük, olyat, amelyben az adatbáziselemekre kiadott *zárak* (lock) akadályozzák meg a nem sorba rendezhető viselkedést. Röviden arról van szó, hogy a tranzakciók zárolják azokat az adatbáziselemeket, amelyekhez hozzáférnek, hogy megakadályozzák azt, hogy ugyanakkor más tranzakciók is hozzáférjenek ezekhez az elemekhez, mivel ekkor felmerülne a nem sorbarendezhetőség kockázata.

Először egy leegyszerűsített zárolási sémával vezetjük be a zárolás fogalmát. Ebben a sémában csak egyféle zár van, amelyet a tranzakcióknak meg kell kapniuk az adatbáziselemre, ha bármilyen műveletet végre akarnak hajtani rajta. Később sokkal valósabb zárolási sémákat tanulmányozunk, különböző zármódokkal.

Zárak

Az ütemező felelős azért, hogy fogadja a tranzakcióktól érkező kéréseket, és vagy megengedi a műveletek végrehajtását, vagy addig késlelteti, amikor már biztonságosan végre tudja hajtani őket. Nézzük meg, hogyan irányítja ezt a döntést a *zártábla* (lock table) felhasználásával.

Az lenne az ideális, ha az ütemező akkor és csak akkor továbbítana egy kérést, ha annak végrehajtása nem vezethetne inkonzisztens adatbázis-állapothoz, miután az összes aktív tranzakciót vagy véglegesen végrehajtottuk, vagy abortáltuk (vagyis sikertelenül befejeztük). Ezt a kérdést viszont túl nehéz lenne valós időben eldönteni. Így minden ütemező csak egy egyszerű tesztet hajt végre a sorbarendezhetőség eldöntésére, azonban letilthat olyan műveleteket is, amelyek önmagukban nem vezetnének inkonzisztenciához. A *zárolási ütemező*, mint a legtöbb ütemező, a konfliktus-sorbarendezhetőséget követeli meg, pedig – mint azt már láttuk – ez erősebb követelmény, mint a sorbarendezhetőség.

Ha az ütemező zárakat használ, akkor a tranzakcióknak – az adatbáziselemek olvasásán és írásán felül – zárakat kell kérniük és feloldaniuk. A zárak használatának két értelemben is helyesnek kell lennie: mind a tranzakciók szerkezetére, mind pedig az ütemezések szerkezetére alkalmazva:

- *Tranzakciók konzisztenciája* (consistency of transactions): 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 később azt fel kell szabadítania.
- *Az ütemezések jogszerűsége* (legality of schedules): 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.

Kibővítjük a jelöléseinket a zárolás és a feloldás műveletekkel:

$l_i(X)$: a T_i tranzakció az X adatbáziselemre *zárolást kér* (lock).
$u_i(X)$: a T_i tranzakció az X adatbáziselem *zárolását feloldja* (unlock).

Így a tranzakciók konzisztenciafeltétele és az ütemezések jogszerűségének a feltétele a következőképpen is kimondható:

- Ha egy T_i tranzakcióban van egy $r_i(X)$ vagy egy $w_i(X)$ művelet, akkor van korábban egy $l_i(X)$ művelet, és van később egy $u_i(X)$ művelet, de a zárolás és az írás/olvasás között nincs $u_i(X)$.
- Ha egy ütemezésben van olyan $l_i(X)$ művelet, amelyet $l_j(X)$ követ, akkor e két művelet között lennie kell egy $u_i(X)$ műveletnek.

Példa. Tekintsük a legelső példánkat, amelyben T_1 hozzáad az A és B adatbáziselemekhez 100-at, T_2 pedig megduplázza az értéküket. Most úgy adjuk meg a tranzakciókat, hogy a zárolási és az aritmetikai műveleteket is leírjuk, bár rendszerint a számításokat nem ábrázoljuk ebben a jelölésben, ugyanis az ütemező sem tudja azt figyelembe venni, amikor arról dönt, hogy engedélyezze vagy elutasítsa a kéréseket:

T_1: $l_1(A); r_1(A); A := A+100; w_1(A); u_1(A); l_1(B); r_1(B); B := B+100; w_1(B); u_1(B);$
T_2: $l_2(A); r_2(A); A := A*2; w_2(A); u_2(A); l_2(B); r_2(B); B := B*2; w_2(B); u_2(B);$

Mindkét tranzakció konzisztens. Mindkettő felszabadítja az A-ra és B-re kiadott zárakat. Továbbá mindkettő csak olyan lépésekben dolgozik A-n és B-n, melyeket megelőzően már zárolták az elemet, és még nem oldották fel a zár alól.

T_1	T_2	A	B
$l_1(A)$; $r_1(A)$;		25	
A := A+100;			
$w_1(A)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;	125	
	A := A*2;		
	$w_2(A)$; $u_2(A)$;	250	
	$l_2(B)$; $r_2(B)$;		25
	B := B*2;		
	$w_2(B)$; $u_2(B)$;		50
$l_1(B)$; $r_1(B)$;			50
B := B+100;			
$w_1(B)$; $u_1(B)$;			150

Az ábrán a két tranzakciónak egy jogszerű ütemezése látható, ugyanis a két tranzakció sohasem zárolja egyidejűleg A-t vagy B-t. Pontosabban: T_2 nem végzi el az $l_2(A)$ műveletet, csak miután T_1 végrehajtotta $u_1(A)$-t, és T_1 nem végzi el az $l_1(B)$ műveletet, csak miután T_2 végrehajtotta $u_2(B)$-t. Láthatjuk a kiszámított értékek nyomon követésével, hogy bár ez az ütemezés jogszerű, mégsem sorba rendezhető. Nemsokára látni fogunk egy további feltételt (a kétfázisú zárolást), amivel biztosíthatjuk, hogy a jogszerű ütemezések konfliktus-sorbarendezhetők legyenek.

A zárolási ütemező

A zároláson alapuló ütemező feladata, hogy akkor és csak akkor engedélyezze a kérések végrehajtását, ha azok jogszerű ütemezéseket eredményeznek. Ezt a döntést segíti a *zártábla*, amely minden adatbáziselemhez megadja azt a tranzakciót, ha van ilyen, amelyik pillanatnyilag zárolja az adott elemet. A zártábla szerkezetéről később lesz szó. Ha viszont csak egyféle zárolás van, mint ahogyan eddig feltételeztük, akkor úgy tekinthetjük a táblát, mint (X,T) párokból álló Zárolások(elem, tranzakció) relációt, ahol a T tranzakció zárolja az X adatbáziselemet. Az ütemezőnek csak le kell kérdeznie ezt a relációt, illetve egyszerű INSERT és DELETE utasításokkal kell módosítania.

Példa. A fenti példában látható ütemezés jogszerű, így a zárolási ütemező engedélyezhetné az összes kérést abban a sorrendben, ahogyan beérkeznek. Néha azonban előfordulhat, hogy nem lehet engedélyezni a kéréseket. Hajtsunk végre a T_1 és T_2 tranzakciókon egy apró, de lényeges változtatást, mégpedig azt, hogy T_1 és T_2 is előbb zárolja B-t, és csak azután oldja fel A zárolását:

T_1: $l_1(A); r_1(A); A := A+100; w_1(A); l_1(B); u_1(A); r_1(B); B := B+100; w_1(B); u_1(B);$
T_2: $l_2(A); r_2(A); A := A*2; w_2(A); l_2(B); u_2(A); r_2(B); B := B*2; w_2(B); u_2(B);$

T_1	T_2	A	B
$l_1(A)$; $r_1(A)$;		25	
A := A+100;			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;	125	
	A := A*2;		
	$w_2(A)$;	250	
	$l_2(B)$; **elutasítva**		
$r_1(B)$; B := B+100;			25
$w_1(B)$; $u_1(B)$;			125
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		125
	B := B*2;		
	$w_2(B)$; $u_2(B)$;		250

Az ábrán látható, hogy amikor a módosított ütemezésben T_2 kéri B zárolását, az ütemezőnek el kell utasítania ezt a kérést, hiszen T_1 még zárolja B-t. Így T_2 áll, és a következő műveleteket a T_1 tranzakció végzi. Végül T_1 végrehajtja $u_1(B)$-t, amely felszabadítja B-t. T_2 most már zárolhatja B-t, amelyet a következő lépésben végre is hajt. Látható, hogy mivel T_2-nek várakoznia kellett, ezért B-t akkor szorozza meg 2-vel, miután T_1 már hozzáadott 100-at, és ez konzisztens adatbázis-állapotot eredményez.

A kétfázisú zárolás

Van egy meglepő feltétel, amellyel biztosítani tudjuk, hogy konzisztens tranzakciók jogszerű ütemezése konfliktus-sorbarendezhető legyen. Ezt a feltételt, amelyet a gyakorlatban elterjedt zárolási rendszerek leginkább követnek, *kétfázisú zárolásnak* (two-phase locking, 2PL) nevezzük:

- Minden tranzakcióban minden zárolási művelet megelőzi az összes zárfeloldási műveletet.

A „két fázis" abból adódik, hogy az első fázisban csak zárolásokat adunk ki, a második fázisban pedig csak megszüntetünk zárolásokat. A kétfázisú zárolás – a konzisztenciához hasonlóan – a tranzakcióban a műveletek sorrendjére egy feltétel. Azt a tranzakciót, amely eleget tesz a 2PL feltételnek, *kétfázisú zárolású tranzakciónak* (two-phase-locked transaction) vagy *2PL tranzakciónak* nevezzük.

Példa. Az első példánkban a tranzakciók nem tesznek eleget a kétfázisú zárolási szabálynak. Például T_1 előbb oldja fel A zárolását, mint zárolja B-t. A második példában található tranzakciók azonban már eleget tesznek a 2PL feltételnek. Látható, hogy mind T_1, mind T_2 A-t és B-t is az első öt műveleten belül zárolja, és a következő öt műveleten belül feloldja a zárakat. Ha összehasonlítjuk a két ábrát, azt is látjuk, hogy a kétfázisú zárolású tranzakciók hogyan működnek együtt az ütemezővel a konzisztencia biztosítására, míg a nem 2PL tranzakciók esetén előfordulhat inkonzisztencia.

Miért működik a kétfázisú zárolás?

Igaz, bár közel sem nyilvánvaló, hogy a 2PL példánkban észlelt előnyei általában is érvényesek. Intuíció alapján mindegyik kétfázisú zárolású tranzakcióról azt gondolhatjuk, hogy rögtön végrehajtásra kerülnek, amint az első zárfeloldási kérés kiadásra kerül. A 2PL tranzakciók egy S ütemezésével konfliktusekvivalens soros ütemezésben a tranzakciók ugyanabban a sorrendben vannak, mint amilyenben az első zárfeloldásaik.

Megnézzük, hogyan lehet konzisztens, kétfázisú zárolású tranzakciók bármely S jogszerű ütemezését átalakítani konfliktusekvivalens soros ütemezéssé. A konverziót legjobban az S-ben részt vevő tranzakciók száma (n) szerinti indukcióval tudjuk leírni. Lényeges, hogy a konfliktusekvivalencia csak az olvasási és írási műveletekre vonatkozik. Amikor felcseréljük az olvasások és írások sorrendjét, akkor figyelmen kívül hagyjuk a zárolási és zárfeloldási műveleteket. Amikor megkaptuk az olvasási és írási műveletek sorrendjét, akkor úgy helyezzük el köréjük a zárolási és zárfeloldási műveleteket, ahogyan azt a különböző tranzakciók megkövetelik. Mivel minden tranzakció felszabadítja az összes zárolást a tranzakció befejezése előtt, tudjuk, hogy a soros ütemezés jogszerű lesz.

Alapeset: Ha $n = 1$, vagyis csak egyetlen tranzakcióból áll az ütemezés, akkor az már önmagában soros, tehát biztosan konfliktus-sorbarendezhető.

Indukció: Legyen S a T_1, T_2, …, T_n n darab konzisztens, kétfázisú zárolású tranzakció műveleteiből álló ütemezés, és legyen T_i az a tranzakció, amelyik a teljes S ütemezésben a legelső zárfeloldási műveletet végzi, mondjuk $u_i(X)$-t. Azt állítjuk, hogy T_i összes olvasási és írási műveletét az ütemezés legelejére tudjuk vinni anélkül, hogy konfliktusműveleteken kellene áthaladnunk.

Tekintsük T_i valamelyik műveletét, mondjuk $w_i(Y)$-t. Megelőzheti-e ezt S-ben valamely konfliktusművelet, például $w_j(Y)$? Ha így lenne, akkor az S ütemezésben az $u_j(Y)$ és az $l_i(Y)$ műveletek az alábbi módon helyezkednének el a műveletsorozatban:

$\ldots; w_j(Y); \ldots; u_j(Y); \ldots; l_i(Y); \ldots; w_i(Y); \ldots$

Mivel T_i az első, amelyik zárat old fel, így S-ben $u_i(X)$ megelőzi $u_j(Y)$-t, vagyis S a következőképpen néz ki:

$\ldots; w_j(Y); \ldots; u_i(X); \ldots; u_j(Y); \ldots; l_i(Y); \ldots; w_i(Y); \ldots$

Az $u_i(X)$ művelet állhat $w_j(Y)$ előtt is. Mindkét esetben $u_i(X)$ $l_i(Y)$ előtt van, ami azt jelenti, hogy T_i nem kétfázisú zárolású, amint azt feltételeztük. Ahogyan beláttuk, hogy nem létezhetnek konfliktuspárok az írásra, ugyanúgy be lehet látni bármely két lehetséges műveletre – az egyiket T_i-ből, a másikat pedig egy T_i-től különböző T_j-ből választva –, hogy nem lehetnek konfliktuspárok.

Bebizonyítottuk, hogy valóban S legelejére lehet vinni T_i összes műveletét konfliktusmentes olvasási és írási műveletekből álló műveletpárok cseréjével. Ezután elhelyezhetjük T_i zárolási és zárfeloldási műveleteit. Így S a következő alakba írható át:

(T_i műveletei) (a többi $n-1$ tranzakció műveletei)

Az $n-1$ tranzakcióból álló második rész szintén konzisztens 2PL tranzakciókból álló jogszerű ütemezés, így alkalmazhatjuk rá az indukciós feltevést. Átalakítjuk a második részt konfliktusekvivalens soros ütemezéssé, így a teljes S konfliktus-sorbarendezhetővé vált.

A holtpont kockázata

Az egyik probléma, amelyet nem lehet a kétfázisú zárolással megoldani, a *holtpontok* (deadlock) bekövetkezésének a lehetősége, vagyis amikor az ütemező arra kényszeríti a tranzakciókat, hogy „örökké” várakozzanak egy olyan adatbáziselemre vonatkozó zárra, amelyet egy másik tranzakció tart zárolva. Példaként tekintsük a megszokott 2PL tranzakcióinkat, de most T_2 A előtt dolgozza fel B-t:

T_1: $l_1(A); r_1(A); A := A+100; w_1(A); l_1(B); u_1(A); r_1(B); B := B+100; w_1(B); u_1(B);$
T_2: $l_2(B); r_2(B); B := B*2; w_2(B); l_2(A); u_2(B); r_2(A); A := A*2; w_2(A); u_2(A);$

A tranzakciós műveletek egy lehetséges végrehajtása a következő:

T_1	T_2	A	B
$l_1(A);\ r_1(A);$		25	
	$l_2(B);\ r_2(B);$		25
A := A+100;			
	B := B*2;		
$w_1(A);$		125	
	$w_2(B);$		50
$l_1(B);$ **elutasítva**	$l_2(A);$ **elutasítva**		

Most egyik tranzakció sem folytatódhat, hanem örökké várakozniuk kell. Látható, hogy nem tudjuk mind a két tranzakciót folytatni, ugyanis ha így lenne, akkor az adatbázis végső állapotában nem teljesülhetne $A = B$.

Ha a holtpont már bekövetkezett, akkor általában nem lehet a helyzeten úgy javítani, hogy minden tranzakció továbbléphessen, azaz legalább egy tranzakciót vissza kell görgetni: abortálni kell, majd újraindítani.

A holtpontkezelés problémája két fő irányból közelíthető meg: vagy valahogy rájövünk, hogy néhány tranzakció holtpontra jutott, és ebből a helyzetből keresünk kiutat (*holtpontérzékelés*), vagy már eleve úgy kezeljük a tranzakciókat, hogy soha ne juthassanak holtpontra (*holtpontmegelőzés*).

A holtpontok érzékelésére és feloldására a legegyszerűbb megoldást az *időtúllépés* (timeout) módszere adja. Időkorlátot vezetünk be, amely arra vonatkozik, hogy az egyes tranzakciók mennyi ideig lehetnek aktívak, és ha ezt a határt túllépik, akkor visszagörgetjük őket. Például egy egyszerű rendszerben, ahol a tipikus tranzakciók ezredmásodpercek alatt lefutnak, az egyperces időkorlátnak tényleg csak a holtpontra jutott tranzakciókra lenne hatása. De ha van néhány összetettebb tranzakció is, akkor az időtúllépés bekövetkezéséhez hosszabb időt választhatunk.

Vegyük észre, hogy ha a holtpontra jutott tranzakció túllépi az időkorlátját, akkor a többi erőforrással együtt az eddig birtokolt zárjairól is lemond. Így tehát van esély arra, hogy a holtponton álló többi tranzakció még azelőtt be tudja fejezni a tevékenységét, mielőtt kifutna az időből. De mivel a holtpontra jutott tranzakciók valószínűleg körülbelül ugyanabban az időpontban indultak (különben az egyik befejeződött volna, még mielőtt a másik elkezdődik), az is lehetséges, hogy a rendszer hamis időtúllépéseket érzékel, azaz úgy görgeti vissza a tranzakciókat, hogy azok már túljutottak a közös holtponton.

A holtpontérzékelésnek egy kifinomultabb módszere a *várakozási gráf* (waits-for graph) használata, amelyben azt tartjuk nyilván, hogy melyik tranzakció melyik másik tranzakció által birtokolt zárakra vár. Ezt a módszert nemcsak a már kialakult holtpontok érzékelésére, hanem azok kialakulásának megelőzésére is használhatjuk. Mi most az utóbbit tekintjük, ami azzal jár, hogy a várakozási gráfot egész idő alatt nyilván kell tartanunk, és az olyan műveleteket, amelyek következtében a gráfban kör alakulna ki, nem szabad megengednünk.

Látni fogjuk, hogy a zártáblában minden `X` adatbáziselemhez létezik egy lista, amelyben azon tranzakciók mellett, amelyek arra várnak, hogy zárolhassák `X`-et, azok is fel vannak sorolva, amelyek rendelkeznek `X` zárjával. A várakozási gráf csúcsai a listában található tranzakcióknak felelnek meg. A gráfban irányított él fut `T`-ből `U`-ba, ha létezik olyan `A` adatbáziselem, melyre

1. `U` zárolja `A`-t,
2. `T` arra vár, hogy zárolhassa `A`-t, és
3. `T` csak akkor kapja meg `A` zárját, ha először `U` lemond róla.

Ha nincsen (irányított) kör a gráfban, akkor végül minden tranzakció be tudja fejezni a működését. Lesz legalább egy olyan tranzakció, amelyik nem vár semelyik másikra, így ez biztosan befejeződhet. Ekkor viszont megint lesz legalább egy tranzakció, amelyik nem várakozik, ezért továbbléphet, és így tovább.

Ha azonban a gráf nem körmentes, akkor a körben részt vevő tranzakciók nem léphetnek tovább, azaz holtpontra jutottak. A holtpontmegelőzési stratégia tehát abból áll, hogy minden olyan tranzakciót visszagörgetünk, amelynek valami olyan igénye van, ami kört idézne elő a várakozási gráfban.

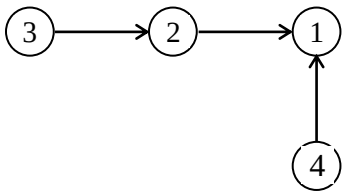
Példa. Tegyük fel, hogy az alábbi négy tranzakcióval rendelkezünk, amelyek mindegyike először olvas egy adatbáziselemet, majd ír egy másikat:

T_1: $l_1(A); r_1(A); l_1(B); w_1(B); u_1(A); u_1(B);$
T_2: $l_2(C); r_2(C); l_2(A); w_2(A); u_2(C); u_2(A);$
T_3: $l_3(B); r_3(B); l_3(C); w_3(C); u_3(B); u_3(C);$
T_4: $l_4(D); r_4(D); l_4(A); w_4(A); u_4(D); u_4(A);$

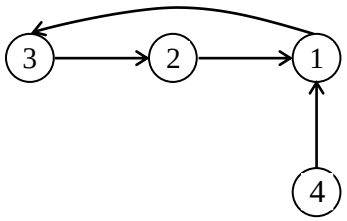
Lépés	T_1	T_2	T_3	T_4
1)	$l_1(A)$; $r_1(A)$;			
2)		$l_2(C)$; $r_2(C)$;		
3)			$l_3(B)$; $r_3(B)$;	
4)				$l_4(D)$; $r_4(D)$;
5)		$l_2(A)$; **elutasítva**		
6)			$l_3(C)$; **elutasítva**	
7)				$l_4(A)$; **elutasítva**
8)	$l_1(B)$; **elutasítva**			

A fenti ábrán egy lehetséges ütemezés kezdeti szakasza látható. Az első négy lépésben mindegyik tranzakció zárolja azt az elemet, amelyet olvasni szeretne. Az 5) lépésben T_2 megpróbálja zárolni A-t, de nem tudja, mert a zár már T_1 birtokában van. T_2 tehát várakozik T_1-re, ezért a várakozási gráfba berajzolunk egy élt a T_2-nek megfelelő csúcsból a T_1-nek megfelelő csúcs felé.

Hasonlóan, a 6) lépésben T_3 nem tudja zárolni C-t T_2 miatt, a 7) lépésben pedig T_4 vall kudarcot A zárolásával T_1 miatt. Az ebben az állapotban egyelőre körmentes várakozási gráf a következő:

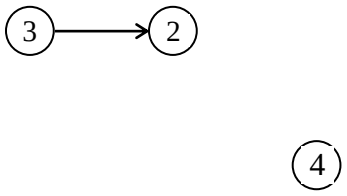


A 8) lépésben T_1-nek várnia kell B zárolásával T_3 miatt. Ha megengednénk T_1-nek, hogy várjon erre a zárra, akkor T_1, T_2 és T_3 mentén kör jönne létre a várakozási gráfban, ahogy ezt az alábbi ábra is mutatja:



Mivel a körben mindegyik tranzakció arra vár, hogy a másik befejeződjön, egyik sem tud továbblépni, vagyis ennek a három tranzakciónak a részvételével holtpont alakul ki. Véletlen egybeesés, hogy T_4 sem fejeződhet be annak ellenére, hogy nincs benne a körben. Az ő előrejutása ugyanis T_1 továbblépésén múlik.

Mivel a kört okozó tranzakciókat visszagörgetjük, így teszünk T_1-gyel is. A várakozási gráf a következőképpen alakul:



T_1 feloldja A zárolását, amelyet vagy T_2, vagy T_4 vesz át. Tegyük fel, hogy a zár T_2 birtokába kerül. T_2 befejeződik, ezáltal feloldódik a zár A-n és C-n. Most T_3, amely C-t akarja zárolni, és T_4 is, amely A-t, lezárulhat. Valamivel később T_1-et újraindítjuk, de nem kaphatja meg sem A, sem B zárját, amíg T_2, T_3 és T_4 be nem fejeződött.

Különböző zármódú zárolási rendszerek

A fentebb vázolt zárolási séma bemutatja a zárolás mögött álló legfőbb elveket, de túl egyszerű ahhoz, hogy a gyakorlatban is használható séma legyen. Az a legfőbb probléma, hogy a T tranzakciónak akkor is zárolnia kell az X adatbáziselemet, ha csak olvasni akarja X-et, írni nem. Nem kerülhetjük el a zárolást ekkor sem, mert 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 vagy más konkurenciavezérlési működésre. Mint már említettük, továbbra is szükséges azt az elemet is zárolni, amelyet olvasunk, ugyanis 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.

Ez indokolja, hogy bevezessük a legelterjedtebb zárolási sémát, amely két különböző zárat alkalmaz: az *osztott zárakat* (shared locks) vagy *olvasási zárakat*, és a *kizárólagos zárakat* (exclusive locks) vagy *írási zárakat*. Intuíció alapján 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 megfelel. Feltételezzük, hogy ha olvasni akarjuk X-et, de írni nem, akkor előnyben részesítjük az osztott zárolást.

Az $sl_i(X)$ jelölést használjuk arra, hogy a T_i tranzakció osztott zárat kér az X adatbáziselemre, az $xl_i(X)$ jelölést pedig arra, hogy a T_i kizárólagos zárat kér X-re. Továbbra is $u_i(X)$-szel jelöljük, hogy T_i feloldja X zárását, vagyis felszabadítja X-et minden zár alól.

Az előzőekben tárgyalt három követelmény (a tranzakciók konzisztenciája, a tranzakciók 2PL feltétele és az ütemezések jogszerűsége) mindegyikének van megfelelője az osztott/kizárólagos zárolási rendszerben:

1. *Tranzakciók konzisztenciája*: Nem írhatunk kizárólagos zár fenntartása nélkül, és nem olvashatunk valamilyen zár fenntartása nélkül. Pontosabban fogalmazva: bármely T_i tranzakcióban
   a) az $r_i(X)$ olvasási műveletet meg kell, hogy előzze egy $sl_i(X)$ vagy egy $xl_i(X)$ úgy, hogy közben nincs $u_i(X)$;
   b) a $w_i(X)$ írási műveletet meg kell, hogy előzze egy $xl_i(X)$ úgy, hogy közben nincs $u_i(X)$.

   Minden zárolást követnie kell egy ugyanannak az elemnek a zárolását feloldó műveletnek.
2. *Tranzakciók kétfázisú zárolása*: A zárolásoknak meg kell előzniük a zárak feloldását. Pontosabban fogalmazva: bármely T_i kétfázisú zárolású tranzakcióban egyetlen $sl_i(X)$ vagy $xl_i(X)$ műveletet sem előzhet meg egyetlen $u_i(Y)$ művelet sem semmilyen Y-ra.
3. *Az ütemezések jogszerűsége*: Egy elemet vagy egyetlen tranzakció zárol kizárólagosan, vagy több is zárolhatja osztottan, de a kettő egyszerre nem lehet. Pontosabban fogalmazva:
   a) Ha $xl_i(X)$ szerepel egy ütemezésben, akkor ezután nem következhet $xl_j(X)$ vagy $sl_j(X)$ valamely i-től különböző j-re anélkül, hogy közben ne szerepelne $u_i(X)$.
   b) Ha $sl_i(X)$ szerepel egy ütemezésben, akkor ezután nem következhet $xl_j(X)$ valamely i-től különböző j-re anélkül, hogy közben ne szerepelne $u_i(X)$.

Az engedélyezett, hogy egy tranzakció ugyanazon elemre kérjen és tartson mind osztott, mind kizárólagos zárat, feltéve, hogy ezzel nem kerül konfliktusba más tranzakciók zárolásaival. Ha a tranzakciók előre tudnák, milyen zárakra lesz szükségük, akkor biztosan csak a kizárólagos zárolást kérnék, de ha nem láthatók előre a zárolási igények, lehetséges, hogy egy tranzakció osztott és kizárólagos zárakat is kér különböző időpontokban.

Példa. Tekintsük az alábbi, osztott és kizárólagos zárakat használó két tranzakciónak egy lehetséges ütemezését:

T_1: $sl_1(A); r_1(A); xl_1(B); r_1(B); w_1(B); u_1(A); u_1(B);$
T_2: $sl_2(A); r_2(A); sl_2(B); r_2(B); u_2(A); u_2(B);$

T_1 is és T_2 is olvassa A-t és B-t, de csak T_1 írja B-t, és egyik sem írja A-t.

T_1	T_2
$sl_1(A);\ r_1(A);$	
	$sl_2(A);\ r_2(A);$
	$sl_2(B);\ r_2(B);$
$xl_1(B);$ **elutasítva**	
	$u_2(A);\ u_2(B);$
$xl_1(B);\ r_1(B);\ w_1(B);$	
$u_1(A);\ u_1(B);$	

Az ábrán T_1 és T_2 műveleteinek olyan ütemezése látható, amelyet T_1 kezd A osztott zárolásával. Ezután T_2 következik, A és B mindegyikét osztottan zárolja. Most T_1-nek lenne szüksége B kizárólagos zárolására, ugyanis olvassa is és írja is B-t. Viszont nem kaphatja meg a kizárólagos zárat, hiszen T_2-nek már osztott zárja van B-n. Így az ütemező várakozni kényszeríti T_1-et. Végül T_2 feloldja B zárját, és ekkor T_1 befejeződhet.

A vázolt ütemezés konfliktus-sorbarendezhető. A konfliktusekvivalens soros sorrend a (T_2, T_1), hiába kezdődött T_1 előbb. Nem bizonyítjuk, de konzisztens 2PL tranzakciók jogszerű ütemezése konfliktus-sorbarendezhető; ugyanazok a meggondolások alkalmazhatók az osztott és kizárólagos zárakra is, mint korábban. Az ábrán T_2 előbb old fel zárat, mint T_1, így azt várjuk, hogy T_2 megelőzi T_1-et a soros sorrendben. Megvizsgálva az olvasási és írási műveleteket, észrevehető, hogy $r_1(A)$-t T_2 összes műveletén át ugyan hátra tudjuk cserélgetni, de $w_1(B)$-t nem tudjuk $r_2(B)$ elé vinni, ami pedig szükséges lenne ahhoz, hogy T_1 megelőzze T_2-t egy konfliktusekvivalens soros ütemezésben.

Kompatibilitási mátrixok

Ha több zármódot használunk, akkor az ütemezőnek valamilyen elvre van szüksége ahhoz, hogy mikor engedélyezzen egy zárolási kérést, ha már adva vannak más zárak is azon az adatbáziselemen. Bár az osztott/kizárólagos rendszerek egyszerűek, a gyakorlatban léteznek a zárolási módoknak összetettebb rendszerei is. A zárolást engedélyező elvek következő fogalmait előbb az egyszerű osztott/kizárólagos rendszerek környezetében vezetjük be.

A *kompatibilitási mátrix* minden egyes zármódhoz rendelkezik egy-egy sorral és egy-egy oszloppal. A sorok egy másik tranzakció által az X elemre elhelyezett záraknak, az oszlopok pedig az X-re kért zármódoknak felelnek meg. A kompatibilitási mátrix használatának szabálya a zárolást engedélyező döntésekre az alábbi:

- Egy X adatbáziselemre C módú zárat akkor és csak akkor engedélyezhetünk, ha a táblázat minden olyan R sorára, amelyre más tranzakció már zárolta X-et R módban, a C oszlopban „igen” szerepel.

Példa. Az ábrán osztott (S) és kizárólagos (X) zárak kompatibilitási mátrixa látható:

	S	X
S	igen	nem
X	nem	nem

Az S oszlop azt mondja meg, hogy akkor engedélyezhetünk osztott zárat egy elemre, ha arra az elemre jelenleg is legfeljebb csak osztott zárak vannak. Az X oszlop azt mondja meg, hogy csak akkor engedélyezhetünk kizárólagos zárat, ha jelenleg nincs más zár az elemen. Látható, hogy ezek a szabályok az ütemezések jogszerűségének a definícióját tükrözik erre a zárolási rendszerre.

Zárak felminősítése

Az a T tranzakció, amelyik osztott zárat helyez X-re, „barátságos" a többi tranzakcióhoz, ugyanis a többinek is lehetősége van X-et T-vel egy időben olvasni. A kérdés az, hogy még barátságosabb-e az a T tranzakció, amelyik beolvasni és új értékkel írni akarja X-et úgy, hogy előbb csak osztott zárat tesz X-re, majd később, amikor T már készen áll az új érték beírására, akkor *felminősíti* a zárat kizárólagossá, vagyis később kéri X kizárólagos zárolását azon túl, hogy már osztott zárat tart fenn X-en. Nincs akadálya, hogy a tranzakció ugyanarra az adatbáziselemre újabb, különböző zármódú kéréseket adjon ki. Továbbra is fenntartjuk azt a megszokott jelölést, hogy $u_i(X)$ a T_i tranzakció által elhelyezett összes zárat feloldja X-en, bár be lehetne vezetni zárolási módoktól függő feloldási műveleteket, ha lenne hasznuk.

Pontosan fogalmazva: Azt mondjuk, hogy a T tranzakció *felminősíti* (upgrade) az L_1 zárját az L_1-nél dominánsabb L_2 zárra az X adatbáziselemen, ha L_2 zárat kér X-re, amelyen már birtokol egy L_1 zárat. L_2 *dominánsabb* L_1-nél, ha a kompatibilitási mátrixban L_2 sorában/oszlopában minden olyan pozícióban „nem" áll, amelyben L_1 sorában/oszlopában „nem" áll. Például az SX zárolási séma esetén X dominánsabb S-nél. (X egyébként minden zármódnál dominánsabb bármelyik zárolási séma esetén, hiszen X sorában és oszlopában is minden pozícióban „nem" szerepel.)

Példa. A következő példában a T_1 tranzakció T_2-vel konkurensen tudja végrehajtani a számításait, amely nem lenne lehetséges, ha T_1 kezdetben kizárólagosan zárolta volna B-t. A két tranzakció a következő:

T_1: $sl_1(A); r_1(A); sl_1(B); r_1(B); xl_1(B); w_1(B); u_1(A); u_1(B);$
T_2: $sl_2(A); r_2(A); sl_2(B); r_2(B); u_2(A); u_2(B);$

Itt T_1 beolvassa A-t és B-t, és végrehajtja a (valószínűleg hosszadalmas) számításokat velük, és a legvégén az eredményt beírja B új értékének. T_1 előbb osztottan zárolja B-t, majd később, miután az A-val és B-vel kapcsolatos számításait befejezte, kér egy kizárólagos zárat B-re. A T_2 tranzakció csak olvassa A-t és B-t, nem ír rájuk.

T_1	T_2
$sl_1(A); r_1(A);$	
	$sl_2(A); r_2(A);$
	$sl_2(B); r_2(B);$
$sl_1(B); r_1(B);$	
$xl_1(B);$ **elutasítva**	
	$u_2(A); u_2(B);$
$xl_1(B); w_1(B);$	
$u_1(A); u_1(B);$	

Az ábra a műveletek egy lehetséges ütemezését mutatja. T_2 egy osztott zárat kap B-re T_1 előtt, de a negyedik sorban T_1 is képes osztottan zárolni B-t. Így T_1 rendelkezésére áll A is és B is, és az értékeik felhasználásával végre tudja hajtani a számításokat. Amikor T_1 megpróbálja B-n a zárat felminősíteni kizárólagossá, az ütemező a kérést elutasítja, és arra kényszeríti T_1-et, hogy várjon addig, amíg T_2 felszabadítja a B-n lévő zárat. Ezután T_1 megkapja a kizárólagos zárat, kiírja B-t, és befejeződik a tranzakció.

Ha T_1 a kezdéskor kért volna kizárólagos zárat B-re, mielőtt beolvasta volna, akkor ezt a kérést az ütemező elutasította volna, ugyanis T_2-nek már volt egy osztott zárja B-n. T_1 nem tudta volna elvégezni a számításait B beolvasása nélkül, így T_1-nek sokkal több dolga lett volna, miután T_2 felszabadította a zárakat. T_1 tehát később fejeződött volna be, ha csak kizárólagos zárat használt volna B-n, mint amikor a felminősítő stratégiát alkalmazta.

Példa. Sajnos a felminősítés válogatás nélküli alkalmazása a holtpontok új forrását jelenti. Tételezzük fel, hogy T_1 és T_2 is beolvassa az A adatbáziselemet, és egy új értéket ír vissza A-ba. Ha mindkét tranzakció a felminősítéssel dolgozik, akkor előbb osztott zárat kapnak A-ra, és azután minősítik ezt át kizárólagossá, így az alábbi eseménysorozat következhet be, amikor T_1 és T_2 közel egyidejűleg kezdődik:

T_1	T_2
$sl_1(A)$;	
	$sl_2(A)$;
$xl_1(A)$; **elutasítva**	
	$xl_2(A)$; **elutasítva**

T_1 és T_2 is kaphat osztott zárat A-ra. Ezután mindkettő megpróbálja ezt felminősíteni kizárólagossá, de az ütemező mindkettőt várakozásra kényszeríti, hiszen a másik már osztottan zárolja A-t. Emiatt egyikük végrehajtása sem folytatódhat; vagy mindkettőnek örökösen kell várakoznia, vagy addig kell várakozniuk, amíg a rendszer fel nem fedezi, hogy holtpont alakult ki, abortálja valamelyik tranzakciót, és a másiknak engedélyezi A-ra a kizárólagos zárat.

Módosítási zárak

A fenti holtpontproblémát el tudjuk kerülni egy harmadik zárolási mód, az úgynevezett *módosítási zárak* (update lock) használatával. Az $ul_i(X)$ módosítási zár a T_i tranzakciónak csak X olvasására ad jogot, X írására nem. Később azonban csak a módosítási zárat lehet felminősíteni írásira, az olvasási zárat nem (azt csak módosításira). Módosítási zárat akkor is engedélyezhetünk X-en, ha X osztott módon már zárolva van, ha azonban X-en már van egy módosítási zár, akkor ez megakadályozza, hogy X bármilyen más újabb zárat (akár osztott, akár módosítási, akár kizárólagos zárat) kapjon. Ennek az az oka, hogy ha nem utasítanánk el ezeket az újabb zárolásokat, akkor előfordulhat, hogy a módosításinak soha sem lenne lehetősége kizárólagossá való felminősítésre, ugyanis mindig valamilyen más zár lenne X-en (a módosítási zár tehát nemcsak a holtpontproblémát oldja meg, hanem a kiéheztetés problémáját is).

Ez a szabály nem szimmetrikus kompatibilitási mátrixot eredményez, ugyanis az U módosítási zár úgy néz ki, mintha osztott zár lenne, amikor kérjük, és úgy néz ki, mintha kizárólagos zár lenne, amikor már megvan. Emiatt az U és az S zárak oszlopai megegyeznek, valamint U és X sorai is megegyeznek:

	S	X	U
S	igen	nem	igen
X	nem	nem	nem
U	nem	nem	nem

Ne feledjük azonban, hogy van egy további feltétel az ütemezések jogszerűségére vonatkozóan, amely nem jelenik meg a mátrixban: egy olyan tranzakció, amelynek van osztott zárja egy X adatbáziselemen, de nincs módosítási zárja, nem kaphat kizárólagos zárat X-re, noha általában nem tiltjuk, hogy egy tranzakció több zárat is fenntartson ugyanazon az elemen.

Példa. A módosítási zárak használata nem befolyásolja a korábbi példát. A harmadik művelet az lenne, hogy T_1 módosítási zárat tenne B-re, nem pedig osztott zárat. A módosítási zárat megkapná, ugyanis csak osztott zárak vannak B-n, és ugyanaz a műveletsorozat fordulna elő.

Módosítási zárakkal megszüntethető viszont a holtpontprobléma. Most mind T_1, mind T_2 előbb módosítási zárat kér A-n, majd később kizárólagos zárat. T_1 és T_2 egy lehetséges leírása az alábbi:

T_1: $ul_1(A); r_1(A); xl_1(A); w_1(A); u_1(A);$
T_2: $ul_2(A); r_2(A); xl_2(A); w_2(A); u_2(A);$

A korábbinak megfelelő eseménysorozat pedig a következő:

T_1	T_2
$ul_1(A);\ r_1(A);$	
	$ul_2(A);$ **elutasítva**
$xl_1(A);\ w_1(A);\ u_1(A);$	
	$ul_2(A);\ r_2(A);$
	$xl_2(A);\ w_2(A);\ u_2(A);$

Itt T_2-t elutasítjuk, amelyik másodikként kérte A módosítási zárolását. Miután T_1 befejeződött, T_2 folytatódhat. A zárolási rendszer hatékonyan megakadályozta T_1 és T_2 konkurens végrehajtását, ebben a példában viszont lényeges mennyiségű konkurens végrehajtás vagy holtpontot, vagy inkonzisztens adatbázis-állapotot eredményez.

Növelési zárak

Egy másik érdekes zárolási mód, amely bizonyos helyzetekben hasznos lehet, a *növelési zár*. Számos tranzakciónak csak az a hatása az adatbázison, hogy növeli vagy csökkenti a tárolt értéket. Ilyen például, amikor pénzt utalunk át az egyik bankszámláról a másikra, vagy amikor egy repülőjegyeket árusító tranzakció csökkenti az adott gépen a szabad ülőhelyek számát.

A növelési műveletek érdekes tulajdonsága, hogy tetszőleges sorrendben kiszámíthatók, ugyanis ha két tranzakció egy-egy konstanst ad hozzá ugyanahhoz az adatbáziselemhez, akkor nem számít, hogy melyiket hajtjuk végre előbb. Másrészt a növelés nem cserélhető fel sem az olvasással, sem az írással. Ha azelőtt vagy azután olvassuk be A-t, hogy valaki növelte, különböző értékeket kapunk, és ha azelőtt vagy azután növeljük A-t, hogy más tranzakció új értéket írt be A-ba, akkor is különböző értékei lesznek A-nak az adatbázisban.

Vezessünk be egy új műveletet, a *növelési műveletet* (increment action), és jelöljük `INC(A,c)`-vel. Ez a művelet megnöveli az A adatbáziselem (ami ilyenkor mindig attribútum) értékét c-vel, amelyről feltételezzük, hogy egyszerű szám konstans. Ha c negatív, akkor valójában csökkentést hajtunk végre. A gyakorlatban az `INC` műveletet a relációsor egy attribútumára alkalmazzuk, annak ellenére, hogy maga a sor, és nem az attribútum a zárolható elem.

Formálisan az `INC(A,c)` művelet a következő lépések atomi végrehajtására szolgál: `READ(A,t); t := t+c; WRITE(A,t);`. Az atomiságnak ez az alakja alsóbb szintű, mint a tranzakcióknak a zárolások által támogatott atomisága.

Szükségünk van a növelési műveletnek megfelelő *növelési zárra* (increment lock), amelyet $il_i(X)$-szel jelölünk. Jelentése: a T_i tranzakció növelési zárat kér az X adatbáziselemre. Az $inc_i(X)$ rövidítést arra a műveletre használjuk, amelyben a T_i tranzakció megnöveli az X adatbáziselemet valamely konstanssal. Annak, hogy pontosan mennyi ez a konstans, nincs jelentősége.

A növelési műveletek és zárak létezése szükségessé teszi, hogy több helyen módosítsuk a konzisztens tranzakciók, a konfliktusok és a jogszerű ütemezések definícióit. A változtatások az alábbiak:

a) Egy konzisztens tranzakció csak akkor végezheti el X-en a növelési műveletet, ha egyidejűleg növelési (vagy kizárólagos) zárat tart fenn rajta. A növelési zár viszont nem teszi lehetővé sem az olvasási, sem az írási műveleteket.

b) Az $inc_i(X)$ művelet konfliktusban áll $r_j(X)$-szel és $w_j(X)$-szel is $j \neq i$-re, de nem áll konfliktusban $inc_j(X)$-szel.

c) Egy jogszerű ütemezésben bármennyi tranzakció bármikor fenntarthat X-en növelési zárat. Ha viszont egy tranzakció növelési zárat tart fenn X-en, akkor egyidejűleg semelyik más tranzakció sem tarthat fenn sem osztott, sem kizárólagos zárat X-en. Ezeket a követelményeket a kompatibilitási mátrix segítségével fejezzük ki:

	S	X	I
S	igen	nem	nem
X	nem	nem	nem
I	nem	nem	igen

Példa. Tekintsünk két tranzakciót, mindkettő beolvassa az A adatbáziselemet, és azután növeli B-t. Lehet, hogy A-t adják hozzá B-hez, vagy egy olyan konstanssal növelik B-t, amelynek kiszámítása valamilyen más módon függ A-tól.

T_1: $sl_1(A); r_1(A); il_1(B); inc_1(B); u_1(A); u_1(B);$
T_2: $sl_2(A); r_2(A); il_2(B); inc_2(B); u_2(A); u_2(B);$

Látható, hogy a tranzakciók konzisztensek, hiszen csak akkor végeznek növelést, amikor növelési zárral rendelkeznek, és csak akkor olvasnak, amikor osztott zárat tartanak fenn. T_1 és T_2 egy lehetséges ütemezése a következő:

T_1	T_2
$sl_1(A); r_1(A);$	
	$sl_2(A); r_2(A);$
	$il_2(B); inc_2(B);$
$il_1(B); inc_1(B);$	
	$u_2(A); u_2(B);$
$u_1(A); u_1(B);$	

T_1 olvassa először A-t, azután T_2 beolvassa A-t, és növeli B-t. Ezután viszont T_1-nek is megengedjük, hogy növelési zárat kapjon B-re, és folytatódjon. Az ütemezőnek egyik kérést sem kell késleltetnie. Például tételezzük fel, hogy T_1 növeli B-t A-val, T_2 pedig növeli B-t 2A-val. Bármelyik sorrendben végrehajthatjuk a tranzakciókat, ugyanis A értéke nem változik, és a növelést is bármely sorrendben elvégezhetjük. Másképpen kifejezve: nézzük meg a nem zárolási műveletek sorozatát az ütemezésben:

S: $r_1(A); r_2(A); inc_2(B); inc_1(B);$

Az utolsó műveletet, $inc_1(B)$-t, előrébb tudjuk hozni a második helyre, ugyanis ez nincs konfliktusban ugyanannak az elemnek egy másik növelésével, és biztosan nincs konfliktusban egy másik elem olvasásával. A cseréknek ez a sorozata mutatja, hogy S konfliktusekvivalens a következő soros ütemezéssel:

$r_1(A); inc_1(B); r_2(A); inc_2(B);$

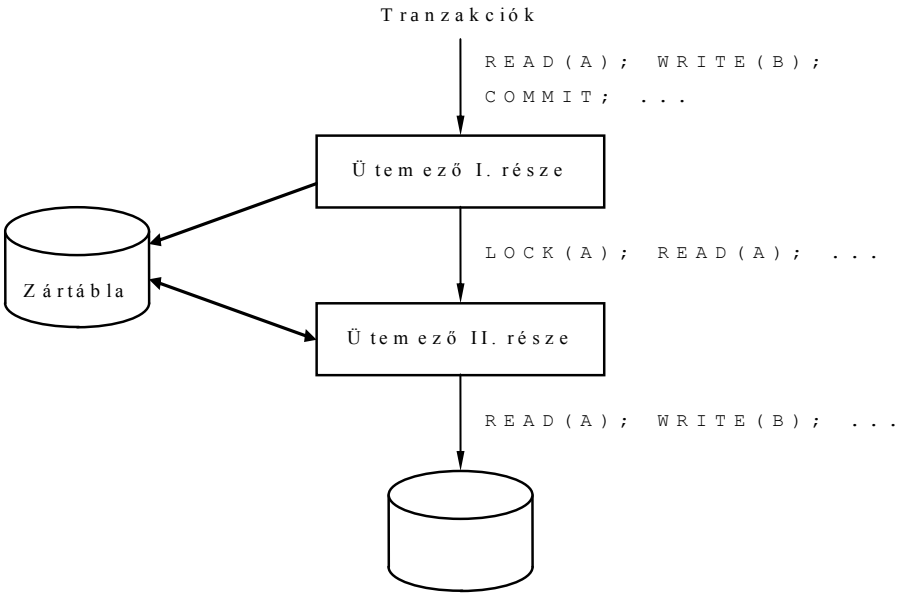
Hasonlóan tudjuk az első műveletet, $r_1(A)$-t, cserékkel a harmadik helyre hátrébb vinni, amely azt a soros ütemezést adja, amelyben T_2 megelőzi T_1-et.

A zárolási ütemező felépítése

Eddig már számos zárolási sémát láttunk, most megnézzük, hogyan működik egy olyan ütemező, amely ezek közül a sémák közül használja valamelyiket. Itt csak a következő elveken alapuló egyszerű ütemező felépítését tekintjük:

1. Maguk a tranzakciók nem kérnek zárakat, vagy figyelmen kívül hagyjuk, hogy ezt teszik. Az ütemező feladata, hogy zárolási műveleteket szúrjon be az adatokhoz hozzáférő olvasási, írási, illetve egyéb műveletek sorába.
2. Nem a tranzakciók, hanem az ütemező oldja fel a zárakat, mégpedig akkor, amikor a tranzakciókezelő a tranzakció véglegesítésére vagy abortálására készül.

Zárolási műveleteket beszúró ütemező



Az ábra egy olyan két részből álló ütemezőt mutat be, amely `READ`, `WRITE`, `COMMIT` és `ABORT` kéréseket fogad a tranzakcióktól. Az ütemező karbantartja a zártáblát, amelyet – bár másodlagosan tárolt adatként ábrázoltunk – lehet, hogy részben vagy egészben a központi memóriában tárolunk. A zártábla által használt központi memória általában nem a lekérdezés-végrehajtás és a naplózás által használt pufferterület része. A zártábla az adatbázis-kezelő rendszernek csak egy komponense, és az operációs rendszer foglal számára helyet, ugyanúgy, mint az adatbázis-kezelő rendszer többi kódjának és belső adatainak.

A tranzakciók által kért műveletek az ütemezőn jutnak keresztül, és az adatbázison kerülnek végrehajtásra általában azonnal. Bizonyos körülmények esetén viszont *késleltetett* a tranzakció, zárolásra vár, és a kérései még nem jutottak el az adatbázishoz. Az ütemező két része a következő műveleteket hajtja végre:

1. Az I. rész fogadja a tranzakciók által generált kérések sorát, és minden adatbázis-hozzáférési művelet elé beszúrja a megfelelő zárolási műveletet. Az ütemező I. részének kell tehát kiválasztania a megfelelő zárolási módot az ütemező által használt zármódok halmazából. Az adatbázis-hozzáférési és zárolási műveleteket ezután átküldi a II. részhez (a `COMMIT` és `ABORT` műveleteket nem).
2. A II. rész fogadja az I. részen keresztül érkező zárolási és adatbázis-hozzáférési műveletek sorozatát. Eldönti, hogy a `T` tranzakció késleltetett-e (mivel zárolásra vár). Ha igen, akkor magát a műveletet késlelteti, azaz hozzáadja azoknak a műveleteknek a listájához, amelyeket a `T` tranzakciónak még végre kell hajtania. Ha `T` nem késleltetett, vagyis az összes előzőleg kért zár már engedélyezve van, akkor megnézi, hogy milyen műveletet kell végrehajtania.
   a) Ha a művelet adatbázis-hozzáférés, akkor továbbítja az adatbázishoz, és végrehajtja.
   b) Ha zárolási művelet érkezik, akkor megvizsgálja a zártáblát, hogy a zár engedélyezhető-e. Ha igen, akkor úgy módosítja a zártáblát, hogy az az éppen engedélyezett zárat is tartalmazza. Ha nem, akkor

egy olyan bejegyzést készít a zártáblában, amely jelzi a zárolási kérést. Az ütemező II. része ezután késlelteti a T tranzakció további műveleteit mindaddig, amíg nem tudja engedélyezni a zárat.

3. Amikor a T tranzakciót véglegesítjük vagy abortáljuk, akkor a tranzakciókezelő COMMIT, illetve ABORT műveletek küldésével értesíti az I. részt, hogy oldja fel az összes T által fenntartott zárat. Ha bármelyik tranzakció várakozik ezen zárfeloldások valamelyikére, akkor az I. rész értesíti a II. részt.
4. Amikor a II. rész értesül, hogy egy X adatbáziselemen felszabadult egy zár, akkor eldönti, hogy melyik az a tranzakció, vagy melyek azok a tranzakciók, amelyek megkapják a zárat X-re. A tranzakciók, amelyek megkapták a zárat, a késleltetett műveleteik közül annyit végrehajtanak, amennyit csak végre tudnak hajtani mindaddig, amíg vagy befejeződnek, vagy egy másik olyan zárolási kéréshez érkeznek el, amely nem engedélyezhető.

Példa. Ha csak egymódú zárak vannak, akkor az ütemező I. részének a feladata egyszerű. Ha bármilyen műveletet lát az X adatbáziselemen, és még nem szúrt be zárolási kérést X-re az adott tranzakcióhoz, akkor beszúrja a kérést. Amikor véglegesítjük vagy abortáljuk a tranzakciót, az I. rész törölheti ezt a tranzakciót, miután feloldotta a zárakat, így az I. részhez igényelt memória nem nő korlátlanul.

Amikor többmódú zárak vannak, az ütemezőnek szüksége lehet arra, hogy azonnal értesüljön, milyen későbbi műveletek fognak előfordulni ugyanazon az adatbáziselemen. Nézzük meg újból az osztott/kizárólagos/módosítási zárak esetét, a felminősítésnél látott példában szereplő tranzakciókat használva. Zárolások nélkül a tranzakciók a következők:

T_1: $r_1(A); r_1(B); w_1(B);$
T_2: $r_2(A); r_2(B);$

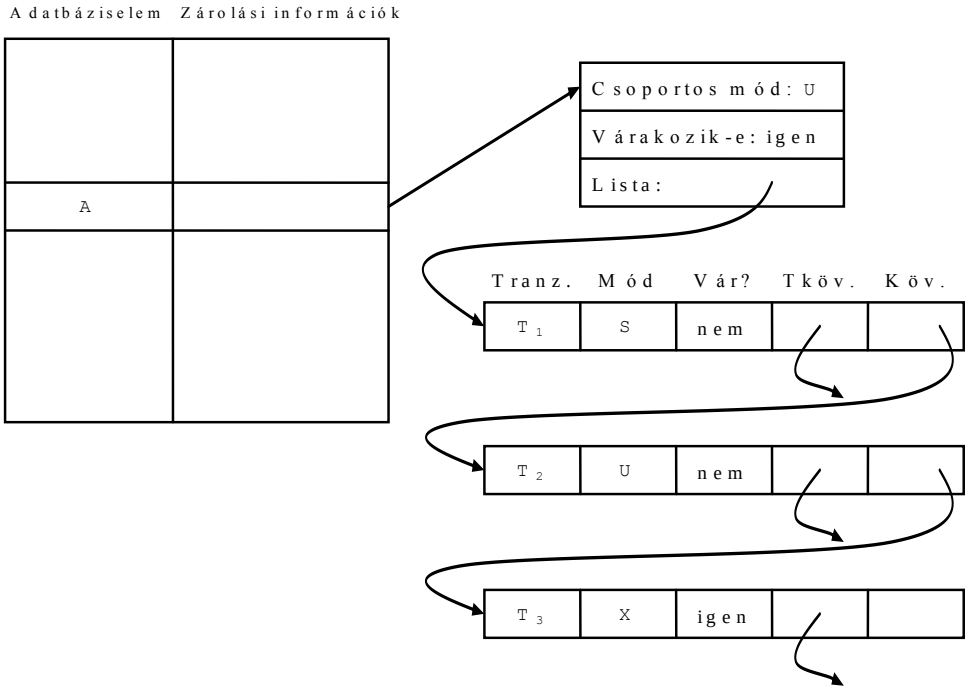
Az ütemező I. részéhez küldött üzenetnek nemcsak az olvasási és írási kéréseket kell tartalmaznia, hanem az ugyanazon az elemen bekövetkező későbbi műveletekre vonatkozó jelzést is. Amikor például az $r_1(B)$ érkezik be, az ütemezőnek tudnia kell, hogy lesz-e később $w_1(B)$ művelet (vagy lehet-e ilyen művelet, ha a T_1 tranzakció kódjában elágazás szerepel). Több módon válhat elérhetővé az információ. Például ha a tranzakció egy lekérdezés, akkor tudjuk, hogy semmit sem fog írni. Ha a tranzakció egy SQL-adatbázist módosító utasítás, akkor a lekérdező processzor azonnal megadhatja azokat az adatbáziselemeket, amelyeket olvashatunk és írhatunk is egyben. Ha a tranzakció egy beágyazott SQL-program, akkor a fordító hozzá tud férni az összes SQL-utasításhoz (és csakis ezekkel lehet írni az adatbázisba), és meghatározhatja, mely adatbáziselemek esélyesek az írásra.

A példánkban tételezzük fel, hogy a felminősítés példájában bemutatott sorrendben következnek be az események. Ekkor T_1 először $r_1(A)$-t adja ki. Mivel nincs később kizárólagos zárrá való felminősítés erre a zárra, az ütemező beszúrja $sl_1(A)$-t az $r_1(A)$ elé. Ezután T_2 kérései ($r_2(A)$ és $r_2(B)$) érkeznek az ütemezőhöz. Megint nincs később felminősítés, így az ütemező I. része a következő műveletsorozatot adja ki: $sl_2(A); r_2(A); sl_2(B); r_2(B);$.

Ezután az $r_1(B)$ művelet érkezik be az ütemezőhöz azzal a figyelmeztetéssel, hogy ezt a zárat fel lehet minősíteni. Az ütemező I. része ekkor kibocsátja $ul_1(B); r_1(B);$-t a II. résznek, amely megnézi a zártáblát, és azt találja, hogy T_1 engedélyezheti a módosítási zárat B-re, ugyanis csak osztott zárak vannak B-n.

Amikor a $w_1(B)$ művelet beérkezik az ütemezőhöz, az I. rész kibocsátja $xl_1(B); w_1(B);$-t. A II. rész viszont nem teljesítheti az $xl_1(B)$ kérést, ugyanis T_2-nek már van osztott zárja B-n. T_1-nek ezt a műveletét és az ezutáni műveleteit késlelteti, egyben tárolja a későbbi végrehajtáshoz. Végül T_2 végrehajtja a véglegesítést, és az I. rész feloldja a zárakat A-n és B-n. Ugyanekkor felfedezi, hogy T_1 várakozik B zárolására. Értesíti a II. részt, amely az $xl_1(B)$ zárolást most már végrehajthatónak találja. Beviszi ezt a zárat a zártáblába, és folytatja T_1 tárolt műveleteinek a végrehajtását mindaddig, ameddig tudja. Esetünkben T_1 befejeződik.

A zártábla



Absztrakt szinten a zártábla egy olyan reláció, amely összekapcsolja az adatbáziselemeket a rájuk vonatkozó zárolási információkkal, ahogyan ezt az ábra mutatja. Azok az elemek, amelyek nincsenek zárolva, nem fordulnak elő a táblában, így a méret csak a zárolt elemek számával arányos, nem pedig a teljes adatbázis méretével.

Az ábrán egy példát láthatunk arra, hogy milyen információk találhatók egy zártáblabejegyzésnél. Ez a példa feltételezi, hogy az ütemező az osztott/kizárólagos/módosítási (SXU) zársémát alkalmazza. Egy tipikus A adatbáziselemhez a bejegyzés a következő komponensekből áll:

1. A *csoportos mód* (group mode) a legszigorúbb feltételek összefoglalása, amivel egy tranzakció szembesül, amikor egy új zárolást kér A-n, azaz a csoportos mód az A-n jelenleg fenntartott zármódok közül a legdominánsabb. Ahelyett, hogy összehasonlítanánk a zárolási kérést a többi tranzakciónak ugyanazon az elemen fenntartott minden zárolásával, egyszerűsíthetjük az engedélyezési/elutasítási döntést azzal, hogy a kérést csak a csoportos móddal hasonlítjuk össze. (A zároláskezelőnek viszont foglalkoznia kell azzal a lehetőséggel, hogy a kérést kiadó tranzakciónak már van egy másik módban zárja ugyanazon az elemen. Például az SXU zárolási rendszerre vonatkoztatva, a zároláskezelő elfogadhat egy X zárra vonatkozó kérést, ha az igénylő tranzakció pont az, amely U zárat tart fenn ugyanazon az elemen. Azoknál a rendszereknél, amelyek nem támogatják, hogy egy tranzakció egy elemen több zárat is tartson, a csoportos mód mindig megadja mindazt, amit a zároláskezelőnek tudnia kell.) Az SXU zárolási sémákhoz egyszerű a szabály:

   Egy csoportos módban:

   a) S azt jelenti, hogy csak osztott zárak vannak;
   b) U azt jelenti, hogy egy módosítási zár van, és lehet még egy vagy több osztott zár is;
   c) X azt jelenti, hogy csak egy kizárólagos zár van, és semmilyen más zár nincs.

   A többi zárolási sémához is mindig találunk a csoportos mód összegzésének megfelelő rendszert.

2. A *várakozási bit* (waiting bit) azt adja meg, hogy van-e legalább egy tranzakció, amely A zárolására várakozik.

3. Az összes olyan tranzakciót leíró lista, amelyek vagy jelenleg zárolják A-t, vagy A zárolására várakoznak. Hasznos információk, amelyeket minden listabejegyzés tartalmazhat:

   a) a zárolást fenntartó vagy a zárolásra váró tranzakció neve;
   b) ennek a zárnak a módja;
   c) a tranzakció fenntartja-e a zárat, vagy várakozik-e a zárra.

   Az ábrán két láncolás szerepel minden bejegyzésnél. Az egyik magukhoz az adatbáziselemre vonatkozó bejegyzésekhez tartozik, a másik (*Tköv.*) pedig egy bizonyos tranzakció összes bejegyzését láncolja össze. Az utóbbi akkor használható, amikor a tranzakciót véglegesítjük vagy abortáljuk, így könnyen megtalálhatjuk az összes zárat, amelyet fel kell oldanunk.

A zárolási kérések kezelése

Tételezzük fel, hogy a T tranzakció zárat kér A-ra. Ha nincs A-ra bejegyzés a zártáblában, akkor biztos, hogy zárak sincsenek A-n, így létrehozhatjuk a bejegyzést, és engedélyezhetjük a kérést. Ha a zártáblában létezik bejegyzés A-ra, akkor ezt felhasználjuk a zárolási kéréssel kapcsolatos döntésünkben. Megkeressük a csoportos módot, amely az ábrán U, vagyis módosítási. Amikor már van módosítási zár egy elemen, akkor semmilyen más zárat nem engedélyezhetünk (kivéve azt az esetet, amikor maga T tartja fenn az U zárat, és a többi zár kompatibilis T kérésével). Tehát T-nek ezt a kérését elutasítjuk, és egy bejegyzést helyezünk el a listában, amely szerint T zárat kért, és a várakozási bitet igazra állítjuk.

Ha a csoportos mód X, vagyis kizárólagos lenne, akkor ugyanez történne. Ha azonban a csoportos mód S, vagyis osztott lenne, akkor lehetne adni egy másik osztott vagy módosítási zárat. Ebben az esetben a listában a T-hez tartozó várakozási bitet hamisra, a csoportos módot pedig U-ra kellene állítani, ha az új zár módosítási zár, egyébként pedig a csoportos mód S maradna. Akár adtunk engedélyt a zárolásra, akár nem, az új listabejegyzést megfelelően beláncoljuk a két mutatón keresztül. Látható, hogy akár engedélyezzük a zárat, akár nem, a zártáblában a bejegyzés megadja az ütemezőnek azt, amit tudnia kell, anélkül hogy megvizsgálná a zárolások listáját.

A zárfeloldások kezelése

Most tételezzük fel, hogy a T tranzakció feloldja az A-n lévő zárakat. Ekkor T bejegyzését A-ra a listában töröljük. Ha a T által fenntartott zár nem egyezik meg a csoportos móddal (például T egy S zárat tart fenn, míg a csoportos mód U), akkor nincs okunk, hogy megváltoztassuk a csoportos módot. Ha viszont a T által fenntartott zár van a csoportos módban, akkor meg kell vizsgálnunk a teljes listát, hogy megtaláljuk az új csoportos módot. Az ábrán látható példában csak egyetlen U zár lehet egy elemen, így ha azt a zárat feloldjuk, az új csoportos mód csak S lehetne (ha maradt még osztott zár), vagy semmi (ha nincs más zár jelenleg fenntartva). (Valójában sohasem lesz „semmi” a csoportos mód, ugyanis ha nincs sem zár, sem zárolási kérés egy elemen, akkor nincs bejegyzés sem a zártáblában erre az elemre. Csak zárolási kérés meglévő zár nélkül pedig szintén nem fordulhat elő.) Ha a csoportos mód X, akkor tudjuk, hogy nincsenek más zárolások, ha pedig a csoportos mód S, akkor el kell döntenünk, hogy van-e további osztott zár.

Ha a várakozási bit igaz, akkor engedélyeznünk kell egy vagy több zárat a kért zárak listájáról. Több különböző megközelítés lehetséges, mindegyiknek megvan a saját előnye:

1. *Első beérkezett első kiszolgálása* (first-come-first-served): Azt a zárolási kérést engedélyezzük, amelyik a legrégebb óta várakozik. Ez a stratégia azt biztosítja, hogy ne legyen kiéheztetés, vagyis a tranzakció ne várjon örökké egy zárra.
2. *Elsőbbségadás az osztott záraknak* (priority to shared locks): Először az összes várakozó osztott zárat engedélyezzük. Ezután egy módosítási zárolást engedélyezünk, ha várakozik ilyen. A kizárólagos

zárolást csak akkor engedélyezzük, ha semmilyen más igény nem várakozik. Ez a stratégia csak akkor engedi a kiéheztetést, ha a tranzakció U vagy X zárolásra vár.

3. *Elsőbbségadás a felminősítésnek* (priority to upgrading): Ha van olyan U zárral rendelkező tranzakció, amely X zárrá való felminősítésre vár, akkor ezt engedélyezzük előbb. Máskülönben a fent említett stratégiák valamelyikét követjük.

Adatbáziselemekből álló hierarchiák kezelése

Térjünk vissza a különféle zárolási sémák feltárásához. Különösen két olyan problémára összpontosítunk, amelyek akkor merülnek fel, amikor fastruktúra tartozik az adatainkhoz:

1. Az első fajta fastruktúra, amelyet figyelembe veszünk, a zárolható elemek (zárolási egységek) hierarchiája. Megvizsgáljuk, hogyan engedélyezünk zárolást mind a nagy elemekre, mint például a relációkra, mind a kisebb elemekre, mint például a reláció néhány sorát tartalmazó blokkokra vagy egyedi sorokra.
2. A másik lényeges hierarchiafajtát képezik a konkurenciavezérlési rendszerekben azok az adatok, amelyek önmagukban faszervezésűek. Ilyenek például a B-fa-indexek. A B-fák csomópontjait adatbáziselemeknek tekinthetjük, így viszont az eddig tanult zárolási sémákat szegényesen használhatjuk, emiatt egy új megközelítésre van szükségünk.

Többszörös szemcsézettségű zárak

A különböző rendszerek különböző méretű adatbáziselemeket zárolnak, mint például relációkat, sorokat, lapokat vagy blokkokat. Bizonyos alkalmazásoknál a kis adatbáziselemek előnyösek, míg másoknál a nagy elemek nyújtják a legtöbbet.

Példa. Tekintsünk egy banki adatbázist. Ha a relációkat kezeljük adatbáziselemekként, akkor így csak egy zárat tudunk kiadni arra a teljes relációra, amely a számlák egyenlegét adja meg, ezért a rendszer nagyon kis konkurenciát engedélyezne. Mivel a legtöbb tranzakció a számlák egyenlegét változtatja, a legtöbb tranzakciónak kizárólagosan kellene zárolnia a számlaegyenlegeket tartalmazó relációt. Így csak egyetlen befizetést vagy kivételt tudnánk egyidejűleg elvégezni, nem számítana, hogy hány olyan processzor van, amely alkalmas lenne ezeknek a tranzakcióknak az elvégzésére. Jobb megközelítés, hogy egyedi lapokat vagy adatblokkokat zárolunk. Így két olyan számla, amelyekhez tartozó sorok külön blokkokban vannak, egyidejűleg módosítható. Ez biztosítja szinte a teljes konkurenciát, amely elérhető a rendszerben. A másik véglet az lenne, ha minden egyes sorra biztosítanánk zárolást, így bármilyen számlahalmazt egyszerre tudnánk módosítani, de a záraknak ennyire finom szemcsézettsége valószínűleg nem érné meg a sok fáradságot.

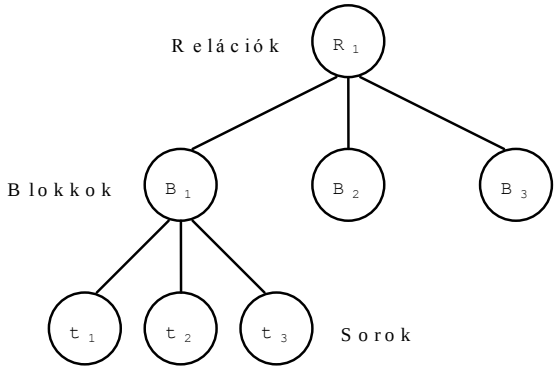
Másik példaként tekintsünk egy dokumentumokból álló adatbázist. Ezeket a dokumentumokat időnként szerkeszteni szokták, és a legtöbb tranzakció teljes dokumentumokhoz fér hozzá. Az adatbáziselem ésszerű megválasztása ekkor a teljes dokumentum. Mivel a legtöbb tranzakció csak olvasási tranzakció (vagyis nem végez írási műveletet), a zárolás csak azért szükséges, hogy elkerüljük a dokumentumok szerkesztés közbeni olvasását. Ha kisebb szemcsézettségű elemeket zárolnánk, mint például bekezdéseket, mondatokat vagy szavakat, akkor ennek semmilyen előnyét sem látnánk, viszont sokkal költségesebb lenne. Az egyetlen tevékenység, amelyet a kisebb szemcsézettségű zárak támogatnának, hogy a dokumentum egy részét tudnánk olvasni a dokumentum szerkesztése közben is.

Bizonyos alkalmazások mind a nagy, mind a kis szemcsézettségű zárakat tudják alkalmazni. Például a fent vázolt banki adatbázisnál világos, hogy blokk vagy sor szintű zárolás is szükséges, de néhány esetben a teljes számlareláció zárolása is szükséges lehet, például azért, hogy ellenőrizzük a számlákat. De ha osztott zárat teszünk a számlarelációra annak érdekében, hogy kiszámoljunk a reláción valamilyen

csoportfüggvényt, és egyidejűleg az egyéni számlák soraihoz kizárólagos zárat adunk, ez könnyen nem sorba rendezhető viselkedéshez vezethet, ugyanis a reláció valójában megváltozik, amíg egy feltehetően befagyasztott másolatát olvassuk a csoportfüggvényes lekérdezéshez.

Figyelmeztető zárak

A probléma megoldásához, hogy hogyan kezeljük az újfajta zárolással kapcsolatos, különféle szemcsézettségű zárakat, bevezetjük a *figyelmeztető zárakat*. Ezek a zárak akkor hasznosak, amikor adatbáziselemek beágyazott vagy hierarchikus struktúrákat mutatnak, amint azt az alábbi ábrán láthatjuk:



Itt az adatbáziselemek három szintjét különböztetjük meg:

1. a relációk a legnagyobb zárolható elemek;
2. minden reláció egy vagy több blokkból vagy lapból épül fel, amelyekben a soraik vannak;
3. minden blokk egy vagy több sort tartalmaz.

Az adatbáziselemek hierarchiáján a zárak kezelésére szolgáló szabályok alkotják a *figyelmeztető protokollt* (warning protocol), amely tartalmazza mind a „közönséges”, mind a „figyelmeztető” zárakat. A zárolási sémát úgy adjuk meg, hogy a közönséges zárak S és X (osztott és kizárólagos) lehetnek. A figyelmeztető zárakat a közönséges zárak elé helyezett I (intention) előtaggal jelöljük. Például IS azt jelenti, hogy szándékunkban áll osztott zárat kapni egy részelemen. A figyelmeztető protokoll szabályai:

1. Ahhoz, hogy elhelyezzünk egy közönséges S vagy X zárat valamely elemen, a hierarchia gyökerénél kell kezdenünk.
2. Ha már annál az elemnél tartunk, amelyet zárolni akarunk, akkor nem kell tovább folytatnunk, hanem kérjük az S vagy X zárolást arra az elemre.
3. Ha az elem, amelyet zárolni szeretnénk, lejjebb van a hierarchiában, akkor elhelyezünk egy figyelmeztetést ezen a csomóponton. Vagyis ha osztott zárat szeretnénk kérni egy részelemen, akkor ebben a csomópontban egy IS zárat kérünk, ha pedig kizárólagos zárat szeretnénk kérni egy részelemen, akkor ebben a csomópontban egy IX zárat kérünk. Amikor a jelenlegi csomópontban kért zárat megkaptuk, akkor ennek a csomópontnak azzal az utód csomópontjával folytatjuk, amelyikhez tartozó részfa tartalmazza azt a csomópontot, amelyet zárolni kívánunk. Ezután megfelelően a 2. vagy 3. lépéssel folytatjuk mindaddig, amíg el nem érjük a keresett csomópontot.

Ahhoz, hogy eldöntsük, engedélyezhetjük-e ezek közül a zárak közül valamelyiket, vagy sem, a következő kompatibilitási mátrixot használjuk:

	IS	IX	S	X
IS	igen	igen	igen	nem
IX	igen	igen	nem	nem
S	igen	nem	igen	nem
X	nem	nem	nem	nem

Ennek a mátrixnak az értelmezéséhez először nézzük meg az IS oszlopot. Ha IS zárat kérünk egy N csomópontban, az N egy leszármazottját szándékozzuk olvasni. Ez a szándék csak abban az esetben okozhat problémát, ha egy másik tranzakció korábban már jogosulttá vált arra, hogy az N által reprezentált teljes adatbáziselemet felülírja, ezért van „nem” az X-hez tartozó sorban. Ha más tranzakció azt tervezi, hogy N-nek csak egy részelemét írja (ezért az N csomóponton egy IX zárat helyezett el), akkor lehetőségünk van arra, hogy engedélyezzük az IS zárat N-en, és a konfliktust alsóbb szinten oldhatjuk meg, ha az írási és olvasási szándék valóban egy közös elemre vonatkozik.

Most tekintsük az IX-hez tartozó oszlopot. Ha az N csomópont egy részelemét szándékozzuk írni, akkor meg kell akadályoznunk az N által képviselt teljes elem olvasását vagy írását. Ezért van „nem” az S és az X zármódok sorában. Azonban az IS oszloppal kapcsolatban leírtaknak megfelelően más tranzakció, amely egy részelemet olvas vagy ír, a potenciális konfliktusokat az adott szinten kezeli le, így az IX nincs konfliktusban egy másik IX-szel vagy IS-sel N-en.

Nézzük most az S-hez tartozó oszlopot. Az N csomópontnak megfeleltetett elem olvasása nincs konfliktusban sem egy másik olvasási zárral N-en, sem egy olvasási zárral N egy részelemén, amelyet N-en egy IS reprezentál. Emiatt „igen”-t találunk az S és az IS sorában is. Azonban egy X vagy egy IX azt jelenti, hogy más tranzakció írni fogja legalább egy részét az N által reprezentált elemnek. Ezért nem tudjuk engedélyezni N teljes olvasását. Ezt fejezik ki a megfelelő „nem” bejegyzések.

Végül az X oszlopban csak „nem” bejegyzések vannak. Nem tudjuk megengedni az N csomópont egyik részének írását sem, ha más tranzakciónak már joga van arra, hogy olvassa vagy írja N-et, vagy arra, hogy megszerezze ezt a jogot N egy részelemére.

Példa. Tekintsük a következő relációt:

```
Film(filmCím, év, hossz, stúdióNév)
```

Tételezzük fel, hogy a teljes relációra és az egyedi sorokra követelünk zárolást. Legyen T_1 egy olyan tranzakció, amely az alábbi kérdést tartalmazza:

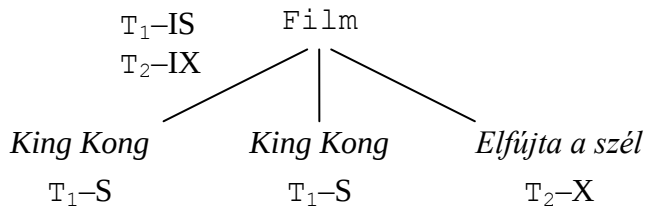
```
SELECT * FROM Film WHERE filmCím = 'King Kong';
```

T_1 azzal kezdődik, hogy IS módon zárolja a teljes relációt. Ezután veszi az egyedi sorokat, és S módú zárolást ad ki azokra, amelyekben a filmCím a megadottal egyezik (legyen két ilyen sor).

Tételezzük fel, hogy mialatt az első lekérdezést végezzük, elkezdődik a T_2 tranzakció, amely a sorok év komponensét változtatja meg:

```
UPDATE Film SET év = 1939 WHERE filmCím = 'Elfújta a szél';
```

Ekkor T_2-nek szüksége van a reláció IX módú zárolására, ugyanis azt tervezi, hogy új értéket ír be az egyik sorba. Ez kompatibilis T_1-nek a relációra vonatkozó IS zárolásával, így a zárat engedélyezzük. Amikor T_2 elérkezik az „Elfújta a szél” című filmhez tartozó sorhoz, ezen a soron nem talál zárat, így megkapja az X módú zárat, és módosítja a sort. Ha T_2 a „King Kong” című filmek valamelyikéhez próbált volna új értéket beírni, akkor várnia kellett volna, amíg T_1 felszabadítja az S zárakat, ugyanis az S és az X nem kompatibilisek. Az ábrán láthatjuk a zárak kollekcióját:



Csoportos mód a szándékzárolásokhoz

A fenti kompatibilitási mátrix olyan helyzetet mutat be, amelyet eddig még nem láttunk a zármódok erejét illetően. A korábbi zárolási sémák többségében valahányszor lehetőségünk volt arra, hogy egy adatbáziselemet egyszerre kétféle módban is zároljunk, ezek közül az egyik dominánsabb volt a másiknál. Például az SXU zárolási séma esetén U dominánsabb S-nél, X pedig mindkettőnél. Egy előnye annak, hogy tudjuk, mindig van egy domináns zár egy elemen, az, hogy több zárolás hatását össze tudjuk foglalni egy csoportos móddal.

A figyelmeztető zárakat is alkalmazó zárolási séma esetén az S és az IX módok közül egyik sem dominánsabb a másiknál. Továbbá egy elemet az S és IX módok mindegyikében zárolhatunk egyidejűleg, feltéve hogy ugyanaz a tranzakció kérte a zárolást. (Vigyázzunk, hogy a „nem” bejegyzések a kompatibilitási mátrixban csak azokra a zárakra alkalmazhatók, amelyeket más tranzakciók tartanak fenn.) Egy tranzakció mindkét zárolást kérheti, ha egy teljes elemet akar beolvasni, és azután a részelemeknek egy valódi részhalmazát akarja írni. Ha egy tranzakciónak S és IX zárolásai is vannak egy elemen, akkor ez korlátozza a többi tranzakciót olyan mértékben, ahogy bármelyik zár teszi. Vagyis elképzelhetünk egy új SIX zárolási módot, amelynek sorai és oszlopai a „nem” bejegyzést tartalmazzák az IS bejegyzés kivételével mindenhol. Az SIX zárolási mód csoportmódként szolgál, ha van olyan tranzakció, amelynek van S, illetve IX módú, de nincs X módú zárolása.

Elképzelhetjük ugyanezt a helyzetet a növelési zárolásokra, vagyis egy tranzakció S és I módban is fenntarthatna zárakat. Ez a helyzet viszont ekvivalens az X módú zárolással, így ekkor X-et használhatnánk csoportos módként.

Nem ismételhető olvasás és a fantomok

Tegyük fel, hogy van egy T_1 tranzakció, amelyben egy lekérdezés sorokat válogat ki egy relációból. Ezután egy T_2 tranzakció módosít vagy töröl a táblából olyan sorokat, amelyek eleget tesznek a lekérdezés feltételének. Ha ezután T_1 újra megpróbálja beolvasni ezeket a sorokat, azt fogja tapasztalni, hogy bizonyos sorok megváltoztak vagy eltűntek. Ezt a szituációt *nem ismételhető olvasásnak* (nonrepeatable read vagy fuzzy read) nevezzük. A nem ismételhető olvasással az a probléma, hogy mást eredményez a lekérdezés másodszori végrehajtása, mint az első. A tranzakció viszont elvárhatja, hogy ha többször végrehajtja ugyanazt a lekérdezést, akkor mindig ugyanazt az eredményt kapja.

Ugyanez a helyzet akkor is, ha a T_2 tranzakció nem töröl vagy módosít, hanem beszúr olyan sorokat, amelyek eleget tesznek a lekérdezés feltételének. A lekérdezés másodszori futtatásakor most is más eredményt kapunk, mint az első alkalommal. Ennek az az oka, hogy olyan sorok jelentek meg a relációban, amelyek az első futtatáskor még nem is léteztek. Az ilyen sorokat nevezzük *fantomoknak* (phantom).

A fenti jelenségek olyan ritkán fordulnak elő a gyakorlatban, hogy a legtöbb adatbázis-kezelő rendszer alapértelmezésben nem is figyel rájuk; annak ellenére, hogy mindkét jelenség nem sorbarendezhető viselkedést eredményezhet. A felhasználó azonban kérheti, hogy a nem ismételhető olvasások és/vagy a fantomolvasások ne fordulhassanak elő egy tranzakció végrehajtása során. Ehhez a tranzakció *elkülönítési szintjét* kell módosítani (lásd később).

A nem ismételhető olvasásokat könnyű megakadályozni: T_1-nek osztott zárat kell kérnie a lekérdezés által kiválasztott sorokra. T_2 így nem tudja azokat kizárólagosan zárolni, amíg T_1 be nem fejeződik vagy nem abortál. Könnyen megelőzhetjük a fantomolvasásokat is, ha többszörös szemcsézettségű zárakat használunk: a T_2 tranzakciónak X módban kell zárolnia a teljes relációt, mielőtt új sorokat szúrna be. Mivel T_1 korábban IS módban zárolta a relációt, ezt a kérést az ütemező először elutasítja, és csak akkor engedélyezi, amikor a T_1 tranzakció már befejeződött, elkerülve ezáltal a nem sorbarendezhető viselkedést.

Faprotokoll

Eddig a beágyazott szerkezetű adatbáziselemekből létrehozott fákkal foglalkoztunk, amelyekben a gyerekek a szülők részei voltak. Most maguknak az elemeknek a kapcsolati sémájából álló fa struktúrákkal foglalkozunk. Az adatbáziselemek diszjunkt adatdarabok, azonban csak egyféleképpen, a szülőkön keresztül lehet elérni egy csomópontot. A B-fák az ilyen típusú adatoknak fontos példái. Tudjuk, hogy csak egy bizonyos útvonalon jutunk el egy elemhez, és ez lényeges szabadságot ad nekünk abban, hogy a kétfázisú zárolási megközelítéstől eltérő módon kezeljük a zárakat.

A fa alapú zárolások indítékai

Tekintsünk egy B-fa-indexet egy olyan rendszerben, amely az egyedi csomópontokat (blokkokat) zárolható adatbáziselemekként kezeli. A csomópont a zárolás szemcsézettségének a megfelelő szintje, ugyanis nem előnyös, ha kisebb darabokat kezelünk elemekként. Ha pedig a teljes B-fát kezeljük adatbáziselemként, akkor ez megakadályozza az index olyan konkurens használatát, mint amilyen elérhető a következőkben tárgyalt működési mechanizmus segítségével.

Ha a zármódoknak egy szabványos halmazát használjuk (mint az osztott, kizárólagos és módosítási zárak), valamint használjuk a kétfázisú zárolást, akkor a B-fa konkurens használata szinte lehetetlen. Ennek az az oka, hogy az indexet használó minden tranzakciónak a B-fa gyökér csomópontját kell először zárolnia. Ha a tranzakció 2PL, akkor nem oldhatja fel a gyökéren a zárolást, amíg meg nem szerezte az összes zárat, amelyre szüksége van, mind a B-fa csomópontjain, mind pedig más adatbáziselemeken. Továbbá mivel elvben bármely tranzakció, amely beszúrásokat vagy törléseket végez, a B-fa gyökerének az átírásával fejeződhet be, a tranzakciónak legalább egy módosítási zárolásra szüksége van a gyökér csomóponton (vagy kizárólagosra, ha nincs módosítási mód). Így csak egyetlen nem csak olvasási tranzakció férhet hozzá bármikor a B-fához.

Mégis az esetek többségében majdnem közvetlenül levezethetjük, hogy egy B-fa gyökér csomópontját nem kell átírni, még akkor sem, ha a tranzakció beszúr vagy töröl egy sort. Például ha a tranzakció beszúr egy sort, de a gyökérnek az a gyereke, amelyhez hozzáférünk, nincs teljesen tele, akkor tudjuk, hogy a beszúrás nem gyűrűzik fel a gyökérig. Hasonlóan, ha a tranzakció egyetlen sort töröl, és a gyökérnek abban a gyerekében, amelyhez hozzáfértünk, a minimálisnál több kulcs és mutató van, akkor biztosak lehetünk abban, hogy a gyökér nem változik meg.

Így amikor a tranzakció a gyökérnek egyik gyereke felé irányul, és észleli azt a (teljesen szokványos) helyzetet, ami kizárja a gyökér átírását (azaz látja, hogy a gyökér biztosan nem változik meg), azonnal szeretnénk feloldani a gyökéren a zárat. Ugyanezt a megfigyelést alkalmazhatjuk a B-fa bármely belső csomópontjának a zárolására is, bár a konkurens B-fánál a legtöbb lehetőség abból származik, hogy a gyökéren a zárat korán oldjuk fel. Sajnos a gyökéren lévő zárolás korai feloldása ellentmond a 2PL-nek, így nem lehetünk biztosak abban, hogy a B-fához hozzáférő tranzakcióknak az ütemezése sorba rendezhető lesz. A megoldás egy speciális protokoll a B-fákhoz hasonló fa struktúrájú adatokhoz hozzáférő tranzakciók részére. A protokoll ellentmond a 2PL-nek, de azt a tényt használja, hogy az elemekhez való hozzáférés lefelé halad a fán a sorbarendezhetőség biztosítása érdekében.

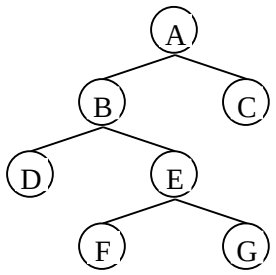
Fa szerkezetű adatok hozzáférési szabályai

Az alábbi megszorítások a zárakon a *faprotokollt* (tree protocol) adják. Tételezzük fel, hogy csak egyféle zár van, amelyet az $l_i(X)$ alakú zárolási kérésekkel ábrázolunk, de ezt az ötletet bármely zárolási módokból álló halmazra általánosíthatjuk. Tételezzük fel, hogy a tranzakciók konzisztensek, az ütemezések jogszerűek (vagyis az ütemező csak akkor engedélyezi a kért zárolásokat, ha azok nincsenek konfliktusban

azokkal a zárakkal, amelyek már a csomóponton vannak), és ugyanakkor nincs kétfázisú zárolási követelmény a tranzakciókon.

1. Egy tranzakciónak az első zárja a fa bármely csomópontján lehet. (A fenti példában az első zárnak mindig a gyökéren kell lennie, mivel a B-fa keresőfa, amelyben a keresés mindig a gyökértől indul.)
2. Rákövetkező zárakat csak akkor lehet szerezni, ha a tranzakciónak jelenleg van zárja a szülő csomóponton.
3. A csomópontok zárját bármikor feloldhatjuk.
4. Egy tranzakció nem zárolhatja újból azt a csomópontot, amelyen feloldotta a zárat, még akkor sem, ha még tartja a csomópont szülőjén a zárat.

Példa. Az alábbi ábra a csomópontok hierarchiáját, a táblázat pedig ezeken az adatokon három tranzakció műveleteit mutatja:



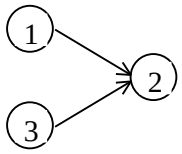
T_1	T_2	T_3
$l_1(A)$; $r_1(A)$;		
$l_1(B)$; $r_1(B)$;		
$l_1(C)$; $r_1(C)$;		
$w_1(A)$; $u_1(A)$;		
$l_1(D)$; $r_1(D)$;		
$w_1(B)$; $u_1(B)$;		
	$l_2(B)$; $r_2(B)$;	
		$l_3(E)$; $r_3(E)$;
$w_1(D)$; $u_1(D)$;		
$w_1(C)$; $u_1(C)$;		
	$l_2(E)$; **elutasítva**	
		$l_3(F)$; $r_3(F)$;
		$w_3(F)$; $u_3(F)$;
		$l_3(G)$; $r_3(G)$;
		$w_3(E)$; $u_3(E)$;
	$l_2(E)$; $r_2(E)$;	
		$w_3(G)$; $u_3(G)$;
	$w_2(B)$; $u_2(B)$;	
	$w_2(E)$; $u_2(E)$;	

T_1 az A gyökéren kezdődik, és lefelé folytatódik B, C és D felé. T_2 B-n kezdődik, és az E felé próbál haladni, de először elutasítjuk, ugyanis T_3-nak már van zárja E-n. A T_3 tranzakció E-n kezdődik, és folytatja F-fel és G-vel. T_1 nem 2PL tranzakció, ugyanis A-n előbb töröljük a zárat, mint hogy megszerezzük a zárat D-n. Hasonlóan T_3 sem 2PL tranzakció, de T_2 véletlenül éppen 2PL.

Miért működik a faprotokoll?

A faprotokoll jogszerű ütemezésben részt vevő konzisztens tranzakciókon konfliktus-sorbarendezhető ütemezést eredményez. A következőképpen definiálhatjuk a megelőzési sorrendet: Azt mondjuk, hogy T_i *megelőzi* T_j-t az S ütemezésben ($T_i <_S T_j$), ha a T_i és T_j tranzakciók egyrészt közösen zárolnak egy csomópontot, másrészt T_i zárolja a csomópontot először.

Példa. A fenti példa S ütemezésében T_1 és T_2 közösen zárolják B-t, és T_1 zárolja először. Így $T_1 <_S T_2$. Azt találjuk még, hogy T_2 és T_3 közösen zárolják E-t, és T_3 zárolja először, tehát $T_3 <_S T_2$. T_1 és T_3 között viszont nincs megelőzés, hiszen nincs olyan csomópont, amelyet közösen zárolnak. Az ezekből a megelőzési relációkból levezetett megelőzési gráf a következő ábrán látható:



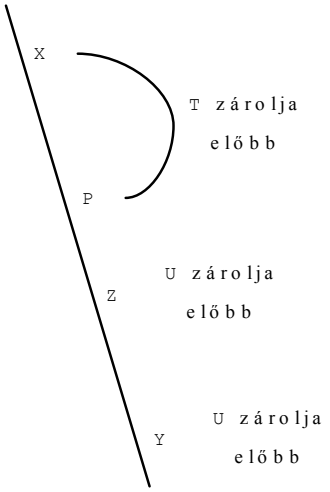
Ha a fent definiált megelőzési relációk alapján rajzolt megelőzési gráf nem tartalmaz kört, akkor azt állítjuk, hogy a tranzakciók bármely topologikus sorrendje egy ekvivalens soros ütemezés. Ebben a példában vagy a (T_1, T_3, T_2) vagy a (T_3, T_1, T_2) az ekvivalens soros ütemezés. Ennek az az oka, hogy az ilyen soros ütemezésben minden egyes csomóponthoz ugyanabban a sorrendben nyúlnak a tranzakciók, mint az eredeti ütemezésben.

Ahhoz, hogy megértsük, hogy a fent leírt megelőzési gráf miért lesz körmentes, ha betartjuk a faprotokoll szabályait, először vegyük észre a következőt:

- Ha két tranzakció közösen zárol néhány elemet, akkor ugyanabban a sorrendben zárolják mindegyiket.

Bizonyítás: Tekintsünk valamilyen T és U tranzakciókat, amelyek két vagy több elemet közösen zárolnak. Minden tranzakció fa formájú halmazát zárolja az elemeknek, és a két fa metszete maga is fa. Mivel most T és U közösen zárolnak elemeket, a metszet nem lehet üres fa. Emiatt van egy „legmagasabb” X elem, amelyet T és U is zárol. Tételezzük fel, hogy T zárolja X-et először, de van egy másik Y elem, amelyet U előbb zárol, mint T. Ekkor az elemekből álló fában van út X-ből Y-ba, és T-nek is és U-nak is zárolnia kell minden elemet az út mentén, ugyanis egyik sem zárolhat úgy egy csomópontot, hogy ne lenne már a szülőjén zárja.

Tekintsük az első olyan elemet az út mentén, amelyet U zárol először, legyen ez Z. Ekkor T előbb zárolja Z-nek a P szülőjét, mint U. Ekkor viszont T még mindig tartja a zárolást P-n, amikor zárolja Z-t, így U még nem zárolhatta P-t, amikor Z-t zárolja. Az nem lehet, hogy Z lenne az első elem, amelyet T és U közösen zárolnak, mivel mindkettő zárolta az ősét, X-et (amely lehet P is, csak Z nem). Így U addig nem zárolhatja Z-t, amíg meg nem szerezte P-n a zárat, amely viszont azután van, hogy T zárolta Z-t. Arra következtetünk, hogy T megelőzi U-t minden csomópontban, amelyet közösen zárolnak.



Most tekintsük a T_1, T_2, …, T_n tranzakciók tetszőleges halmazát, amely eleget tesz a faprotokollnak, és az S ütemezésnek megfelelően zárolja a fa valamely csomópontjait. Azok a tranzakciók, amelyek zárolják a gyökeret, ezt valamilyen sorrendben végzik, és olyan szabály alapján, amelyet éppen megfigyeltünk:

- Ha T_i előbb zárolja a gyökeret, mint T_j, akkor T_i minden T_j-vel közösen zárolt csomópontot előbb zárol, mint T_j. Vagyis $T_i <_S T_j$, de nem igaz $T_j <_S T_i$.

A fa csomópontjainak száma szerinti teljes indukcióval megmutathatjuk, hogy a teljes tranzakcióhalmazhoz létezik az S-sel ekvivalens soros sorrend:

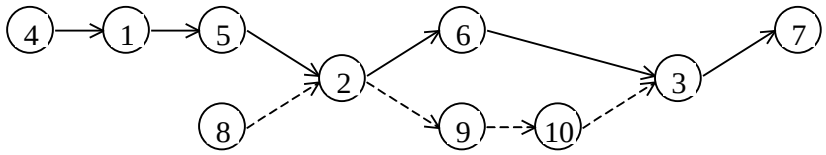
Alapeset: Ha csak egyetlen csomópont van, a gyökér, akkor ahogyan már megfigyeltük, a megfelelő sorrend az, amelyben a tranzakciók a gyökeret zárolják.

Indukció: Ha egynél több csomópont van a fában, tekintsük a gyökér mindegyik részfájához az olyan tranzakciókból álló halmazt, amelyek egy vagy több csomópontot zárolnak abban a részfában. A gyökeret záró tranzakciók több részfához is tartozhatnak, de egy olyan tranzakció, amely nem zárolja a gyökeret, csak egyetlen részfához tartozik. Például a fenti táblázatban található tranzakciók közül csak T_1 zárolja a gyökeret, és az mindkét részfához tartozik: a B gyökerű és a C gyökerű fához is. T_2 és T_3 viszont csak a B gyökerű fához tartozik.

Az indukciós feltevés szerint létezik soros sorrend az összes olyan tranzakcióhoz, amelyek ugyanabban a tetszőleges részfában zárolnak csomópontokat. Csupán egybe kell olvasztanunk a különböző részfákhoz tartozó soros sorrendeket. Mivel a tranzakcióknak ezekben a listáiban csak azok a tranzakciók közösek, amelyek zárolják a gyökeret, és megállapítottuk, hogy ezek a tranzakciók minden közös csomópontot ugyanabban a sorrendben zárolnak, ahogy a gyökeret zárolják, nem fordulhat elő két gyökeret záró tranzakció különböző sorrendben két részlistán. Pontosabban: ha T_i és T_j előfordul a gyökér valamely C gyermekéhez tartozó listán, akkor ezek C-t ugyanabban a sorrendben zárolják, mint a gyökeret, és emiatt a listán is ebben a sorrendben fordulnak elő. Így felépíthetjük a soros sorrendet a teljes tranzakcióhalmazhoz azokból a tranzakciókból kiindulva, amelyek a gyökeret zárolják, a megfelelő sorrendjükben, és beleolvasztjuk azokat a tranzakciókat, amelyek nem zárolják a gyökeret, a részfák soros sorrendjével konzisztens tetszőleges sorrendben.

Példa. Legyen T_1, T_2, …, T_{10} 10 darab tranzakció, és ezekből T_1, T_2 és T_3 ugyanebben a sorrendben zárolja a gyökeret. Tegyük fel, hogy a gyökérnek van két gyereke, az elsőt T_1-től T_7-ig zárolják a tranzakciók, a másikat pedig T_2, T_3, T_8, T_9 és T_{10} zárolja. Legyen az első részfához a soros sorrend (T_4, T_1, T_5, T_2, T_6, T_3, T_7). Ennek a sorrendnek T_1-et, T_2-t és T_3-at ebben a sorrendben kell tartalmaznia. A másik részfához tartozó soros sorrend legyen (T_8, T_2, T_9, T_{10}, T_3). Mint az előző esetben, a T_2 és T_3 tranzakciók, amelyek a gyökeret zárolják, abban a sorrendben fordulnak elő, ahogyan a gyökeret zárolták.

Ezeknek a tranzakcióknak a soros sorrendjére felállított megszorításokat a következő ábra mutatja:



A folyamatos nyilak a gyökér első gyerekének a rendezése szerinti megszorításokat jelölik, a szaggatott nyilak pedig a másik gyereknél lévő rendezést jelölik. Ennek a gráfnak több topologikus sorrendje létezik, az egyik: (T_4, T_8, T_1, T_5, T_2, T_9, T_6, T_{10}, T_3, T_7).

Konkurenciavezérlés időbélyegzőkkel

A következőkben a zárolástól különböző két másik módszert nézünk meg, amelyeket néhány rendszerben használnak a tranzakciók sorbarendezhetőségének biztosítására:

1. *Időbélyegzés* (timestamping, timestamp ordering – TO): Minden tranzakcióhoz hozzárendelünk egy „időbélyegzőt”. Minden adatbáziselem utolsó olvasását és írását végző tranzakció időbélyegzőjét rögzítjük, és összehasonlítjuk ezeket az értékeket, hogy biztosítsuk, hogy a tranzakciók időbélyegzőinek megfelelő soros ütemezés ekvivalens legyen a tranzakciók tényleges ütemezésével.
2. *Érvényesítés* (validation): Megvizsgáljuk a tranzakciók időbélyegzőit és az adatbáziselemeket, amikor a tranzakció véglegesítésre kerül. Ezt az eljárást a tranzakciók *érvényesítésének* nevezzük. Az a soros ütemezés, amely az érvényesítési idejük alapján rendezi a tranzakciókat, ekvivalens kell, hogy legyen a tényleges ütemezéssel.

Mindkét megközelítés *optimista* abban az értelemben, hogy feltételezik, nem fordul elő nem sorba rendezhető viselkedés, és csak akkor tisztázza a helyzetet, amikor ez nyilvánvalóan nem teljesül. Ezzel ellentétben minden zárolási módszer azt feltételezi, hogy „a dolgok rosszra fordulnak”, hacsak a tranzakciókat azonnal meg nem akadályozzák abban, hogy nem sorba rendezhető viselkedésük alakuljon ki. Az optimista megközelítések abban különböznek a zárolásoktól, hogy az egyetlen ellenszerük, amikor valami rosszra fordul, hogy azt a tranzakciót, amely nem sorba rendezhető viselkedést okozna, abortálják, majd újraindítják. A zárolási ütemezők ezzel ellentétben késleltetik a tranzakciókat, de nem abortálják őket, hacsak nem alakul ki holtpont. (Késleltetés az optimista megközelítések esetén is előfordulhat, annak érdekében, hogy kevesebb abortálásra legyen szükség.) Általában az optimista ütemezők akkor jobbak a zárolásinál, amikor sok tranzakció csak olvasási műveleteket hajt végre, ugyanis az ilyen tranzakciók önmagukban soha nem okozhatnak nem sorba rendezhető viselkedést.

Időbélyegzők

Annak érdekében, hogy az időbélyegzést konkurenciavezérlési módszerként használjuk, az ütemezőnek minden egyes `T` tranzakcióhoz hozzá kell rendelnie egy egyedi számot, a `TS(T)` *időbélyegzőt* (timestamp). Az időbélyegzőket növekvő sorrendben kell kiadni abban az időpontban, amikor a tranzakció az elindításáról először értesíti az ütemezőt. Két lehetséges megközelítés az időbélyegzők generálásához:

a) Az egyik lehetőség, hogy az időbélyegzőket a rendszeróra felhasználásával hozzuk létre, feltéve, hogy az ütemező nem működik annyira gyorsan, hogy két tranzakcióhoz ugyanazt az értéket rendelné időbélyegzőként.

b) A másik megközelítés szerint az ütemező karbantart egy számlálót. Minden alkalommal, amikor egy tranzakció elindul, a számláló növekszik eggyel, és ez az új érték lesz a tranzakció időbélyegzője. Ebben a megközelítésben az időbélyegzőknek semmi közük sincs az időhöz, azonban azzal a – bármely időbélyegző-generáló rendszer esetén szükséges – fontos tulajdonsággal rendelkeznek, miszerint egy később elindított tranzakció nagyobb időbélyegzőt kap, mint egy korábban elindított tranzakció.

Bármelyik módszert is használjuk az időbélyegzők generálására, az ütemezőnek karban kell tartania a jelenleg aktív tranzakciók és időbélyegzőik tábláját.

Ahhoz, hogy időbélyegzőket használjunk konkurenciavezérlési módszerként, minden egyes `X` adatbáziselemhez hozzá kell rendelnünk két időbélyegzőt és esetlegesen egy további bitet:

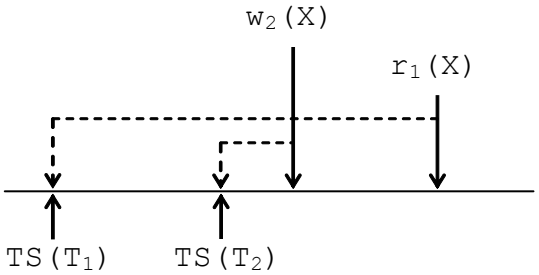
1. `RT(X)`: `X` olvasási ideje (read time), amely a legmagasabb időbélyegző, ami egy olyan tranzakcióhoz tartozik, amely már olvasta `X`-et.
2. `WT(X)`: `X` írási ideje (write time), amely a legmagasabb időbélyegző, ami egy olyan tranzakcióhoz tartozik, amely már írta `X`-et.

3. C(X): X véglegesítési bitje (commit bit), amely akkor és csak akkor igaz, ha a legújabb tranzakció, amely X-et írta, már véglegesítve van. Ez a bit nem feltétlenül szükséges, és az a célja, hogy elkerüljük azt a helyzetet, amelyben egy T tranzakció egy másik U tranzakció által írt adatokat olvas be, és utána U-t abortáljuk. Ez a probléma, amikor T nem véglegesített adatok „piszkos olvasását” hajtja végre, az adatbázis-állapot inkonzisztenssé válását is okozhatja. Így bármely ütemezőhöz szükség van olyan mechanizmusra, amely megakadályozza a piszkos olvasást (bár a gyakorlatban az adatbázis-kezelő rendszerek általában a felhasználóra bízzák, hogy megengedhetők-e a piszkos olvasások; lásd később a „nem olvasásbiztos” tranzakcióelkülönítési szintet).

Fizikailag nem megvalósítható viselkedések

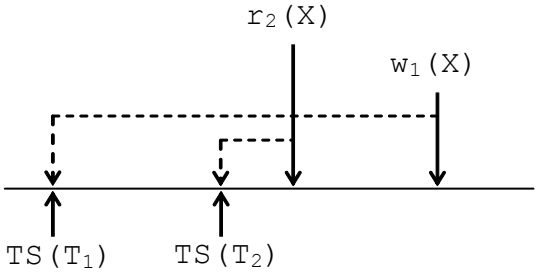
Hogy megértsük az időbélyegzőn alapuló ütemező felépítését és szabályait, tudnunk kell, hogy az ütemező feltételezi, hogy a tranzakciók időbélyegző szerinti sorrendje egyúttal olyan soros sorrend, amely a végrehajtás sorrendjét is jelenti. Így az ütemező feladata azon túl, hogy hozzárendeli az időbélyegzőket a tranzakciókhoz, és módosítja RT-t, WT-t és C-t az egyes adatbáziselemekhez kötődően, még az is, hogy ellenőrzi, amikor egy olvasás vagy írás fordul elő, hogy az úgy történt volna-e valós időben is, ha minden tranzakciót azonnal, az időbélyegző által jelzett időpillanatban hajtottunk volna végre. Ha nem, akkor azt mondjuk, hogy a viselkedés *fizikailag nem megvalósítható* (physically unrealizable behavior). Kétféle probléma merülhet fel:

1. *Túl késői olvasás* (read too late): A T_1 tranzakció megpróbálja olvasni az X adatbáziselemet, de X írási ideje azt jelzi, hogy X jelenlegi értékét azután írtuk, miután T_1-et már elméletileg végrehajtottuk, vagyis $TS(T_1) < WT(X)$. A következő ábra mutatja ezt a problémát:



   A vízszintes tengely jelenti a valós időt. A szaggatott vonalak kapcsolják össze a tényleges eseményt azzal az időponttal, amikor a tranzakciók időbélyegzője szerint elméletileg végre kellett volna hajtani az eseményt. Látjuk, hogy a T_2 tranzakciót a T_1 tranzakció után indítottuk el, mégis X értékét előbb írta, mint hogy T_1 beolvasta volna. T_1-nek nem a T_2 által írt értéket kellene olvasnia, ugyanis elméletileg T_2-t T_1 után hajtjuk végre. T_1-nek viszont nincs más választása, ugyanis X-nek a T_2 által írt értéke az egyetlen, amelyet T_1 most be tud olvasni. A megoldás, hogy T_1-et abortáljuk, amikor ez a probléma felmerül.

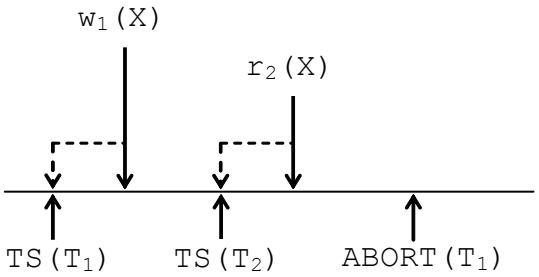
2. *Túl késői írás* (write too late): A T_1 tranzakció megpróbálja írni az X adatbáziselemet, de X olvasási ideje azt jelzi, hogy van egy másik tranzakció is, amelynek a T_1 által beírt értéket kellene olvasnia, ám ehelyett más értéket olvas, vagyis $TS(T_1) < RT(X)$. A következő ábra mutatja ezt a problémát:



Az ábra egy T_2 tranzakciót mutat, amelyet T_1 után indítottunk el, mégis előbb olvassa X-et, mint T_1-nek lehetősége lett volna írni. Amikor T_1 megpróbálja írni X-et, úgy találjuk, hogy $RT(X) > TS(T_1)$, ami azt jelenti, hogy a T_2 tranzakció már beolvasta X-et, amelyet elméletileg T_1 végrehajtása után kellett volna elvégeznie.

A piszkos adatok problémái

Van egy problémákból álló osztály, amelynek kezelésére bevezették a véglegesítési bitet. A problémák egyike a „piszkos olvasás”, amelyet a következő ábra szemléltet:

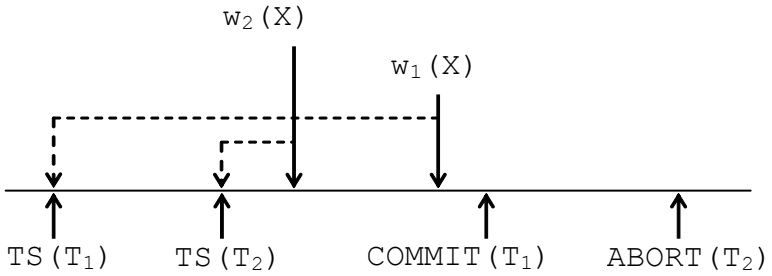


Itt a T_2 tranzakció olvassa X-et, amelyet utoljára T_1 írt. T_1 időbélyegzője kisebb, mint T_2-é, és a valóságban a T_2 általi olvasás a T_1 általi írás után történik, tehát úgy tűnik, hogy az esemény fizikailag megvalósítható. Mégis lehetséges, hogy miután T_2 beolvasta a T_1 által X-be írt értéket, a T_1 tranzakciót abortáljuk; például azért, mert T_1 talált valami hibát a saját működésében (például nullával való osztás), vagy az ütemező kényszeríti ki T_1 abortálását, mivel az valamilyen fizikailag nem megvalósítható viselkedést eredményező műveletet próbált végezni. Így, bár nincs fizikailag nem megvalósítható abban, hogy T_2 olvassa X-et, mégis jobb a T_2 általi olvasást azutánra elhalasztani, hogy T_1 véglegesítését vagy abortálását már elvégeztük, különben az ütemezésünk nem lesz konfliktus-sorbarendezhető. Azt, hogy T_1 még nincs véglegesítve, onnan tudjuk, hogy a $C(X)$ véglegesítési bit hamis.

A piszkos olvasás problémája véglegesítési bit nélkül is megoldható: Amikor abortálunk egy T tranzakciót, meg kell néznünk, hogy vannak-e olyan tranzakciók, amelyek olvastak egy vagy több T által írt adatbáziselemet. Ha igen, akkor azokat is abortálnunk kell. Ebből aztán további abortálások következhetnek, azokból megint újabbak, és így tovább. Ezt a szituációt *kaszkádolt visszagörgetésnek* (cascading rollback) nevezzük. Ez a megoldás azonban alacsonyabb fokú konkurenciát engedélyez, mint a véglegesítési bit bevezetése és a késleltetés, ráadásul előfordulhat, hogy *nem helyreállítható ütemezést* (nonrecoverable schedule) kapunk. Ez abban az esetben következik be, ha az egyik abortálandó tranzakciót már véglegesítettük.

Drasztikus, de nagyon egyszerű megoldás a piszkos olvasás problémájára, hogy minden olyan tranzakciót abortálunk, amely piszkos adatot szeretne olvasni. Végül megoldást jelenthet a többváltozatú időbélyegzés alkalmazása is (lásd később).

Egy másik lehetséges problémát a következő ábra szemléltet:



Itt T_2, a T_1-nél későbbi időbélyegzővel rendelkező tranzakció írja először X-et. Amikor T_1 írni próbál, a megfelelő művelet semmit sem végez, tehát elhagyható. Nyilvánvalóan nincs más T_3 tranzakció, amelynek X-ből a T_1 által beírt értéket kellene beolvasnia, és ehelyett a T_2 által írt értéket olvasná, ugyanis ha T_3 megpróbálná olvasni X-et, abortálnia kellene a túl késői olvasás miatt. X későbbi olvasásainál a T_2 által írt értéket kell olvasni, vagy X még későbbi, de nem T_1 által írt értékét. Ezt az ötletet, miszerint azokat az írásokat kihagyhatjuk, amelyeknél későbbi írási idejű írást már elvégeztünk, *Thomas-féle írási szabálynak* (Thomas' write rule) nevezzük.

A Thomas-féle írási szabállyal azonban van egy lényegi probléma. Ha T_2-t később abortáljuk, amint az az ábrán látható, akkor X-nek a T_2 által írt értékét ki kell törölnünk, továbbá az előző értéket és írási időt vissza kell állítanunk. Minthogy T_1-et véglegesítettük, úgy látszik, hogy X T_1 által írt értékét kell a későbbi olvasásokhoz használnunk. Mi viszont kihagytuk a T_1 általi írást, és már túl késő, hogy helyrehozhassuk ezt a hibát.

A problémát a következőképpen kezelhetjük: Amikor a T_1 tranzakció írja az X adatbáziselemet, és azt látjuk, hogy X írási ideje nagyobb T_1 időbélyegzőjénél (azaz $TS(T_1) < WT(X)$), valamint hogy az X-et író tranzakció (T_2) még nincs véglegesítve (azaz $C(X)$ hamis), akkor T_1-et késleltetjük mindaddig, amíg $C(X)$ igazzá nem válik; vagy azért, mert T_2 véglegesítődik, vagy azért, mert abortál. Ha T_2 véglegesítődik, akkor T_1 írását elhagyjuk, ha viszont abortál, akkor végrehajtjuk.

Természetesen most is létezik másik megoldás: a fenti feltételek teljesülése esetén T_1-et a késleltetés helyett egyszerűen visszagörgetjük. Nyilván ez a megoldás alacsonyabb fokú konkurenciát engedélyez, mint a véglegesítési bit bevezetése és a késleltetés, ráadásul ha a piszkos olvasásokat is visszagörgetéssel kezeljük, akkor ez az abortálás tovább növeli a kaszkádolt visszagörgetés és a nem helyreállítható ütemezés kockázatát. Végül a harmadik megoldás ebben az esetben is a többváltozatú időbélyegzés alkalmazása.

Látható, hogy az időbélyegzési technika alapváltozatában (amikor nem használunk véglegesítési bitet és nincs késleltetés) nem léphet fel holtponti helyzet, előfordulhat viszont kaszkádolt visszagörgetés és nem helyreállítható ütemezés.

Az időbélyegzőn alapuló ütemezések szabályai

Összegezhetjük azokat a szabályokat, amelyeket az időbélyegzőket használó ütemezőnek követnie kell ahhoz, hogy biztosan konfliktus-sorbarendezhető ütemezést kapjunk. Mi most az időbélyegzésnek a véglegesítési bittel bővített változatát tekintjük. Az ütemezőnek egy T tranzakciótól érkező olvasási vagy írási kérésre adott válaszában az alábbi választásai lehetnek:

a) Engedélyezi a kérést.
b) Abortálja T-t (ha T „megsérti a fizikai valóságot"), és egy új időbélyegzővel újraindítja. Azt az abortálást, amelyet újraindítás követ, gyakran *visszagörgetésnek* (rollback) nevezzük.
c) Késlelteti T-t, és később dönti el, mi történjen (ha a kérés olvasás, és az olvasás piszkos is lehet, illetve ha a kérés írás, és alkalmazható lehet a Thomas-féle írási szabály).

A szabályok a következők:

1. Tegyük fel, hogy az ütemezőhöz érkező kérés $r_T(X)$:
   a) Ha $TS(T) \geq WT(X)$, az olvasás fizikailag megvalósítható:
      i) Ha $C(X)$ igaz vagy $TS(T) = WT(X)$, engedélyezzük a kérést. Ha $TS(T) > RT(X)$, akkor $RT(X) := TS(T)$, egyébként nem változtatjuk meg $RT(X)$-et.
      ii) Ha $C(X)$ hamis és $TS(T) > WT(X)$, késleltessük T-t addig, amíg $C(X)$ igazzá nem válik (azaz az X-et utoljára író tranzakció nem véglegesítődik vagy abortál).
   b) Ha $TS(T) < WT(X)$, az olvasás fizikailag nem megvalósítható: Visszagörgetjük T-t, vagyis abortáljuk, és újraindítjuk egy új, nagyobb időbélyegzővel.
2. Tegyük fel, hogy az ütemezőhöz érkező kérés $w_T(X)$:
   a) Ha $TS(T) \geq RT(X)$ és $TS(T) \geq WT(X)$, az írás fizikailag megvalósítható, és az alábbiakat kell végrehajtani:
      i) X új értékének beírása;
      ii) $WT(X) := TS(T)$;
      iii) $C(X)$:= hamis.
   b) Ha $TS(T) \geq RT(X)$, de $TS(T) < WT(X)$, akkor az írás fizikailag megvalósítható, de X-nek már egy későbbi értéke van.
      i) Ha $C(X)$ igaz, az X előző írását végző tranzakció véglegesítve van, így egyszerűen figyelmen kívül hagyjuk X T általi írását; megengedjük, hogy T folytatódjon, és ne változtassa meg az adatbázist.
      ii) Ha viszont $C(X)$ hamis, akkor késleltetnünk kell T-t, mégpedig az 1. a) ii) pontban leírtak szerint.
   c) Ha $TS(T) < RT(X)$, az írás fizikailag nem megvalósítható, és T-t vissza kell görgetnünk.
3. Tegyük fel, hogy az ütemezőhöz érkező kérés T véglegesítése (COMMIT T). Meg kell találnunk (egy, az ütemező által karbantartott lista alapján) az összes olyan X adatbáziselemet, amelybe T írt utoljára ($WT(X) = TS(T)$), és állítsuk be a hozzájuk tartozó $C(X)$ biteket igazra. Ha vannak X „véglegesítésére” várakozó tranzakciók az 1. a) ii) és a 2. b) ii) pontoknak megfelelően (ezeket a tranzakciókat az ütemező egy másik listáján találjuk meg), akkor meg kell ismételnünk ezen tranzakciók olvasási vagy írási kísérleteit.
4. Tegyük fel, hogy az ütemezőhöz érkező kérés T abortálása (ABORT T) vagy visszagörgetése, mint az 1. b) vagy a 2. c) esetben. Ekkor visszavonjuk az abortált tranzakció azon írásait, amelyek olyan X adatbáziselemekre vonatkoznak, amelyekre $WT(X) = TS(T)$. Ez azt jelenti, hogy visszaállítjuk ezen adatbáziselemeknek és azok írási idejének régi értékét (azt, amelyik a legnagyobb írási időhöz tartozik), valamint igazra állítjuk a véglegesítési bitet, ha az írási időhöz tartozó tranzakció már véglegesítődött. Ezenkívül „visszavonjuk” T olvasásait is, azaz visszaállítjuk az olyan T által olvasott adatbáziselemek olvasási idejének régi (legnagyobb) értékét, amelyekre $RT(X) = TS(T)$. Ezután bármely olyan tranzakcióra, amely egy X elem T általi írása miatt várakozik (1. a) ii) és 2. b) ii)), meg kell ismételnünk az olvasási vagy írási kísérletet, és meglátjuk, hogy a művelet most jogszerű-e.

Példa. A következő ábrán három tranzakció (T_1, T_2 és T_3) ütemezése látható, amelyek három adatbáziselemhez (A, B és C) férnek hozzá:

T_1	T_2	T_3	A	B	C
200	150	175	RT = 0 WT = 0 C = igaz	RT = 0 WT = 0 C = igaz	RT = 0 WT = 0 C = igaz
r_1(B);				RT = 200	
	r_2(A);		RT = 150		
		r_3(C);			RT = 175
w_1(B);				WT = 200 C = hamis	
w_1(A);			WT = 200 C = hamis		
	w_2(C);				
	abortál		RT = 0		
véglegesítődik			C = igaz	C = igaz	
		w_3(A);			

Az események előfordulásának ideje szokás szerint lefelé nő. Legyen kezdetben minden adatbáziselemhez az olvasási és az írási idő is 0. A tranzakciók abban a pillanatban kapnak időbélyegzőt, amikor értesítik az ütemezőt az elindításukról. Most például bár T_1 hajtja végre az első adathozzáférést, mégsem neki van a legkisebb időbélyegzője. Tegyük fel, hogy T_2 az első, amelyik az indításáról értesíti az ütemezőt, T_3 volt a következő, és T_1-et indítottuk el utoljára.

Az első műveletben T_1 beolvassa B-t. Mivel B írási ideje kisebb, mint T_1 időbélyegzője, ez az olvasás fizikailag megvalósítható, és engedélyezzük a végrehajtást. B olvasási idejét 200-ra, T_1 időbélyegzőjére állítjuk. A második és a harmadik olvasási művelet hasonlóan jogszerű, és mindegyik adatbáziselem olvasási idejének értékét az őt olvasó tranzakció időbélyegzőjére állítjuk.

A negyedik lépésben T_1 írja B-t. Mivel B olvasási ideje nem nagyobb, mint T_1 időbélyegzője, az írás fizikailag megvalósítható. Mivel B írási ideje nem nagyobb, mint T_1 időbélyegzője, ténylegesen végre kell hajtanunk az írást. Amikor ezt elvégeztük, B írási idejét 200-ra növeljük, amely az őt felülíró T_1 tranzakció időbélyegzője. Ezután hasonlóan járunk el A-val.

Ezután T_2 megpróbálja írni C-t. C-t viszont már beolvasta a T_3 tranzakció, amelyet elméletileg a 175-ös időpontban hajtottunk végre, míg T_2-nek az értéket a 150-es időpontban kellett volna beírnia. Így T_2 olyan dologgal próbálkozik, amely fizikailag nem megvalósítható viselkedést eredményezne, tehát T_2-t vissza kell görgetnünk.

Az utolsó lépés, hogy T_3 írja A-t. Mivel A olvasási ideje (150) kevesebb, mint T_3 időbélyegzője (175), az írás jogszerű. Viszont A-nak már egy későbbi értéke van tárolva ebben az adatbáziselemben, mégpedig a T_1 által – elméletileg a 200-as időpontban – beírt érték. T_3-at tehát nem görgetjük vissza, de be sem írjuk az értéket. (Feltesszük, hogy T_1 időközben véglegesítődött.)

Többváltozatú időbélyegzés

Az időbélyegzés egyik fontos változata, a *többváltozatú időbélyegzés* (multiversion timestamping, multiversion timestamp ordering – MVTO, multiversion concurrency control – MVCC) karbantartja az adatbáziselemek régi változatait is a magában az adatbázisban tárolt jelenlegi változaton kívül. A cél az, hogy megengedjünk olyan $r_T(X)$ olvasásokat, amelyek egyébként a T tranzakció abortálását okoznák (ugyanis X jelenlegi változatát egy T-nél későbbi tranzakció írta felül). Ilyenkor T-t X megfelelő régebbi változatának beolvasásával folytatjuk. A módszer különösen hasznos, ha az adatbáziselemek lemezblokkok vagy lapok, ugyanis ekkor csak annyit kell a pufferkezelőnek biztosítania, hogy bizonyos blokkok a memóriában legyenek, amelyek néhány jelenleg aktív tranzakció számára hasznosak lehetnek.

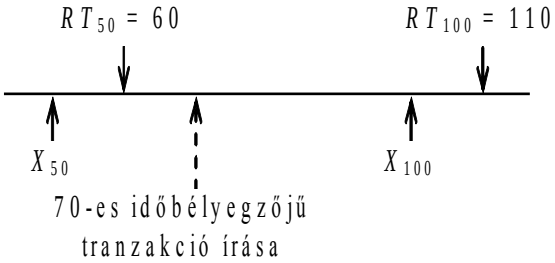
Példa. Tekintsük a következő ábrán szereplő, az A adatbáziselemhez hozzáférő tranzakciókat:

T_1	T_2	T_3	T_4	A
150	200	175	225	RT = 0 WT = 0
$r_1(A)$;				RT = 150
$w_1(A)$;				WT = 150
	$r_2(A)$;			RT = 200
	$w_2(A)$;			WT = 200
		$r_3(A)$;		
		abortál		
			$r_4(A)$;	RT = 225

Ezek a tranzakciók egy hagyományos, időbélyegzőn alapuló ütemező alatt működnek. Amikor T_3 megpróbálja olvasni A-t, azt találja, hogy WT(A) nagyobb, mint a saját időbélyegzője, így abortálni kell. Viszont megvan A-nak a T_1 által írt, és a T_2 által felülírt régi értéke, amely alkalmas lenne T_3-nak, hogy olvassa. Ebben a változatában A-nak 150 volt az írási ideje, ami kevesebb, mint T_3 175-ös időbélyegzője. Ha A-nak ez a régi értéke hozzáférhető lenne, T_3 engedélyt kaphatna az olvasásra, még ha ez A-nak nem is a „jelenlegi" értéke.

A többváltozatú időbélyegzést használó ütemező az alábbiakban különbözik a fent leírt ütemezőtől:

1. Amikor egy új $w_T(X)$ írás fordul elő, ha ez jogszerű, akkor az X adatbáziselemnek egy új változatát hozzuk létre, amelynek az írási ideje TS(T), és X_t-vel fogunk rá hivatkozni, ahol $t = TS(T)$.
2. Amikor egy $r_T(X)$ olvasás fordul elő, az ütemező megkeresi X-nek azt az X_t változatát, amelyre $t \le TS(T)$, de nincs más $X_{t'}$ változata, amelyre $t < t' \le TS(T)$ lenne. Vagyis X-nek azt a változatát olvassa be T, amelyet T elméleti végrehajtása előtt közvetlenül írtak.
3. Az írási időket egy elem változataihoz rendeljük, és soha nem változtatjuk meg.
4. Az olvasási időket szintén rendelhetjük a változatokhoz. Arra használjuk őket, hogy ne kelljen visszautasítanunk bizonyos írásokat, mégpedig azokat, amelyek ideje nagyobb vagy egyenlő, mint az őt időben közvetlenül megelőző változat olvasási ideje. Ha csak az utolsó változat olvasási idejét tartanánk nyilván, akkor az ilyen írásokat el kellene utasítanunk. A problémát a következő ábra szemlélteti:



   X változatai X_{50} és X_{100}. X_{50} a 60-as időpontban olvasásra került, és megjelent a 70-es időbélyegzőjű T tranzakció általi új írás. Ez az írás jogszerű, mert $RT_{50} \le TS(T)$. Ha csak az utolsó változat 110-es olvasási idejét tárolnánk, akkor erről az írásról nem tudnánk eldönteni, hogy jogszerű-e, ezért abortálnunk kellene T-t.
5. Amikor egy X_t változat t írási ideje olyan, hogy nincs t-nél kisebb időbélyegzőjű aktív tranzakció, akkor törölhetjük X-nek az X_t-t megelőző változatait.

Példa. Tekintsük újból az előző példában szereplő műveleteket, de most használjunk többváltozatú időbélyegzést:

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225	RT = 0		
$r_1(A)$;				olvasás, RT = 150		
$w_1(A)$;					létrehozás, RT = 150	
	$r_2(A)$;				olvasás, RT = 200	
	$w_2(A)$;					létrehozás, RT = 200
		$r_3(A)$;			olvasás	
			$r_4(A)$;			olvasás, RT = 225

A-nak három változata létezik: A_0, amelyik a tranzakciók elindítása előtt létezik, A_{150}, amelyet T_1 írt, és A_{200}, amelyet T_2 írt. Az ábra mutatja azt az eseménysorozatot, amikor az egyes változatokat létrehozzuk, illetve beolvassuk. T_3-at most nem kell abortálni, ugyanis be tudja olvasni A-nak egy korábbi változatát.

A többváltozatú időbélyegzés tehát kiküszöböli a túl késői olvasásokat. Mi a helyzet a piszkos olvasással és a Thomas-féle írási szabály problémájával? Piszkos olvasás most is előfordulhat, de most nemcsak a tranzakció késleltetésével vagy abortálásával tehetünk ellene, hanem azzal is, hogy olvasáskor megkeressük az adatbáziselem utolsó olyan változatát, amelyet vagy maga az olvasó tranzakció, vagy egy, az olvasó tranzakció indulásakor már *véglegesített* tranzakció hozott létre. Így sosem olvasunk piszkos adatot, nem kell késleltetnünk egy tranzakciót sem, ráadásul nem fordulhat elő túl késői írás sem, hiszen a „túl későn író" tranzakció még nem lehetett véglegesítve az olvasó tranzakció indulásakor, amelynek emiatt nincs szüksége a „túl későn írt" értékre. Ezt a technikát (amelyet az Oracle is alkalmaz) *pillanatkép-elkülönítésnek* (snapshot isolation) nevezzük. Hátránya, hogy nem garantálja a sorbarendezhetőséget.

A Thomas-féle írási szabály pedig nem alkalmazható többváltozatú időbélyegzés esetén (legalábbis eredeti formájában), még akkor is létrehozzuk az adatbáziselem „új" változatát, ha az régebbi, mint a legújabb változat.

Időbélyegzők és zárolások

Általában az időbélyegzés azokban a helyzetekben kiváló, amikor a tranzakciók többsége csak olvasási, vagy ritka az az eset, hogy konkurens tranzakciók ugyanazt az elemet próbálják meg olvasni és írni. Az erősen konfliktusos helyzetekben jobb a zárolásokat használni. Ehhez az ökölszabályhoz az érvek az alábbiak:

- A zárolások gyakran késleltetik a tranzakciókat azzal, hogy a zárakra várnak, és még holtpontok is kialakulhatnak, amikor néhány tranzakció hosszú ideje várakozik, és ekkor az egyiket vissza kell görgetni.
- Időbélyegzés használatakor viszont ha a konkurens tranzakciók gyakran olvasnak és írnak közös elemeket, akkor a visszagörgetés lesz gyakori, ami még több késedelmet okoz, mint egy zárolási rendszer.

Bizonyos rendszerek érdekes kompromisszumot alkalmaznak: Az ütemező felosztja a tranzakciókat csak olvasási tranzakciókra és olvasási/írási tranzakciókra. Az olvasási/írási tranzakciókat kétfázisú zárolást használva hajtjuk végre úgy, hogy a zárolt elemek hozzáférését megakadályozzuk a többi tranzakciónak. A csak olvasási tranzakciókat a többváltozatú időbélyegzéssel hajtjuk végre. Amikor az olvasási/írási tranzakciók létrehozzák egy adatbáziselem új változatait, ezeket a változatokat úgy kezeljük, ahogyan fentebb leírtuk. Egy csak olvasási tranzakciónak megengedjük, hogy egy adatbáziselem bármelyik olyan változatát olvassa, amely korábban jött létre, mint a tranzakció időbélyegzője. Csak olvasási tranzakciókat emiatt soha nem kell abortálnunk, és csak nagyon ritkán kell késleltetnünk.

Konkurenciavezérlés érvényesítéssel

Az *érvényesítés* (validation, Kung–Robinson-modell) az optimista konkurenciavezérlés másik típusa, amelyben a tranzakcióknak megengedjük, hogy zárolások nélkül hozzáférjenek az adatokhoz, és a megfelelő időben ellenőrizzük a tranzakció sorba rendezhető viselkedését. Az érvényesítés alapvetően abban különbözik az időbélyegzéstől, hogy itt az ütemező nyilvántartást vezet arról, mit tesznek az aktív tranzakciók, ahelyett hogy az összes adatbáziselemhez feljegyezné az olvasási és írási időt. Mielőtt a tranzakció írni kezdene értékeket az adatbáziselemekbe, egy „érvényesítési fázison" megy keresztül, amikor a beolvasott és kiírandó elemek halmazait összehasonlítjuk más aktív tranzakciók írásainak halmazaival. Ha fellép a fizikailag nem megvalósítható viselkedés kockázata, a tranzakciót visszagörgetjük.

Az érvényesítésen alapuló ütemező felépítése

Ha az érvényesítést használjuk konkurenciavezérlési módszerként, az ütemezőnek meg kell adnunk minden `T` tranzakcióhoz a `T` által olvasott és a `T` által írt adatbáziselemek halmazát: `RS(T)` az *olvasási halmaz*, `WS(T)` az *írási halmaz*. A tranzakciókat három fázisban hajtjuk végre:

1. *Olvasás*. Az első fázisban a tranzakció beolvassa az adatbázisból az összes szükséges elemet az olvasási halmazába, majd kiszámítja a lokális változóiban az összes eredményt, amelyet ki fog írni, ezzel meghatározva az írási halmazt is.
2. *Érvényesítés*. A második fázisban az ütemező érvényesíti a tranzakciót oly módon, hogy összehasonlítja az olvasási és írási halmazait a többi tranzakcióéval. Az érvényesítési eljárást később részletezzük. Ha az érvényesítés hibát jelez, akkor a tranzakciót visszagörgetjük, egyébként pedig folytatódik a harmadik fázissal.
3. *Írás*. A harmadik fázisban a tranzakció az írási halmazában lévő elemek értékeit kiírja az adatbázisba.

Intuitív alapon minden sikeresen érvényesített tranzakcióról azt gondolhatjuk, hogy az érvényesítés pillanatában került végrehajtásra. Így az érvényesítésen alapuló ütemező a tranzakciók feltételezett soros sorrendjével dolgozik. Annak a döntésnek az alapja, hogy érvényesítsen-e egy tranzakciót vagy sem, az, hogy a tranzakciók viselkedése konzisztens legyen ezzel a soros sorrenddel. A döntés segítéséhez az ütemező fenntart három halmazt:

1. *KEZD*: a már elindított, de még nem teljesen érvényesített tranzakciók halmaza. Ebben a halmazban az ütemező minden `T` tranzakcióhoz karbantartja `KEZD(T)`-t, amely `T` indításának időpontja.
2. *ÉRV*: a már érvényesített, de a harmadik fázisban az írásokat még be nem fejezett tranzakciók halmaza. Ebben a halmazban az ütemező minden `T` tranzakcióhoz karbantartja `KEZD(T)`-t, és `T` érvényesítésekor `ÉRV(T)`-t. `ÉRV(T)` az az idő, amikor `T` végrehajtását gondoljuk a végrehajtás feltételezett soros sorrendjében.
3. *BEF*: a harmadik fázist befejezett tranzakciók halmaza. Ezekhez a `T` tranzakciókhoz az ütemező rögzíti `KEZD(T)`-t, `ÉRV(T)`-t, és `T` befejezésekor `BEF(T)`-t. Elméletben ez a halmaz nő, de – mint látni fogjuk – nem kell megjegyeznünk a `T` tranzakciót, ha `BEF(T)` $<$ `KEZD(U)` bármely `U` aktív tranzakcióra (vagyis $\forall$`U` $\in$ *KEZD* $\cup$ *ÉRV* esetén). Az ütemező így időnként tisztogathatja a *BEF* halmazt, hogy megakadályozza méretének korlátlan növekedését.

Az érvényesítési szabályok

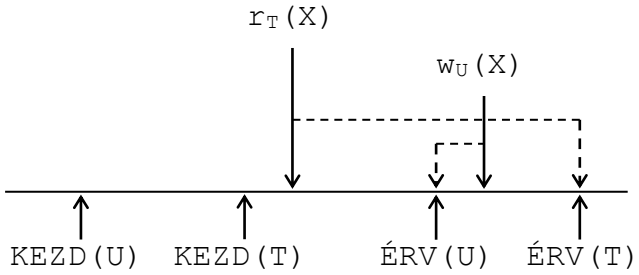
Ha az ütemező elvégzi a fenti halmazok karbantartását, akkor segítségükkel észlelheti a tranzakciók feltételezett soros sorrendjének (azaz a tranzakciók érvényesítési sorrendjének) bármely lehetséges

megsértését. A szabályok megértése végett először vizsgáljuk meg, hogy mi lehet hibás, amikor a T tranzakciót megpróbáljuk érvényesíteni:

1. Tegyük fel, hogy van olyan U tranzakció, melyre teljesülnek a következő feltételek:

   a) $U \in \acute{E}RV \cup BEF$, vagyis U-t már érvényesítettük.
   b) $BEF(U) > KEZD(T)$, vagyis U nem fejeződött be T indítása előtt. (Ha $U \in \acute{E}RV$, vagyis U még nem fejeződött be T érvényesítésekor, akkor BEF(U) technikailag nem definiált, de az biztos, hogy KEZD(T)-nél nagyobbnak kell lennie.)
   c) $RS(T) \cap WS(U) \neq \emptyset$, legyen X egy eleme ennek a halmaznak.

   Ekkor lehetséges, hogy U azután írja X-et, miután T olvassa azt („túl korai olvasás”). Elképzelhető az is, hogy U még nem írta X-et. Az előbbi eset a következő ábrán látható:

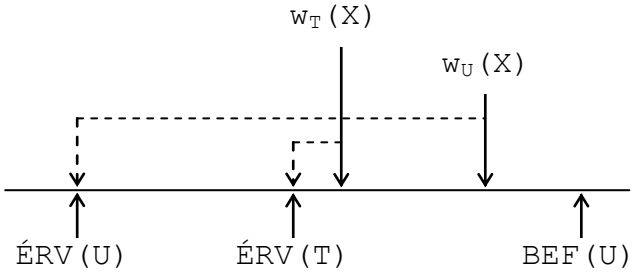


   A szaggatott vonalak kapcsolják össze a valós idejű eseményeket azzal az idővel, amikor be kellett volna következniük, ha a tranzakciókat az érvényesítés pillanatában hajtottuk volna végre. Mivel nem tudjuk, hogy T beolvasta-e az U-tól származó értéket, vissza kell görgetnünk T-t, hogy elkerüljük annak kockázatát, hogy T és U műveletei nem lesznek konzisztensek a feltételezett soros sorrenddel.

2. Tegyük fel, hogy van olyan U tranzakció, melyre teljesülnek a következő feltételek:

   a) $U \in \acute{E}RV$, vagyis U-t már érvényesítettük.
   b) $BEF(U) > \acute{E}RV(T)$, vagyis U-t nem fejeztük be, mielőtt T az érvényesítési fázisába lépett. (Ez a feltétel valójában mindig teljesül, mivel U még biztosan nem fejeződött be.)
   c) $WS(T) \cap WS(U) \neq \emptyset$, legyen X egy eleme ennek a halmaznak.

   Ekkor a lehetséges problémát a következő ábra szemlélteti:

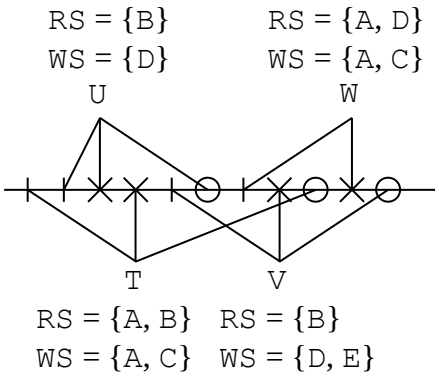


   Mind T-nek, mind U-nak írnia kell X értékét, és ha megengedjük T érvényesítését, lehetséges, hogy U előtt fogja írni X-et („túl korai írás”). Mivel nem lehetünk biztosak a dolgunkban, visszagörgetjük T-t, hogy biztosan ne szegjük meg azt a feltételezett soros sorrendet, amelyben T követi U-t.

A fent leírt két problémával kerülhetünk csak olyan helyzetbe, amikor a T által végzett művelet fizikailag nem megvalósítható. Az 1. esetben ha U T elindítása előtt fejeződött volna be, akkor T biztosan olyan X értéket olvasna, amelyet vagy U, vagy valamely későbbi tranzakció írt. A 2. esetben ha U T érvényesítése előtt fejeződik be, akkor biztos, hogy U T előtt írta X-et. Ezek alapján a T tranzakció érvényesítésére vonatkozó észrevételeinket az alábbi szabállyal foglalhatjuk össze:

- Összehasonlítjuk RS(T)-t WS(U)-val, és ellenőrizzük, hogy RS(T) ∩ WS(U) = ∅ minden olyan érvényesített U-ra, amely még nem fejeződött be T elindítása előtt, vagyis U ∈ *ÉRV* ∪ *BEF* és BEF(U) > KEZD(T).
- Összehasonlítjuk WS(T)-t WS(U)-val, és ellenőrizzük, hogy WS(T) ∩ WS(U) = ∅ minden olyan érvényesített U-ra, amely még nem fejeződött be T érvényesítése előtt, vagyis U ∈ *ÉRV* és BEF(U) > ÉRV(T).

Példa.



Az ábra egy idővonalat ábrázol, amely mentén négy tranzakció (T, U, V és W) végrehajtási és érvényesítési kísérletei láthatók. I-vel jelöltük az indítást, X-szel az érvényesítést, O-val pedig a befejezést. Az ábrán láthatók az egyes tranzakciók olvasási és írási halmazai. T-t indítjuk el elsőnek, de U-t érvényesítjük először.

1. Amikor U-t érvényesítjük, nincs más érvényesített tranzakció, így nem kell semmit sem ellenőriznünk. U-t érvényesítjük, és beírjuk az új értéket a D adatbáziselembe.
2. Amikor T-t érvényesítjük, U már érvényesítve van, de még nincs befejezve. Így ellenőriznünk kell, hogy T-nek sem az olvasási, sem az írási halmazában nincs semmi közös WS(U) = {D}-vel. Mivel RS(T) = {A, B} és WS(T) = {A, C}, mindkét halmazzal a metszet üres, tehát T-t érvényesítjük.
3. Amikor V-t érvényesítjük, U már érvényesítve van és befejeződött, T pedig szintén érvényesítve van, de még nem fejeződött be. Továbbá V-t U befejeződése előtt indítottuk el. Így össze kell hasonlítanunk mind RS(V)-t, mind WS(V)-t WS(T)-vel, azonban csak RS(V)-t kell összehasonlítanunk WS(U)-val. Az eredmények:
   - RS(V) ∩ WS(T) = {B} ∩ {A, C} = ∅;
   - WS(V) ∩ WS(T) = {D, E} ∩ {A, C} = ∅;
   - RS(V) ∩ WS(U) = {B} ∩ {D} = ∅.

   Ezek alapján V-t érvényesítjük.
4. Amikor W-t érvényesítjük, azt tapasztaljuk, hogy U már W elindítása előtt befejeződött, így nem kell elvégeznünk W és U összehasonlítását. T W érvényesítése előtt fejeződött be, de nem fejeződött be W elindítása előtt, ezért csak RS(W)-t kell összehasonlítanunk WS(T)-vel. V már érvényesítve van, de még nem fejeződött be, így össze kell hasonlítanunk mind RS(W)-t, mind WS(W)-t WS(V)-vel. Az eredmények:
   - RS(W) ∩ WS(T) = {A, D} ∩ {A, C} = {A};
   - RS(W) ∩ WS(V) = {A, D} ∩ {D, E} = {D};
   - WS(W) ∩ WS(V) = {A, C} ∩ {D, E} = ∅.

Mivel a metszetek nem mind üresek, `W`-t nem érvényesítjük, hanem visszagörgetjük, így nem ír értéket sem `A`-ba, sem `C`-be.

Többprocesszoros rendszerek esetén ha több ütemező végzi a feldolgozást, akkor lehet, hogy egyszerre érvényesítenek több tranzakciót. Ebben az esetben a többprocesszoros rendszer olyan szinkronizációs működésére kell támaszkodnunk, amely biztosítja, hogy az érvényesítés atomi tevékenységként kerüljön végrehajtásra. Egyprocesszoros rendszereken ha csak egy ütemező fut, akkor azt gondolhatjuk az érvényesítésről és az ütemező többi tevékenységéről, hogy egy pillanat alatt hajtódnak végre. Ebben az esetben tehát nem fordulhat elő, hogy egy tranzakció érvényesítése egy másik tranzakció érvényesítése alatt fejeződik be.

A három konkurenciavezérlési technika működésének összehasonlítása

A sorbarendezhetőség biztosításához három megközelítést néztünk meg: a zárolást, az időbélyegzést és az érvényesítést. Hasonlítsuk őket össze először a tárigény szempontjából:

- *Zárolás*: A zártábla által lefoglalt tár a zárolt adatbáziselemek számával arányos.
- *Időbélyegzés*: Egy naiv megvalósításban minden adatbáziselem olvasási és írási idejéhez szükségünk van tárra, akár hozzáférünk az adott elemhez, akár nem. Egy körültekintőbb megvalósítás azonban az összes olyan időbélyegzőt mínusz végtelen értékűnek tekinti, amely a legkorábbi aktív tranzakciónál korábbi tranzakcióhoz tartozik, és nem rögzíti ezeket. Ez esetben a zártáblával analóg méretű táblában tudjuk tárolni az olvasási és írási időket, amelyben csak a legújabban elért adatbáziselemek szerepelnek.
- *Érvényesítés*: Tárat használunk az időbélyegzőkhöz és minden jelenleg aktív tranzakció olvasási/írási halmazaihoz, hozzávéve még egy pár olyan tranzakciót, amelyek azután fejeződnek be, miután valamelyik jelenleg aktív tranzakció elkezdődött.

Így mindegyik megközelítésben az összes aktív tranzakcióra felhasznált tár a tranzakciók által elért adatbáziselemek számának az összegével megközelítőleg arányos. Az időbélyegzés és az érvényesítés kicsit több helyet használhat fel, ugyanis nyomon kell követnünk a korábban véglegesített tranzakciók bizonyos hozzáféréseit, amelyeket a zártábla nem rögzítene. Az érvényesítéssel kapcsolatban egy lényeges probléma, hogy a tranzakcióhoz tartozó írási halmazt az írások elvégzése előtt kell már ismernünk (de a tranzakció számításainak befejeződése után).

Összehasonlíthatjuk a módszereket abból a szempontból is, hogy késleltetés nélkül befejeződnek-e a tranzakciók. A három módszer hatékonysága attól függ, hogy vajon a tranzakciók közötti egymásra hatás erős vagy gyenge, azaz milyen valószínűséggel akar egy tranzakció hozzáférni egy olyan elemhez, amelyhez egy konkurens tranzakció már hozzáfért:

- A zárolás késlelteti a tranzakciókat, azonban elkerüli a visszagörgetéseket, még ha erős is az egymásra hatás. Az időbélyegzés és az érvényesítés nem késlelteti a tranzakciókat, azonban visszagörgetést okozhatnak, amely a késleltetésnek egy problémásabb formája, azonfelül erőforrásokat is pazarol.
- Ha gyenge az egymásra hatás, akkor sem az időbélyegzés, sem az érvényesítés nem okoz sok visszagörgetést, és előnyösebbek lehetnek a zárolásnál, ugyanis ezeknek általában alacsonyabbak a költségei, mint a zárolási ütemezőnek.
- Amikor szükséges a visszagörgetés, az időbélyegzés hamarabb feltárja a problémákat, mint az érvényesítés, amely mindig hagyja, hogy a tranzakció elvégezze az összes belső munkáját, mielőtt megnézné, hogy vissza kell-e görgetni a tranzakciót.

Az Oracle konkurenciavezérlési technikája

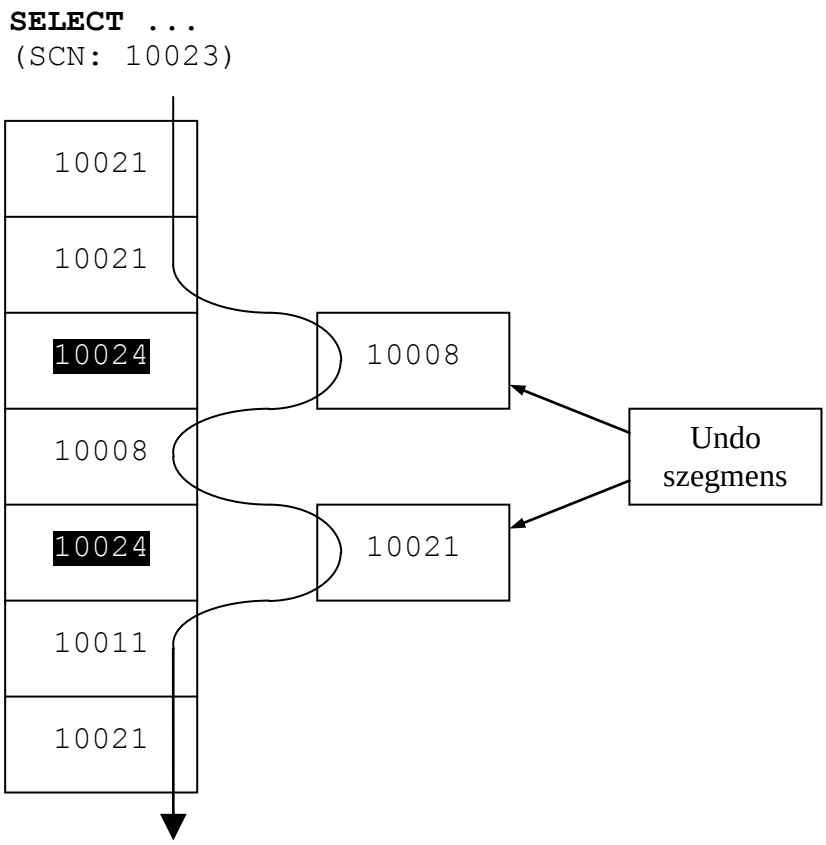
Az alábbi információk forrása az Oracle Database Concepts — Data Concurrency and Consistency.

Az Oracle a kétfázisú zárolás és a pillanatkép-elkülönítés kombinációját használja a konkurenciavezérléshez. Felhasználói szinten a zárolási egység lehet a tábla vagy annak egy sora. A zárakat az ütemező automatikusan helyezi el és oldja fel, de lehetőség van arra is, hogy a felhasználó (alkalmazás) kérjen zárat.

Az olvasási konzisztencia szintjei

Az Oracle minden lekérdezés számára biztosítja az *utasítás szintű olvasási konzisztenciát*, azaz a lekérdezés által olvasott adatok véglegesítettek, és egy időpillanatból (alapértelmezésben a lekérdezés kezdetének pillanatából) származnak. Emiatt a lekérdezés sohasem olvas piszkos (nem véglegesített) adatot, és nem látja azokat a változtatásokat sem, amelyeket a lekérdezés végrehajtása alatt véglegesített tranzakciók eszközöltek. Kérhetjük egy tranzakció összes lekérdezése számára is az olvasási konzisztencia biztosítását, ez a *tranzakció szintű olvasási konzisztencia*. Ezt úgy érhetjük el, hogy a tranzakciót sorba rendezhető vagy csak olvasás módban futtatjuk (lásd lejjebb). Ekkor a tranzakció által tartalmazott összes lekérdezés a tranzakció indításakor fennálló adatbázis-állapotot látja, kivéve a tranzakció által korábban végrehajtott módosításokat.

A kétféle olvasási konzisztencia biztosításához az adatbázisszervernek egy olvasáskonzisztens adathalmazt kell előállítania, amikor egy tábla egyszerre lekérdezés és módosítás alatt is áll. E cél eléréséhez az Oracle az undo információkat használja fel. Amikor egy felhasználó adatmódosítást hajt végre, az Oracle undo bejegyzéseket készít, amelyeket undo (vagy rollback) szegmensekbe ír. Az undo szegmensek tárolják azon adatok régi értékeit, amelyeket még nem véglegesített vagy nemrég véglegesített tranzakciók változtattak meg. Így ugyanazon adatnak több, különböző időpontokból származó változata létezhet az adatbázisban. Az adatbázisszerver az adatok különböző időpontokban létező pillanatképeit használja fel arra, hogy biztosítsa az adatok olvasáskonzisztens nézeteit és lehetővé tegye a nemblokkoló lekérdezéseket (lásd később). Amint egy lekérdezés vagy tranzakció megkezdi működését, meghatározódik a *system change number* (SCN) aktuális értéke. Az SCN a blokkokhoz mint adatbáziselemekhez tartozó időbélyegzőnek tekinthető. Ahogy a lekérdezés olvassa az adatblokkokat, az Oracle összehasonlítja azok SCN-jét (utolsó módosításának „idejét”) az aktuális lekérdezés SCN értékével, és csak az annál nem nagyobb SCN-nel rendelkező véglegesített blokkokat olvassa be a tábla területéről. A nagyobb SCN-nel rendelkező blokkok esetén az undo adatokból rekonstruálja az adott blokk azon verzióját, amelyhez a legnagyobb olyan SCN érték tartozik, amely kisebb, mint a lekérdezésé, és már véglegesített tranzakció hozta létre. Ezeket a rekonstruált adatblokkokat *konzisztens olvasási klónoknak* (consistent read clones) nevezzük. A következő ábra illusztrálja a folyamatot:



Előfordulhat, hogy az undo szegmensből már nem állítható elő a keresett blokk szükséges korábbi változata. Ha az undo információk menedzselése automatikus, akkor létezik egy aktuális *undo megtartási idő* (undo retention period), amely az a minimális időtartam, ameddig az Oracle megpróbálja megtartani a régi undo információkat, mielőtt felülírná őket. Azokat a régi (véglegesített tranzakcióhoz tartozó) undo bejegyzéseket, amelyek régebbiek az aktuális undo megtartási időnél, *lejártnak* (expired) nevezzük; ezek felülírhatók újabb tranzakciók bejegyzéseivel. Az undo megtartási időnél kisebb idejű régi bejegyzések nem lejártak, ezeket az Oracle igyekszik megtartani a konzisztens olvasások és a flashback műveletek (egy tábla valamely múltbéli állapotán végrehajtott műveletek) biztosításához.

Ha az undo táblaterület az AUTOEXTEND opcióval lett létrehozva (a DBCA által automatikusan létrehozott UNDOTBS1 például ilyen), akkor az Oracle úgy állítja be dinamikusan az undo megtartási időt, hogy az valamivel nagyobb legyen, mint a rendszer leghosszabb ideig futó aktív lekérdezésének a végrehajtási ideje. Ha a lejárt undo információk által elfoglalt tárterület fogyóban van, akkor – a nem lejárt undo információk felülírása helyett – megnöveli a táblaterület méretét. Ha a táblaterülethez megadtuk a MAXSIZE opciót, és a táblaterület mérete eléri az abban megadott méretet, akkor nem lejárt undo bejegyzések is felülíródhatnak.

Ha az undo táblaterület fix méretű, akkor az Oracle úgy állítja be dinamikusan az undo megtartási időt, hogy az a lehető legnagyobb legyen a táblaterület nagyságát és a rendszer terheltségét figyelembe véve. Ez a lehető legnagyobb megtartási idő általában lényegesen nagyobb, mint a leghosszabb ideig futó aktív lekérdezés végrehajtási ideje. Ha túl kicsire méretezzük az undo táblaterületet, akkor a hosszan futó tranzakciók abortálhatnak egy „snapshot too old” hibaüzenet kíséretében, ami azt jelenti, hogy nincs elegendő undo információ az olvasási konzisztencia biztosításához.

Ha garantálni szeretnénk a hosszan futó lekérdezések vagy a flashback műveletek sikeres végrehajtását, kérhetjük a *megtartási garanciát* (retention guarantee). Ekkor az Oracle soha nem írja felül a nem lejárt undo bejegyzéseket, még akkor sem, ha emiatt az új tranzakciók nem tudnak lefutni (mivel nincs hely az

undo bejegyzéseik tárolására). A megtartási garancia nélkül az adatbázisszerver felülírhatja a nem lejárt undo bejegyzéseket, ha kevés a tárhely, ezáltal csökkentve a megtartási időt.

A tranzakcióelkülönítési szintek

Az SQL92 ANSI/ISO szabvány a tranzakcióelkülönítés négy szintjét definiálja, amelyek abban különböznek egymástól, hogy az alábbi három jelenség közül melyeket engedélyezik:

- *piszkos olvasás:* a tranzakció olyan adatot olvas, amelyet egy másik, még nem véglegesített tranzakció írt;
- *nem ismételhető (fuzzy) olvasás:* a tranzakció újraolvas olyan adatokat, amelyeket már korábban beolvasott, és azt találja, hogy egy másik, már véglegesített tranzakció módosította vagy törölte őket;
- *fantomok olvasása:* a tranzakció újra végrehajt egy lekérdezést, amely egy adott keresési feltételnek eleget tevő sorokkal tér vissza, és azt találja, hogy egy másik, már véglegesített tranzakció további sorokat szúrt be, amelyek szintén eleget tesznek a feltételnek.

A négy tranzakcióelkülönítési szint a következő:

Elkülönítési szint	**Piszkos olvasás**	**Nem ismételhető olvasás**	**Fantomok olvasása**
nem olvasásbiztos (read uncommitted)	lehetséges	lehetséges	lehetséges
olvasásbiztos (read committed)	nem lehetséges	lehetséges	lehetséges
megismételhető olvasás (repeatable read)	nem lehetséges	nem lehetséges	lehetséges
sorba rendezhető (serializable)	nem lehetséges	nem lehetséges	nem lehetséges

Az Oracle ezek közül az olvasásbiztos és a sorba rendezhető elkülönítési szinteket ismeri, valamint egy *csak olvasás* (read-only) módot, amely nem része a szabványnak.

- *Olvasásbiztos:* Ez az alapértelmezett tranzakcióelkülönítési szint. Egy tranzakció minden lekérdezése csak a lekérdezés (és nem a tranzakció) elindítása előtt véglegesített adatokat látja. Piszkos olvasás sohasem történik. A lekérdezés két végrehajtása között azonban a lekérdezés által olvasott adatokat más tranzakciók megváltoztathatják, ezért előfordulhat nem ismételhető olvasás és fantomok olvasása is. Olyan környezetekben célszerű ezt a szintet választani, amelyekben várhatóan kevés tranzakció kerül egymással konfliktusba.
- *Sorba rendezhető:* A sorba rendezhető tranzakciók csak a tranzakció elindítása előtt véglegesített változásokat látják, valamint azokat, amelyeket maga a tranzakció hajtott végre `INSERT`, `UPDATE` és `DELETE` utasítások segítségével. A sorba rendezhető tranzakciók nem hajtanak végre nem ismételhető olvasásokat, és nem olvasnak fantomokat. Ezt a szintet olyan környezetekben célszerű használni, amelyekben nagy adatbázisok vannak, és rövidek a tranzakciók, amelyek csak kevés sort módosítanak, valamint ha kicsi az esélye annak, hogy két konkurens tranzakció ugyanazokat a sorokat módosítja, illetve ahol a hosszú (sokáig futó) tranzakciók elsősorban csak olvasási tranzakciók. Az Oracle csak akkor engedi egy sor módosítását egy sorba rendezhető tranzakciónak, ha az adott sor korábbi változtatásait olyan tranzakciók hajtották végre, amelyek még a sorba rendezhető tranzakció elindítása előtt véglegesítődtek. Amennyiben egy sorba rendezhető tranzakció megpróbál módosítani vagy törölni egy sort, amelyet egy olyan tranzakció változtatott meg, amely a sorba rendezhető tranzakció indításakor még nem véglegesítődött, az Oracle hibaüzenetet ad: „Cannot serialize access for this transaction”. **Ne feledjük, hogy a neve ellenére a sorba rendezhető elkülönítési szint valójában pillanatkép-elkülönítést használ, és nem garantálja a sorbarendezhetőséget!**

- *Csak olvasás:* A csak olvasás elkülönítési szint hasonló a sorba rendezhető elkülönítési szinthez, kivéve hogy a csak olvasó tranzakciók nem engedik meg az adatmódosítást a tranzakcióban, hacsak nem a SYS felhasználó futtatja azt. A csak olvasó tranzakciók így nem futhatnak bele a fent leírt hibába. Ez az elkülönítési szint akkor hasznos, ha olyan jelentéseket készítünk, amelyek tartalmának konzisztensnek kell lennie a tranzakció kezdetekor fennálló adatbázis-állapottal.

Az elkülönítési szintet a következő utasítások valamelyikének a tranzakció elején történő kiadásával adhatjuk meg:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET TRANSACTION READ ONLY;
```

A zárolási rendszer

Mind az olvasásbiztos, mind a sorba rendezhető tranzakciók használják a sor szintű zárolást, ezáltal egy `T` tranzakciónak várnia kell, ha olyan sort próbál írni, amelyet egy még nem véglegesített konkurens `U` tranzakció módosított. `T` megvárja, míg `U` véglegesítődik vagy abortál, és felszabadítja a zárat. Ha `U` abortál, akkor `T` végrehajthatja a korábban zárolt sor módosítását, függetlenül az elkülönítési szintjétől, mintha `U` nem is létezett volna. Ha azonban `U` véglegesítődik, akkor `T` csak akkor hajthatja végre a módosítást, ha az elkülönítési szintje az olvasásbiztos. Egy sorba rendezhető tranzakció ilyenkor „Cannot serialize access" hibaüzenetet ad, mert `U` módosításának véglegesítése `T` kezdete után történt.

A zárakat az Oracle automatikusan kezeli, amikor SQL-utasításokat hajt végre. Mindig a legkevésbé szigorú zármódot alkalmazza, így biztosítja a legmagasabb fokú konkurenciát. Lehetőség van arra is, hogy a felhasználó kérjen zárat.

Egy tranzakcióban szereplő SQL-utasításoknak adott zárak általában a tranzakció befejeződéséig fennmaradnak (kétfázisú zárolás). Az Oracle akkor szabadítja fel a zárakat, amikor a tranzakció véglegesítődik vagy abortál. Ezenkívül egy mentési pont után kapott zárak akkor is felszabadulnak, ha visszagörgetjük a tranzakciót a mentési pontig. Ilyenkor azonban csak olyan tranzakciók kaphatják meg a most felszabaduló erőforrások zárjait, amelyek nem várakoztak az eddig zárolt erőforrásokra. A várakozó tranzakciók tovább várakoznak, amíg az eredeti tranzakciót nem véglegesítjük, vagy teljesen vissza nem görgetjük.

Zártípusok

Az Oracle a zárakat a következő általános kategóriákba sorolja:

- DML-zárak (adatzárak): az adatok védelmére szolgálnak;
- DDL-zárak (szótárzárak): a sémaobjektumok (pl. táblák) szerkezetének a védelmére valók;
- belső zárak: a belső adatszerkezetek, adatfájlok védelmére szolgálnak, kezelésük teljesen automatikus.

DML-zárak két szinten léteznek: vannak sor szintű zárak (TX) és tábla szintű zárak (TM). A DML-utasítások hatására a tranzakciók mindkét szinten automatikusan kapják a zárakat. Sorok szintjén csak egyféle zármód létezik, a kizárólagos. A többváltozatú időbélyegzés és a sor szintű zárolás kombinációja azt eredményezi, hogy a tranzakciók csak akkor versengenek az adatokért, ha ugyanazokat a sorokat próbálják meg elérni. Az Oracle olvasókra és írókra vonatkozó zárolási szabályai a következők:

- Egy sor csak akkor kerül zárolásra, ha módosítja egy író.
- Egy sor írója blokkolja (késlelteti) ugyanazon sor egy konkurens íróját.
- Egy olvasó sosem blokkol egy írót, hacsak az olvasó nem a `SELECT ... FOR UPDATE` utasítást használja, amely zárolja is a beolvasott sorokat.

- Egy író sosem blokkol egy olvasót. Ha egy író módosít egy sort, az Oracle az undo adatokat használja, hogy a sor konzisztens nézetét biztosítsa az olvasóknak.

A `FOR UPDATE` nélküli lekérdezések tehát sohasem járnak zárolásokkal, így más tranzakciók is lekérdezhetik vagy akár módosíthatják a lekérdezett táblát, akár a kérdéses sorokat is. Mivel a `FOR UPDATE` nélküli lekérdezések – zárolások híján – nem blokkolhatnak más műveleteket, az Oracle gyakran hívja az ilyen lekérdezéseket *nemblokkoló lekérdezéseknek*. Másrészt a lekérdezések sohasem várnak zárfeloldásra, mindig végrehajtódhatnak.

Egy tranzakció TX zárat kap minden egyes sorra, amelyet az alábbi utasítások módosítanak: `INSERT`, `UPDATE`, `DELETE`, `MERGE` vagy `SELECT … FOR UPDATE`. Ha egy tranzakció zárat kap egy sorra, akkor a sort tartalmazó táblára is zárat kap, hogy elkerüljük az olyan DDL-utasításokat, amelyek felülírnák a tranzakció változtatásait.

Egy tranzakció TM zárat kap, ha a táblát az alábbi utasítások módosítják: `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `SELECT … FOR UPDATE` vagy `LOCK TABLE`. Táblák szintjén ötféle zármódot különböztetünk meg: *row share* (RS) vagy *subshare* (SS), *row exclusive* (RX) vagy *subexclusive* (SX), *share* (S), *share row exclusive* (SRX) vagy *share-subexclusive* (SSX) és *exclusive* (X). A következő táblázat összefoglalja, hogy az egyes utasítások milyen zármódot vonnak maguk után, és hogy milyen zármódokkal kompatibilisek:

SQL-utasítás	Zármód	RS	RX	S	SRX	X
`SELECT … FROM tábla`	-	I	I	I	I	I
`INSERT INTO tábla`	RX	I	I	N	N	N
`UPDATE tábla`	RX	I*	I*	N	N	N
`MERGE INTO tábla`	RX	I	I	N	N	N
`DELETE FROM tábla`	RX	I*	I*	N	N	N
`SELECT … FROM tábla … FOR UPDATE`	RX	I*	I*	N	N	N
`LOCK TABLE tábla IN ROW SHARE MODE`	RS	I	I	I	I	N
`LOCK TABLE tábla IN ROW EXCLUSIVE MODE`	RX	I	I	N	N	N
`LOCK TABLE tábla IN SHARE MODE`	S	I	N	I	N	N
`LOCK TABLE tábla IN SHARE ROW EXCLUSIVE MODE`	SRX	I	N	N	N	N
`LOCK TABLE tábla IN EXCLUSIVE MODE`	X	N	N	N	N	N

* Igen, ha egy másik tranzakció nem tart fenn konfliktusos sor szintű zárat, különben várakozik.

Az egyes zármódok részletesen a következők:

- Az RS zár azt jelzi, hogy a zárat fenntartó tranzakció sorokat zárolt a táblában, és később módosítani kívánja őket. Az RS a legkevésbé szigorú zármód, amely a legmagasabb fokú konkurenciát biztosítja.
- Az RX zár általában azt jelzi, hogy a zárat fenntartó tranzakció módosította a tábla egyes sorait, vagy kiadott egy `SELECT … FOR UPDATE` utasítást.
- Ha egy tranzakció S zárat birtokol egy táblán, akkor más tranzakció csak lekérdezheti a táblát (`SELECT … FOR UPDATE` használata nélkül). Módosítások csak akkor megengedettek, ha csak egyetlen tranzakciónak van S zárja a táblán. Mivel több tranzakció is fenntarthat egyidejűleg S zárat ugyanazon a táblán, ez a zár nem elegendő a tábla módosíthatóságának biztosításához.
- Az SRX zár szigorúbb az S zárnál. Egy adott táblán egy időpillanatban csak egy tranzakció tarthat fenn SRX zárat. Más tranzakciók csak lekérdezhetik a táblát (a `SELECT … FOR UPDATE` kivételével), de nem módosíthatják.
- Az X a legszigorúbb zármód, amely kizárólagos írási hozzáférést biztosít az ilyen zárat birtokló tranzakciónak. Egy adott táblán egy időpillanatban csak egy tranzakció tarthat fenn X zárat.

A módosító DML-utasítások és a `SELECT … FOR UPDATE` utasítás az érintett sorokra kizárólagos sor szintű zárakat helyeznek, így más tranzakciók nem módosíthatják vagy törölhetik a zárolt sorokat, amíg a zárakat elhelyező tranzakció nem véglegesítődik vagy abortál. A módosító utasítást tartalmazó tranzakciónak a sor szintű zárakon kívül az érintett sorokat tartalmazó táblára is szüksége van legalább egy

RX módú zárra. Ha a tartalmazó tranzakció már fenntart egy S, SRX vagy X zárat a kérdéses táblán (amelyek szigorúbbak az RX zárnál), akkor az RX zárra nincs szükség, ha pedig RS zárat tartott fenn, akkor azt az Oracle automatikusan felminősíti RX zárrá.

Ha az utasítás alkérdést vagy implicit kérdést tartalmaz, akkor a lekérdezett sorok nem kapnak sor szintű zárat. A DML-utasításokba ágyazott alkérdések és implicit kérdések garantáltan konzisztensek a lekérdezés kezdetekor fennálló adatbázis-állapottal, és nem látják a tartalmazó módosító utasítás által véghezvitt változtatásokat.

Egy tranzakcióban lévő lekérdezés látja a tranzakció korábbi módosító utasításai által végrehajtott változtatásokat, de nem látja más tranzakciók nem véglegesített módosításait.

Zárak felminősítése és kiterjesztése

Táblák szintjén az Oracle automatikusan felminősít egy zárat erősebb módúvá, amikor szükséges. Ha például egy tranzakció RS módú zárat tart fenn egy táblán, és a tranzakció egy DML-utasítása módosítani szeretné a tábla néhány sorát, az RS mód automatikusan felminősül RX módra. Mivel sorok szintjén csak egyfajta zármód létezik (kizárólagos), nincs szükség felminősítésre.

Zárak *kiterjesztésének* (lock escalation) nevezzük azt a folyamatot, amikor a szemcsézettség egy szintjén (pl. sorok szintjén) lévő zárakat az adatbázis-kezelő rendszer a szemcsézettség egy magasabb szintjére (pl. a tábla szintjére) emeli. Például ha a felhasználó sok sort zárol egy táblában, egyes rendszerek ezeket automatikusan kiterjesztik a teljes táblára. Ezáltal csökken a zárak száma, viszont nő a zárolt elemek zármódjának erőssége. Az Oracle nem alkalmazza a zárkiterjesztést, mivel az megnöveli a holtpontok kialakulásának kockázatát. Tegyük fel, hogy egy rendszer szeretné kiterjeszteni a T_1 tranzakció sor szintű zárait a teljes táblára, de nem teheti meg a T_2 tranzakció által fenntartott zárak miatt. Ha a T_2 tranzakciónak szintén szüksége van a sor szintű zárainak kiterjesztésére ugyanarra a táblára, holtpont alakul ki.