Transakcie

Ján Mazák

FMFI UK Bratislava

Transakcie

Transakcie, ktoré sa vykonajú celé, alebo vôbec nič z nich:

- rieši konzistenciu dát z hľadiska používateľov databázy;
- rieši problém so zlyhaniami (hardvér, výpadky siete či elektriny);
- môže ich paralelne prebiehať mnoho, navzájom o sebe nevedia.

Operácie v rámci transakcie:

- start (BEGIN TRANSACTION)
- read (SELECT)
- write (INSERT, UPDATE, DELETE)
- COMMIT (úspešné ukončenie) / ROLLBACK (abort, zrušenie)

Transakcie

Príklad transakcie (PostgreSQL):

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
COMMIT;
```

Transakcie

Príklad transakcie (Python):

```
command = 'INSERT INTO student VALUES (?, ?)'
cursor = conn.cursor()
cursor.execute(command, ('a', 'b'))
cursor.execute(command, ('c', 'd'))
conn.commit()
```

Príkaz pre začiatok transakcie sa odošle automaticky.
Databázové drivery zväčša umožňujú nastaviť tzv. *autocommit* mód, keď je každý vykonávaný príkaz transakciou sám osebe.

Transakcie

Za prelínanie jednotlivých transakcií zodpovedá scheduler, ktorý vytvára rozvrh obsahujúci jednotlivé operácie (tiež história). Scheduler nevidí do budúcnosti a pre každú požiadavku jednotlivých transakcií musí ihneď rozhodnúť, čo s ňou:

- vykonať hneď
- počkať a skúsiť požiadavku zaradiť neskôr
- abortnúť transakciu (napr. ak sa požiadavka nebude dať vykonať ani v budúcnosti, alebo ako prevenciu deadlocku)

Transakcie

Dôvody abortu transakcie:

- explicitné želanie klienta
- výpadok na strane klienta
- výpadok na strane servera
- strata spojenia medzi klientom a serverom
- scheduler usúdi, že nie je možné v transakcii pokračovať (napr. by došlo k neriešiteľnému konfliktu s inou tx)

Klient musí počítať s tým, že dôjde k abortu jeho transakcie, aj keď všetky vykonal správne.

Požiadavky na transakcie — ACID

- Atomicity — tx sa vykoná celá, alebo vôbec
- Consistency — tx mení stav db z konzistentného na konzistentný (za toto zodpovedá klient)
- Isolation — efekt paralelných transakcií je taký istý, ako keby boli vykonané postupne (v nejakom poradí)
- Durability — efekt tx musí po commite zotrvať aj pri výpadkoch systému

Reálne systémy niekedy čiastočne oslabia tieto garancie, aby zvýšili priepustnosť systému.

Rozvrhy

Sériový:

T_1	T_2
r(A)	
w(A)	
commit	
	r(B)
	w(C)
	commit

Nie sériový,
ale „ekvivalentný“:

T_1	T_2
r(A)	
	r(B)
w(A)	
	w(C)
	commit
commit	

Maximálna miera paralelizmu bez neželaných efektov:
view-sériovateľnosť, čiže view-ekvivalencia nejakému sériovému rozvrhu.

Rozvrhy

Dva rozvrhy R a S sú view-ekvivalentné, ak:

- pozostávajú z tých istých transakcií;
- ak T_i je prvá tx v R, ktorá číta hodnotu objektu X, tak T_i je prvá taká aj v S;
- ak operácia o_i z T_i číta hodnotu zapísanú operáciou o_j z T_j v R, tak je to tak aj v S;
- ak T_i je posledná tx, ktorá zapisuje hodnotu objektu X v R, tak T_i je posledná taká aj v S;
- tieto podmienky platia aj naopak (po výmene R a S).

Čiže rozvrhy majú rovnaký efekt na výsledný stav databázy aj na výpočet jednotlivých transakcií.

(Táto aj niektoré nasledujúce definície vynechávajú zopár technických detailov.)

Rozvrhy

V niektorých prípadoch môžu existovať rozvrhy, ktoré sú ekvivalentné sériovým, ale nie sú view-sériovateľné — vidno to však až z analýzy operácií iných ako READ a WRITE, a tie DBMS nevidí.
View-sériovateľnosť je nepraktická: je NP-ťažké rozhodnúť, či existuje view-ekvivalentný sériový rozvrh. Preto sa prax zameriava na užšiu triedu rozvrhov (menej paralelizmu), ktorá sa dobre generuje: konflikt-sériovateľné rozvrhy.

Rozvrhy

Dve operácie na tom istom objekte z rôznych transakcií sú konfliktné okrem prípadu read-read (čiže r-w, w-r, w-w).
Dva rozvrhy R a S sú konflikt-ekvivalentné, ak:

- pozostávajú z tých istých transakcií;
- poradie v rámci každej dvojice konfliktných operácií je v R a S rovnaké.

Rozvrh je konflikt-sériovateľný, ak je konflikt-ekvivalentný nejakému sériovému rozvrhu.

Rozvrhy

T_1	T_2	T_3
w(A)		
	r(B)	
	r(A)	
		r(B)
		w(B)
w(B)		

Tento rozvrh *nie je konflikt-sériovateľný*, lebo v akomkoľvek konflikt-ekvivalentom sériovom rozvrhu by T_1 musela byť pred T_2 (modrá dvojica operácií), T_2 pred T_3 (zelená) a T_3 pred T_1 (červená). (V tzv. precedenčnom grafe, ktorý možno zostaviť z hrán zodpovedajúcich konfliktným operáciám, sa vyskytuje cyklus.)

Rozvrhy

V niektorých prípadoch možno z požiadaviek na sériovateľnosť zľaviť.

- read-only transakcie
- zisťovanie databázových štatistík pre účely optimalizácie dotazov (stačí nám aproximácia výsledkov)
- distribuované databázy (zabezpečiť sériovateľnosť môže byť pridrahé)

Vo všeobecnosti si volíme kompromis medzi výkonom a mierou izolácie (a konzistencie).

Obnoviteľnosť

Aby bolo možné zabezpečiť požiadavku na trvalé uchovanie dát (durability), musia byť rozvrhy odolné voči výpadkom DBMS.
Najjednoduchším riešením je pred každou operáciou do logu (žurnálu) zapísať, že ju ideme vykonať. Jednotlivé operácie môžu byť pomerne náročné (napr. po pridaní záznamu treba upraviť index či spustiť trigger) a nedajú sa vykonávať atomicky, ale samotný zápis do logu áno. Log zároveň rieši problémy s rôznymi vyrovnávacími pamäťami (cache manažér môže pozdržať zápis bloku na disk apod.).

Obnoviteľnosť

V prípade pádu systému sa len pozrieme do logu, odstránime efekt transakcií, ktoré treba abortnúť (UNDO), a vykonáme operácie commitnutých transakcií nanovo (REDO).

Správa logu využíva prostriedky operačného systému a nemusí byť úplne spoľahlivá: príkladom je sekcia „How To Corrupt Your Database Files“ v dokumentácii SQLite (link).

Obnoviteľnosť

Príklad rozvrhu, ktorý nie je obnoviteľný:

T_1	T_2
w(A)	
	r(A)
	w(B)
	commit
commit	

Dvojica modrých operácií predstavuje dirty read: T_2 číta hodnotu, ktorá ešte nie je commitnutá, a ak by došlo k výpadku po commite T_2, ale pred commitom T_1, hodnoty vypočítané T_2 budú vychádzať z falošnej hodnoty objektu A (ktorá nikdy nebola zapísaná do db).

Obnoviteľnosť

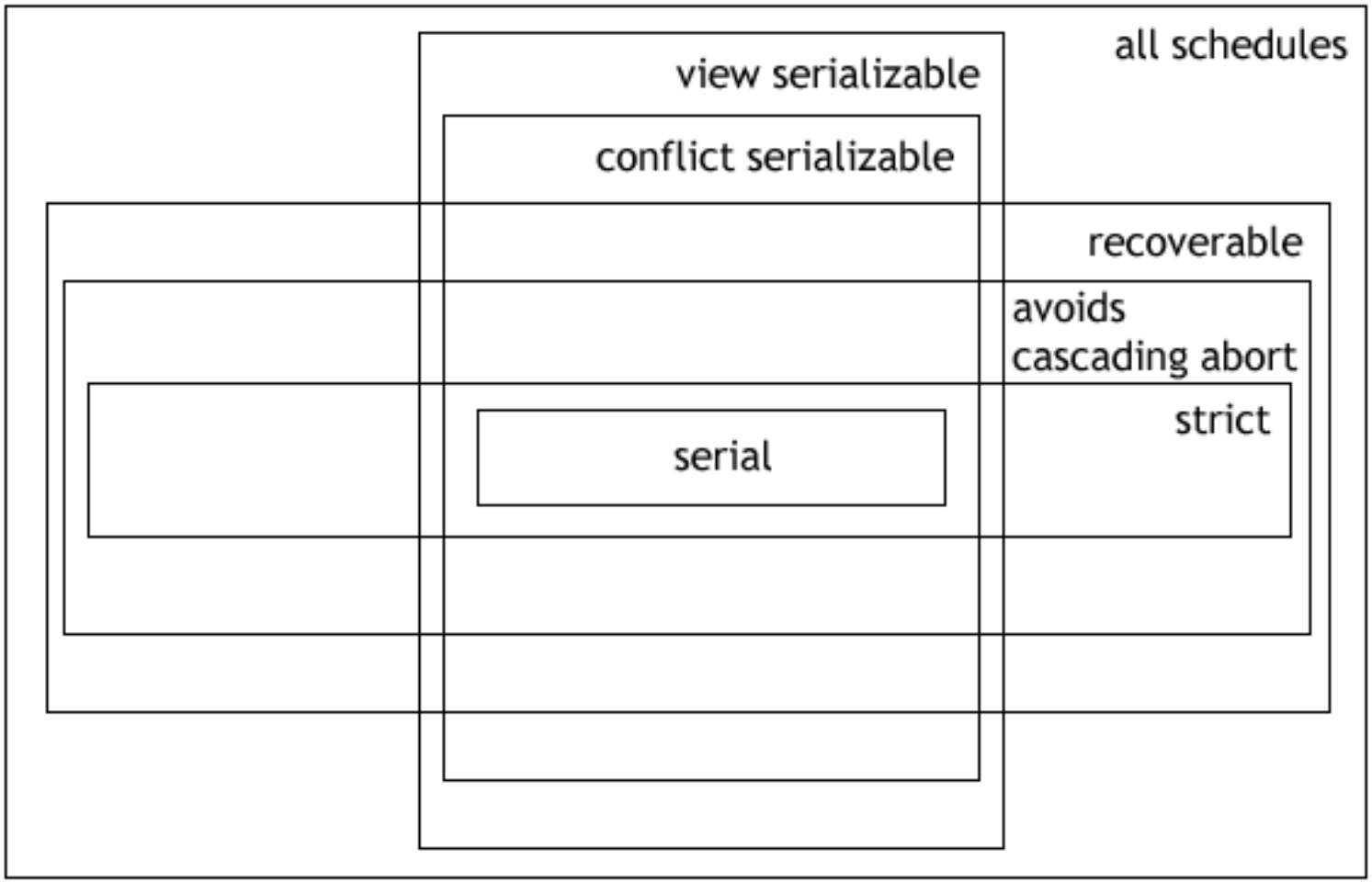
Uvedený rozvrh sa dá napraviť výmenou poradia commitov.

T_1	T_2
w(A)	
	r(A)
	w(B)
commit	
	commit

Je tu však ďalší problém: abort T_1 (miesto commitu) povedie k vynútenému abortu T_2 (*cascading abort*). Takto sa abort môže šíriť do ďalších transakcií a musíme zahodiť všetku prácu, čo vykonali.

Riešenie: generovať rozvrhy, ktoré vôbec neobsahujú *dirty read* (trieda ACA — avoids cascading aborts), príp. ani *dirty write* (striktné rozvrhy).

Rozvrhy



Všeobecne zamerané DBMS využívajú najmä konflikt-sériovateľné striktné rozvrhy.

Rozvrhy

Metódy na generovanie vyhovujúcich rozvrhov:

- dvojfázové zamykanie
- časové pečiatky
- validácia
- multi-version concurrency control (MVCC)
- ...

Zámky

Základnou technikou na generovanie konflikt-sériovateľných rozvrhov je použitie zámkov. Zámky môžu mať rôznu granularitu (záznam, celá tabuľka apod.). Typy zámkov:

- exkluzívne (write lock)
- zdieľané, len na čítanie (read lock)
- upgrade lock atď.

Pred vykonaním operácie musí transakcia dostať zámok, ktorý jej umožní túto operáciu vykonať. V minimalistickej verzii sa využijú len exkluzívne zámky, čiže prvá tx dostane prístup k objektu a ostatné musia čakať (alebo budú abortnuté). Využitie zdieľaných zámkov výrazne urýchli prácu systému, v ktorom sa veľa číta a menej zapisuje.

Dvojfázové zamykanie

Princíp dvojfázového zamykania (2PL) — v prvej fáze transakcia zámky získava, v druhej už len uvoľňuje. Pre dvojicu konfliktných operácií dochádza aj ku konfliktu zámkov (nemôžu byť pridelené súčasne pre obe operácie).

2PL generuje len konflikt-sériovateľné rozvrhy.

(Predpokladajme opak. Potom musí v precedenčnom grafe existovať cyklus, povedzme $T_1 \to T_2 \to \cdots \to T_1$. Ak T_i predchádza T_j, tak v príslušnej dvojici konfliktných operácií T_j dostane zámok až po tom, čo ho T_i uvoľnila. Potom však T_1 musí získať zámok až potom, čo ho uvoľnila, a to je spor s definíciou 2PL.)

Striktné dvojfázové zamykanie

Ak pridáme k 2PL požiadavku na uvoľnovanie zámkov len pri commite (Strict 2PL), zníži to mieru paralelizmu (niektoré rozvrhy generované 2PL už nebudú vyhovovať), ale zabezpečí generovanie striktných rozvrhov.

Po zápise hodnoty do zdieľaného objektu bude transakcia držať exkluzívny zámok na tento objekt až do svojho commitu a medzitým žiadna tx nemôže na tento objekt získať zámok, čiže žiaden dirty read (ani write) nemôže nastať.

Poznámka: Zdieľané zámky by bolo možné uvoľňovať aj skôr ako pri commite, ale v praxi to nevedie k pozorovateľnému zrýchleniu a pridáva overhead. Uvoľňovanie zámkov pri commite je veľmi ľahké implementovať v podobe online algoritmu: nepotrebujeme vidieť „budúce“ operácie tx, aby sme vedeli, kedy uvoľňovať zámky.

Dvojfázové zamykanie

Rozvrh vytvorený 2PL môže viesť k deadlocku: vzniku cyklu transakcií, z ktorých každá čaká, kým predošlá uvoľní nejaký zámok, ktorý ona potrebuje.
Deadlocky možno riešiť v momente, keď nastanú (udržiavame graf čakajúcich transakcií a detekujeme v ňom cykly), alebo ich vzniku predchádzať niektorou zo známych metód (idea je zachovať staršie transakcie, ktoré už nejakú prácu vykonali):

- wait-die
  (ak novšia tx žiada zámok od staršej, abortneme novšiu);
- wound-wait
  (ak staršia tx žiada zámok od novšej, spôsobí jej abort).

Izolácia transakcií podľa SQL štandardu

Štandard SQL vychádza z neželaných javov pri paralelnom behu transakcií.

- dirty read
- nonrepeatable read (tx prečíta 2x po sebe ten istý záznam, ale dostane rôzne výsledky)
- phantom read (tx si vyžiada 2x záznamy určené tou istou podmienkou, ale dostane rôzne výsledky)
- serialization anomaly (výsledok rozvrhu niekoľkých tx nezodpovedá žiadnemu ich sériovému rozvrhu)

Izolácia transakcií podľa SQL štandardu

Štandardné úrovne izolácie a ich implementácia v PostgreSQL:

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Default by mal byť podľa štandardu `SERIALIZABLE`, ale pre väčšinu DBMS to tak nie je. Úroveň izolácie si vie transakcia zvoliť na začiatku, napr.

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Existujú aj možnosti nad rámec štandardu, napr. `READ ONLY` transakcie.

Izolácia transakcií podľa SQL štandardu

PostgreSQL využíva MVCC (výhodou oproti zamykaniu je, že čítanie a zapisovanie sa navzájom neblokujú). Pred verziou 9.1 (cca 2011) v PostgreSQL chýbala implementácia skutočného SERIALIZABLE levelu (lebo pre MVCC to vymysleli až v 2008).

Paralelné spracovanie môže viesť k vzniku chýb, za ktoré samotná tx nemôže a nevie ich ovplyvniť, napr.

```
ERROR:  could not serialize access due to concurrent update
```

Nepovinné čítanie o nesúlade teórie a praxe izolácie transakcií:
A Critique of ANSI SQL Isolation Levels (2007) (link)

Literatúra

- https://cs186berkeley.net/notes/note11/
- https://cs186berkeley.net/notes/note12/
- https://www.db-book.com/slides-dir/PDF-dir/ch17.pdf
- https://www.db-book.com/slides-dir/PDF-dir/ch18.pdf
- https://www.db-book.com/slides-dir/PDF-dir/ch19.pdf
- https://en.wikipedia.org/wiki/Database_transaction_scheduling
- Transaction isolation in PostgreSQL
- Transaction isolation in SQLite