MVCC: dlaczego czytający nie blokuje piszącego

> Wielowersyjność sprawia, że każda transakcja widzi migawkę danych z określonego momentu, więc odczyt nigdy nie musi czekać na zapis ani zapis na odczyt — kosztem tego, że stare wersje wierszy trzeba gdzieś trzymać i w końcu posprzątać.

Skróty użyte na tej stronie (2)

Skrót	Co znaczy
MVCC	Multi-Version Concurrency Control: baza trzyma kilka wersji wiersza, żeby czytający nie blokował piszącego
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Po co to istnieje

> **Pojęcie: MVCC (Multi-Version Concurrency Control, wielowersyjność).** Mechanizm, w którym silnik nie zmienia wiersza „w miejscu" przy każdym `UPDATE`, tylko tworzy jego nową wersję obok starej, oznaczając obie informacją o tym, która transakcja i kiedy je stworzyła. Dzięki temu każda transakcja, w momencie swojego rozpoczęcia, dostaje spójną migawkę bazy danych — widzi dokładnie te wersje wierszy, które były zatwierdzone w chwili, gdy ta migawka powstała, niezależnie od tego, co inne transakcje robią równolegle. Odczyt nie musi czekać na blokadę trzymaną przez piszącego, bo nie sięga po tę samą, aktualnie modyfikowaną wersję wiersza — sięga po wersję sprzed rozpoczęcia zapisu, która wciąż istnieje osobno. To jest cała odpowiedź na pytanie z tytułu tej strony: czytający i piszący pracują na dwóch różnych wersjach tego samego logicznego wiersza, więc nie ma czego blokować między nimi.



Transakcja A czyta starą wersję wiersza z własnej migawki, podczas gdy transakcja B tworzy nową wersję przez UPDATE (kliknięcie otwiera pełny rozmiar)

Bez MVCC, w modelu opartym wyłącznie na blokadach opisanych na stronie Blokady: współdzielone, wyłączne, eskalacja, każdy odczyt musiałby czekać, aż transakcja modyfikująca dany wiersz zwolni swoją blokadę wyłączną — nawet gdy odczyt nie potrzebuje najświeższej wersji, tylko dowolnej spójnej. MVCC eliminuje ten konflikt kosztem innego: silnik musi przechowywać wiele wersji tego samego wiersza naraz i w pewnym momencie te już niepotrzebne wersje posprzątać.

Jak to działa

Rozwiń: jak to działa pod spodem

PostgreSQL: xmin/xmax, martwe krotki, VACUUM. Każdy wiersz w PostgreSQL ma dwie ukryte kolumny systemowe, `xmin` i `xmax` — numer transakcji, która ten wiersz stworzyła, i numer transakcji, która go usunęła albo zastąpiła nowszą wersją (`0`, jeśli wiersz wciąż jest aktualny). `UPDATE` w PostgreSQL nigdy nie nadpisuje istniejącego wiersza — zapisuje nową krotkę z własnym `xmin`, a starą oznacza swoim numerem transakcji jako `xmax`, zamiast fizycznie ją usuwać. Stara wersja, nazywana martwą krotką, zostaje na dysku, dopóki żadna aktywna transakcja jej nie potrzebuje do zbudowania swojej migawki — dopiero wtedy proces `VACUUM` może ją fizycznie usunąć i odzyskać zajmowane przez nią miejsce.

```
SELECT xmin, xmax, id, status FROM zamowienia WHERE id = 104;
```

Przykład:

```
 xmin | xmax |  id |    status
------+------+-----+---------------
 7213 |    0 | 104 | w realizacji
(1 row)
```

`VACUUM` jest potrzebny nie z dobroci serca, tylko dlatego, że bez niego martwe krotki zajmują coraz więcej miejsca na dysku — zjawisko nazywane bloatem: tabela fizycznie rośnie znacznie ponad rozmiar swoich aktualnie żywych danych, bo stare wersje wierszy nigdy nie są odzyskiwane automatycznie w tle, tylko przy okazji `VACUUM`a (uruchamianego zwykle automatycznie przez `autovacuum`, ale możliwego też ręcznie). Drugi, poważniejszy powód to wraparound — numer transakcji `xmin`/`xmax` w PostgreSQL jest licznikiem o skończonej długości (32 bity), który po wyczerpaniu zawija się do zera; gdyby doszło do tego bez wcześniejszego „zamrożenia" starych krotek przez `VACUUM`, wiersze zapisane dawno temu mogłyby nagle wyglądać, jakby zostały utworzone w przyszłości względem bieżących transakcji, czyniąc je niewidocznymi. PostgreSQL chroni się przed tym, wymuszając agresywny `VACUUM` na długo przed realnym zbliżeniem się licznika do granicy, ale zaniedbana baza, w której `autovacuum` był wyłączony albo nie nadążał, może w skrajnym przypadku zostać przez silnik przełączona w tryb tylko do odczytu, żeby zapobiec faktycznej utracie danych.

Oracle i MySQL/InnoDB: segmenty UNDO. Oba silniki realizują MVCC inaczej niż PostgreSQL — zamiast trzymać starą wersję wiersza obok nowej w tej samej tabeli, zapisują informację, jak cofnąć zmianę, w osobnej strukturze zwanej segmentem UNDO (Oracle) albo UNDO log (InnoDB). Transakcja, która potrzebuje starszej wersji wiersza do zbudowania swojej migawki, odtwarza ją, cofając bieżącą wersję o kolejne zapisane operacje UNDO, zamiast czytać osobno przechowywaną kopię. To działa świetnie, dopóki miejsce na dane UNDO starcza — gdy transakcja trwa bardzo długo, a segment UNDO jest zbyt mały albo skonfigurowany na zbyt krótkie przechowywanie historii, silnik może nadpisać dane UNDO potrzebne tej starej transakcji, zanim ta zdąży ich użyć. Oracle zgłasza to wprost błędem:

```
ORA-01555: snapshot too old (rollback segment too small)
```

MySQL/InnoDB nie zwraca identycznego komunikatu, ale cierpi na analogiczny problem — bardzo długo trwająca transakcja z otwartą migawką uniemożliwia InnoDB odzyskanie starych wpisów UNDO, prowadząc do wzrostu segmentu UNDO (`ibdata` albo osobnych plików `undo tablespace`) proporcjonalnego do liczby zmian wykonanych przez inne transakcje w tym czasie — zjawisko analogiczne do bloatu w PostgreSQL, tylko po stronie UNDO zamiast głównej tabeli.

SQL Server: wersjonowanie w tempdb przy RCSI. SQL Server domyślnie nie używa MVCC — domyślny `READ COMMITTED` opiera się na blokadach współdzielonych, tak jak opisuje to strona Blokady: współdzielone, wyłączne, eskalacja. Dopiero włączenie `READ_COMMITTED_SNAPSHOT` (RCSI) albo pełnej izolacji `SNAPSHOT`, opisanej na stronie Poziomy izolacji i co który naprawdę gwarantuje, przełącza silnik na wersjonowanie wierszy — a wtedy stare wersje trafiają nie do tej samej bazy danych, tylko do specjalnego obszaru bazy systemowej `tempdb`, zwanego version store. Konsekwencja praktyczna jest bezpośrednia: włączenie RCSI zwiększa obciążenie `tempdb`, więc na serwerach, gdzie `tempdb` był traktowany jako mało istotny (mała alokacja, wolny dysk), włączenie tego trybu potrafi ujawnić zupełnie nowe wąskie gardło wydajnościowe, niezwiązane z samymi tabelami użytkownika.

Cena MVCC. We wszystkich trzech podejściach cena jest tej samej natury: miejsce na dysku (albo w `tempdb`) zajęte przez wersje wierszy, które żadna aktywna transakcja już nie potrzebuje, oraz konieczność jakiegoś mechanizmu porządkującego — `VACUUM` w PostgreSQL, automatyczne zwalnianie segmentów UNDO w Oracle i InnoDB, czyszczenie `version store` w `tempdb` w SQL Server. Zaniedbanie tego sprzątania w każdym z trzech przypadków kończy się tym samym objawem z innej strony: bazą rosnącą szybciej, niż rosną realne dane, i coraz większym ryzykiem, że długo trwająca transakcja natrafi na dane, których nie da się już odtworzyć.

Silnik	Gdzie trzyma stare wersje	Mechanizm sprzątający	Typowy objaw zaniedbania
PostgreSQL	martwe krotki w tej samej tabeli	`VACUUM` / `autovacuum`	bloat, ryzyko wraparound
Oracle	segmenty UNDO (rollback segments)	automatyczne zarządzanie UNDO (`UNDO_RETENTION`)	`ORA-01555: snapshot too old`
MySQL / MariaDB (InnoDB)	UNDO log	wątek `purge` InnoDB	rosnący rozmiar `undo tablespace`
SQL Server (przy RCSI/SNAPSHOT)	`version store` w `tempdb`	automatyczne czyszczenie `tempdb`	wzrost `tempdb`, spadek wydajności

Częste błędy

Długo trwająca transakcja blokująca sprzątanie, mimo że sama nie trzyma żadnej blokady wyłącznej. Transakcja otwarta poleceniem `BEGIN` i pozostawiona bezczynnie z ustawionym poziomem izolacji dającym spójną migawkę (nawet zwykły `REPEATABLE READ`) uniemożliwia `VACUUM`owi usunięcie martwych krotek nowszych niż moment jej rozpoczęcia — bo formalnie ta transakcja wciąż może ich potrzebować, choćby nigdy więcej nie wykonała żadnego zapytania. Efekt jest identyczny jak przy zapomnianej otwartej transakcji ze strony BEGIN, COMMIT, ROLLBACK, SAVEPOINT, tylko widoczny nie jako blokada, a jako rosnący bloat tabeli, którego przyczyna bywa trudniejsza do znalezienia.

Zakładanie, że MVCC oznacza brak blokad w ogóle. MVCC eliminuje konflikt między czytającym a piszącym, ale nie eliminuje konfliktu między dwoma piszącymi — dwie transakcje próbujące zmienić ten sam wiersz wciąż czekają na siebie nawzajem, dokładnie tak jak opisuje strona Blokady: współdzielone, wyłączne, eskalacja. MVCC rozwiązuje jedną konkretną kolizję (odczyt kontra zapis), nie wszystkie możliwe kolizje współbieżności naraz.

Interpretowanie `ORA-01555` jako błędu uszkodzonych danych. Komunikat „snapshot too old" nie oznacza, że dane w bazie są niepoprawne — oznacza, że konkretna, długo trwająca transakcja nie zdążyła odczytać potrzebnej jej starszej wersji wiersza, zanim segment UNDO ją nadpisał. Naprawą jest zwykle skrócenie czasu trwania długich zapytań albo zwiększenie retencji UNDO (

`UNDO_RETENTION`), nie naprawa danych, które nigdy nie były uszkodzone.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — migawka a bieżący stan

Sesja A w PostgreSQL otwiera transakcję na poziomie `REPEATABLE READ` i wykonuje `SELECT status FROM zamowienia WHERE id = 104;`, dostając `w realizacji`. Sesja B, w osobnej, autocommitowej transakcji, zaraz potem wykonuje `UPDATE zamowienia SET status = 'zrealizowane' WHERE id = 104;` i się zatwierdza. Sesja A, wciąż w tej samej transakcji, pyta ponownie o status tego samego wiersza. Co zobaczy i dlaczego to nie jest brudny odczyt ani jego brak wynikający z blokady?

Odpowiedź

Sesja A wciąż zobaczy `w realizacji` — na poziomie `REPEATABLE READ` PostgreSQL buduje migawkę raz, na początku transakcji, i trzyma się jej do końca, niezależnie od tego, ile razy sesja A odczyta ten sam wiersz. To nie jest efekt żadnej blokady — sesja B swobodnie zmieniła i zatwierdziła swoją wersję wiersza, nie czekając na nic. Sesja A po prostu nie widzi tej nowej wersji, bo MVCC pokazuje jej dokładnie tę migawkę danych, jaka istniała w momencie rozpoczęcia jej transakcji — nowa wersja `xmin` z UPDATE'u sesji B jest dla niej niewidoczna, dopóki sama nie zakończy swojej transakcji i nie zacznie nowej.

Pytania kontrolne

1. Dlaczego PostgreSQL potrzebuje `VACUUM`a, a Oracle nie ma dokładnego odpowiednika tego samego procesu?

Odpowiedź

Bo PostgreSQL trzyma stare wersje wierszy fizycznie w tej samej tabeli co nowe — `VACUUM` musi je aktywnie znaleźć i usunąć, żeby odzyskać miejsce. Oracle trzyma historię zmian osobno, w segmentach UNDO zorganizowanych jako log operacji cofających, a nie jako pełne kopie starych wierszy w tabeli głównej — zarządzanie ich zwalnianiem odbywa się automatycznie w ramach mechanizmu UNDO, bez osobnego, widocznego dla administratora polecenia analogicznego do `VACUUM`.

2. Baza PostgreSQL ma wyłączony `autovacuum` od kilku miesięcy na dużej, często aktualizowanej tabeli. Jaki będzie tego najbardziej prawdopodobny, obserwowalny skutek?

Odpowiedź

Tabela zajmie na dysku znacznie więcej miejsca, niż wynikałoby z liczby jej aktualnie żywych wierszy — martwe krotki po każdym `UPDATE` i `DELETE` gromadzą się bez sprzątania, co nazywa się bloatem. Zapytania na tej tabeli, w tym pełne skany, zaczną czytać więcej danych z dysku niż potrzeba, bo silnik musi przechodzić też przez martwe krotki, żeby dotrzeć do żywych — objawiające się to będzie stopniowym spadkiem wydajności bez żadnej zmiany w liczbie rzeczywistych, aktualnych danych.

Powiązane: Anomalie współbieżności: brudny odczyt, niepowtarzalny odczyt, fantom · Poziomy izolacji i co który naprawdę gwarantuje · Blokady: współdzielone, wyłączne, eskalacja · Log transakcyjny: WAL, redo, transaction log

Źródła: dokumentacja PostgreSQL, rozdziały „Concurrency Control" i „Routine Vacuuming" · dokumentacja Oracle Database Concepts, rozdział „Data Concurrency and Consistency", sekcja „Undo Data" · dokumentacja MySQL 8.0, rozdział „InnoDB Multi-Versioning" · dokumentacja Microsoft SQL Server, „Row Versioning-Based Isolation Levels"

Revision #47
Created 2026-09-24 22:42:32 UTC by Tomasz Konieczny
Updated 2026-10-26 14:11:35 UTC by Tomasz Konieczny