Zakleszczenie (deadlock): jak powstaje i jak je czytać z logu

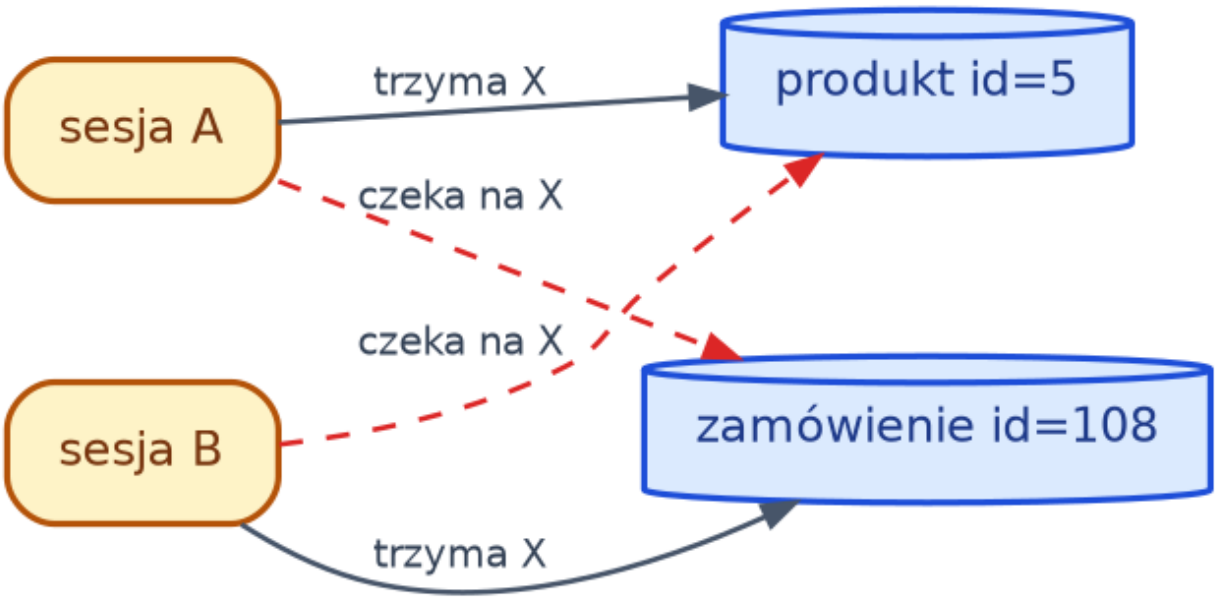
> Dwie transakcje, każda czekająca na blokadę, którą trzyma ta druga — żadna nie ustąpi sama, więc silnik musi wybrać jedną z nich na ofiarę i przerwać ją, żeby druga mogła kontynuować.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
SQL	Structured Query Language: język zapytań do relacyjnych baz danych
XML	eXtensible Markup Language: format tekstowy do zapisu ustrukturyzowanych danych w znacznikach, tutaj używany przez PowerShell do zapisu zrzutu obiektów poleceniem Export-Clixml

Po co to istnieje

> **Pojęcie: zakleszczenie.** Sytuacja, w której dwie (albo więcej) transakcje wzajemnie czekają na blokady trzymane przez siebie nawzajem, tworząc cykl, z którego żadna z nich nie może się wydostać bez interwencji z zewnątrz. Zakleszczenie różni się od zwykłego oczekiwania na blokadę opisanego na stronie Blokady: współdzielone, wyłączne, eskalacja tym, że zwykłe oczekiwanie się kończy — transakcja blokująca prędzej czy później zawoła `COMMIT` albo `ROLLBACK` i zwolni blokadę. W zakleszczeniu żadna z transakcji nigdy sama z siebie nie dojdzie do tego punktu, bo każda czeka właśnie na to, czego potrzebuje ta druga.



Sesja A trzyma produkt i czeka na zamówienie, sesja B trzyma zamówienie i czeka na produkt: cykl zamknięty (kliknięcie otwiera pełny rozmiar)

Minimalny scenariusz zakleszczenia potrzebuje dwóch sesji i dwóch obiektów, do których obie sesje sięgają w odwrotnej kolejności:

```
Czas   | Sesja A                                                | Sesja B
-------|--------------------------------------------------------
|-------------------------------------------------------
t1     | BEGIN;                                                  |
t2     |                                                          | BEGIN;
t3     | UPDATE produkty SET ilosc = ilosc - 1 WHERE id = 5;     |
       | -- A trzyma blokadę wyłączną na produkcie 5             |
t4     |                                                          | UPDATE zamowienia SET
status = 'anulowane' WHERE id = 108;
       |                                                          | -- B trzyma blokadę
wyłączną na zamówieniu 108
t5     | UPDATE zamowienia SET status = 'wstrzymane' WHERE id = 108;
       | -- A czeka na blokadę B, trzymaną na zamówieniu 108     |
t6     |                                                          | UPDATE produkty SET ilosc
= ilosc + 1 WHERE id = 5;
       |                                                          | -- B czeka na blokadę A,
trzymaną na produkcie 5
t7     | -- A czeka na B, B czeka na A — cykl zamknięty,         |
```

```
        | -- żadna nie może kontynuować                                  |
```

Sesja A trzyma blokadę na produkcie 5 i czeka na zamówienie 108, sesja B trzyma blokadę na zamówieniu 108 i czeka na produkt 5 — obie czekają, obie trzymają dokładnie to, na co czeka ta druga, i żadna z nich sama z siebie nigdy nie ustąpi.

Jak silnik wykrywa zakleszczenie i kogo wybiera na ofiarę. Każdy z czterech silników z tej książki utrzymuje wewnętrzny graf oczekiwania — kto na kogo czeka — i cyklicznie (PostgreSQL co ustawiony `deadlock_timeout`, domyślnie raz na sekundę; pozostałe silniki analogicznie, z własnymi wewnętrznymi interwałami) sprawdza, czy w tym grafie powstał cykl. Gdy cykl zostanie znaleziony, silnik musi przerwać dokładnie jedną z transakcji uczestniczących w cyklu, żeby zwolnić blokadę i pozwolić reszcie kontynuować. Wybór ofiary nie jest przypadkowy w tym sensie „rzut monetą" — większość silników bierze pod uwagę koszt już wykonanej pracy (transakcja, która zrobiła mniej, jest tańsza do wycofania) i czas trwania transakcji, ale ostateczny wybór pozostaje w praktyce nieprzewidywalny z perspektywy autora aplikacji: nie ma gwarancji, że ofiarą zawsze padnie „mniej ważna" transakcja w sensie biznesowym.

Jak to działa

Rozwiń: jak to działa pod spodem

Komunikat z PostgreSQL. Transakcja wybrana na ofiarę dostaje błąd zamiast zawiśnięcia w nieskończoność:

Przykład:

```
ERROR:  deadlock detected
DETAIL:  Process 8842 waits for ShareLock on transaction 7213811; blocked by process 8791.
Process 8791 waits for ShareLock on transaction 7213809; blocked by process 8842.
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,3) in relation "zamowienia"
```

Sekcja `DETAIL` pokazuje dokładnie ten cykl — proces 8842 czeka na proces 8791, a proces 8791 czeka z powrotem na proces 8842. PostgreSQL zapisuje pełną treść obu konfliktujących zapytań do logu serwera (nie do komunikatu widocznego dla klienta), więc pełną diagnozę — które dokładnie polecenia SQL się zderzyły — trzeba szukać w pliku logu, zwykle pod ścieżką skonfigurowaną parametrem `log_directory`, filtrując po słowie `deadlock detected`.

Deadlock graph z MSSQL. SQL Server, gdy ma włączoną flagę śledzenia `1222` albo zbiera zdarzenia przez Extended Events, zapisuje pełny graf zakleszczenia w formacie XML do

dziennika błędów — skrócona wersja tego, co się w nim znajduje, wygląda tak:

Przykład:

```
deadlock-list
 deadlock victim=process1
  process-list
   process id=process1 waitresource=KEY: 5:72057594043432960 (produkty)
    executionStack: UPDATE zamowienia SET status = 'wstrzymane' WHERE id = 108;
   process id=process2 waitresource=KEY: 5:72057594043498496 (zamowienia)
    executionStack: UPDATE produkty SET ilosc = ilosc + 1 WHERE id = 5;
```

`deadlock victim` wskazuje, który proces SQL Server zdecydował się przerwać — `process1` w tym zrzucie. Pełny graf, dostępny w Extended Events pod sesją `system_health` (włączoną domyślnie od SQL Server 2008) albo w dzienniku błędów przy włączonej fladze `1222`, pokazuje też, jakie zasoby (`waitresource`) i w jakim trybie każdy proces trzymał i o co się ubiegał.

Gdzie szukać w logu każdego silnika:

Silnik	Gdzie szukać deadlocków
PostgreSQL	log serwera (`log_directory`), fraza `deadlock detected`
SQL Server	sesja `system_health` w Extended Events (domyślnie włączona) albo dziennik błędów przy fladze `1222`
Oracle	`alert log` instancji oraz plik `.trc` w `background_dump_dest`, komunikat `ORA-00060: deadlock detected while waiting for resource`
MySQL / MariaDB (InnoDB)	`SHOW ENGINE INNODB STATUS;`, sekcja `LATEST DETECTED DEADLOCK`

Przykład:

```
mysql> SHOW ENGINE INNODB STATUS\G
------------------------
LATEST DETECTED DEADLOCK
------------------------
*** (1) TRANSACTION:
UPDATE produkty SET ilosc = ilosc - 1 WHERE id = 5
*** (2) TRANSACTION:
UPDATE zamowienia SET status = 'anulowane' WHERE id = 108
*** WE ROLL BACK TRANSACTION (2)
```

InnoDB, w odróżnieniu od pozostałych trzech silników, trzyma tylko *ostatnie* wykryte zakleszczenie w pamięci — trzeba je odczytać zaraz po wystąpieniu, zanim kolejne je nadpisze, chyba że włączona jest globalna zmienna `innodb_print_all_deadlocks`, która kieruje każdy zrzut do standardowego logu błędów MySQL zamiast tylko do `SHOW ENGINE INNODB STATUS`.

Czas do wykrycia. Wykrywanie cyklu nie jest natychmiastowe — silnik musi najpierw poczekać na czas dłuższy niż zwykłe, krótkotrwałe oczekiwanie na blokadę, zanim w ogóle zacznie sprawdzać, czy to oczekiwanie jest cyklem, czy tylko chwilowym zajęciem. PostgreSQL kontroluje ten czas parametrem `deadlock_timeout`, domyślnie ustawionym na jedną sekundę:

Przykład:

```
produkcja=# SHOW deadlock_timeout;
 deadlock_timeout
------------------
 1s
(1 row)
```

Przez tę jedną sekundę transakcja uczestnicząca w przyszłym zakleszczeniu wygląda dokładnie tak samo jak transakcja, która po prostu czeka na zwolnienie zwykłej, niekonfliktowej blokady — dopiero po przekroczeniu tego czasu PostgreSQL sprawdza graf oczekiwania i, jeśli faktycznie znajdzie cykl, przerywa jedną z transakcji. Zbyt niska wartość tego parametru obciąża silnik częstym sprawdzaniem grafu bez potrzeby; zbyt wysoka wydłuża czas, przez jaki aplikacja czeka na komunikat błędu zamiast dostać go od razu.

Częste błędy

Sięganie do obiektów w różnej kolejności w różnych częściach aplikacji. Fragment kodu, który zawsze aktualizuje najpierw `produkty`, a potem `zamowienia`, i drugi fragment, który robi to w odwrotnej kolejności, tworzą dokładnie warunki na zakleszczenie ze scenariusza wyżej — żaden z osobna nie jest błędny, problem powstaje dopiero, gdy oba działają współbieżnie. Naprawą jest ustalenie jednej, obowiązującej w całej aplikacji kolejności dostępu do tabel (na przykład zawsze alfabetycznie po nazwie tabeli) i trzymanie się jej we wszystkich miejscach kodu, które modyfikują więcej niż jedną tabelę w jednej transakcji.

Długie transakcje trzymające blokady przez cały czas swojego działania. Transakcja, która otwiera się, wykonuje pierwszy `UPDATE`, potem czeka na odpowiedź zewnętrznego serwisu przez API, i dopiero wtedy wykonuje drugi `UPDATE` przed `COMMIT`em, trzyma pierwszą blokadę przez cały czas oczekiwania na sieć — wielokrotnie dłużej, niż gdyby oba `UPDATE`y wykonały się jeden po drugim bez przerwy. Im dłużej transakcja trzyma blokady, tym większe okno czasowe, w którym inna transakcja może utworzyć z nią cykl oczekiwania. Skrócenie transakcji do wyłącznie operacji

bazodanowych, z wywołaniami sieciowymi wykonanymi przed `BEGIN` albo po `COMMIT`, ogranicza to okno do minimum.

Skan sekwencyjny bez indeksu blokujący więcej wierszy niż potrzeba. `UPDATE zamowienia SET status = 'anulowane' WHERE klient_id = 2;` na tabeli bez indeksu na `klient_id` zmusza silnik do przejrzenia każdego wiersza tabeli, i w niektórych silnikach (zwłaszcza przy wyższych poziomach izolacji) zakłada blokady albo blokady przerw na więcej wierszy niż te faktycznie pasujące do warunku — zwiększając szansę na konflikt z inną transakcją, która nigdy by nie kolidowała, gdyby silnik trafił od razu do właściwych wierszy przez indeks. Strona Czym jest indeks — i dlaczego to nie jest „przyspieszacz tabeli" rozwija, dlaczego brak właściwego indeksu kosztuje więcej niż tylko czas samego zapytania.

Brak ponawiania transakcji po stronie aplikacji. Zakleszczenie jest normalnym, oczekiwanym zjawiskiem w systemie z wysoką współbieżnością zapisu, nie błędem, który sygnalizuje uszkodzenie danych — transakcja wybrana na ofiarę traci swoją pracę i musi zacząć od nowa, ale sam fakt jej przerwania nie oznacza niczego złego z danymi. Aplikacja, która przy błędzie zakleszczenia (`ERROR: deadlock detected` w PostgreSQL, `1213 (40001): Deadlock found` w MySQL, `ORA-00060` w Oracle, błąd `1205` w SQL Server) po prostu pokazuje użytkownikowi komunikat błędu zamiast automatycznie ponowić całą transakcję od `BEGIN`, marnuje wbudowaną w silnik obsługę tego zjawiska — ponawianie transakcji po otrzymaniu kodu błędu zakleszczenia to standardowy, oczekiwany wzorzec, nie obejście awarii.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — naprawa kolejności

Dwie procedury w tej samej aplikacji obie aktualizują `produkty` i `zamowienia` w jednej transakcji. Procedura „rezerwacja towaru" robi to w kolejności `produkty` → `zamowienia`, procedura „anulowanie zamówienia" robi to w kolejności `zamowienia` → `produkty`. Obie są logicznie poprawne z osobna. Jak zmienić kod, żeby zlikwidować ryzyko zakleszczenia między nimi, nie zmieniając niczego w logice biznesowej każdej z nich?

Odpowiedź

Ustalamy jedną obowiązującą kolejność dostępu do tabel dla całej aplikacji — na przykład zawsze najpierw `produkty`, potem `zamowienia`, niezależnie od tego, która procedura akurat działa — i przepisujemy procedurę „anulowanie zamówienia" tak, żeby też sięgała po `produkty`

przed `zamowienia`, mimo że z punktu widzenia jej własnej logiki kolejność wydawała się dowolna. Skoro obie procedury zawsze biorą blokady w tej samej kolejności, żadna para transakcji nie może już utworzyć cyklu wzajemnego oczekiwania — jedna zawsze zdąży zablokować oba obiekty, zanim druga zdąży zablokować pierwszy z nich.

Pytania kontrolne

1. Dlaczego zakleszczenie nigdy nie rozwiąże się samo, w odróżnieniu od zwykłego oczekiwania na zwolnienie blokady?

Odpowiedź

Bo w zwykłym oczekiwaniu transakcja blokująca prędzej czy później kończy swoją pracę niezależnie od tego, co robi transakcja czekająca — jej `COMMIT` albo `ROLLBACK` nie zależy od niczego, na co czeka druga strona. W zakleszczeniu obie transakcje czekają na siebie nawzajem, więc żadna z nich nigdy nie dojdzie do własnego `COMMIT`/`ROLLBACK` bez zewnętrznej interwencji — potrzebny jest mechanizm wykrywania cyklu i przymusowe przerwanie jednej ze stron, bo cykl sam z siebie nigdy się nie otworzy.

2. Aplikacja łapie błąd zakleszczenia i loguje go jako błąd krytyczny wymagający natychmiastowej interwencji administratora. Czy to właściwa reakcja?

Odpowiedź

Zwykle nie — pojedyncze, sporadyczne zakleszczenie w systemie z współbieżnym zapisem jest zjawiskiem oczekiwanym, a właściwą reakcją aplikacji jest automatyczne ponowienie przerwanej transakcji, nie alarmowanie administratora. Interwencja ma sens dopiero wtedy, gdy zakleszczenia występują z rosnącą częstotliwością albo systematycznie dotyczą tych samych dwóch fragmentów kodu — to sygnał, że warto sprawdzić kolejność dostępu do obiektów albo długość transakcji, nie że coś jest nagle zepsute.

Powiązane: Blokady: współdzielone, wyłączne, eskalacja · BEGIN, COMMIT, ROLLBACK, SAVEPOINT · Czym jest indeks — i dlaczego to nie jest „przyspieszacz tabeli" · Anomalie współbieżności: brudny odczyt, niepowtarzalny odczyt, fantom

Źródła: dokumentacja PostgreSQL, rozdział „Explicit Locking", sekcja „Deadlocks" · dokumentacja Microsoft SQL Server, „Deadlocking" i „system_health Extended Events Session" · dokumentacja Oracle Database, komunikat błędu ORA-00060 · dokumentacja MySQL 8.0, rozdział „InnoDB Deadlocks"

Revision #47
Created 2026-09-24 22:42:31 UTC by Tomasz Konieczny
Updated 2026-10-26 14:11:33 UTC by Tomasz Konieczny