O'REILLY®

Wydajna praca z MySQL

Efektywne i bezpieczne zarządzanie bazami danych

Helion

Daniel Nichter

Tytuł oryginału: Efficient MySQL Performance: Best Practices and Techniques

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-9290-8

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/wydpra
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 8.
Transakcje

MySQL ma nietransakcyjne silniki, takie jak MyISAM, ale domyślnie używanym jest transakcyjny InnoDB. Dlatego każde zapytanie, nawet zwykłe `SELECT`, jest domyślnie wykonywane w ramach transakcji.

Materiał zamieszczony w tym rozdziale może Cię nie zainteresować, jeśli używasz innego silnika bazy danych, takiego jak Aria lub MyRocks. Prawdopodobnie jednak korzystasz z InnoDB i wówczas warto powtórzyć, że każde zapytanie MySQL jest transakcją.

Z perspektywy inżyniera transakcja jawi się w sposób koncepcyjny: `BEGIN`, wykonanie zapytań i `COMMIT`. Ufamy serwerowi MySQL (i silnikowi InnoDB), że prawidłowo zapewni obsługę właściwości ACID: niepodzielność, spójność, izolację i trwałość. Gdy obciążenie aplikacji — zapytania, indeksy, dane i wzorce dostępu — jest doskonale zoptymalizowane, transakcje nie powodują problemów związanych z wydajnością działania. (Większość tematów dotyczących baz danych jest nieproblematycznych, gdy obciążenie zostało dobrze zoptymalizowane). Jednak w tle transakcje oznaczają wywołanie całego nowego świata rozważań, ponieważ spełnienie właściwości ACID przy jednoczesnym zachowaniu wydajności działania nie jest łatwym zadaniem. Na szczęście MySQL wyróżnia się obsługą transakcji.

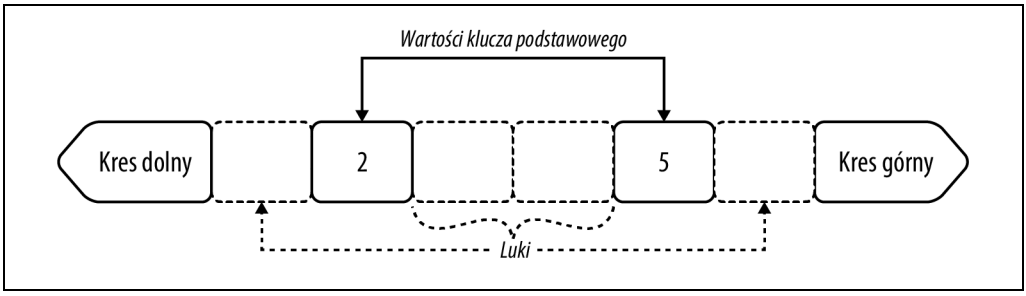
Podobnie jak w przypadku omówionego w poprzednim rozdziale opóźnienia replikacji, przedstawienie wewnętrznego mechanizmu działania transakcji wykracza poza zakres tematyczny tej książki. Mimo to zrozumienie kilku podstawowych koncepcji ma duże znaczenie i pomaga uniknąć najczęściej pojawiające się problemy, które przenoszą transakcje z najniższych poziomów MySQL na szczyty umysłów inżynierów. Nawet pobieżne zrozumienie zagadnień pomaga zapobiec wielu problemom.

W tym rozdziale przeanalizuję transakcje MySQL pod kątem unikania najczęściej występujących problemów. Materiał podzieliłem na pięć podrozdziałów. W pierwszym zajmę się nakładaniem blokad w kontekście poziomów izolacji transakcji. W drugim wyjaśnię, jak silnik InnoDB zarządza współbieżnym dostępem do danych i jednocześnie gwarantuje zachowanie właściwości ACID (technika MVCC i dzienniki przywracania). W trzecim przedstawię długość listy historii i wyjaśnię, jak wskazuje ona problematyczne transakcje. Czwarty podrozdział wymienia najczęściej pojawiające się problemy związane z transakcjami i podpowiada, jak ich unikać. Piąty to pierwszy krok w kierunku przedstawiania w MySQL szczegółów dotyczących transakcji.

Nakładanie blokad na rekordy

Operacje odczytu nie powodują nakładania blokad na rekordy (wyjątkiem są zapytania `SELECT ... FOR SHARE` i `SELECT ... FOR UPDATE`), natomiast operacje zapisu zawsze wiążą się z blokowaniem rekordów. Ten fakt jest prosty i akceptowany. Trudne pytanie brzmi: które rekordy muszą być zablokowane? To oczywiście dotyczy rekordów, które są zapisywane. Jednak w transakcji `REPEATABLE READ` silnik InnoDB może nałożyć blokadę na znacznie większą liczbę rekordów, niż jest zapisywana. W tym podrozdziale to zilustruję i wyjaśnię, dlaczego tak się dzieje. Jednak najpierw muszę przenieść terminologię na natywny język dotyczący blokowania danych InnoDB.

Skoro tabele są indeksami (przypomnij sobie punkt „Tabele InnoDB są indeksami" z rozdziału 2.), rekordy są *rekordami indeksu*. Nakładanie blokad InnoDB będzie przeanalizowane w kategoriach *nakładania blokad na rekordy indeksu*, ponieważ mogą one zawierać luki. *Luka* to zakres wartości między dwoma rekordami indeksu, jak pokazałem na rysunku 8.1. W omawianym przykładzie mamy klucz podstawowy z dwoma rekordami, dwa pseudorekordy (kresy dolny i górny) oraz trzy luki.



Rysunek 8.1. Luki w rekordzie indeksu

Rekordy indeksu są przedstawione w postaci prostokątów zawierających wewnątrz wartości indeksów — w omawianym przykładzie to 2 i 5. Pseudorekordy indeksy zostały przedstawione w postaci strzałek na obu końcach indeksu: *kres dolny* (tzw. infimum) i *kres górny* (tzw. supremum). Każde drzewo typu B indeksu zawiera dwa pseudorekordy: kres dolny przedstawia wszystkie wartości indeksu mniejsze niż wartość minimalnego rekordu indeksu (w omawianym przykładzie to 2), natomiast kres górny przedstawia wszystkie wartości indeksu większe niż wartość maksymalnego rekordu indeksu (w omawianym przykładzie to 5). Rekordy indeksu nie rozpoczynają się w miejscu elementu 2 i nie kończą na elemencie 5. Z technicznego punktu widzenia rozpoczynają się od kresu dolnego i kończą na kresie górnym, a przykłady zamieszczone w tym podrozdziale pokazują wagę tych szczegółów. Luki zostały przedstawione w postaci prostokątów zaznaczonych przerywanymi liniami i pozbawionych wartości indeksu. Jeżeli klucz podstawowy ma postać pojedynczej 4-bajtowej liczby całkowitej bez znaku, wówczas trzy luki w omawianym przykładzie są następujące (zastosowałem notację odstępów):

- `[0, 2)`
- `(2, 5)`
- `(5, 4294967295]`

Podczas analizowania blokad nakładanych na rekordy pojawia się określenie *rekord indeksu* zamiast po prostu *rekord*, ponieważ rekord indeksu zawiera luki, a mówienie o rekordzie z lukami może okazać się dezorientujące. Na przykład jeśli aplikacja ma dwa rekordy z wartościami 2 i 5, to nie wiąże się z luką w rekordach zawierających wartości 3 i 4, ponieważ być może to nie są prawidłowe wartości dla aplikacji. Z kolei w przypadku indeksu między wartościami rekordu indeksu 2 i 5 wartości 3 i 4 tworzą lukę rekordu indeksu (przy założeniu, że mamy do czynienia z kolumnami typu liczb całkowitych). Krótko mówiąc, aplikacja działa na rekordach, natomiast silnik InnoDB nakłada na nie blokady, korzystając z rekordów indeksu. Przykłady zamieszczone w tym podrozdziale pokazują, że blokady luk są zaskakująco dominujące i bezsprzecznie ważniejsze niż poszczególne blokady rekordów indeksu.

Pojęcie *blokada danych* odwołuje się do wszystkich typów blokad. Istnieje wiele typów blokad danych, natomiast w tabeli 8.1 wymieniłem jedynie podstawowe blokady danych InnoDB.

Tabela 8.1. Podstawowe blokady danych InnoDB

Typ blokady	Skrót	Blokada luki	Blokady
Blokada rekordu indeksu	`REC_NOT_GAP`		Blokada pojedynczego rekordu indeksu
Blokada luki	`GAP`	✓	Blokada luki przed (mniej niż) rekordem indeksu
Blokada następnego klucza		✓	Blokada pojedynczego rekordu indeksu i luki przed nim
Blokada zamiaru wstawienia	`INSERT_INTENTION`		Pozwala na operację `INSERT` w luce

Najlepszym sposobem na zrozumienie podstaw blokad danych InnoDB jest wykorzystanie rzeczywistych transakcji, blokad i ilustracji.

Począwszy od wydania MySQL 8.0.16 blokady danych można łatwo przeanalizować za pomocą tabel funkcjonalności Performance Schema `data_locks` i `data_lock_waits`. W omówionych tutaj przykładach zostały wykorzystane te właśnie tabele tej funkcjonalności.

Wydanie MySQL 5.7 i starsze wymaga najpierw polecenia `SET GLOBAL innodb_sta tus_` ↪`output_locks=ON`, które z kolei wymaga uprawnienia MySQL `SUPER`. Następnie można wykonać zapytanie `SHOW ENGINE INNODB STATUS` i przejrzeć dane wyjściowe w celu znalezienia odpowiednich informacji o transakcjach i blokadach. To nie jest proste — nawet eksperci zmagają się z dokładnym przeanalizowaniem danych wyjściowych. Ponieważ MySQL 5.7 nie jest bieżącym wydaniem serwera, w tym podrozdziale nie będę korzystał z danych wyjściowych wygenerowanych przez tę wersję. Jednak wydanie MySQL 5.7 wciąż jest powszechnie używane, dlatego odsyłam Cię do mojego posta *MySQL Data Locks: Mapping 8.0 to 5.7* (*https://hackmysql.com/post/mysql-data-locks-mapping-80-to-57/*), zawierającego ilustrowany przewodnik dotyczący mapowania danych wyjściowych blokady danych z MySQL 5.7 na MySQL 8.0.

W tym miejscu wykorzystamy sprawdzoną, choć uproszczoną wersję tabeli `elem`, tworzoną przez kod przedstawiony na listingu 8.1.

Listing 8.1. Kod tworzący uproszczoną tabelę elem

```
CREATE TABLE `elem` (
  `id` int unsigned NOT NULL,
  `a`  char(2) NOT NULL,
  `b`  char(2) NOT NULL,
  `c`  char(2) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_a` (`a`)
) ENGINE=InnoDB;

+----+-----+----+----+
| id | a   | b  | c  |
+----+-----+----+----+
|  2 | Au  | Be | Co |
|  5 | Ar  | Br | C  |
+----+-----+----+----+
```

Tabela `elem` jest niemalże taka sama jak wcześniej, ale teraz nieunikatowy indeks `idx_a` obejmuje jedynie kolumnę `a`. Istnieją tylko dwa rekordy tworzące dwie wartości klucza podstawowego, jak pokazałem na rysunku 8.1. Ponieważ blokady rekordów to tak naprawdę blokady rekordów indeksu, a nie ma indeksu obejmującego kolumny `b` i `c`, można zignorować te dwie kolumny. Zostały uwzględnione jedynie w celu pokazania pełni przykładu i jako wyraz nostalgii za prostszymi rozdziałami, np. rozdziałem 2., w których blokady rekordów były po prostu blokadami rekordów.

Skoro zmienna `autocommit` (*https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_autocommit*) jest domyślnie zdefiniowana, przedstawione tutaj przykłady rozpoczynają się od słowa kluczowego `BEGIN` wyraźnie wskazującego na rozpoczęcie transakcji. Blokady będą zwolnione po zakończeniu transakcji. Dlatego transakcja jest utrzymywana w działaniu (brak słowa kluczowego `COMMIT` lub `ROLLBACK`) w celu przeanalizowania blokad danych nałożonych przez polecenie SQL znajdujące się po `BEGIN` (lub oczekujące na nałożenie). Na końcu każdego przykładu blokady danych zostaną wyświetlone przez wykonanie zapytania do tabeli `performance_schema.data_locks`.

Blokada rekordu indeksu i następnego klucza

Zapytanie `UPDATE` do tabeli `elem` z użyciem klucza podstawowego w celu dopasowania rekordów powoduje nałożenie czterech blokad danych w domyślnym poziomie izolacji transakcji, `REPEATABLE READ`:

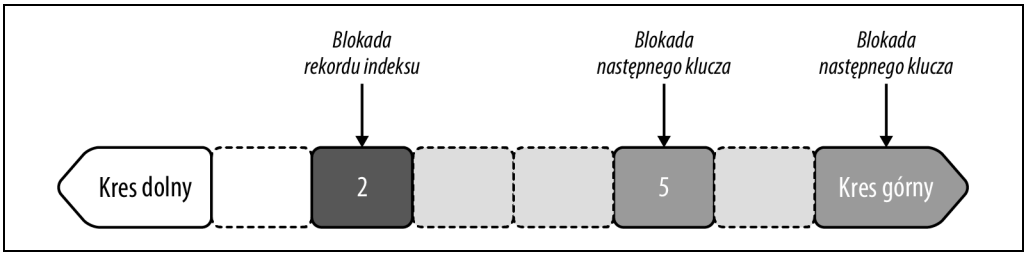
```
BEGIN;
UPDATE elem SET c='' WHERE id BETWEEN 2 AND 5;

SELECT index_name, lock_type, lock_mode, lock_status, lock_data
FROM   performance_schema.data_locks
WHERE  object_name = 'elem';

+------------+-----------+---------------+-------------+-----------------------+
| index_name | lock_type | lock_mode     | lock_status | lock_data             |
+------------+-----------+---------------+-------------+-----------------------+
| NULL       | TABLE     | IX            | GRANTED     | NULL                  |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 2                     |
```

```
| PRIMARY    | RECORD    | X             | GRANTED     | supremum pseudo-record|
| PRIMARY    | RECORD    | X             | GRANTED     | 5                     |
+------------+-----------+---------------+-------------+-----------------------+
```

Przed zilustrowaniem i wyjaśnieniem tych blokad danych chciałbym pokrótce przedstawić znaczenie poszczególnych rekordów:

- Pierwszy rekord to *blokada tabeli*, na co wskazuje wartość kolumny `lock_type`. InnoDB to silnik bazy danych stosujący blokady na poziomie rekordu, podczas gdy MySQL wymaga również blokad na poziomie tabeli (zajrzyj do punktu „Czas blokady" w rozdziale 1.). Blokada tabeli będzie dla każdej tabeli wymienionej w zapytaniach transakcji. Informacje o blokadach tabeli zamieściłem tutaj w celu przekazania pełnego obrazu sytuacji. Możesz je zignorować, ponieważ koncentrujemy się na blokadach rekordów indeksu.
- Drugi rekord to *blokada rekordu indeksu* dla wartości 2 klucza podstawowego, na co wskazują wszystkie kolumny. Tajemniczą kolumną jest `lock_mode`: wartość `X` oznacza blokadę na wyłączność (nieużyta w tym przykładzie wartość `S` oznacza blokadę współdzieloną), `REC_NOT_GAP` zaś oznacza blokadę rekordu indeksu.
- Trzeci rekord to *blokada następnego klucza* dla pseudorekordu kresu górnego. W kolumnie `lock_mode` samotna wartość `X` oznacza blokadę na wyłączność, a `S` blokadę współdzieloną następnego klucza. Potraktuj ją jako wartość `X,NEXT_KEY`.
- Czwarty rekord to *blokada następnego klucza* dla wartości 5 klucza podstawowego. Także tutaj samotna wartość `X` w kolumnie `lock_mode` oznacza blokadę na wyłączność, a `S` współdzieloną następnego klucza. Potraktuj ją jako wartość `X,NEXT_KEY`.

Na rysunku 8.2 pokazałem wpływ wywierany przez te blokady danych.



Rysunek 8.2. Blokady rekordu indeksu i następnego klucza dla klucza podstawowego w transakcji o poziomie izolacji REPEATABLE READ

Zablokowane rekordy indeksu są oznaczone ciemniejszym kolorem, natomiast niezablokowane są białe. Blokada rekordu indeksu dla wartości 2 klucza podstawowego jest oznaczona jeszcze ciemniejszym kolorem. Ten rekord indeksu został zablokowany, ponieważ odpowiadający mu rekord dopasowuje warunek tabeli `id BETWEEN 2 AND 5`.

Blokada następnego klucza dla wartości 5 klucza podstawowego została oznaczona kolorem średnioszarym, a luka przed nią kolorem jasnoszarym. Ten rekord indeksu jest zablokowany, ponieważ odpowiadający mu rekord również dopasował warunek tabeli. Luka przed tym rekordem została zablokowana, gdyż jest to blokada następnego klucza. Luka obejmuje nieistniejące wartości 3 i 4 klucza podstawowego (dla których nie ma odpowiadających im rekordów).

Podobnie blokada następnego klucza dla pseudorekordu kresu górnego jest w kolorze średnioszarym, a luka przed nim w kolorze jasnoszarym. Luka obejmuje wszystkie wartości klucza podstawowego większe niż 5. Interesujące pytanie brzmi: dlaczego nakładamy blokadę na pseudorekord kresu górnego, która *obejmuje* wszystkie wartości klucza podstawowego większe niż 5, podczas gdy warunek tabeli *wyklucza* wartości klucza podstawowego większe niż 5? Odpowiedź na to pytanie jest równie interesująca, ale poznasz ją w dalszej części rozdziału.

Sprawdź teraz, czy na luki faktycznie zostały nałożone blokady. W tym celu spróbuj wstawić rekord (używając innej transakcji z włączoną opcją jej automatycznego zatwierdzania):

```
mysql> INSERT INTO elem VALUES (3, 'Au', 'B', 'C');
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction

+------------+-----------+-----------------------+-------------+-----------+
| index_name | lock_type | lock_mode             | lock_status | lock_data |
+------------+-----------+-----------------------+-------------+-----------+
| PRIMARY    | RECORD    | X,GAP,INSERT_INTENTION | WAITING     | 5         |
....

mysql> INSERT INTO elem VALUES (6, 'Au', 'B', 'C');
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction

+------------+-----------+--------------------+-------------+--------------------+
| index_name | lock_type | lock_mode          | lock_status | lock_data          |
+------------+-----------+--------------------+-------------+--------------------+
| PRIMARY    | RECORD    | X,INSERT_INTENTION | WAITING     | supremum pseudo... |
```

W pierwszym zapytaniu `INSERT` mamy przekroczenie czasu oczekiwania na nałożenie na lukę między wartościami 2 i 5 blokady zamiaru wstawienia danych, do której miałaby zostać wstawiona nowa wartość (3). Wprawdzie kolumna `lock_data` podaje wartość 5, ale ten rekord indeksu *nie* został zablokowany, ponieważ to nie jest rekord indeksu ani blokada następnego klucza. To jest blokada zamiaru wstawienia, czyli specjalny rodzaj blokady luki (dla zapytania `INSERT`). Dlatego zostaje zablokowana luka przed wartością 5. Więcej informacji na temat blokad zamiaru wstawienia znajdziesz w dalszej części rozdziału.

W drugim zapytaniu `INSERT` mamy przekroczenie czasu oczekiwania na nałożenie blokady następnego klucza dla pseudorekordu kresu górnego, ponieważ nowa wartość, 6, jest większa niż bieżąca wartość maksymalna, 5. Dlatego byłaby wstawiona między rekordem indeksu wartości maksymalnej i pseudorekordem kresu górnego.

Te zapytania `INSERT` potwierdzają słuszność mechanizmu pokazanego na rysunku 8.2: zostaje zablokowany niemal cały indeks, z wyjątkiem wartości mniejszych niż 2. Dlaczego silnik InnoDB używa blokad następnego rekordu, które nakładają blokady na luki zamiast na rekordy indeksu? Ponieważ poziom izolacji transakcji został określony jako `REPEATABLE READ`, ale to niepełna odpowiedź. Pełna odpowiedź nie jest wcale taka prosta, więc proszę jeszcze o chwilę cierpliwości. Dzięki nałożeniu blokad na luki przed rekordami indeksu, których dotyczyło zapytanie, blokady następnego klucza izolują całe zakresy rekordów indeksu, do których zapytanie uzyskuje dostęp — to oznacza litera *I* we właściwościach ACID: izolacja. Chroni to przed zjawiskiem o nazwie *rekordy widma* (*https://dev.mysql.com/doc/refman/8.0/en/innodb-next-key-locking.html*) lub *odczyty widma*, gdy później transakcja odczytuje rekordy, które nie były odczytywane wcześniej.

Te rekordy są określane mianem *widm*, ponieważ niczym duchy pojawiają się w tajemniczy sposób. (Słowo *widmo* (ang. *phantom*) zostało faktycznie użyte w standardzie ANSI SQL-92). Rekordy widma stanowią złamanie zasady izolacji i dlatego są niedozwolone na niektórych poziomach izolacji transakcji. Czas na prawdziwie tajemniczy fragment tego wyjaśnienia: standard ANSI SQL-92 *zezwala* na rekordy widma na poziomie izolacji transakcji `REPEATABLE READ`, natomiast InnoDB im zapobiega za pomocą blokad następnego klucza. Nie będę się tutaj zagłębiać bardziej w to, dlaczego InnoDB nie pozwala używać rekordów widm na poziomie izolacji transakcji `REPEATABLE READ`. Ta wiedza nie zmienia faktu, a ponadto dość często zdarza się w serwerach baz danych implementowanie poziomów izolacji transakcji inaczej, niż wskazuje na to standard[1]. W celu przedstawienia pełnego obrazu sytuacji muszę dodać, że standard ANSI SQL-92 zabrania używania rekordów widm jedynie w najwyższym poziomie izolacji transakcji: `SERIALIZABLE`. Silnik InnoDB obsługuje ten poziom, ale nie będę go omawiał w tym rozdziale, ponieważ nie jest zbyt często stosowany. Poziom izolacji transakcji `REPEATABLE READ` jest domyślny w MySQL, a InnoDB używa blokad następnego klucza w celu uniknięcia rekordów widm na poziomie izolacji transakcji `REPEATABLE READ`.

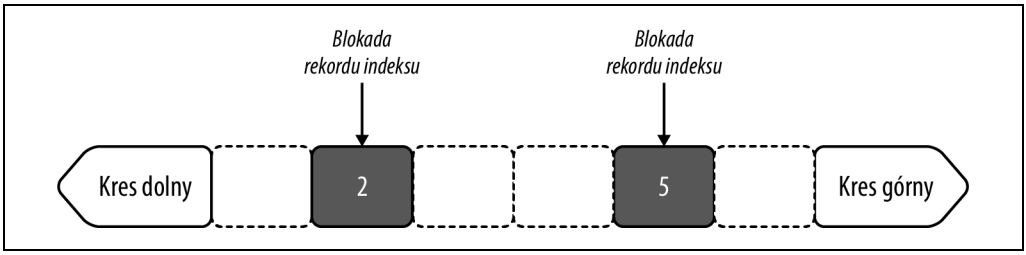
Poziom izolacji transakcji `READ COMMITTED` wyłącza nakładanie blokad na luki, co obejmuje także blokady następnego klucza. Aby się o tym przekonać, zmień poziom izolacji transakcji na `READ COMMITTED`:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;
UPDATE elem SET c='' WHERE id BETWEEN 2 AND 5;

SELECT index_name, lock_type, lock_mode, lock_status, lock_data
FROM   performance_schema.data_locks
WHERE  object_name = 'elem';
+------------+-----------+---------------+-------------+-----------+
| index_name | lock_type | lock_mode     | lock_status | lock_data |
+------------+-----------+---------------+-------------+-----------+
| NULL       | TABLE     | IX            | GRANTED     | NULL      |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 2         |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 5         |
+------------+-----------+---------------+-------------+-----------+
```

Polecenie `SET TRANSACTION` ma zastosowanie jedynie dla następnej transakcji. Po następnej transakcji kolejne będą używały domyślnego poziomu izolacji transakcji. Więcej informacji na ten temat znajdziesz w dokumentacji polecenia `SET TRANSACTION` (*https://dev.mysql.com/doc/refman/8.0/en/set-transaction.html*).

To samo zapytanie `UPDATE` na poziomie izolacji transakcji `READ COMMITTED` spowoduje nałożenie blokad rekordów indeksów tylko na dopasowane rekordy, jak pokazałem na rysunku 8.3.

[1] Jeżeli chcesz dowiedzieć się więcej na ten temat, zapoznaj się z dokumentem *A Critique of ANSI SQL Isolation Levels* (*https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf*), który można uznać za lekturę obowiązkową w zakresie poziomów izolacji obsługiwanych przez standard ANSI SQL-92.



Rysunek 8.3. Blokady rekordu indeksu dla klucza podstawowego w transakcji o poziomie izolacji READ COMMITTED

Dlaczego nie można użyć poziomu izolacji transakcji `READ COMMITTED`? To pytanie ma związek z cechą wzorców dostępu (zobacz punkt „Izolacja transakcji" w rozdziale 4.) całkowicie zależną od aplikacji, a nawet od zapytania. Innymi słowy poziom izolacji transakcji `READ COMMITTED` ma dwa ważne efekty uboczne:

- Ponowne wykonanie tego samego polecenia odczytu może zwrócić inne rekordy.
- Ponownie wykonane to samo polecenie zapisu może dotyczyć innych rekordów.

Te efekty uboczne wyjaśniają, dlaczego silnik InnoDB nie musi używać spójnych migawek dla odczytów lub nakładać blokady na luki między rekordami w trakcie operacji zapisu: poziom izolacji transakcji `READ COMMITTED` pozwala transakcji na odczytywanie lub zapisywanie odmiennych rekordów (dla zatwierdzonych zmian) w różnym czasie. (W dalszej części rozdziału, w podrozdziale „MVCC i dzienniki przywracania", znajdziesz definicję wyrażenia *spójna migawka*). Uważnie przeanalizuj te efekty uboczne dla swojej aplikacji. Jeżeli masz pewność, że nie spowodują odczytania, zapisania lub zwrócenia przez transakcję nieprawidłowych danych, wówczas poziom izolacji transakcji `READ COMMITTED` zmniejsza ilość blokad i wielkość dziennika przywracania, co może poprawić wydajność działania.

Blokady luk

Blokady lub są czysto zaporowe: uniemożliwiają innym transakcjom wstawianie rekordów w luce. Tylko na tym polega ich działanie.

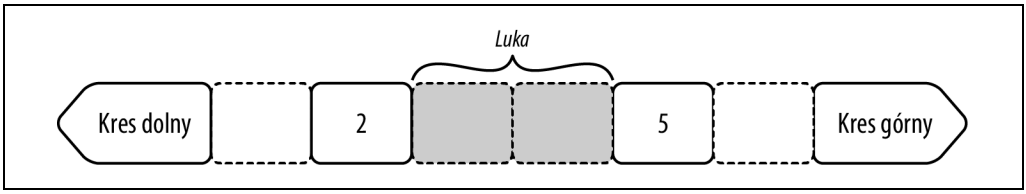
Wiele transakcji może nakładać blokady na te same luki, ponieważ wszystkie blokady luk są ze sobą zgodne. Skoro jednak blokada luki uniemożliwia *innym* transakcjom wstawianie rekordów w luce, tylko jedna transakcja może wstawiać rekordy w luce, gdy jest jedyną transakcją nakładającą blokadę na tę lukę. Dwie lub więcej blokad na tej samej luce uniemożliwia wszystkim transakcjom wstawianie rekordów do tej luki.

Cel blokady luki jest jasno określony: uniemożliwienie innym transakcjom wstawiania rekordów w luce. Nałożenie blokady na lukę nie jest trudnym zadaniem — może to zrobić każde zapytanie, które uzyskuje dostęp do danej luki. Nawet odczyt niczego może spowodować nałożenie blokady na lukę, uniemożliwiając w ten sposób wstawianie rekordów.

```
BEGIN;
SELECT * FROM elem WHERE id = 3 FOR SHARE;

SELECT index_name, lock_type, lock_mode, lock_status, lock_data
```

```
FROM   performance_schema.data_locks
WHERE  object_name = 'elem';
+------------+-----------+-----------+-------------+-----------+
| index_name | lock_type | lock_mode | lock_status | lock_data |
+------------+-----------+-----------+-------------+-----------+
| NULL       | TABLE     | IS        | GRANTED     | NULL      |
| PRIMARY    | RECORD    | S,GAP     | GRANTED     | 5         |
+------------+-----------+-----------+-------------+-----------+
```

Na pierwszy rzut oka wydaje się, że zapytanie SELECT jest nieszkodliwe: SELECT na poziomie izolacji transakcji REPEATABLE READ używa spójnej migawki, natomiast FOR SHARE nakłada jedynie blokady współdzielone, więc nie będzie blokować innych operacji odczytu. Co ważniejsze, zapytanie SELECT nie dopasowuje żadnych rekordów: tabela elem ma wartości 2 i 5 klucza podstawowego, ale już nie wartość 3. Brak dopasowanych rekordów to brak blokad, prawda? Nie! Przez uzyskanie dostępu do luki za pomocą READ REPEATABLE i SELECT ... FOR SHARE przyjmujesz założenie o istnieniu pojedynczej blokady luki, jak pokazałem na rysunku 8.4.



Rysunek 8.4. Pojedyncza blokada luki

Użyłem określenia *pojedyncza* blokada, ponieważ nie towarzyszy jej blokada następnego klucza ani blokada zamiaru wstawienia danych. Istnieje tylko ta blokada. Wszystkie blokady luk — współdzielone lub na wyłączność — uniemożliwiają innym transakcjom wstawianie rekordów w luce. To niewinnie wyglądające zapytanie SELECT okazało się w rzeczywistości podstępną blokadą. Im większa luka, tym blokada obejmuje więcej rekordów, co dokładnie omówię w następnym punkcie, na przykładzie indeksu wtórnego.

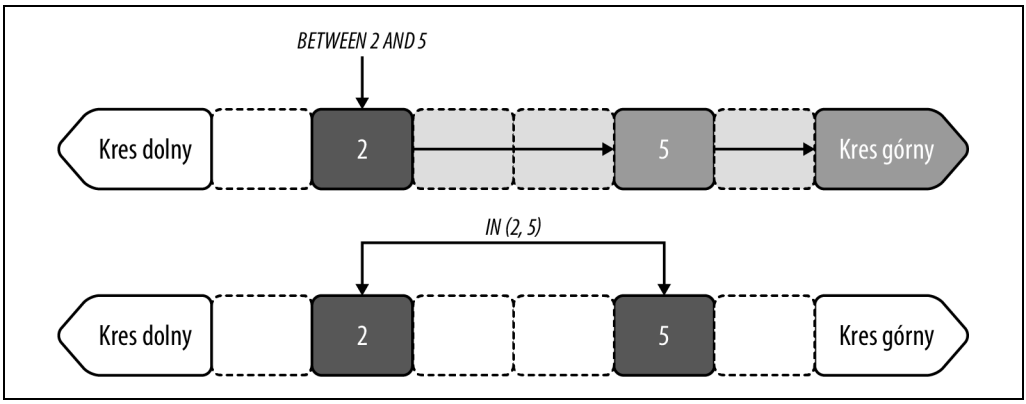
Możliwość łatwego nałożenia blokady luki poprzez uzyskanie dostępu do luki to część odpowiedzi na interesujące pytanie, które pojawiło się we wcześniejszej części rozdziału: dlaczego nakładamy blokadę na pseudorekord kresu górnego, która *obejmuje* wszystkie wartości klucza podstawowego większe niż 5, podczas gdy warunek tabeli *wyklucza* wartości klucza podstawowego większe niż 5? Pozwól mi jeszcze bardziej podnieść poziom napięcia. Spójrz na pierwotne zapytanie i blokowane przez nie dane:

```
BEGIN;
UPDATE elem SET c='' WHERE id BETWEEN 2 AND 5;

+------------+-----------+---------------+-------------+------------------------+
| index_name | lock_type | lock_mode     | lock_status | lock_data              |
+------------+-----------+---------------+-------------+------------------------+
| NULL       | TABLE     | IX            | GRANTED     | NULL                   |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 2                      |
| PRIMARY    | RECORD    | X             | GRANTED     | supremum pseudo-record |
| PRIMARY    | RECORD    | X             | GRANTED     | 5                      |
+------------+-----------+---------------+-------------+------------------------+
```

Teraz spójrz na to samo zapytanie, ale zawierające klauzulę `IN` zamiast `BETWEEN`:

```
BEGIN;
UPDATE elem SET c='' WHERE id IN (2, 5);

+------------+-----------+---------------+-------------+-----------+
| index_name | lock_type | lock_mode     | lock_status | lock_data |
+------------+-----------+---------------+-------------+-----------+
| NULL       | TABLE     | IX            | GRANTED     | NULL      |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 2         |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 5         |
+------------+-----------+---------------+-------------+-----------+
```

Obie transakcje wykorzystują poziom izolacji transakcji `REPEATABLE READ` i oba zapytania powodują wygenerowanie tego samego planu `EXPLAIN`: dostęp zakresu na podstawie klucza podstawowego. Jednak nowe zapytanie powoduje nałożenie blokad jedynie na dopasowane rekordy. Na czym polega sztuczka? Na rysunku 8.5 pokazałem, co się dzieje podczas wykonywania obu tych zapytań.



Rysunek 8.5. Dostęp zakresu dla klauzuli BETWEEN kontra IN w przypadku poziomu izolacji transakcji REPEATABLE READ

Dostęp do rekordu z wykorzystaniem klauzuli `BETWEEN` odbywa się zgodnie z oczekiwaniami: od wartości 2 do 5 z wszystkimi, które znajdują się między nimi. W najprostszym ujęciu sekwencja dostępu rekordu podczas użycia klauzuli `BETWEEN` przedstawia się następująco:

1. Odczytanie rekordu o wartości indeksu 2.
2. Dopasowanie rekordu: nałożenie blokady na rekord.
3. Następna wartość indeksu: 5.
4. Poruszanie się przez lukę od wartości 2 do wartości 5.
5. Odczytanie rekordu o wartości indeksu 5.
6. Dopasowanie rekordu: blokada następnego klucza.
7. Następna wartość indeksu: kres górny.
8. Poruszanie się przez lukę od wartości 5 do kresu górnego.
9. Koniec indeksu: blokada następnego klucza.

Natomiast sekwencja dla dostępu rekordu z użyciem klauzuli `IN` jest znacznie prostsza:

1. Odczytanie rekordu o wartości indeksu 2.
2. Dopasowanie rekordu: nałożenie blokady na rekord.
3. Odczytanie rekordu o wartości indeksu 5.
4. Dopasowanie rekordu: nałożenie blokady na rekord.

Pomimo wygenerowania tego samego planu `EXPLAIN` i dopasowania tych samych rekordów to zapytanie inaczej uzyskuje dostęp do rekordów. Pierwotna wersja zapytania (z klauzulą `BETWEEN`) uzyskuje dostęp do luk i dlatego używa blokad następnego klucza do zablokowania tych luk. Natomiast nowe zapytanie (`IN`) nie uzyskuje dostępu do luk i dlatego używa blokad rekordów indeksu. Niech to Cię nie zmyli: wersja oparta na klauzuli `IN` nie eliminuje blokad luk. Jeżeli warunkiem tabeli nowego zapytania będzie `IN (2, 3, 5)`, wówczas dostęp do luki między wartościami 2 i 5 spowoduje nałożenie blokady luki (nie blokady następnego klucza):

```
BEGIN;
UPDATE elem SET c='' WHERE id IN (2, 3, 5);

+------------+-----------+---------------+-------------+-----------+
| index_name | lock_type | lock_mode     | lock_status | lock_data |
+------------+-----------+---------------+-------------+-----------+
| NULL       | TABLE     | IX            | GRANTED     | NULL      |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 2         |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 5         |
| PRIMARY    | RECORD    | X,GAP         | GRANTED     | 5         |
+------------+-----------+---------------+-------------+-----------+
```

W tym przypadku masz pojedynczą blokadę luki: `X,GAP`. Zwróć uwagę na pewna kwestię: to nie jest blokada następnego klucza dla pseudorekordu kresu górnego, ponieważ klauzula `IN (2, 3, 5)` nie ma dostępu do tej luki. Uważaj na lukę.

Nakładanie blokad na luki można dość łatwo wyłączyć przez użycie poziomu izolacji transakcji `READ COMMITTED`. Ten poziom nie wymaga nakładania blokad na luki (lub blokad następnego klucza), ponieważ rekordy indeksu w lukach mogą się zmieniać, a podczas wykonywania każde zapytanie uzyskuje dostęp do najnowszych zmian (zatwierdzonych rekordów). Nawet pojedyncza blokada luki spowodowana przez zapytanie `SELECT * FROM elem WHERE id = 3 FOR SHARE` zostanie unieważniona przez poziom izolacji transakcji `READ COMMITTED`.

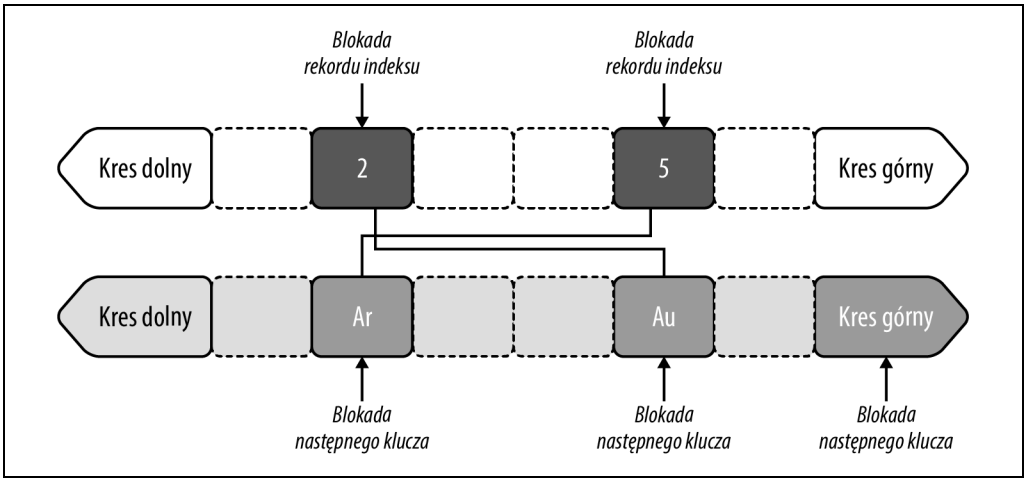
Indeksy wtórne

Indeksy wtórne, szczególnie te nieunikatowe, prowadzą do rozległych potencjalnych konsekwencji związanych z nadkładaniem blokad na rekordy. Przypomnij sobie, że uproszczona wersja tabeli `elem` (zobacz listing 8.1) ma nieunikatowy indeks dla kolumny `a`. Mając to na uwadze, zobacz, jak przedstawione tutaj zapytanie `UPDATE` korzystające z poziomu izolacji transakcji `REPEATABLE READ` nałoży blokady rekordów indeksu dla indeksu wtórnego i klucza podstawowego:

```
BEGIN;
UPDATE elem SET c='' WHERE a BETWEEN 'Ar' AND 'Au';

SELECT   index_name, lock_type, lock_mode, lock_status, lock_data
```

```
FROM     performance_schema.data_locks
WHERE    object_name = 'elem'
ORDER BY index_name;
+------------+-----------+---------------+-------------+-----------------------+
| index_name | lock_type | lock_mode     | lock_status | lock_data             |
+------------+-----------+---------------+-------------+-----------------------+
| NULL       | TABLE     | IX            | GRANTED     | NULL                  |
| a          | RECORD    | X             | GRANTED     | supremum pseudo-record |
| a          | RECORD    | X             | GRANTED     | 'Au', 2               |
| a          | RECORD    | X             | GRANTED     | 'Ar', 5               |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 2                     |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 5                     |
+------------+-----------+---------------+-------------+-----------------------+
```

Na rysunku 8.6 pokazałem te sześć blokad rekordów: cztery dotyczą indeksu wtórnego i dwie klucza podstawowego.



Rysunek 8.6. Blokady następnego rekordu w indeksie wtórnym w przypadku poziomu izolacji transakcji REPEATABLE READ

Zapytanie `UPDATE` dopasowało jedynie dwa rekordy, ale nałożyło blokadę na cały indeks wtórny, co uniemożliwiło wstawienie jakichkolwiek wartości. Blokady indeksu wtórnego są podobne do pokazanych na rysunku 8.2. Jednak teraz mamy blokadę następnego klucza dla pierwszego rekordu indeksu w rekordzie indeksu wtórnego: to krotka `('Ar', 5)`, w której 5 to wartość klucza podstawowego. Blokada następnego klucza powoduje odizolowanie zakresu od nowych wartości duplikatu „Ar”. Na przykład to uniemożliwia wstawienie krotki `('Ar', 1)`, która w przypadku sortowania danych znalazłaby się przed krotką `('Ar', 5)`.

Normalnie silnik InnoDB nie nakłada blokady na cały indeks wtórny. Tak się dzieje w omawianym przykładzie, ponieważ mamy tylko dwa rekordy indeksu (zarówno w kluczu podstawowym, jak i w nieunikatowym indeksie wtórnym). Przypomnij sobie punkt „Wyjątkowa selektywność” w rozdziale 2.: im mniejsza selektywność, tym większe luki. Oto przykład ekstremalny: jeżeli nieunikatowy indeks ma pięć unikatowych wartości równo rozłożonych w 100 000 rekordów, wówczas mamy 20 000 rekordów indeksu na każdy rekord (100 000 rekordów / liczebność 5) lub inaczej 20 000 rekordów indeksu na lukę.

Im niższa selektywność indeksu, tym większe luki rekordów indeksu.

Poziom izolacji transakcji `READ COMMITTED` pozwala uniknąć nakładania blokad na luki, nawet w przypadku nieunikatowych indeksów wtórnych, ponieważ jedynie dopasowane rekordy będą blokowane za pomocą blokad rekordów indeksu. Nie pójdziemy tutaj drogą na skróty i przeanalizujemy blokady danych InnoDB nakładane na nieunikatowe indeksy wtórne dla różnego rodzaju zmian w danych.

Na końcu poprzedniego punktu zmiana klauzuli `BETWEEN` na klauzulę `IN` zapobiegła nałożeniu blokady na lukę, choć takie rozwiązanie nie sprawdza się w przypadku indeksów nieunikatowych. W rzeczywistości silnik InnoDB *nakłada* blokadę luki w takim przypadku:

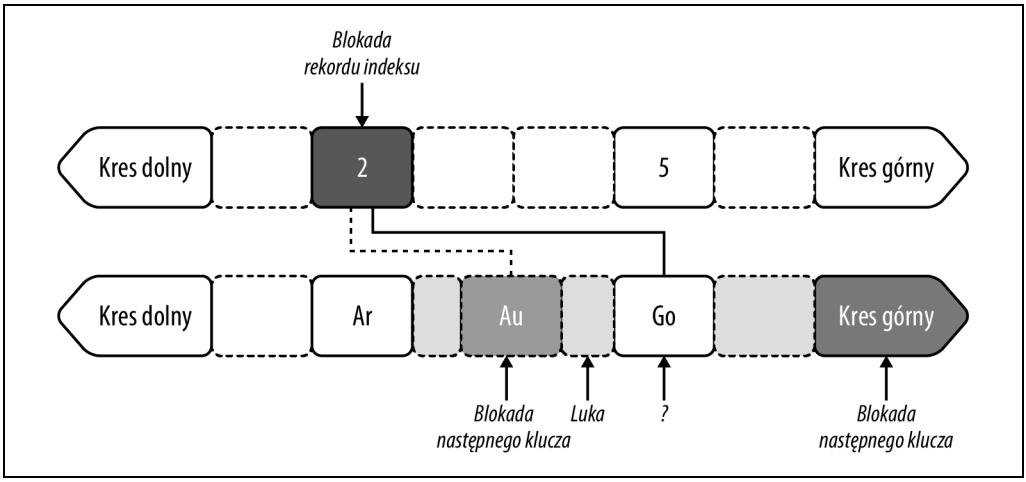
```
BEGIN;
UPDATE elem SET c='' WHERE a IN ('Ar', 'Au');

SELECT   index_name, lock_type, lock_mode, lock_status, lock_data
FROM     performance_schema.data_locks
WHERE    object_name = 'elem'
ORDER BY index_name;
+------------+-----------+---------------+-------------+-----------------------+
| index_name | lock_type | lock_mode     | lock_status | lock_data             |
+------------+-----------+---------------+-------------+-----------------------+
| a          | RECORD    | X,GAP         | GRANTED     | 'Au', 2               |
```

Z danych wyjściowych usunąłem pierwotne blokady danych (są one identyczne), aby podkreślić nową blokadę luki dla krotki `('Au', 2)`. Szczerze mówiąc, ta blokada luki jest zbędna dzięki blokadzie następnego klucza nałożonej na tę samą krotkę. Jednak to nie jest wynik nieprawidłowego stosowania blokad lub dostępu do danych. Dlatego niech tak będzie. Nie zapominaj również o tym, że InnoDB kryje w sobie wiele cudów i tajemnic. Jak wyglądałoby bez nich życie?

Trzeba koniecznie przeanalizować blokady danych, ponieważ silnik InnoDB kryje w sobie wiele niespodzianek. Wprawdzie ten podrozdział zawiera dość dokładne informacje, ale to jedynie wierzchołek góry lodowej — temat blokad w silniku InnoDB jest głęboki i kryje wiele tajemnic. Na przykład jakie mogą być wymagania blokady danych w przypadku zmiany „Au” na „Go”? Przeanalizuj blokady danych w przypadku takiej zmiany:

```
BEGIN;
UPDATE elem SET a = 'Go' WHERE a = 'Au';

+------------+-----------+---------------+-------------+-----------------------+
| index_name | lock_type | lock_mode     | lock_status | lock_data             |
+------------+-----------+---------------+-------------+-----------------------+
| NULL       | TABLE     | IX            | GRANTED     | NULL                  |
| a          | RECORD    | X             | GRANTED     | supremum pseudo-record |
| a          | RECORD    | X             | GRANTED     | 'Au', 2               |
| a          | RECORD    | X,GAP         | GRANTED     | 'Go', 2               |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 2                     |
+------------+-----------+---------------+-------------+-----------------------+
```

Na rysunku 8.7 pokazałem wizualnie nasze cztery blokady danych.



Rysunek 8.7. Uaktualnienie wartości nieunikatowego indeksu wtórnego dla poziomu izolacji transakcji REPEATABLE READ

Wartość „Au” zniknęła — zastąpiona przez „Go” — a mimo to silnik InnoDB wciąż nakłada blokadę następnego klucza na krotkę (`'Au', 2`). Nowa wartość „Go” nie ma blokady rekordu indeksu ani następnego klucza, lecz jedynie blokadę luki przed krotką (`'Go', 2`). Jak wygląda kwestia nałożenia blokady na nowy rekord indeksu „Go”? Czy mamy do czynienia z tego samego rodzaju efektem ubocznym poziomu izolacji transakcji `REPEATABLE READ`? Zmienimy teraz poziom izolacji transakcji i ponownie przeanalizujemy blokady danych:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;
UPDATE elem SET a = 'Go' WHERE a = 'Au';

+------------+-----------+---------------+-------------+-----------+
| index_name | lock_type | lock_mode     | lock_status | lock_data |
+------------+-----------+---------------+-------------+-----------+
| NULL       | TABLE     | IX            | GRANTED     | NULL      |
| a          | RECORD    | X,REC_NOT_GAP | GRANTED     | 'Au', 2   |
| PRIMARY    | RECORD    | X,REC_NOT_GAP | GRANTED     | 2         |
+------------+-----------+---------------+-------------+-----------+
```

Zmiana poziomu izolacji transakcji na `REPEATABLE COMMITED` zgodnie z oczekiwaniami powoduje wyeliminowanie blokady luki, ale gdzie znajduje się blokada — jakakolwiek — dla nowej wartości „Go”? Na początku podrozdziału stwierdziłem, że „operacje zapisu zawsze wiążą się z blokowaniem rekordów”, natomiast w omawianej sytuacji silnik InnoDB informuje o braku blokad dla operacji zapisu…

A jeśli Ci powiem, że silnik InnoDB został tak zoptymalizowany, że potrafi blokować bez nakładania blokad? Następny rodzaj blokady danych, zamiar wstawienia, wykorzystamy do zagłębienia się w mechanizm nakładania blokad w InnoDB i wyjaśnienia tej tajemnicy.

Blokada zamiaru wstawienia

Blokada zamiaru wstawienia to specjalny typ blokady luki oznaczający, że transakcja wstawi rekord do luki, gdy nie została na nią nałożona blokada przez inne transakcje. Tylko blokada luki może zablokować blokadę zamiaru wstawienia. (Zapamiętaj: *blokada luki* obejmuje blokadę następnego klucza, ponieważ druga z wymienionych jest połączeniem blokady rekordu indeksu i blokady luki). Blokada zamiaru wstawienia jest zgodna z innymi blokadami zamiaru wstawienia (czyli ich nie blokuje). To ma duże znaczenie dla wydajności działania zapytania `INSERT`, ponieważ pozwala jednocześnie wielu transakcjom wstawiać różne rekordy do tej samej luki. W jaki sposób silnik InnoDB radzi sobie z obsługą duplikatów kluczy? Do tego pytania powrócę po przedstawieniu innych faktów dotyczących blokady zamiaru wstawienia, dzięki którym odpowiedź stanie się znacznie jaśniejsza.

Blokada luki *uniemożliwia* wykonanie zapytania `INSERT`. Natomiast blokada zamiaru wstawienia *pozwala* na wykonanie zapytania `INSERT`.

Blokada zamiaru wstawienia jest szczególna z trzech powodów:

- Blokada zamiaru wstawienia nie nakłada blokady na lukę, ponieważ jak sugeruje słowo *zamiar*, oznacza ona *późniejsze* działanie: wstawienie rekordu, *gdy* nie będzie blokad luki nałożonych przez inne transakcje.
- Blokada zamiaru wstawienia jest nakładana i zgłaszana tylko w przypadku konfliktu z innymi blokadami nałożonymi przez inne transakcje. W przeciwnym razie blokada zamiaru wstawienia nie jest nakładana i zgłaszana przez transakcję wstawiającą rekord.
- Jeżeli została nałożona blokada zamiaru wstawienia, będzie użyta jednokrotnie i natychmiast zwolniona, przy czym silnik InnoDB nadal będzie informował o jej istnieniu, aż do zakończenia transakcji.

W takim sensie blokada zamiaru wstawienia nie jest blokadą, ponieważ nie blokuje dostępu. To bardziej warunek oczekiwania, używany przez InnoDB do zasygnalizowania, kiedy transakcja może wykonać zapytanie `INSERT`. Możliwość nałożenia blokady zamiaru wstawienia jest takim sygnałem. Natomiast jeśli transakcja nie musi czekać z powodu braku konfliktów między blokadami luk, wówczas nie czeka i nie zobaczysz blokady zamiaru wstawienia, ponieważ nie została ona nałożona.

Spójrz na blokadę zamiaru wstawienia w akcji. Rozpoczynam od nałożenia blokady na lukę między wartościami 2 i 5 klucza podstawowego, a potem w drugiej transakcji następuje próba wstawienia rekordu z wartością 3 klucza podstawowego:

```
-- Pierwsza transakcja
BEGIN;
UPDATE elem SET c='' WHERE id BETWEEN 2 AND 5;

-- Druga transakcja
BEGIN;
INSERT INTO elem VALUES (3, 'As', 'B', 'C');
```

```
+------------+-----------+-----------------------+-------------+-----------+
| index_name | lock_type | lock_mode             | lock_status | lock_data |
+------------+-----------+-----------------------+-------------+-----------+
| PRIMARY    | RECORD    | X,GAP,INSERT_INTENTION | WAITING     | 5         |
```

Wartość `X,GAP,INSERT_INTENTION` w kolumnie `lock_mode` to blokada zamiaru wstawienia. Jest również przedstawiana w postaci `X,INSERT_INTENTION` (nie w omawianym przykładzie) podczas blokowania i wstawiania do luki między rekordem indeksu wartości maksymalnej i pseudorekordem kresu górnego.

Pierwsza transakcja nakłada blokadę luki przed wartością 5 klucza podstawowego. Ta blokada luki uniemożliwia drugiej transakcji wstawienie rekordu do luki, więc powoduje ona utworzenie blokady zamiaru wstawienia i czeka na jej nałożenie. Gdy pierwsza transakcja zostanie zatwierdzona (lub wycofana), nastąpi zwolnienie blokady luki, nałożenie blokady zamiaru wstawienia i druga transakcja może wstawić rekord.

```
-- Pierwsza transakcja
COMMIT;

-- Druga transakcja
-- Wykonanie zapytania INSERT

+------------+-----------+-----------------------+-------------+-----------+
| index_name | lock_type | lock_mode             | lock_status | lock_data |
+------------+-----------+-----------------------+-------------+-----------+
| NULL       | TABLE     | IX                    | GRANTED     | NULL      |
| PRIMARY    | RECORD    | X,GAP,INSERT_INTENTION | GRANTED     | 5         |
+------------+-----------+-----------------------+-------------+-----------+
```

Jak wyjaśniłem już wcześniej, silnik InnoDB będzie kontynuował zgłaszanie blokady zamiaru wstawienia, nawet pomimo tego, że po jej nałożeniu zostaje ona użyta jednokrotnie i natychmiast zwolniona. W efekcie wydaje się, że istnieje blokada na lukę, choć to tylko złudzenie — sztuczka InnoDB w celu zmuszenia programisty do sięgnięcia głębiej. Możesz potwierdzić, że masz do czynienia ze złudzeniem. Wystarczy wstawić do luki inny rekord z wartością 4 klucza podstawowego, a zobaczysz, że blokada nie istnieje. Dlaczego silnik InnoDB kontynuuje informowanie o blokadzie zamiaru wstawienia, której tak naprawdę nie ma? O tym wie tylko kilku śmiertelników, a to i tak nie ma znaczenia. Spróbuj spojrzeć nieco dalej poza to złudzenie, a zobaczysz, czym ono *jest*: w przeszłości transakcja nakładała blokadę przed wstawieniem rekordu do luki.

W celu przedstawienia pełnego obrazu sytuacji i przejścia do głębszych aspektów nakładania blokad InnoDB, zwłaszcza pod kątem blokad zamiaru wstawienia, zamieszczam dane, które zobaczysz, gdy zapytanie `INSERT` nie nałoży blokady luki:

```
BEGIN;
INSERT INTO elem VALUES (9, 'As', 'B', 'C'); -- Does not block
+------------+-----------+-----------+-------------+-----------+
| index_name | lock_type | lock_mode | lock_status | lock_data |
+------------+-----------+-----------+-------------+-----------+
| NULL       | TABLE     | IX        | GRANTED     | NULL      |
+------------+-----------+-----------+-------------+-----------+
```

W ogóle brak blokad rekordów indeksu. To jest ogólny sposób działania blokady zamiaru wstawienia. Jednak tym tematem zająłem się, aby nieco dokładniej omówić nakładanie blokad w silniku InnoDB. Przejdę więc dalej, zadając pytanie, które nas doprowadziło do tego miejsca: dlaczego nie ma blokady rekordu indeksu (lub następnego klucza) dla nowo wstawionego rekordu? Mamy do czynienia z tą samą tajemnicą, z którą zetknęliśmy się nieco wcześniej: brak blokady dla nowej wartości „Go".

Oto i tajemnica: silnik InnoDB ma blokady jawne i niejawne, a informuje tylko o blokadach jawnych[2]. Blokady jawne istnieją jako struktury blokad w pamięci i dlatego InnoDB może o nich informować. Natomiast blokady niejawne nie istnieją: brakuje struktury blokady, więc InnoDB nie ma o czym informować.

W poprzednim przykładzie, `INSERT INTO elem VALUES (9, 'As', 'B', 'C')`, istniał rekord indeksu dla nowego rekordu, choć ten nowy rekord nie został jeszcze zatwierdzony (z powodu braku zatwierdzenia transakcji). Jeżeli inna transakcja spróbuje nałożyć blokadę na rekord, wykryje trzy warunki:

- Rekord nie został zatwierdzony.
- Rekord należy do innej transakcji.
- Rekord nie jest wyraźnie zablokowany.

A teraz czas na magię: transakcja żądająca — czyli ta, która próbuje nałożyć blokadę na rekord indeksu — konwertuje blokadę niejawną na jawną *w imieniu* transakcji, która utworzyła dany rekord indeksu. To oznacza, że jedna transakcja tworzy blokadę dla innej transakcji — ale to jeszcze nie jest dezorientujące. Skoro transakcja żądająca tworzy blokadę, którą próbuje nałożyć, to na pierwszy rzut oka wydaje się, że InnoDB informuje o transakcji oczekującej na blokadę, którą sama nałożyła — transakcja blokuje samą siebie. Istnieje sposób, by zobaczyć to złudzenie, ale to wymaga jeszcze większego zagłębienia się w temat.

Mam nadzieję, że jako inżynier używający MySQL, chcąc osiągnąć dużą wydajność działania MySQL, nie musisz nigdy schodzić aż tak głęboko w temat blokad InnoDB. Omawiam ten temat tak szczegółowo z dwóch powodów. Po pierwsze, mimo wszelkich złudzeń podstawy nakładania blokad na rekordy InnoDB w odniesieniu do poziomów izolacji transakcji nie sprawiają trudności i są możliwe do zaakceptowania. Zyskujesz doskonałe przygotowanie do obsługi wszystkich najczęściej pojawiających się problemów związanych z blokowaniem rekordów — a nawet więcej. Po drugie, silnik InnoDB zmusił mnie do tego, ponieważ zbyt długo dokładnie go analizowałem. A kiedy wszystko się rozmyło, wiedziałem, że przekroczyłem punkt, po którym nie ma już powrotu. Nie pytaj, dlaczego nakładana jest blokada pseudorekordu kresu górnego poza zakresem warunku tabeli. Nie pytaj również, dlaczego istnieją nadmiarowe blokady luk. Nie pytaj o powody konwersji blokad niejawnych. Wyświadcz sobie przysługę i nie zadawaj kolejnych przychodzących Ci na myśl pytań.

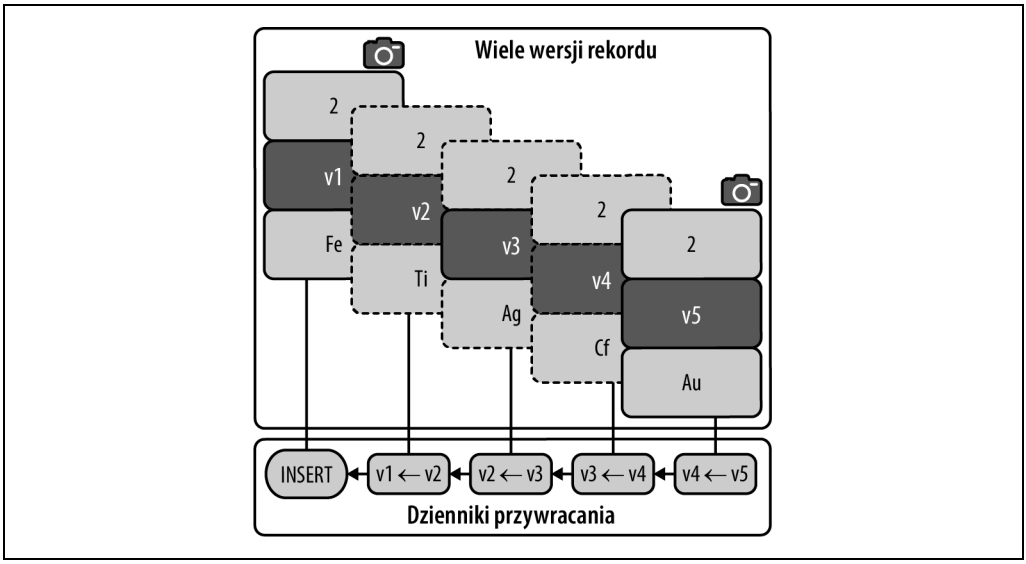
[2] Dziękuję Jakubowi Łopuszańskiemu za poinformowanie mnie o tej tajemnicy i za jej wyjaśnienie.

MVCC i dzienniki przywracania

Silnik InnoDB używa mechanizmu kontroli współbieżności (ang. *multiversion concurrency control*, MVCC) i dzienników przywracania do zapewnienia obsługi właściwości *A*, *C* oraz *I* zasady ACID. (W celu zapewnienia obsługi właściwości *D* InnoDB używa dziennika zdarzeń transakcji (zapoznaj się z podpunktem „Dziennik zdarzeń transakcji" w rozdziale 6.). *Mechanizm kontroli współbieżności* oznacza, że zmiany w rekordzie powodują utworzenie jego nowej wersji. Ten mechanizm nie jest unikatowy dla silnika InnoDB, to metoda dość powszechnie używana przez wiele magazynów danych. Po utworzeniu rekordu znajduje się on w wersji pierwszej. Po pierwszym uaktualnieniu rekordu jest on w wersji drugiej. Podstawy MVCC są proste, choć sam mechanizm szybko staje się znacznie bardziej złożony i interesujący.

Użycie pojęcia *dzienniki przywracania* jest celowym uproszczeniem, ponieważ pełna strategia rejestrowania danych wykorzystywanych do przywracania jest skomplikowana. Pojęcie to wystarczająco precyzyjnie wskazuje przeznaczenie i wpływ na wydajność działania.

Dzienniki przywracania rejestrują informacje o tym, jak wycofać zmiany i przywrócić rekord do jego poprzedniej wersji. Na rysunku 8.8 pokazałem przykładowy rekord z pięcioma wersjami i pięcioma dziennikami przywracania, co pozwala MySQL wycofać zmiany i przywrócić poprzednie wersje rekordu.



Rysunek 8.8. Rekord zawierający pięć wersji i pięć dzienników przywracania

Przypomnij sobie punkt „Tabele InnoDB są indeksami" w rozdziale 2.: przedstawiony tutaj rekord pochodzi z tabeli `elem`, ma wartość 2 klucza podstawowego i jest pokazany jako węzeł liścia klucza podstawowego. Dla zachowania zwięzłości na rysunku pokazałem jedynie wartość klucza

podstawowego, 2, wersję rekordu, od v1 do v5, oraz wartość kolumny, „Au”, dla wersji 5. Pozostałe kolumny tabeli, `b` i `c`, nie zostały pokazane.

Wersja 5 (na dole rysunku 8.8) to bieżąca wersja rekordu odczytywana przez wszystkie nowe transakcje, ale zacznę od początku. Ten rekord został utworzony z wartością „Fe” — jest to wersja 1, widoczna w lewym górnym rogu rysunku. Dla tej wersji jest dziennik przywracania, ponieważ zapytanie `INSERT` tworzy pierwszą wersję rekordu. Następnie kolumna `a` zostaje zmodyfikowana (`UPDATE`) i wartością rekordu jest teraz „T” — jest to wersja 2. Po utworzeniu wersji 2 MySQL tworzy także dziennik przywracania umożliwiający wycofanie dwóch zmian, co spowoduje przywrócenie rekordu do wersji 1. (W następnym akapicie wyjaśnię, dlaczego wersja 1 ma na rysunku kontur w postaci linii ciągłej [i ikonę przedstawiającą aparat fotograficzny], wersja 2 zaś ma kontur w postaci linii przerywanej). Następnie kolumna `a` została zmodyfikowana i zawiera wartość „Ag” — jest to wersja 3. MySQL tworzy dziennik przywracania pozwalający wycofać trzy zmiany. Ten dziennik jest powiązany z poprzednimi, więc MySQL może — jeśli zachodzi potrzeba — wycofać np. jedną zmianę i przywrócić rekord do wersji 2. Dalej mamy dwa kolejne uaktualnienia rekordu: wartość „Cf” (wersja 4) i „Au” (wersja 5).

Istnieją dwa zbiory dzienników przywracania: *dzienniki przywracania operacji wstawiania* dla zapytań `INSERT` i *dzienniki przywracania operacji uaktualniania* dla zapytań `UPDATE` i `DELETE`. Dla zachowania prostoty używam pojęcia dzienników przywracania obejmującego oba te zbiory.

Wersja 1 ma na rysunku kontur w postaci linii ciągłej i ikonę aparatu fotograficznego, ponieważ aktywna transakcja (niepokazana) przechowuje spójną migawkę tego punktu w historii bazy danych. Pozwól mi wyjaśnić to zdanie. Silnik InnoDB zapewnia obsługę czterech poziomów izolacji transakcji (*https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html*), przy czym tylko dwa z nich są powszechnie używane: `REPEATABLE READ` (domyślny) i `READ COMMITED`.

Na poziomie izolacji transakcji `REPEATABLE READ` pierwsza operacja odczytu powoduje utworzenie *spójnej migawki* (lub krócej *migawki*): wirtualny widok bazy danych (wszystkie tabele) w chwili wykonywania zapytania `SELECT`. Ta migawka jest przechowywana aż do zakończenia transakcji i używana przez wszystkie kolejne operacje odczytu w celu uzyskania dostępu do rekordów w danym punkcie historii bazy danych. Zmiany wprowadzone przez inne transakcje po tym punkcie nie będą widoczne w pierwotnej transakcji. Przyjmując założenie, że inne transakcje modyfikują bazę danych, migawka pierwotnej transakcji staje się coraz starszym widokiem bazy danych, gdy transakcja jest wciąż aktywna (brak wykonanego polecenia `COMMIT` lub `ROLLBACK`). To tak, jakby pierwotna transakcja zatrzymała się na latach osiemdziesiątych ubiegłego wieku i jedynymi muzykami, których można słuchać, byliby Pat Benatar, Stevie Nicks i Taylor Dayne: starzy, ale wciąż świetni.

Skoro wersja 5 przedstawia bieżącą postać rekordu, nowe transakcje tworzą migawkę w tym punkcie historii bazy danych. Dlatego wersja 5 na rysunku ma kontur w postaci linii ciągłej i ikonę aparatu fotograficznego. W tym miejscu może się zrodzić ważne pytanie: dlaczego wersje 2, 3 i 4 wciąż istnieją pomimo braku transakcji przechowujących migawki w tych punktach historii bazy danych? One istnieją z powodu migawki dla wersji 1, ponieważ MySQL używa dzienników przywracania do przywracania starszych wersji rekordu.

MySQL używa dzienników przywracania do przywracania starszych wersji rekordu dla migawki.

Bardzo łatwo można zrekonstruować sytuację pokazaną na rysunku 8.8. Przede wszystkim natychmiast po wstawieniu rekordu do tabeli trzeba rozpocząć transakcję i tym samym utworzyć migawkę wersji 1 rekordu. W tym celu wystarczy wykonać zapytanie `SELECT`:

```
BEGIN;

SELECT a FROM elem WHERE id = 2;

-- Zwrot rekordu w wersji 1: 'Fe'
```

Ponieważ nie zostało wykonane polecenie `COMMIT`, transakcja wciąż jest aktywna i przechowuje migawkę całej bazy danych. Dlatego w omawianym przykładzie wersją rekordu jest 1. Tę transakcję nazwiemy *transakcją pierwotną*.

Następnie czterokrotnie uaktualniamy rekord w celu otrzymania jego wersji 5:

```
-- Włączony tryb autocommit
UPDATE elem SET a = 'Ti' WHERE id = 2;
UPDATE elem SET a = 'Ag' WHERE id = 2;
UPDATE elem SET a = 'Cf' WHERE id = 2;
UPDATE elem SET a = 'Au' WHERE id = 2;
```

Tryb `autocommit` (*https://dev.mysql.com/doc/refman/8.0/en/innodb-autocommit-commit-rollback.html*) jest włączony domyślnie w MySQL i dlatego pierwsza (aktywna) transakcja wymaga wyraźnego użycia polecenia `BEGIN`, natomiast cztery kolejne zapytania `UPDATE` już nie. W tej chwili MySQL znajduje się w stanie pokazanym na rysunku 8.8.

Jeżeli w pierwotnej transakcji ponownie będzie wykonane zapytanie `SELECT a FROM elem WHERE id = 2`, nastąpi odczytanie wersji 5 (to nie pomyłka), ale transakcja „zobaczy", że ta wersja jest nowsza niż punkt w historii uchwycony przez migawkę. W efekcie MySQL skorzysta z dzienników przywracania w celu przywrócenia wersji 1 rekordu, która jest spójna z migawką utworzoną przez pierwsze zapytanie `SELECT`. Po zatwierdzeniu pierwotnej transakcji i przyjęciu założenia, że żadna inna aktywna transakcja nie korzysta ze starych migawek, MySQL może się pozbyć wszystkich powiązanych dzienników przywracania, ponieważ nowe transakcje zawsze rozpoczynają się od bieżącej wersji rekordu. Gdy transakcje działają świetnie, cały proces nie ma znaczenia z perspektywy wydajności działania. Jednak doskonale wiesz, że problematyczne transakcje mogą mieć negatywny wpływ na wydajność działania całego procesu — w dalszej części rozdziału wyjaśnię, dlaczego i jak tak się dzieje. Jednak wcześniej muszę przedstawić jeszcze nieco szczegółów dotyczących mechanizmu kontroli współbieżności i dzienników przywracania.

Na poziomie izolacji transakcji `READ COMMITTED` każda operacja odczytu powoduje utworzenie nowej migawki. W efekcie każda taka operacja ma dostęp do najnowszej zatwierdzonej wersji rekordu i stąd nazwa `READ COMMITTED`. Skoro migawki są używane, dzienniki przywracania wciąż są tworzone, ale to nie stanowi problemu na tym poziomie izolacji transakcji, ponieważ poszczególne migawki są przechowywane jedynie przez czas trwania operacji odczytu. Jeżeli odczyt trwa

bardzo długo *i* baza danych oferuje dużą przepustowość zapisu, będzie można zauważyć nagromadzenie dzienników przywracania (jako wzrost długości listy historii). W przeciwnym razie poziom izolacji transakcji `READ COMMITTED` praktycznie nie prowadzi do gromadzenia dzienników przywracania.

Migawki mają wpływ jedynie ma operacje odczytu (`SELECT`) i nigdy nie są używane podczas zapisu. Operacja zapisu zawsze po cichu odczytuje bieżące rekordy, nawet jeśli transakcja nie może ich „zobaczyć" za pomocą zapytania `SELECT`. Taka podwójna wizja zapobiega chaosowi. Na przykład wyobraź sobie inną transakcję wstawiającą nowy rekord z kluczem podstawowym o wartości 11. Jeżeli pierwotna transakcja spróbuje wstawić rekord z taką samą wartością klucza podstawowego, wówczas MySQL zgłosi błąd powielonego klucza, ponieważ wartość klucza podstawowego istnieje nawet pomimo tego, że transakcja jej nie widzi podczas wykonywania zapytania `SELECT`. Co więcej, migawki są bardzo spójne — transakcja nie ma możliwości przesunięcia migawki do nowszego punktu w historii bazy danych. Jeżeli aplikacja wykonująca transakcję wymaga nowszej migawki, musi zatwierdzić transakcję i rozpocząć nową, która utworzy nowszą migawkę.

Operacja zapisu powoduje wygenerowanie dzienników przywracania przechowywanych aż do zakończenia transakcji — niezależnie od poziomu izolacji transakcji. Dotychczas koncentrowałem się na dziennikach przywracania w odniesieniu do przywracania starych wersji rekordów dla migawek. Jednak tych dzienników używają również polecenia `ROLLBACK` do wycofywania zmian wprowadzonych przez operacje zapisu.

Jest jeszcze jedna kwestia dotycząca MVCC: dzienniki przywracania są zapisywane w puli bufora InnoDB. Jak zapewne pamiętasz z podpunktu „Opróżnianie stron" w rozdziale 6., tzw. „*różne strony* zawierają różne wewnętrzne struktury danych, nieomówione w tej książce". Te różne strony obejmują dzienniki przywracania (i wiele innych wewnętrznych struktur danych). Skoro dzienniki przywracania znajdują się na stronach puli bufora, wykorzystują pamięć i okresowo są zapisywane na dysku.

Kilka zmiennych systemowych i wskaźników jest powiązanych z dziennikami przywracania. Inżynier korzystający z MySQL musi znać i monitorować tylko jeden z nich, HLL, o którym pokrótce wspomniałem w rozdziale 6., a jego dokładne omówienie znajduje się w następnym podrozdziale. Poza tym mechanizm kontroli współbieżności i dzienniki przywracania działają bezproblemowo, o ile aplikacja unika najczęściej występujących problemów, które wymienię w dalszej części rozdziału. Jednym z takich problemów są porzucone transakcje. Dlatego teraz go unikniemy przez zatwierdzenie pierwotnej transakcji:

```
COMMIT;
```

Żegnaj, spójna migawko. Żegnajcie, dzienniki przywracania. Witaj, wielkości listy historii…

Wielkość listy historii

Wielkość listy historii (ang. *history list length*, HLL) podaje ilość starych wersji rekordu, które nie zostały usunięte.

Historycznie (to niezamierzona gra słów) wartość HLL była trudna do zdefiniowania z powodu skomplikowania pełnej struktury wycofywania wprowadzonych zmian:

```
Segmenty wycofywania
└── Sloty przywracania
    └── Segmenty dziennika przywracania
        └── Dzienniki przywracania
            └── Rekordy dzienników przywracania
```

Ten poziom skomplikowania zaciemnia proste relacje zachodzące między dziennikami przywracania i HLL, włączając w to także jednostkę pomiaru. Najprostszą funkcjonalną (choć z technicznego punktu widzenia niezbyt prawidłową) jednostką HLL są *zmiany*. Jeżeli wartość HLL wynosi 10 000, można to odczytać jako 10 000 zmian. Dzięki lekturze poprzedniego podrozdziału wiesz, że zmiany są przechowywane (nie usunięte) w pamięci (nie zapisane na dysku) w celu zapewnienia możliwości przywracania starych wersji rekordu. Dlatego można powiedzieć, że HLL podaje liczbę starych wersji rekordu, które nie zostały usunięte ani zapisane na dysku.

Większa niż 100 000 wartość HLL stanowi problem, którego nie należy ignorować. Mimo że prawdziwa techniczna natura HLL jest nieuchwytna — nawet dla ekspertów od MySQL — jej użyteczność pozostaje jasna i niezaprzeczalna: HLL to zwiastun problemów dotyczących transakcji. Zawsze należy monitorować wartość HLL (zajrzyj do podpunktu „Wskaźnik HLL" w rozdziale 6.), ostrzegać o jego zbyt wysokim poziomie (ponad 100 000) oraz rozwiązywać problem, ponieważ bez wątpienia to jest jeden z najczęściej występujących problemów, który omówię w następnym podrozdziale.

Wprawdzie jestem przeciwnikiem generowania ostrzeżeń po przekroczeniu wartości progowych (zobacz punkt „Szukanie wiatru w polu (wartości progowe)" w rozdziale 6.), ale wartość HLL jest tutaj wyjątkiem. Informowanie o przekroczeniu wartości 100 000 dla HLL jest dobrym i wykonalnym zadaniem.

Generuj komunikat ostrzeżenia, gdy wartość HLL przekroczy 100 000.

Teoretycznie HLL ma wartość maksymalną, ale wydajność działania MySQL znacznie spadnie jeszcze na długo przed osiągnięciem tej wartości maksymalnej[3]. Na przykład kilka tygodni temu miałem do czynienia z pewną sytuacją. Egzemplarz MySQL działający w chmurze uległ awarii z wartością HLL wynoszącą 200 000 — zgromadzenie takiej listy historii zajęło długo wykonywanej transakcji cztery godziny, po których egzemplarz MySQL uległ awarii i doprowadził do dwugodzinnego przestoju.

Ponieważ rejestrowanie danych na potrzeby przywracania jest niewiarygodnie efektywne, istnieje dość duża swoboda w zakresie wartości HLL, po której przekroczeniu wydajność działania MySQL zacznie spadać lub — w najgorszym wypadku — ulegnie awarii. Spotkałem się z egzemplarzem MySQL ulegającym awarii przy wartości HLL wynoszącej 200 000, a także widziałem egzemplarze działające świetnie z wartościami HLL znacznie przekraczającymi 200 000. Jedno jest bezsprzeczne, a mianowicie jeżeli wartość HLL będzie rosła niezauważona, *doprowadzi* do problemu: zauważalnego spadku wydajności działania lub awarii MySQL.

[3] W pliku *storage/innobase/trx/trx0purge.cc* kodu źródłowego MySQL 8.0 blok debugowania generuje komunikat ostrzeżenia, gdy wartość HLL jest większa niż 2 000 000.

Mam nadzieję, że staniesz się pierwszym w historii inżynierem, który używa serwera MySQL i nie ma problemu z wartością HLL. Wprawdzie to ambitny cel, ale zachęcam Cię do podjęcia próby jego osiągnięcia. Z tego powodu celowo zalałem egzemplarz MySQL powodzią zapytań `UPDATE` i tym samym doprowadziłem do wzrostu wartości HLL — zgromadzone zostały tysiące starych wersji rekordu. W tabeli 8.2 przedstawiłem wpływ wartości HLL na czas udzielenia odpowiedzi na zapytanie w aktywnej transakcji wykorzystującej poziom izolacji transakcji `REPEATABLE READ`.

Tabela 8.2. Wpływ wartości HLL na czas udzielenia odpowiedzi na zapytanie

Wartość HLL	Czas odpowiedzi (ms)	Wzrost względem wartości bazowej (%)
0	0,200 ms	
495	0,612 ms	206%
1089	1,012 ms	406%
2079	1,841 ms	821%
5056	3,673 ms	1737%
11546	8,527 ms	4164%

Ten przykład *nie* oznacza, że wartość HLL będzie wydłużała czas udzielenia odpowiedzi na zapytanie o przedstawione tutaj wartości. Tabela jedynie potwierdza, że wartość HLL może wydłużyć czas udzielenia odpowiedzi na zapytanie. Na podstawie informacji zamieszczonych wcześniej w tym rozdziale wiesz, dlaczego tak się dzieje: zapytanie `SELECT` w aktywnej transakcji na poziomie izolacji `REPEATABLE READ` ma spójną migawkę rekordu 5 (`id=5`), natomiast zapytania `UPDATE` dla tego rekordu powodują wygenerowanie nowych wersji rekordu. W trakcie każdego wykonania zapytania `SELECT` podejmowany jest trud przedarcia się przez dzienniki przywracania w celu przywrócenia pierwotnej wersji dla spójnej migawki. To zadanie jest coraz trudniejsze i prowadzi do systematycznego wydłużania czasu udzielenia odpowiedzi na zapytanie.

Wydłużający się czas udzielenia odpowiedzi na zapytanie jest wystarczającym dowodem. Jednak jesteśmy profesjonalistami, więc udowodnimy to niezbicie. Pod koniec poprzedniego podrozdziału wspomniałem, że dzienniki przywracania są przechowywane w postaci stron puli bufora InnoDB. W efekcie zapytanie `SELECT` powinno mieć dostęp do nadmiernej liczby stron. Aby to udowodnić, posłużę się dystrybucją Percona Server (*https://www.percona.com/software/mysql-database/percona-server*), ponieważ jej rozbudowane dane wyjściowe dziennika zdarzeń wolno wykonywanych zapytań podają liczbę odmiennych stron, do których zapytanie uzyskało dostęp (o ile serwer został skonfigurowany z użyciem zmiennej `log_slow_verbosity = innodb`):

```
# Query_time: 0.008527
# InnoDB_pages_distinct: 366
```

Normalnie zapytanie `SELECT` w tym przykładzie uzyskuje dostęp do pojedynczej strony w celu wyszukania jednego rekordu za pomocą klucza podstawowego. Jednak jeśli spójna migawka dla zapytania `SELECT` jest stara (a wartość HLL duża), wówczas silnik InnoDB musi przedzierać się przez setki stron dzienników przywracania, aby przywrócić rekord do starej wersji.

Mechanizm kontroli współbieżności, dzienniki przywracania i wartość HLL to normalny i dobry kompromis: odrobina wydajności działania za dużą współbieżność. Tylko jeśli wartość HLL jest naprawdę ogromna — więcej niż 100 000 — należy podjąć działanie i usunąć przyczynę wzrostu tej wartości. Gwałtowny wzrost wartości HLL jest jednym z najczęściej występujących problemów.

Najczęściej pojawiające się problemy

Problemy związane z transakcjami mają źródło w zapytaniach tworzących te transakcje oraz wynikają z tego, jak szybko aplikacja wykonuje te zapytania i zatwierdza transakcje. Wprawdzie pojedyncza transakcja działająca w trybie `autocommit` (*https://dev.mysql.com/doc/refman/8.0/en/innodb-autocommit-commit-rollback.html*) jest pod względem technicznym transakcją, która może spowodować problemy omówione w tym podrozdziale (wyjątkiem jest problem omówiony w punkcie „Transakcje porzucone"), ale skoncentruję się na składających się z wielu poleceń transakcjach — rozpoczynających się od słowa kluczowego `BEGIN` (lub `START TRANSACTION`), wykonujących wiele zapytań i kończących się słowem kluczowym `COMMIT` (lub `ROLLBACK`). Wpływ takiej transakcji na wydajność działania może być większy niż suma wpływu jej poszczególnych elementów — czyli zapytań tworzących transakcję — ponieważ blokady i dzienniki przywracania są przechowywane aż do chwili zatwierdzenia (lub wycofania) transakcji. Pamiętaj: serwer MySQL jest bardzo cierpliwy — nawet zbyt cierpliwy. Jeżeli aplikacja nie zatwierdzi transakcji, MySQL będzie czekać, nawet jeśli konsekwencje aktywnej transakcji okażą się katastrofalne.

Na szczęście żaden z tych problemów nie jest trudny do wykrycia lub usunięcia. Wartość HLL jest zwiastunem większości problemów związanych z transakcją i dlatego należy monitorować HLL (więcej informacji na ten temat znajdziesz w punkcie „Wskaźnik HLL" w rozdziale 6. i podrozdziale „Wielkość listy historii" we wcześniejszej części tego rozdziału). Aby niepotrzebnie nie zaśmiecać wyjaśnień poszczególnych problemów, informacje dotyczące wyszukiwania i zgłaszania problematycznych transakcji znajdziesz w podrozdziale „Zgłaszanie problemów" w dalszej części rozdziału.

Ogromne transakcje (wielkość transakcji)

Ogromna transakcja modyfikuje *nadmierną* liczbę rekordów. Ile rekordów to wielkość *nadmierna*? To jest wartość względna, a inżynierowie zawsze ją poznają. Na przykład jeśli transakcja zmodyfikowała 250 000 rekordów i wiesz, że w całej bazie danych znajduje się tylko 500 000 rekordów, wówczas mamy do czynienia z nadmierną liczbą zmodyfikowanych rekordów. (Ewentualnie jest to przynajmniej podejrzany wzorzec dostępu — zobacz punkt „Zbiór wynikowy" w rozdziale 4.).

Ogólnie rzecz biorąc, *wielkość transakcji* odwołuje się do liczby zmodyfikowanych rekordów: im ich więcej, tym większa transakcja. W przypadku grup replikacji MySQL (*https://dev.mysql.com/doc/refman/8.0/en/group-replication.html*) *wielkość transakcji* ma nieco odmienne znaczenie — zapoznaj się z sekcją *Group Replication Limitations* podręcznika użytkownika (*https://dev.mysql.com/doc/refman/8.0/en/group-replication-limitations.html*).

Jeżeli transakcja działa na domyślnym poziomie izolacji transakcji, `REPEATABLE READ`, można bezpiecznie założyć, że nałożyła blokady na więcej rekordów, niż zostało zmodyfikowanych, co wynika z istnienia blokad luk, zgodnie z informacjami zamieszczonymi w punkcie „Blokady luk". Jeżeli transakcja działa na poziomie izolacji transakcji `READ COMMITTED`, nakłada blokady jedynie na modyfikowane rekordy. W każdym razie ogromna transakcja oznacza ogromne źródło rywalizacji blokad, co może doprowadzić do degradacji przepustowości zapisu i czasu udzielenia odpowiedzi.

Nie zapominaj o replikacji (zobacz rozdział 7.): ogromne transakcje są podstawowym źródłem opóźnienia replikacji (zobacz punkt „Przepustowość transakcji" w rozdziale 7.) i zmniejszenia efektywności replikacji wielowątkowej (zobacz punkt „Zmniejszenie opóźnienia replikacji — replikacja wielowątkowa" w rozdziale 7.).

Ogromne transakcje mogą być bardzo wolno zatwierdzane (lub wycofywane), jak to dokładniej omówiłem wcześniej w tym rozdziale, a także w punkcie „Zdarzenia binarnego dziennika zdarzeń" w rozdziale 7. i na rysunku 6.7 w rozdziale 6. Bardzo łatwo i szybko można zmodyfikować rekordy, ponieważ zmiany danych zachodzą w pamięci. Natomiast zatwierdzenie jest bardziej wymagające, gdyż MySQL wykonuje całkiem sporo pracy w trakcie trwałego zapisywania danych i ich replikacji.

Mniejsze transakcje są lepsze. O ile mniejsze? To również jest względne i skomplikowane do ustalenia, ponieważ jak już wcześniej wspomniałem, transakcje są rozliczane podczas zatwierdzania, co oznacza konieczność kalibracji wielu podsystemów. (Staje się to jeszcze bardziej skomplikowane, gdy trzeba uwzględnić chmurę, która przeważnie ogranicza możliwość dostosowania szczegółów do własnych potrzeb, takich jak IOPS). Z wyjątkiem operacji hurtowych wymagających kalibracji wielkości partii (zobacz punkt „Wielkość operacji hurtowej" w rozdziale 3.) kalibracja wielkości transakcji nie jest zwykle konieczna, ponieważ wprawdzie problem jest typowy, ale najczęściej jednorazowy: znaleziony, usunięty i nie pojawia się ponownie (przynajmniej przez jakiś czas). Z dalszej części rozdziału dowiesz się, jak wyszukiwać ogromne transakcje.

Rozwiązaniem jest znalezienie zapytania (lub zapytań) w transakcji modyfikującego zbyt wiele rekordów, a następnie zmiana go w taki sposób, aby modyfikował mniej rekordów. To jednak zależy w całości od zapytania, przeznaczenia aplikacji i powodu modyfikowania zbyt wielu rekordów. Niezależnie od powodu informacje przedstawione w rozdziałach od 1. do 4. pozwolą Ci zrozumieć zapytanie i je poprawić.

Jeżeli ściśle stosujesz się do reguły najmniejszej ilości danych (zobacz punkt „Im mniej danych, tym lepiej" w rozdziale 3.), to wielkość transakcji może nigdy nie stanowić problemu.

Długo wykonywane transakcje

Długo wykonywana transakcja wymaga zbyt wiele czasu na jej ukończenie (zatwierdzenie lub wycofanie). *Zbyt wiele*, czyli ile? To zależy:

- dłużej, niż wynosi czas akceptowalny dla aplikacji lub jej użytkowników;
- wystarczająco długo, aby powodować problemy (prawdopodobnie rywalizację) z innymi transakcjami;
- wystarczająco długo, aby spowodować wygenerowanie komunikatu ostrzeżenia dotyczącego wielkości listy historii.

O ile aktywnie nie zajmujesz się rozwiązywaniem problemów z wydajnością działania, punkty drugi i trzeci prawdopodobnie zwrócą Twoją uwagę na długo wykonywane transakcje.

Zakładając, że aplikacja nie czeka między zapytaniami (co jest problemem omówionym w następnym punkcie), długo wykonywane transakcje mają dwa powody:

- zapytania tworzące daną transakcję są zbyt wolne;
- aplikacja wykonuje zbyt wiele zapytań w transakcji.

Pierwszy powód można wyeliminować za pomocą technik omówionych w pierwszych pięciu rozdziałach książki. Pamiętaj: dzienniki przywracania i blokady rekordów dla wszystkich zapytań w transakcji pozostają aż do zatwierdzenia transakcji. Z drugiej strony to oznacza, że optymalizacja wolno wykonywanych zapytań w celu wyeliminowania problemu długo wykonywanych transakcji przynosi dodatkowe korzyści: poszczególne zapytania stają się szybsze *i* transakcja jako całość staje się szybsza, co może poprawić ogólną przepustowość transakcji. Natomiast wadą jest to, że długo wykonywana transakcja może być wystarczająco szybka dla aplikacji, ale zbyt wolna dla innych transakcji. Na przykład załóżmy, że wykonanie transakcji zajmuje 1 sekundę, co nie stanowi problemu dla aplikacji. Jednak w trakcie tej sekundy zostaje nałożona blokada na rekordy wymagane przez inną, szybszą transakcję. To prowadzi do problemu trudnego do wykrycia podczas debugowania, ponieważ szybka transakcja może być wykonywana wolno w środowisku produkcyjnym, a szybko w izolacji, podczas jej analizowania w laboratorium (np. w Twoim laptopie). Różnica polega na tym, że współbieżność i rywalizacja transakcji w środowisku produkcyjnym prawdopodobnie nie będzie istniała w środowisku testowym. W takim przypadku trzeba debugować blokady danych, co z wielu powodów nie będzie łatwe, a najłatwiejszy z nich to ulotność blokad danych. Zapoznaj się z informacjami dotyczącymi tabeli 8.1 oraz skonsultuj się z administratorem baz danych lub ekspertem od MySQL.

Drugi powód można wyeliminować przez modyfikację aplikacji w celu wykonywania mniejszej liczby zapytań w transakcji. Ten problem pojawia się, gdy aplikacja ma wykonać operację hurtową lub bez żadnych ograniczeń programowo generuje zapytania w transakcji. Rozwiązaniem jest zmniejszenie lub ograniczenie liczby zapytań w transakcji. Nawet jeśli transakcja nie jest długo wykonywana, i tak będzie to najlepszą praktyką, ponieważ gwarantuje, że transakcja nie stanie się długo wykonywaną. Na przykład być może na początku aplikacja wstawia jedynie 5 rekordów w ramach transakcji, a kilka lat później, gdy aplikacja zdobyła miliony użytkowników, wstawiane jest już 500 rekordów w trakcie transakcji z powodu braku wbudowanych ograniczeń.

W podrozdziale „Zgłaszanie problemów” zobaczysz, jak wyszukiwać długo wykonywane transakcje.

Transakcje przeciągające się

Przeciągająca się transakcja czeka zbyt długo po wykonaniu polecenia `BEGIN`, między zapytaniami lub przed wykonaniem polecenia `COMMIT`. Przeciągająca się transakcja prawdopodobnie stanie się długo wykonywaną, choć przyczyny jej powstania są inne: oczekiwanie między zapytaniami (przedłużające się) zamiast oczekiwania na zapytania (długo wykonywane).

W praktyce przeciągająca się transakcja jawi się jako długo wykonywana, ponieważ efekt końcowy jest dokładnie taki sam: długi czas udzielenia odpowiedzi na transakcję. Konieczna jest analiza transakcji w celu ustalenia, czy długi czas udzielenia odpowiedzi wynika z przeciągającej się transakcji czy z wolno wykonywanych zapytań. Inżynierowie (i eksperci od MySQL), którzy nie przeprowadzają takiej analizy, często każdą wolną transakcję określają jako długo wykonywaną.

Nie ulega wątpliwości, że między zapytaniami zawsze będzie jakiś czas oczekiwania (przynajmniej ze względu na opóźnienie sieci związane z wysłaniem zapytania i otrzymaniem wyniku na nie), ale podobnie jak w dwóch poprzednich problemach, gdy zobaczysz transakcję, to wiesz, że jest przeciągająca się. Mówiąc w przenośni: całość będzie znacznie większa niż tworzące ją elementy. Natomiast ujmując to w sposób techniczny: czas udzielenia odpowiedzi na transakcję od polecenia `BEGIN` do `COMMIT` jest znacznie dłuższy niż czas udzielenia odpowiedzi na poszczególne zapytania składające się na tę transakcję.

Ponieważ przeciągające się transakcje oznaczają oczekiwanie *między* zapytaniami (po poleceniu `BEGIN` i przed poleceniem `COMMIT`), serwer MySQL nie jest tutaj winny: oczekiwanie jest spowodowane przez aplikację, a powodów jest niezliczona ilość. Dość częstą przyczyną jest wykonywanie przez aplikację czasochłonnej logiki, gdy transakcja jest aktywna, zamiast przed transakcją lub po niej. Jednak czasami nie można tego uniknąć. Spójrz na przedstawiony tutaj przykład:

```
BEGIN;
SELECT <rekord>
--
-- Czasochłonna logika aplikacji na podstawie rekordu
--
UPDATE <rekord>
COMMIT;
```

W takim przypadku rozwiązanie zależy od logiki aplikacji. Najlepiej zacząć od zadania sobie najbardziej podstawowego pytania: czy te zapytania muszą być transakcją? Czy rekord może ulec zmianie po jego odczytaniu, ale przed uaktualnieniem? Jeżeli rekord ulegnie zmianie, to czy to może doprowadzić do uszkodzenia logiki? Jeżeli nic innego nie pomaga, czy można wykorzystać poziom izolacji transakcji `READ COMMITTED` w celu wyłączenia blokad luk? Inżynierowie są sprytni i potrafią znaleźć rozwiązania dla tego typu problemów. Pierwszym krokiem jest znalezienie problemu, co dokładniej omówię w dalszej części rozdziału.

Transakcje porzucone

Transakcja porzucona to transakcja aktywna, ale bez aktywnego połączenia z klientem. Mamy dwie podstawowe przyczyny istnienia takich transakcji:

- wyciek połączenia aplikacji;
- półzamknięte połączenia.

Błąd w aplikacji może doprowadzić do wycieku połączeń bazy danych (podobnie jak w przypadku pamięci lub wątków): działający na poziomie kodu obiekt połączenia opuszcza zakres, więc nie jest już używany, choć odwołanie do niego nadal znajduje się w kodzie. W takim przypadku

obiekt ani nie jest zamknięty, ani nie zwolnił pamięci (prawdopodobnie powoduje również mały wyciek pamięci). Poza profilowaniem na poziomie aplikacji, debugowaniem lub wykrywaniem wycieku w celu bezpośredniego zweryfikowania błędu możesz go potwierdzić także pośrednio, jeśli ponowne uruchomienie aplikacji rozwiąże problem porzuconych transakcji (zostaną zakończone). MySQL można sprawdzić pod kątem niebezpieczeństwa istnienia porzuconych transakcji (dokładnie wyjaśniłem to w podrozdziale „Zgłaszanie problemów"), ale nie można potwierdzić tego błędu w MySQL, ponieważ serwer nie wie, czy dane połączenie zostało porzucone.

Półzamknięte połączenie nie zdarzą się w normalnej sytuacji, ponieważ MySQL wycofuje transakcję, gdy klient zamknie połączenie z jakiegokolwiek powodu, leżącego po stronie MySQL lub systemu operacyjnego. Jednak problemy występujące poza serwerem MySQL i systemem operacyjnym mogą doprowadzić do sytuacji, w której klient zamknie połączenie, a MySQL nie — wówczas mamy do czynienia z tzw. *półzamkniętym* połączeniem. MySQL jest szczególnie podatny na półzamknięte połączenia, ponieważ jego protokół sieciowy jest niemal w całości poleceniem i odpowiedzią: klient przekazuje polecenie, MySQL zaś udziela odpowiedzi. (Jeżeli chcesz wiedzieć, klient przekazuje zapytanie do serwera MySQL za pomocą pakietu `COM_QUERY` (*https://dev.mysql.com/doc/internals/en/com-query.html*)). Między poleceniem i odpowiedzią klient i MySQL pozostają w całkowitej ciszy — nie zostaje przekazany ani jeden bajt. Wprawdzie brzmi to dobrze, ale jednocześnie oznacza, że półzamknięte połączenia pozostają niezauważone, aż do upłynięcia `wait_timeout` (*https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_wait_timeout*) sekund — domyślnie jest to 28 800 (8 godzin).

Niezależnie od tego, czy błąd aplikacji prowadzi do wycieku połączenia, czy półzamknięte połączenie zostało przeoczone, jeśli to miało miejsce w trakcie aktywnej transakcji (nie zatwierdzonej), wynik końcowy zawsze jest ten sam: transakcja pozostaje aktywna. Każda spójna migawka lub blokada danych również pozostaje aktywna, ponieważ MySQL nie wie, że transakcja została porzucona.

Prawdę mówiąc, MySQL lubi ciszę, podobnie jak ja. Ale za ciszę mi nie płacą, więc w następnym podrozdziale wyjaśnię, jak można wyszukiwać i zgłaszać wszystkie cztery omówione problemy związane z transakcjami.

Zgłaszanie problemów

Funkcjonalność Performance Schema (*https://dev.mysql.com/doc/refman/8.0/en/performance-schema.html*) w MySQL umożliwia zgłaszanie dokładnych informacji o transakcjach. W chwili powstawania tej książki nie było żadnego narzędzia, które by to ułatwiało. Żałuję, że nie mogę powiedzieć Ci czegoś o istniejących narzędziach open source przeznaczonych do tego celu — po prostu ich nie ma. Zamieszczone tutaj polecenia SQL są zgodne z obecnym stanem wiedzy. Gdy pojawią się nowe technologie w tym zakresie, poinformuję Cię o tym na stronie MySQL Transaction Reporting (*https://hackmysql.com/mysql-transaction-reporting/*). Zanim to nastąpi, pracę trzeba wykonać w staroświecki sposób, czyli za pomocą techniki *kopiuj i wklej*.

Aktywne transakcje — najnowsze

Polecenie SQL zamieszczone na listingu 8.2 powoduje wygenerowanie raportu dotyczącego ostatniego zapytania dla wszystkich aktywnych transakcji trwających dłużej niż 1 sekundę. Taki raport pozwala odpowiedzieć na pytanie: które transakcje są długo wykonywane i co tak dokładnie w tym momencie robią?

Listing 8.2. Raport dotyczący ostatniego zapytania dla transakcji wykonywanych dłużej niż 1 sekundę

```
SELECT
  ROUND(trx.timer_wait/1000000000000,3) AS trx_runtime,
  trx.thread_id AS thread_id,
  trx.event_id AS trx_event_id,
  trx.isolation_level,
  trx.autocommit,
  stm.current_schema AS db,
  stm.sql_text AS query,
  stm.rows_examined AS rows_examined,
  stm.rows_affected AS rows_affected,
  stm.rows_sent AS rows_sent,
  IF(stm.end_event_id IS NULL, 'running', 'done') AS exec_state,
  ROUND(stm.timer_wait/1000000000000,3) AS exec_time
FROM
       performance_schema.events_transactions_current trx
  JOIN performance_schema.events_statements_current   stm USING (thread_id)
WHERE
      trx.state = 'ACTIVE'
  AND trx.timer_wait > 1000000000000 * 1\G
```

W celu zwiększenia czasu należy zmienić wartość 1 przed `\G`. Zegary funkcjonalności Performance Schema używają pikosekund, więc zapis `1000000000000 * 1` oznacza 1 sekundę.

Dane wyjściowe polecenia z listingu 8.2 przedstawiają się następująco:

```
*************************** 1. row ***************************
    trx_runtime: 20729.094
      thread_id: 60
   trx_event_id: 1137
isolation_level: REPEATABLE READ
     autocommit: NO
             db: test
          query: SELECT * FROM elem
  rows_examined: 10
  rows_affected: 0
      rows_sent: 10
     exec_state: done
      exec_time: 0.038
```

Oto nieco więcej informacji o poszczególnych polach (kolumnach) danych wyjściowych z listingu 8.2:

`trx_runtime`

Wyrażony w sekundach z dokładnością do milisekund czas wykonywania transakcji (aktywnej). W omawianym przykładzie zapomniałem o tej transakcji i dlatego pozostała aktywna przez prawie 6 godzin.

`thread_id`

Identyfikator wątku połączenia klienta wykonującego transakcję. Ta wartość zostanie wykorzystana w punkcie „Aktywna transakcja — historia". Zdarzenia funkcjonalności Performance Schema używają identyfikatorów wątków i zdarzeń w celu powiązania danych, odpowiednio, z połączeniami klienta i zdarzeniami. Identyfikatory wątków są inne niż identyfikatory procesów w komponentach MySQL.

`trx_event_id`

Identyfikator zdarzenia transakcji. Ta wartość zostanie wykorzystana w punkcie „Aktywna transakcja — historia".

`isolation_level`

Poziom izolacji transakcji, `REPEATABLE READ` lub `READ COMMITTED`. (Pozostałe poziomy izolacji transakcji, `SERIALIZABLE` i `READ UNCOMMITTED`, są rzadko używane. Jeżeli widzisz je tutaj wymienione, może to być skutkiem błędu w aplikacji). Przypomnij sobie podrozdział „Nakładanie blokad na rekordy": poziom izolacji transakcji wpływa na blokady rekordów oraz na to, czy zapytanie `SELECT` będzie używało spójnej migawki.

`autocommit`

Jeżeli wartością jest `YES`, wówczas tryb `autocommit` został włączony i to jest transakcja w postaci pojedynczego polecenia. Natomiast wartość `NO` oznacza, że transakcja rozpoczęła się od słowa kluczowego `BEGIN` (lub `START TRANSACTION`) i prawdopodobnie jest transakcją składającą się z wielu poleceń.

`db`

Bieżąca baza danych zapytania (`query`). Bieżąca baza danych oznacza polecenie `USE db`. Zapytanie może uzyskiwać dostęp do innych baz danych dzięki używaniu nazw tabel zawierających także nazwy baz danych, np. `db.table`.

`query`

Ostatnie zapytanie wykonane bądź wykonywane przez transakcję. Jeżeli `exec_state = running`, zapytanie jest aktualnie wykonywane przez transakcję. Z kolei wartość `exec_state = done` określa, że `query` to ostatnie zapytanie wykonane przez transakcję. W obu przypadkach transakcja jest aktywna (nie została zatwierdzona), przy czym ten drugi przypadek oznacza bezczynność transakcji w odniesieniu do wykonywania zapytania.

`rows_examined`

Całkowita liczba rekordów przeanalizowanych przez zapytanie. Nie obejmuje to wcześniejszych zapytań wykonanych przez transakcję.

`rows_affected`

Całkowita liczba rekordów zmodyfikowanych przez zapytanie. Nie obejmuje to wcześniejszych zapytań wykonanych przez transakcję.

`rows_sent`

Całkowita liczba rekordów przekazanych przez zapytanie (zbiór wynikowy). Nie obejmuje to wcześniejszych zapytań wykonanych przez transakcję.

`exec_state`

Wartość `done` oznacza, że transakcja jest bezczynna w odniesieniu do wykonywania zapytania, a ostatnio wykonane przez transakcję zapytanie zostało podane w polu `query`. Natomiast wartość `running` oznacza, że transakcja jest w trakcie wykonywania zapytania. W obu przypadkach transakcja jest aktywna (nie została zatwierdzona).

`exec_time`

Wyrażony w sekundach (z dokładnością do milisekund) czas wykonywania zapytania (`query`).

Tabele funkcjonalności Performance Schema `events_transactions_current` i `events_statements_current` zawierają więcej pól, w omawianym przykładzie przedstawiłem jedynie najważniejsze.

Omawiane zgłoszenie jest bardzo cenne, ponieważ może ujawnić wszystkie cztery najczęściej występujące problemy z transakcjami (zobacz podrozdział „Najczęściej pojawiające się problemy"):

Ogromne transakcje

Sprawdź wartości pól `rows_affected` (rekordy zmodyfikowane) i `rows_sent` (rekordy przekazane), aby poznać wielkość transakcji wyrażoną w rekordach. Poeksperymentuj przed dodanie warunków takich jak `trx.rows_affected > 1000`.

Długo wykonywane transakcje

Dostosuj wartość `1` na końcu warunku `trx.timer_wait > 1000000000000 * 1` w celu filtrowania długo wykonywanych zapytań.

Transakcje przeciągające się

Jeżeli wartość `exec_state = done` pozostaje przez jakiś czas, transakcja się przeciąga. Skoro zgłoszenie wyświetla jedynie ostatnie zapytanie aktywnych transakcji, to zapytanie powinno szybko zostać zmienione, a wartość `exec_state = done` powinna zniknąć.

Transakcje porzucone

Jeżeli wartość `exec_state = done` utrzymuje się przez dłuższy czas, to istnieje niebezpieczeństwo, że transakcja została porzucona, ponieważ nie jest zgłaszana po zatwierdzeniu.

Dane wyjściowe takiego zgłoszenia powinny być ulotne, ponieważ aktywne transakcje powinny być kończone. Jeżeli zgłoszenie informuje o transakcji wielokrotnie, wówczas prawdopodobnie wystąpił w niej jeden z najczęstszych problemów (zobacz podrozdział „Najczęściej pojawiające się problemy"). W takiej sytuacji skorzystaj z wartości `thread_id` i `statement_event_id` (zobacz punkt „Aktywna transakcja — historia") w celu sprawdzenia historii (wcześniejszych zapytań), co może pomóc w ustaleniu problemu z transakcją.

Aktywne transakcje — podsumowanie

Polecenie SQL zamieszczone na listingu 8.3 powoduje wygenerowanie podsumowania zapytań wykonywanych przez wszystkie transakcje aktywne dłużej niż 1 sekundę. Ten raport pozwala odpowiedzieć na pytanie: które transakcje są długo wykonywane i jak wiele zadań muszą one wykonywać?

Tabela Information Schema INNODB_TRX

Używanie tabel funkcjonalności Performance Schema to najlepsza praktyka i przyszłość zgłaszania informacji dotyczących wydajności działania MySQL. Jednak tabele funkcjonalności Information Schema (*https://dev.mysql.com/doc/refman/8.0/en/information-schema.html*) w MySQL nadal są powszechnie stosowane i mogą dostarczać informacje o długo wykonywanych transakcjach — wystarczy wykonać zapytanie do tabeli `information_schema.innodb_trx`:

```
SELECT
  trx_mysql_thread_id AS process_id,
  trx_isolation_level,
  TIMEDIFF(NOW(), trx_started) AS trx_runtime,
  trx_state,
  trx_rows_locked,
  trx_rows_modified,
  trx_query AS query
FROM
  information_schema.innodb_trx
WHERE
  trx_started < CURRENT_TIME - INTERVAL 1 SECOND\G

*************************** 1. row ***************************
         process_id: 13
trx_isolation_level: REPEATABLE READ
        trx_runtime: 06:43:33
          trx_state: RUNNING
    trx_rows_locked: 4
  trx_rows_modified: 1
              query: NULL
```

W tym przykładzie wartością `query` jest `NULL`, ponieważ transakcja nie wykonuje żadnego zapytania. Jeżeli byłoby jakiekolwiek zapytanie, zostałoby podane w polu `query`.

Doradzam używanie tabel funkcjonalności Performance Schema, ponieważ zawierają one znacznie więcej informacji szczegółowych — praktycznie wszystko, co trzeba wiedzieć o działaniach zachodzących w MySQL. We wszystkich przykładach zamieszczonych w tej książce korzystałem z funkcjonalności Performance Schema, o ile było to możliwe. W rzadkich przypadkach pewne informacje wciąż są dostępne jedynie za pomocą funkcjonalności Information Schema.

Aby dowiedzieć się więcej na temat `information_schema.innodb_trx`, zapoznaj się z sekcją *The INFORMATION_SCHEMA INNODB_TRX Table* (*https://dev.mysql.com/doc/refman/8.0/en/information-schema-innodb-trx-table.html*) podręcznika użytkownika MySQL.

Listing 8.3. Raport zawierający podsumowanie transakcji

```
SELECT
  trx.thread_id AS thread_id,
  MAX(trx.event_id) AS trx_event_id,
  MAX(ROUND(trx.timer_wait/1000000000000,3)) AS trx_runtime,
  SUM(ROUND(stm.timer_wait/1000000000000,3)) AS exec_time,
  SUM(stm.rows_examined) AS rows_examined,
  SUM(stm.rows_affected) AS rows_affected,
  SUM(stm.rows_sent) AS rows_sent
```

```
FROM
       performance_schema.events_transactions_current trx
  JOIN performance_schema.events_statements_history   stm
    ON stm.thread_id = trx.thread_id AND stm.nesting_event_id = trx.event_id
WHERE
      stm.event_name LIKE 'statement/sql/%'
  AND trx.state = 'ACTIVE'
  AND trx.timer_wait > 1000000000000 * 1
GROUP BY trx.thread_id\G
```

W celu wydłużenia czasu należy zmienić wartość 1 znajdującą się przed `\G`. Pola danych wyjściowych są takie same jak omówione w poprzednim punkcie. Jednak ten raport agreguje wcześniejsze zapytania poszczególnych transakcji. Transakcje przeciągające się (te, które aktualnie nie wykonują zapytania) mogły wcześniej wykonać wiele pracy, co zostanie ujawnione w tym raporcie.

Gdy zapytanie zakończy wykonywanie, zostanie zarejestrowane w `performance_schema.events_statements_history` i jednocześnie pozostanie w tabeli `performance_schema.events_statements_current`. Dlatego omawiany raport zawiera ukończone zapytania i nieumieszczane w tej drugiej tabeli, dopóki aktywne zapytania nie będą odfiltrowane.

Ten raport lepiej nadaje się do wyszukiwania ogromnych transakcji (zobacz punkt „Ogromne transakcje (wielkość transakcji)”), ponieważ zawiera informacje o wcześniej wykonanych zapytaniach.

Aktywna transakcja — historia

Polecenie SQL zamieszczone na listingu 8.4 powoduje wygenerowanie historii zapytań wykonywanych przez pojedynczą transakcję. Ten raport pozwala odpowiedzieć na pytanie: ile zadań zostało wykonanych przez poszczególne transakcje? Zera należy zastąpić wartościami `thread_id` i `trx_event_id` pochodzącymi z danych wyjściowych wygenerowanych przez polecenie z listingu 8.2.

Listing 8.4. Raport zawierający historię transakcji

```
SELECT
  stm.rows_examined AS rows_examined,
  stm.rows_affected AS rows_affected,
  stm.rows_sent AS rows_sent,
  ROUND(stm.timer_wait/1000000000000,3) AS exec_time,
  stm.sql_text AS query
FROM
  performance_schema.events_statements_history stm
WHERE
       stm.thread_id = 0
  AND  stm.nesting_event_id = 0
ORDER BY stm.event_id;
```

Nie zapomnij o zastąpieniu zer wartościami pochodzącymi z danych wyjściowych wygenerowanych przez polecenie na listingu 8.2:

- Zero w `stm.thread_id = 0` zastąp wartością `thread_id`.
- Zero w `stm.nesting_event_id = 0` zastąp wartością `trx_event_id`.

Dane wyjściowe polecenia z listingu 8.4 przedstawiają się następująco:

```
+---------------+--------------+-----------+-----------+--------------------+
| rows_examined | rows_affected | rows_sent | exec_time | query              |
+---------------+--------------+-----------+-----------+--------------------+
|            10 |             0 |        10 |     0.000 | SELECT * FROM elem  |
|             2 |             1 |         0 |     0.003 | UPDATE elem SET ... |
|             0 |             0 |         0 |     0.002 | COMMIT             |
+---------------+--------------+-----------+-----------+--------------------+
```

Pomijając polecenie `BEGIN` rozpoczynające transakcję, omawiana transakcja wykonała jeszcze dwa zapytania, a następnie polecenie `COMMIT`. Pierwszym zapytaniem było `SELECT`, drugim `UPDATE`. To nie jest przykład przykuwający uwagę, ale ma pokazać historię wykonywanych zapytań transakcji i podstawowe wskaźniki zapytania. Historia okazuje się nieoceniona podczas debugowania problematycznych transakcji, ponieważ pozwala sprawdzić, które zapytania są wolno wykonywane (`exex_time`) lub ogromne (pod względem liczby rekordów), a także punkt, w którym działanie aplikacji przeciąga się (gdy wiadomo, że transakcja będzie wykonywała więcej zapytań).

Transakcje zatwierdzone — podsumowanie

Trzy wcześniej omówione raporty dotyczyły transakcji aktywnych, ale zatwierdzone transakcje również są interesujące. Polecenie SQL na listingu 8.5 powoduje wygenerowanie raportu zawierającego podstawowe wskaźniki dotyczące transakcji zatwierdzonych (ukończonych). Można go potraktować jako dziennik zdarzeń wolno wykonywanych zapytań dla transakcji.

Listing 8.5. Raport zawierający podstawowe wskaźniki dotyczące zatwierdzonych transakcji

```
SELECT
  ROUND(MAX(trx.timer_wait)/1000000000,3) AS trx_time,
  ROUND(SUM(stm.timer_end-stm.timer_start)/1000000000,3) AS query_time,
  ROUND((MAX(trx.timer_wait)-SUM(stm.timer_end-stm.timer_start))/1000000000, 3)
    AS idle_time,
  COUNT(stm.event_id)-1 AS query_count,
  SUM(stm.rows_examined) AS rows_examined,
  SUM(stm.rows_affected) AS rows_affected,
  SUM(stm.rows_sent) AS rows_sent
FROM
      performance_schema.events_transactions_history trx
 JOIN performance_schema.events_statements_history   stm
   ON stm.nesting_event_id = trx.event_id
WHERE
      trx.state = 'COMMITTED'
  AND trx.nesting_event_id IS NOT NULL
GROUP BY
  trx.thread_id, trx.event_id;
```

Zapytanie przedstawione na listingu 8.5 powoduje wygenerowanie danych wyjściowych zawierających m.in. następujące pola:

`trx_time`
: Całkowity czas wykonywania transakcji, wyrażony w milisekundach z dokładnością do mikrosekundy.

`query_time`
: Całkowity czas wykonywania zapytania, wyrażony w milisekundach z dokładnością do mikrosekundy.

`idle_time`
: Całkowity czas wykonywania transakcji pomniejszony o czas wykonywania zapytania, wyrażony w milisekundach z dokładnością do mikrosekundy. Czas bezczynności wskazuje, jak długo aplikacja nie ma nic do zrobienia podczas wykonywania zapytań w transakcji.

`query_count`
: Liczba zapytań wykonanych w transakcji.

`rows_*`
: Całkowita liczba rekordów odpowiednio przeanalizowanych, zmodyfikowanych i przekazanych przez wszystkie zapytania wykonane w transakcji.

Dane wyjściowe wygenerowane przez polecenie z listingu 8.5 są podobne do następujących:

```
+----------+----------+-----------+---------+-----------+-----------+-----------+
| trx_time | qry_time | idle_time | qry_cnt | rows_exam | rows_affe | rows_sent |
+----------+----------+-----------+---------+-----------+-----------+-----------+
| 5647.892 |    1.922 |  5645.970 |       2 |        10 |         0 |        10 |
|    0.585 |    0.403 |     0.182 |       2 |        10 |         0 |        10 |
+----------+----------+-----------+---------+-----------+-----------+-----------+
```

W omawianym przykładzie ta sama transakcja została wykonana dwukrotnie: pierwszy raz ręcznie, drugi po skopiowaniu i wklejeniu kodu transakcji. Ręczne wykonanie zajęło 5,6 sekundy (5647,892 ms), a przez większość czasu była bezczynna ze względu na konieczność wpisania polecenia. Natomiast czas transakcji wykonanej programowo powinien składać się głównie z czasu wykonywania zapytania, jak pokazałem w drugim rekordzie danych wyjściowych: 403 mikrosekundy czasu wykonywania i tylko 182 mikrosekundy czasu oczekiwania.

Podsumowanie

W tym rozdziale przeanalizowałem transakcje MySQL w odniesieniu do unikania najczęściej występujących problemów z transakcjami. Oto najważniejsze wnioski płynące z zamieszczonego tutaj materiału:

- Poziom izolacji transakcji wpływa na blokowanie rekordów (blokady danych).
- Do podstawowych blokad InnoDB zaliczamy: *blokady rekordów indeksu* (blokada nakładana na pojedynczy rekord indeksu), *blokady następnego klucza* (blokada nakładana na pojedynczy rekord indeksu plus luka znajdująca się przed nim), *blokady luk* (blokada zakresu (luki) między dwoma rekordami indeksu) i *blokada zamiaru wstawienia* (pozwala wykonać zapytanie `INSERT` wstawiające dane do luki; to bardziej warunek oczekiwania niż faktyczna blokada).
- Domyślny poziom izolacji transakcji, `REPEATABLE READ`, wykorzystuje blokowanie luk w celu odizolowania zakresu rekordów, których dotyczyło zapytanie.
- Poziom izolacji transakcji `READ COMMITTED` powoduje wyeliminowanie blokad luk.
- Silnik InnoDB używa tzw. *spójnych migawek* na poziomie izolacji transakcji `REPEATABLE READ`, aby zapytaniom odczytu (`SELECT`) umożliwić zwrot tych samych rekordów pomimo zmian wprowadzanych w nich przez inne transakcje.
- Spójne migawki wymagają od silnika InnoDB zapisywania w tzw. dziennikach przywracania zmian w rekordach, dzięki którym można przywracać starsze wersje rekordów.

- Wskaźnik wielkości listy historii (HLL) podaje liczbę starszych wersji rekordu, które nie zostały usunięte ani zapisane na dysku.
- Wartość HLL jest zwiastunem zbliżającego się nieszczęścia — zawsze należy ją monitorować i zgłaszać, gdy jest większa niż 100 000.
- Blokada danych i dzienniki przywracania są zwalniane po zakończeniu transakcji za pomocą zapytania `COMMIT` lub `ROLLBACK`.
- Do czterech najczęściej występujących problemów dotyczących transakcji zaliczamy: ogromne transakcje (modyfikacja zbyt wielu rekordów), długo wykonywane transakcje (długi czas odpowiedzi między poleceniami `BEGIN` i `COMMIT`), transakcje przeciągające się (zbędne oczekiwanie między zapytaniami) i transakcje porzucone (połączenie klienta nagle zniknęło w aktywnej transakcji).
- Tabele funkcjonalności Performance Schema w MySQL umożliwiają generowanie szczegółowych raportów dotyczących transakcji.
- Wydajność działania transakcji jest równie ważna jak wydajność działania zapytania.

W następnym rozdziale skoncentruję się na najczęściej pojawiających się wyzwaniach dotyczących serwera MySQL i sposobach ich pokonywania.

Ćwiczenia praktyczne: ostrzeżenie dotyczące wielkości listy historii

Celem tego ćwiczenia jest wygenerowanie komunikatu ostrzeżenia, gdy wielkość listy historii (HLL) przekroczy 100 000 (zapoznaj się z podrozdziałem „Wielkość listy historii"). Dokładne rozwiązanie zależy od używanego systemu monitorowania (zbierania wskaźników) i ostrzegania, choć w zasadzie nie będzie różniło się od ostrzegania o innych wskaźnikach. Konieczne będzie wykonanie dwóch zadań:

- zebranie i podanie wartości HLL,
- utworzenie komunikatu ostrzeżenia w przypadku przekroczenia wartości 100 000 wskaźnika HLL.

Wszystkie narzędzia monitorowania powinny być w stanie pobierać wartość HLL i informować o niej. Jeżeli używane przez Ciebie narzędzie monitorowania tego nie potrafi, poważnie rozważ jego zmianę, ponieważ wartość HLL to jeden z najważniejszych wskaźników. Zapoznaj się z dokumentacją narzędzia monitorowania, aby dowiedzieć się, jak pobierać wartość HLL i informować o niej. Wprawdzie wartość HLL może się często zmieniać, ale mamy duże pole manewru, zanim serwerowi MySQL zacznie coś grozić ze względu na wysoką wartość HLL. Dlatego wartość HLL można podawać dość rzadko: co minutę.

Gdy narzędzie potrafi monitorować i podawać wartość HLL, zdefiniuj komunikat ostrzeżenia dla wartości większej niż 100 000 przez 20 minut. Przypomnij sobie punkt „Szukanie wiatru w polu (wartości progowe)" z rozdziału 6.: być może trzeba będzie zmienić wartość progową 20 minut. Pamiętaj jednak, że wartość HLL większa niż 100 000 dłużej niż przez 20 minut zdecydowanie jest nienormalna.

Jeżeli musisz ręcznie pobrać wartość HLL, skorzystaj z następującego zapytania:

```
SELECT name, count
FROM   information_schema.innodb_metrics
WHERE  name = 'trx_rseg_history_len';
```

Kiedyś wartość HLL była pobierana z danych wyjściowych zapytania `SHOW ENGINE INNODB STATUS` — poszukaj `History list length` w sekcji `TRANSACTIONS`.

Mam nadzieję, że nigdy nie skorzystasz z tego komunikatu ostrzeżenia. Jednak jego zdefiniowanie jest uznawane za najlepszą praktykę i może uratować wiele aplikacji przed awarią. Komunikat ostrzeżenia dotyczącego HLL jest Twoim przyjacielem.

Ćwiczenia praktyczne: analiza blokad rekordów

Celem tego ćwiczenia jest przeanalizowanie blokad rekordów dla rzeczywistych zapytań wykonywanych przez Twoją aplikację i, o ile to możliwe, zrozumienie, dlaczego zapytanie nakłada poszczególne blokady. Użyłem tutaj sformułowania *jeżeli możliwe*, ponieważ blokady rekordów nakładane przez silnik InnoDB bywają trudne do zbadania.

Podczas wykonywania ćwiczenia skorzystaj z programistycznego lub testowego egzemplarza MySQL, a nie ze środowiska produkcyjnego. Ponadto użyj wydania MySQL 8.0.16 lub nowszego, ponieważ oferuje ono lepsze informacje dotyczące blokad danych pochodzące z tabeli `data_lock` w Performance Schema, jak wyjaśniłem w podrozdziale „Nakładanie blokad na rekordy". Jeżeli masz do dyspozycji jedynie MySQL 5.7, wówczas analizę danych musisz przeprowadzić za pomocą zapytania `SHOW ENGINE INNODB STATUS`. Ilustrowany przewodnik dotyczący mapowania danych wyjściowych blokad danych z MySQL 5.7 w MySQL 8.0 znajdziesz w artykule *MySQL Data Locks: Mapping 8.0 to 5.7* (*https://hackmysql.com/post/mysql-data-locks-mapping-80-to-57/*).

Korzystaj z rzeczywistych definicji danych i jak najbardziej rzeczywistych danych (rekordów). Jeżeli istnieje możliwość, skopiuj dane ze środowiska produkcyjnego i wczytaj je w programistycznym lub testowym egzemplarzu MySQL.

Jeżeli istnieją szczególnie interesujące Cię zapytania lub transakcje, zacznij od przeanalizowania dotyczących ich blokad danych. W przeciwnym razie rozpocznij od wolno wykonywanych zapytań — przypomnij sobie informacje z podpunktu „Profil zapytania" w rozdziale 1.

Skoro blokady są zwalniane po zakończeniu transakcji, konieczne jest użycie wyraźnie zdefiniowanych transakcji, jak to omówiłem w podrozdziale „Nakładanie blokad na rekordy":

```
BEGIN;

--
-- Wykonanie jednego lub wielu zapytań
--

SELECT index_name, lock_type, lock_mode, lock_status, lock_data
FROM   performance_schema.data_locks
WHERE  object_name = 'elem';
```

Wartość `elem` zastąp nazwą swojej tabeli i pamiętaj o wykonaniu polecenia `COMMIT` lub `ROLLBACK`, aby zwolnić blokady.

Aby zmienić poziom izolacji transakcji dla następnej (i tylko dla następnej) transakcji, przed poleceniem `BEGIN` musisz wykonać zapytanie `SET TRANSACTION ISOLATION LEVEL READ COMMITTED`.

To jest ćwiczenie dla eksperta, więc wysiłek włożony w jego zrozumienie jest osiągnięciem. Gratulacje.

Efektywny MySQL

To książka skierowana do programistów, którzy znają podstawy MySQL, choć niekoniecznie na zaawansowanym poziomie, i chcą zgłębić wiedzę na temat tej technologii, aby móc z niej korzystać w maksymalnie wydajny sposób. Stawia na naukę efektywności pracy — pokazuje i objaśnia rozwiązania pozwalające na szybkie i proste posługiwanie się relacyjnymi bazami danych. Co ważne, treść została zilustrowana wieloma przykładami dotyczącymi różnych elementów i mechanizmów MySQL, których zastosowanie umożliwia wysyłanie do bazy danych zoptymalizowanych zapytań.

Daniel Nichter prezentuje dobre praktyki, po które warto sięgać, aby tworzyć wydajny kod. Korzystając z wieloletniego doświadczenia w pracy z relacyjnymi bazami danych, podaje gotowe rozwiązania wraz z wyjaśnieniami umożliwiającymi wysyłanie nawet skomplikowanych zapytań przy minimalnym obciążeniu serwera. To cenne źródło wiedzy niezbędnej do szybkiej nauki praktycznego zastosowania MySQL.

W książce między innymi:

- przykłady kodu i jego zastosowania
- porady dotyczące tworzenia szybkich zapytań generujących jak najmniejsze obciążenie serwera
- opisy narzędzi i mechanizmów do diagnostyki i optymalizacji działania baz danych MySQL

Daniel Nichter — administrator baz danych, od 15 lat specjalizuje się w optymalizacji działania MySQL. W firmie Percona stworzył liczne narzędzia, które usprawniają pracę programistów i administratorów i z których na szeroką skalę korzystają obecnie największe przedsiębiorstwa IT. Wielokrotny prelegent na konferencjach i laureat nagród programistycznych, w tym MySQL Community Award.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-9290-8

9 788328 392908

Cena: 89,00 zł