MySQL pod obstrzałem

Historia bazy danych MySQL sięga wczesnych lat 90., gdy dość popularnym rozwiązaniem była oferowana przez Hughes Technologies baza miniSQL. W tamtym czasie mySQL, gdyż także pod taką nazwą znany jest produkt HT, nie był demonem prędkości, a brak wsparcia dla indeksów czy trudności z obsługą wielu połączeń klienckich jednocześnie odbijały się czkawką w bardziej zaawansowanych aplikacjach. Rozwiązanie tych problemów miał Michael „Monty" Widenius, autor interfejsu do niskopoziomowego łączenia się z ISAM storage i indeksowania danych, który szybko skontaktował się z Davidem Hughesem, aby wspólnie zaoferować rynkowi nowy produkt…

Rozmowy obu panów nie doprowadziły jednak do połączenia sił, głównie z powodu zaawansowanych prac nad mSQL 2.0. Hughes Technologies nie widziało więc potrzeby integracji z UNIREG Wideniusa, który z kolei postanowił dostarczyć na rynek swoje rozwiązanie, zgodne z API mSQL. Choć prace nad nowym systemem oficjalnie ruszyły w 1994, to już zaledwie rok później, w maju 1995 roku, wydana została jej pierwsza wersja. MySQL ujrzał światło dzienne.

U PODSTAW

Przez lata MySQL budził skrajne emocje. Brak obsługi transakcji czy wsparcia dla kluczy obcych w wielu silnikach składowania danych nie przysparzały mu zwolenników, którzy często określali go mianem produktu bazodano-podobnego, nienadającego się do zastosowania w większych projektach. Dziś jednak tę bazę można spotkać w wielu popularnych serwisach internetowych, w tym Facebooku, Google czy Twitterze, gdzie wraz z innymi rozwiązaniami NoSQL tworzy kompletny ekosystem do składowania danych. I to właśnie w architekturze pluginowej oraz w możliwości dołączania do serwera nowych silników danych o odmiennych charakterystykach można dopatrywać się sukcesu MySQL-a.

Projektując nową bazę danych, można wybierać spośród wielu wbudowanych silników, w zależności od konkretnych potrzeb, w tym m.in.:

- InnoDB, oferujące choćby wsparcie dla kluczy obcych, transakcje ACID, blokowanie danych na poziomie wierszy, partycjonowanie czy automatyczne klastrowanie danych,
- Memory, składujące wszystkie dane w pamięci RAM, dla zwiększenia przepustowości zapytań i zmniejszenia czasu odpowiedzi,
- Archive, pozwalający składować dane historyczne i automatycznie je kompresować,
- Blackhole, stanowiący odpowiednik systemowego /dev/null i wbrew pozorom pomocny w wielu przypadkach, wspomagał np. replikację części danych w początkowym okresie rozwoju Facebooka,
- Federated, pozwalający na lokalny dostęp do danych zgromadzonych na zdalnych serwerach, bez potrzeby ich replikacji i synchronizacji,
- NDB, przeznaczony do budowy wysoko-dostępnych klastrów z automatyczną replikacją danych.

Gamę tę uzupełniają rozwiązania zewnętrznych producentów, z których warto tu wspomnieć choćby rozszerzający funkcjonalności InnoDB silnik XtraDB, komercyjny ScaleDB dla data warehousingu czy TokuDB, oferujący wysoki stopień kompresji danych oraz indeksy Fractal Tree. Możliwości oferowane przez poszczególne silniki są ogromne, choć oczywiście w wielu przypadkach warto gruntownie zapoznać się z ich dokumentacją.

Jednak MySQL dziś to nie tylko możliwość wykorzystania konkretnego silnika składowania danych, ale także wersji i producenta samego serwera. Oprócz pierwotnej implementacji rozwijanej od 2008 roku przez Oracle do dyspozycji są także wspierane przez Percona Server czy MariaDB, oferujące w wielu aspektach dodatkowe funkcjonalności i jeszcze lepszą wydajność.

W tej serii artykułów postaram się przedstawić kluczowe możliwości poszczególnych rozwiązań wbudowanych w MySQL i przykłady ich wykorzystania w praktyce. Zobaczmy więc, jak MySQL zachowuje się pod obstrzałem.

DANE, DANE, DANE

Wydajność zapytań kierowanych do bazy danych zależy od bardzo wielu czynników. Z jednej strony istotną rolę odgrywają tu wszelkie aspekty sprzętowe, jak ilość i szybkość procesorów, wielkość pamięci operacyjnej i czas dostępu do dysków. Z drugiej ważne są aspekty związane z przetwarzaniem zapytań SQL, ich strukturą, ilością analizowanych informacji czy ogólną liczbą równoległych odwołań do serwera. I wreszcie z trzeciej strony wszystko to spaja schemat bazy, określający struktury poszczególnych tabel, indeksów i widoków.

Zdarza się, iż raz zaprojektowany schemat nie ulega zmianom przez długie miesiące, a jednak wraz z upływem czasu da się zauważyć istotną degradację czasów odpowiedzi poszczególnych zapytań. Przyczyn takiego stanu rzeczy nie trzeba szukać daleko. To dane, napływające często z bardzo dużą szybkością do naszej bazy powodują ten problem. Wyobraźmy sobie w tym momencie system emitujący reklamy, gdzie każdego dnia generowanych jest średnio 3 miliony odsłon, z których każda musi zostać zapisana na potrzeby pozostałych mechanizmów aplikacji. W skali dnia liczba ta nie jest szczególnie wielka, natomiast w skali roku oznacza to już ponad 1 miliard wierszy! Co będzie po kolejnym roku lub gdy ze względu na rozwój biznesu zapisywać będziemy 5 milionów odsłon dziennie?

Co więc zrobić w sytuacji, gdy ilość danych zaczyna negatywnie odbijać się na naszych zapytaniach i wydajności całego systemu? Odpowiedzi jest zaskakująco wiele, możemy przykładowo skalować wertykalnie warstwę sprzętową, implementować cache i bufory na dane czy też przeprowadzić migrację do jednej z popularnych baz NoSQL. Możliwość zastosowania Cassandry, HBase czy Riaka zwykle brzmi naprawdę kusząco, przecież w końcu w projekcie pojawi się błysk nowości! Jednak tak istotna zmiana wymaga odpowiedniego przygotowania i know-how, a tymczasem w naszym arsenale pozostaje szereg technik optymalizacji bazy danych, które pozwolą nam pozostać w kręgu SQL przez dość długi czas.

PARTYCJONOWANIE DANYCH

Analizując naszą bazę w poszukiwaniu pomysłów na optymalizacje, zastanówmy się przez chwilę nad sposobem organizacji naszych danych i głównymi scenariuszami ich analizy. Postawienie właściwych pytań jest tu kluczowe. Jeśli nasza aplikacja reklamowa agreguje informacje o odsłonach, aby następnie przedstawić dane dobowe z okresu ostatnich 30/60/90 dni na podstawowych dashboardach, to czy zapytanie musi wykorzystywać całość zgromadzonych w tabeli danych?

Rozważmy następującą strukturę tabeli i zapytanie (klucze obce zostały tu pominięte dla uproszczenia):

```
CREATE TABLE ad_display (
  id INT NOT NULL AUTO_INCREMENT,
  ad_id INT NOT NULL,
  user_id INT NOT NULL,
  device_id INT NOT NULL,
  displayed TIMESTAMP NOT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB;

EXPLAIN
SELECT ad_id, DATE(displayed), COUNT(*) AS cnt
FROM ad_display
WHERE displayed BETWEEN '2016-05-01' AND ,2016-08-31'
GROUP BY DATE(displayed)
ORDER BY DATE(displayed) DESC \G;

********************** 1. row **********************
           id: 1
  select_type: SIMPLE
        table: ad_display
         type: range
possible_keys: displayed
          key: displayed
      key_len: 8
          ref: NULL
         rows: 911
        extra: Using where; Using temporary; Using file
sort
1 row in set (0.00 sec)
```

Jeśli tabela zawiera dane z kilku lat i naprawdę sporo danych, użycie indeksu może nie być już wystarczające…

Tutaj z pomocą przychodzi mechanizm *partycjonowania* danych. Pozwala ono rozproszyć dane w ramach jednej tabeli na szereg mniejszych, osobnych tabel, ukrywając całkowicie ten fakt przez klientami łączącymi się z bazą. Zamiast pojedynczej tabeli z 1 miliardem wierszy, moglibyśmy więc mieć 4 tabele z 250 milionami rekordów każda, co powinno korzystnie wpłynąć na performance systemu. Mniejsze ilości danych oznaczają mniejsze pliki i indeksy, które będą skanowane jeszcze szybciej, a dodatkowo być może część najstarszych danych nigdy nie zostanie załadowana do pamięci. Pozostaje się jedynie zastanowić, w jaki sposób zorganizowane są nasze dane? Czy potrzebujemy analizować dane w kontekście czasu, czy też może raczej całość danych np. w kontekście konkretnego klienta? Odpowiedź na to pytanie pozwoli zbudować tzw. klucz partycjonujący, względem którego MySQL zrealizuje podział danych. Dobrze wybrany klucz skróci czas dostępu do danych, nawet kilkaset razy, natomiast źle… Cóż, zapewne pogorszą się czasy odpowiedzi wszystkich zapytań w aplikacji i efekt będzie odwrotny od pożądanego.

Partycjonowanie RANGE

Partycjonowanie `RANGE` przypisuje wiersze do określonych partycji, bazując na zdefiniowanych uprzednio zakresach wartości dla każdej z partycji. Przyjrzyjmy się następującej definicji tabeli, w której kluczem partycjonującym stały się zakresy dat składowanych w kolumnie `displayed`:

```
CREATE TABLE ad_display_partitioned (
    ad_id INT NOT NULL,
    user_id INT NOT NULL,
    device_id INT NOT NULL,
    displayed TIMESTAMP NOT NULL
) ENGINE=InnoDB
PARTITION BY RANGE(UNIX_TIMESTAMP(displayed)) (
  PARTITION p0 VALUES LESS THAN (UNIX_TIMESTAMP('2016-01-01 00:00:00')),
  PARTITION p1 VALUES LESS THAN (UNIX_TIMESTAMP('2016-02-01 00:00:00')),
  PARTITION p2 VALUES LESS THAN (UNIX_TIMESTAMP('2016-03-01 00:00:00')),
  PARTITION p3 VALUES LESS THAN (UNIX_TIMESTAMP('2016-04-01 00:00:00')),
  PARTITION p4 VALUES LESS THAN (UNIX_TIMESTAMP('2016-05-01 00:00:00')),
  PARTITION p5 VALUES LESS THAN (UNIX_TIMESTAMP('2016-06-01 00:00:00')),
  PARTITION p6 VALUES LESS THAN (MAXVALUE)
);
```

Określono tu 6 partycji obejmujących okresy miesięczne, za wyjątkiem partycji `p0` (wszystkie dane z lat wcześniejszych) i `p6` (wszystkie dane od czerwca 2016). Warto tu pamiętać o sytuacjach brzegowych, aby zawsze istniała partycja zdolna przyjąć nasze dane.

Upewnijmy się także, jak serwer widzi tak zdefiniowaną tabelę partycjonowaną.

```
mysql> SHOW TABLES;
+------------------------+
| Tables_in_test         |
+------------------------+
| ad_display             |
| ad_display_partitioned |
+------------------------+
2 rows in set (0.00 sec)

# ls -l /var/lib/mysql/test
total 896
-rw-rw---- 1 mysql mysql   8704 Aug 03 11:28 ad_display.frm
-rw-rw---- 1 mysql mysql 163840 Aug 03 11:30 ad_display.ibd
-rw-rw---- 1 mysql mysql   8678 Aug 03 11:30 ad_display_partitioned.frm
-rw-rw---- 1 mysql mysql     48 Aug 03 11:30 ad_display_partitioned.par
-rw-rw---- 1 mysql mysql  98304 Aug 03 11:30 ad_display_partitioned#P#p0.ibd
-rw-rw---- 1 mysql mysql  98304 Aug 03 11:30 ad_display_partitioned#P#p1.ibd
-rw-rw---- 1 mysql mysql  98304 Aug 03 11:30 ad_display_partitioned#P#p2.ibd
-rw-rw---- 1 mysql mysql  98304 Aug 03 11:30 ad_display_partitioned#P#p3.ibd
-rw-rw---- 1 mysql mysql  98304 Aug 03 11:30 ad_display_partitioned#P#p4.ibd
-rw-rw---- 1 mysql mysql  98304 Aug 03 11:30 ad_display_partitioned#P#p5.ibd
-rw-rw---- 1 mysql mysql 131072 Aug 03 11:30 ad_display_partitioned#P#p6.ibd
-rw-rw---- 1 mysql mysql     65 Aug 03 11:28 db.opt
```

Zgodnie z oczekiwaniami tabela `ad_display_partitioned` została podzielona na kilka osobnych tabel składowanych w odrębnych plikach, z jednolitym interfejsem dostępu z poziomu serwera MySQL.

Partycjonowanie LIST

Partycjonowanie `LIST` pozwala przypisać do partycji dane na podstawie uprzednio określonych zbiorów wartości. Model ten sprawdza się, gdy zbiory te nie są zbyt obszerne, a sam podział zapewnia w miarę równe rozłożenia danych pomiędzy partycje.

W przypadku analizowanej tabeli podział typu LIST byłby możliwy np. na podstawie identyfikatora typu urządzenia, które wygenerowało odsłonę reklamy. Wówczas zapytania przetwarzające dane np. z konkretnych przeglądarek czy klientów mobilnych zostałyby przyśpieszone.

```
PARTITION BY LIST (device_id) (
  PARTITION p0 VALUES IN (1, 2, 3, 4, 5),
  PARTITION p1 VALUES IN (6, 7, 8),
  PARTITION p2 VALUES IN (9, 10, 11, 12)
)
```

Użycie partycjonowania LIST powinno być jednak poprzedzone gruntowną analizą danych, aby uniknąć przeładowania części partycji.

Partycjonowanie HASH, LINEAR HASH, KEY

Partycjonowanie HASH pozwala na automatyczne rozłożenie danych względem klucza partycjonującego na określoną liczbę partycji. Klucz ten może być zarówno nazwą kolumny składującej wartości typu integer, jak i wyrażeniem zwracającym wartość tego typu. Dane są przypisywane do partycji na podstawie wyniku operacji modulo z liczbą partycji.

```
PARTITION BY HASH(device_id)
PARTITIONS 4
```

W sytuacji, gdy tabela zawiera spore ilości danych i dość często wykonywane są operacje maintenance typu dodanie bądź usunięcie partycji, praktyczniejsze będzie wykorzystanie modelu LINEAR HASH. Spowoduje on mniej równomierne wypełnienie danymi poszczególnych partycji, jednak zarządzanie nimi zostanie istotnie przyśpieszone.

MySQL oferuje wreszcie partycjonowanie KEY, w którym funkcja partycjonująca jest z góry określona przez serwer.

Niezależnie od wybranego modelu tutaj także należy dokonać analizy, czy zaproponowany podział danych pomiędzy partycje pomoże przyśpieszyć jej działanie.

Odcinanie partycji

Partycjonowanie tabeli nie miałoby w zasadzie żadnego sensu, gdyby serwer MySQL nie potrafił wykonać tzw. *partition pruning*. Idea za tym zachowaniem jest bardzo prosta: *nie skanuj partycji, w których nie może być odpowiednich danych*. Znając klucz partycjonujący oraz dane z zapytania SQL, serwer bazy danych może zdecydować o pominięciu przetwarzania wybranych partycji i w ten właśnie sposób przyśpieszyć wykonanie wielu zapytań.

Powróćmy zatem do pierwszego zapytania, zmieńmy jednak tabelę, z której pochodzić będą dane:

```
EXPLAIN PARTITIONS
SELECT ad_id, DATE(displayed), COUNT(*) AS cnt
FROM ad_display_partitioned
WHERE displayed BETWEEN '2016-05-01' AND ,2016-08-31'
GROUP BY DATE(displayed)
ORDER BY DATE(displayed) DESC \G;
```

```
********************** 1. row **********************
              id: 1
     select_type: SIMPLE
           table: ad_display_partitioned
      partitions: p5,p6
            type: range
   possible_keys: displayed
             key: NULL
         key_len: NULL
             ref: NULL
            rows: 848
           extra: Using where; Using temporary; Using file
sort
1 row in set (0.00 sec)
```

Plan zapytania pokazuje odcięcie większości partycji, przeskanowane zostały jedynie partycje p5 i p6. Voila! Oczywiście efekt będzie tym lepszy, im więcej danych zostanie odciętych podczas wykonywania zapytania.

Co jednak będzie, gdy zapytanie SQL nie będzie zawierało odwołań do klucza partycjonującego? Niestety w takim przypadku MySQL nie będzie potrafił dokonać odcięcia, zamiast tego przeskanuje wszystkie zdefiniowane partycje. Zapytanie to będzie prawdopodobnie wolniejsze niż w przypadku pojedynczej, niepartycjonowanej tabeli, ze względu na narzut związany z otwieraniem i analizowaniem poszczególnych partycji. I z tego właśnie powodu dobór klucza do najczęstszego modelu przetwarzania danych jest tak bardzo istotny! Czasem warto także rozważyć zmianę niektórych części aplikacji, aby zwiększyć odsetek zapytań korzystających z *partition pruningu*.

Subpartycjonowanie

W przypadku, gdy baza danych zawiera naprawdę dużo informacji i partycjonowanie tylko częściowo rozwiązuje ten problem, można rozważyć użycie tzw. *subpartycjonowania*. Pozwala ono dokonać dalszego podziału w ramach już istniejącej partycji (każdej z osobna), przez co jeszcze bardziej skrócić czas dostępu do danych. Oczywiście pod warunkiem wystąpienia w zapytaniu SQL danych pozwalających na wykorzystanie odcięcia partycji i subpartycji.

MySQL oferuje subpartycjonowanie dla tabel partycjonowanych w modelu RANGE i LIST, gdzie subpartycje mogą wykorzystywać modele HASH oraz KEY. Jednak pomimo tych ograniczeń, w praktyce sprawdza się to znakomicie.

W poniższym przykładzie finalna liczba partycji to 28, 7 partycji RANGE podzielonych na 4 subpartycje HASH każda.

```
PARTITION BY RANGE(UNIX_TIMESTAMP(displayed))
SUBPARTITION BY HASH(device_id)
SUBPARTITIONS 4 (
  PARTITION p0 VALUES LESS THAN (UNIX_TIMESTAMP('2016-01-01 00:00:00')),
  PARTITION p1 VALUES LESS THAN (UNIX_TIMESTAMP('2016-02-01 00:00:00')),
  PARTITION p2 VALUES LESS THAN (UNIX_TIMESTAMP('2016-03-01 00:00:00')),
  PARTITION p3 VALUES LESS THAN (UNIX_TIMESTAMP('2016-04-01 00:00:00')),
  PARTITION p4 VALUES LESS THAN (UNIX_TIMESTAMP('2016-05-01 00:00:00')),
  PARTITION p5 VALUES LESS THAN (UNIX_TIMESTAMP('2016-06-01 00:00:00')),
  PARTITION p6 VALUES LESS THAN (MAXVALUE)
);
```

ZARZĄDZANIE PARTYCJAMI

W momencie startu projektu zwykle trudno będzie określić, które tabele i jak powinny być partycjonowane, to przyjdzie z czasem. Na szczęście MySQL oferuje mechanizmy pozwalające na modyfikację partycjonowania oraz wprowadzanie nowych partycji na już istniejących tabelach za pomocą ALTER TABLE.

```
ALTER TABLE ad_display_partitioned DROP PARTITION p6;
ALTER TABLE ad_display_partitioned ADD PARTITION (
  PARTITION p6 VALUES LESS THAN (UNIX_TIMESTAMP('2016-07-01
  00:00:00')),
  PARTITION p7 VALUES LESS THAN (MAXVALUE)
);
```

Operacje przebudowania, optymalizacji czy naprawy poszczególnych partycji również nie nastręczają wielu problemów. Niektóre silniki składowania danych mogą jednak nie wspierać pewnych operacji na poziomie partycji, pozwalając jednak na ich wykonanie na poziomie całej tabeli.

```
ALTER TABLE ad_display_partitioned REBUILD PARTITION p0;
ALTER TABLE ad_display_partitioned OPTIMIZE PARTITION p1, p2;
ALTER TABLE ad_display_partitioned ANALYZE PARTITION p3;
ALTER TABLE ad_display_partitioned REPAIR PARTITION p4;
ALTER TABLE ad_display_partitioned CHECK PARTITION p5;
```

Partycjonowanie wraz z operacjami managementu można wykorzystać także w dość nietypowy sposób. Na przykład oferujący dobrą kompresję danych silnik Archive nie wspiera operacji aktualizacji, usuwania oraz indeksowania danych. Jednak wprowadzenie dobrego klucza partycjonującego i wykorzystanie go w roli częściowego indeksu potrafi niesamowicie zwiększyć przepustowość bazy. A zmodyfikowane dane macierzyste zawsze można odtworzyć całymi partycjami.

SILVER BULLET?

Decydując się na partycjonowanie bazy danych, w pierwszej kolejności należy zdecydować, które tabele i jak powinny zostać podzielone, bowiem nie każda tabela powinna temu podlegać! W tym celu warto przeanalizować logi aplikacji i zidentyfikować najczęściej wykonywane zapytania, poznać ich przeznaczenie oraz strukturę analizowanych danych. Wszystko po to, aby zaprojektowany klucz partycjonujący faktycznie działał. Pomocne może być także wykonanie kilku prób na kopii bazy danych i porównanie skuteczności poszczególnych rozwiązań. Nie zostawiajmy tu niczego przypadkowi.

Czy przypadkiem nie znaleźliśmy więc mitycznego silver bullet? Niestety nie… Wprowadzenie partycjonowania wiąże się także z pewnymi ograniczeniami, które mogą wystąpić na poziomie samego serwera, jak i konkretnych silników składowania danych. W szczególności interesujące są tu powiązania pomiędzy kluczem partycjonującym a kluczami głównymi i unikalnymi tabeli. Jednak w tym miejscu odsyłam do dokumentacji, która bardzo precyzyjnie definiuje wszystkie wymogi i ograniczenia.

Natomiast dobrze spartycjonowana baza danych odwdzięczy nam się niesamowitą wydajnością, a osiągnięcie 100-krotnego przyśpieszenia zapytań tylko na tym poziomie nie powinno nikogo specjalnie dziwić.

MARIUSZ GIL
mariusz.gil@sourceministry.com

Z branżą IT związany od ponad 14 lat, w tym czasie pracował m.in. dla Naszej-Klasy, Gadu-Gadu, Adv.pl, a także jako konsultant w zakresie tematyki skalowalności i wydajności aplikacji internetowych. Interesuje się m.in. zagadnieniami związanymi z budową skalowalnych i wydajnych rozwiązań webowych, zarówno od strony architektury aplikacji, jak i infrastruktury serwerowej. Pasjonuje się tematami związanymi z analizą dużych zbiorów danych.
Trener w Bottega IT Solutions.