helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

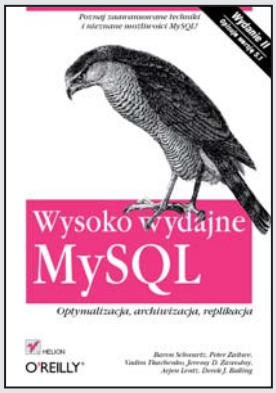
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Wysoko wydajne MySQL. Optymalizacja, archiwizacja, replikacja. Wydanie II



Autor: Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy D. Zawodny, Arjen Lentz, Derek J. Balling
Tłumaczenie: Robert Górczyński
ISBN: 978-83-246-2055-5
Tytuł oryginału: High Performance MySQL: Optimization, Backups, Replication, and More, 2nd edition
Format: 168x237, stron: 712

Poznaj zaawansowane techniki i nieznane możliwości MySQL!

- Jak za pomocą MySQL budować szybkie i niezawodne systemy?
- Jak przeprowadzać testy wydajności?
- Jak optymalizować zaawansowane funkcje zapytań?

MySQL jest ciągle udoskonalanym i rozbudowywanym oprogramowaniem. Stale zwiększa się także liczba jego użytkowników, wśród których nie brak wielkich korporacji. Wynika to z niezawodności i ogromnej, wciąż rosnącej wydajności tego systemu zarządzania. MySQL sprawdza się także w bardzo wymagających środowiskach, na przykład aplikacjach sieciowych, ze względu na dużą elastyczność i możliwości, takie jak zdolność wczytywania silników magazynu danych jako rozszerzeń w trakcie działania bazy.

Książka „Wysoko wydajne MySQL. Optymalizacja, archiwizacja, replikacja. Wydanie II" szczegółowo prezentuje zaawansowane techniki, dzięki którym można w pełni wykorzystać cały potencjał, drzemiący w MySQL. Omówiono w niej praktyczne, bezpieczne i pozwalające na osiągnięcie dużej wydajności sposoby skalowania aplikacji. Z tego przewodnika dowiesz się, w jaki sposób projektować schematy, indeksy i zapytania. Poznasz także zaawansowane funkcje MySQL, pozwalające na uzyskanie maksymalnej wydajności. Nauczysz się tak dostrajać serwer MySQL, system operacyjny oraz osprzęt komputerowy, aby wykorzystywać pełnię ich możliwości.

- Architektura MySQL
- Testy wydajności i profilowanie
- Optymalizacja schematu i indeksowanie
- Optymalizacja wydajności zapytań
- Przechowywanie kodu
- Umieszczanie komentarzy w kodzie składowym
- Konfiguracja serwera
- Dostrajanie i optymalizacja wyszukiwania pełnotekstowego
- Skalowalność i wysoka dostępność
- Wydajność aplikacji
- Kopia zapasowa i odzyskiwanie
- Interfejs SQL dla poleceń spreparowanych
- Bezpieczeństwo

Twórz doskonale dostrojone aplikacje MySQL

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2008

Spis treści

Przedmowa **7**

Wprowadzenie **9**

1. Architektura MySQL **19**
Architektura logiczna MySQL 19
Kontrola współbieżności 22
Transakcje 24
Mechanizm Multiversion Concurrency Control 31
Silniki magazynu danych w MySQL 32

2. Określanie wąskich gardeł: testy wydajności i profilowanie **51**
Dlaczego warto przeprowadzić testy wydajności? 52
Strategie przeprowadzania testów wydajności 53
Taktyki przeprowadzania testów wydajności 56
Narzędzia do przeprowadzania testów wydajności 61
Przykładowe testy wydajności 64
Profilowanie 73
Profilowanie systemu operacyjnego 95

3. Optymalizacja schematu i indeksowanie **99**
Wybór optymalnego rodzaju danych 100
Podstawy indeksowania 115
Strategie indeksowania w celu osiągnięcia maksymalnej wydajności 125
Studium przypadku z zakresu indeksowania 150
Obsługa indeksu oraz tabeli 155
Uwagi dotyczące silników magazynowania danych 168

4. Optymalizacja wydajności zapytań 171
Podstawy powolnych zapytań: optymalizacja dostępu do danych 171
Sposoby restrukturyzacji zapytań 176
Podstawy wykonywania zapytań 179
Ograniczenia optymalizatora zapytań MySQL 198
Optymalizacja określonego rodzaju zapytań 207
Zmienne zdefiniowane przez użytkownika 217

5. Zaawansowane funkcje MySQL 223
Bufor zapytań MySQL 223
Przechowywanie kodu wewnątrz MySQL 236
Funkcje zdefiniowane przez użytkownika 248
System kodowania znaków i kolejność sortowania 255
Ograniczenia klucza zewnętrznego 270
Tabele Merge i partycjonowane 271
Transakcje rozproszone (XA) 280

6. Optymalizacja konfiguracji serwera 283
Podstawy konfiguracji 284
Składnia, zasięg oraz dynamizm 285
Ogólne dostrajanie 289
Dostrajanie zachowania operacji I/O w MySQL 299
Dostosowanie współbieżności MySQL 314

7. Optymalizacja systemu operacyjnego i osprzętu 325
Co ogranicza wydajność MySQL? 326
W jaki sposób wybrać procesor dla MySQL? 326
Wybór osprzętu komputerowego dla serwera podległego 337
Optymalizacja wydajności macierzy RAID 338
Urządzenia Storage Area Network oraz Network Attached Storage 345
Używanie woluminów składających się z wielu dysków 347
Stan systemu operacyjnego 356

8. Replikacja 363
Ogólny opis replikacji 363
Konfiguracja replikacji 367
Szczegóły kryjące się za replikacją 375
Topologie replikacji 382
Replikacja i planowanie pojemności 397
Administracja replikacją i jej obsługa 399
Problemy związane z replikacją i sposoby ich rozwiązywania 409
Jak szybka jest replikacja? 428

9. Skalowalność i wysoka dostępność 431
Terminologia 432
Skalowalność MySQL 434
Wysoka dostępność 469

10. Optymalizacja na poziomie aplikacji 479
Ogólny opis wydajności aplikacji 479
Kwestie związane z serwerem WWW 482

11. Kopia zapasowa i odzyskiwanie 495
Ogólny opis 496
Wady i zalety rozwiązania 500
Zarządzanie kopią zapasową binarnych dzienników zdarzeń i jej tworzenie 510
Tworzenie kopii zapasowej danych 512
Odzyskiwanie z kopii zapasowej 523
Szybkość tworzenia kopii zapasowej i odzyskiwania 535
Narzędzia służące do obsługi kopii zapasowej 536
Kopie zapasowe za pomocą skryptów 543

12. Bezpieczeństwo 547
Terminologia 547
Podstawy dotyczące kont 548
Bezpieczeństwo systemu operacyjnego 566
Bezpieczeństwo sieciowe 567
Szyfrowanie danych 575
MySQL w środowisku chroot 579

13. Stan serwera MySQL 581
Zmienne systemowe 581
SHOW STATUS 582
SHOW INNODB STATUS 589
SHOW PROCESSLIST 602
SHOW MUTEX STATUS 603
Stan replikacji 604
INFORMATION_SCHEMA 605

14. Narzędzia zapewniające wysoką wydajność 607
Narzędzia interfejsu 607
Narzędzia monitorowania 609
Narzędzia analizy 619
Narzędzia MySQL 622
Źródła dalszych informacji 625

A Przesyłanie dużych plików 627

B Używanie polecenia EXPLAIN 631

C Używanie silnika Sphinx w MySQL 647

D Usuwanie błędów w blokadach 675

Skorowidz 685

ROZDZIAŁ 4.

Optymalizacja wydajności zapytań

W poprzednim rozdziale przeanalizowano sposoby optymalizacji schematu, która jest jednym z niezbędnych warunków osiągnięcia wysokiej wydajności. Jednak praca jedynie nad schematem nie wystarczy — trzeba również prawidłowo zaprojektować zapytania. Jeżeli zapytania okażą się niewłaściwie przygotowane, nawet najlepiej zaprojektowany schemat bazy nie będzie działał wydajnie.

Optymalizacja zapytania, optymalizacja indeksu oraz optymalizacja schematu idą ręka w rękę. Wraz z nabywaniem doświadczenia w tworzeniu zapytań MySQL czytelnik odkryje także, jak projektować schematy pozwalające na efektywną obsługę zapytań. Podobnie zdobyta wiedza z zakresu projektowania zoptymalizowanych schematów wpłynie na rodzaj zapytań. Ten proces wymaga czasu, dlatego też autorzy zachęcają, aby powrócić do rozdziałów bieżącego i poprzedniego po zdobyciu większej wiedzy.

Rozdział ten rozpoczyna się od ogólnych rozważań dotyczących projektowania zapytań — omówione są tu elementy, na które powinno się zwrócić uwagę w pierwszej kolejności, jeśli zapytania nie działają zgodnie z oczekiwaniami. Następnie nieco dokładniej zostaną przedstawione zagadnienia dotyczące optymalizacji zapytań oraz wewnętrznego działania serwera. Autorzy zademonstrują, jak można poznać sposób wykonywania określonego zapytania przez MySQL, a także zmienić plan wykonywania zapytania. Wreszcie zostaną przedstawione fragmenty zapytań, w których MySQL nie przeprowadza zbyt dobrej optymalizacji. Czytelnik pozna również wzorce optymalizacji pomagające MySQL w znacznie efektywniejszym wykonywaniu zapytań.

Celem autorów jest pomoc czytelnikowi w dokładnym zrozumieniu sposobu, w jaki MySQL faktycznie wykonuje zapytania. Pozwoli to na zorientowanie się, co jest efektywne lub nieefektywne, umożliwi wykorzystanie zalet bazy danych MySQL oraz ułatwi unikanie jej słabych stron.

Podstawy powolnych zapytań: optymalizacja dostępu do danych

Najbardziej podstawowym powodem słabej wydajności zapytania jest fakt, że obejmuje ono zbyt dużą ilość danych. Niektóre zapytania po prostu muszą dokładnie przebadać ogromną ilość danych, więc w takich przypadkach niewiele można zrobić. Jednak to nietypowa sytuacja, większość błędnych zapytań można zmodyfikować, aby uzyskiwały dostęp do mniejszej ilości danych. Autorzy odkryli, że użyteczne jest analizowanie zapytań o słabej wydajności przy zastosowaniu dwóch kroków. Oto one.

1. Określenie, czy *aplikacja* pobiera więcej danych, niż potrzebuje. Zazwyczaj oznacza to uzyskanie dostępu do zbyt wielu rekordów, ale może również polegać na uzyskiwaniu dostępu do zbyt wielu kolumn.
2. Określenie, czy *serwer MySQL* analizuje więcej rekordów, niż potrzebuje.

Czy zapytanie bazy danych obejmuje dane, które są niepotrzebne?

Niektóre zapytania dotyczą większej ilości danych niż potrzeba, później część danych i tak jest odrzucana. Wymaga to dodatkowej pracy ze strony serwera MySQL, zwiększa obciążenie sieci[1], a także zużywa pamięć i zasoby procesora serwera aplikacji.

Poniżej przedstawiono kilka typowych błędów.

Pobieranie liczby rekordów większej, niż to konieczne
: Najczęściej popełnianym błędem jest przyjęcie założenia, że MySQL dostarcza wyniki na żądanie, a nie generuje pełnego zbioru wynikowego i zwraca go. Autorzy często spotykali się z tym błędem w aplikacjach zaprojektowanych przez osoby znające zagadnienia związane z systemami baz danych. Programiści ci używali technik, takich jak wydawanie poleceń `SELECT` zwracających wiele rekordów, a następnie pobierających pierwsze *N* rekordów i zamykających zbiór wynikowy (np. pobranie stu ostatnich artykułów dla witryny informacyjnej, podczas gdy na stronie głównej było wyświetlanych tylko dziesięć z nich). Tacy programiści sądzą, że baza danych MySQL dostarczy im dziesięć rekordów, a następnie zakończy wykonywanie zapytania. W rzeczywistości MySQL generuje pełny zbiór wynikowy. Biblioteka klienta pobiera wszystkie dane i odrzuca większość. Najlepszym rozwiązaniem jest dodanie do zapytania klauzuli `LIMIT`.

Pobieranie wszystkich kolumn ze złączenia wielu tabel
: Jeżeli programista chce pobrać wszystkich aktorów występujących w filmie *Academy Dinosaur*, nie należy tworzyć zapytania w następujący sposób:

```
mysql> SELECT * FROM sakila.actor
    -> INNER JOIN sakila.film_actor USING(actor_id)
    -> INNER JOIN sakila.film USING(film_id)
    -> WHERE sakila.film.title = 'Academy Dinosaur';
```

: Powyższe zapytanie zwróci wszystkie kolumny z wszystkich trzech tabel. W zamian trzeba utworzyć następujące zapytanie:

```
mysql> SELECT sakila.actor.* FROM sakila.actor...;
```

Pobieranie wszystkich kolumn
: Zawsze warto podejrzliwie spojrzeć na zapytania typu `SELECT *`. Czy naprawdę potrzebne są wszystkie kolumny? Prawdopodobnie nie. Pobieranie wszystkich kolumn uniemożliwia optymalizację w postaci np. zastosowania indeksu pokrywającego, a ponadto zwiększa obciążenie serwera wynikające z wykonywania operacji I/O, większego zużycia pamięci i mocy obliczeniowej procesora.

  Niektórzy administratorzy baz danych z wymienionych powyżej powodów w ogóle uniemożliwiają wykonywanie poleceń `SELECT *`. Takie rozwiązanie ogranicza również ryzyko wystąpienia problemów, gdy ktokolwiek zmieni listę kolumn tabeli.

[1] Obciążenie sieci ma jeszcze poważniejsze znaczenie, jeżeli aplikacja znajduje się na serwerze innym od samego serwera MySQL. Jednak transfer danych między MySQL i aplikacją nie jest bez znaczenia nawet wtedy, kiedy i MySQL, i aplikacja znajdują się na tym samym serwerze.

Oczywiście, zapytanie pobierające ilość danych większą, niż faktycznie potrzeba, nie zawsze będzie złe. W wielu analizowanych przypadkach programiści twierdzili, że takie marnotrawne podejście upraszcza proces projektowania, a także pozwala na używanie tego samego fragmentu kodu w więcej niż tylko jednym miejscu. To dość rozsądne powody, przynajmniej tak długo, jak długo programista jest świadom kosztów mierzonych wydajnością. Pobieranie większej ilości danych, niż w rzeczywistości potrzeba, może być użyteczne także w przypadku stosowania w aplikacji pewnego rodzaju buforowania lub po uwzględnieniu innych korzyści. Pobieranie i buforowanie pełnych obiektów może być bardziej wskazane niż wykonywanie wielu oddzielnych zapytań, które pobierają jedynie fragmenty obiektu.

Czy MySQL analizuje zbyt dużą ilość danych?

Po upewnieniu się, że zapytania *pobierają* jedynie potrzebne dane, można zająć się zapytaniami, które podczas generowania wyniku *analizują* zbyt wiele danych. W bazie danych MySQL najprostsze metody oceny kosztu zapytania to:

- czas wykonywania zapytania,
- liczba przeanalizowanych rekordów,
- liczba zwróconych rekordów.

Żadna z wymienionych miar nie jest doskonałym sposobem pomiaru kosztu zapytania, ale w przybliżeniu określają one ilość danych, do których MySQL musi wewnętrznie uzyskać dostęp, aby wykonać zapytanie. W przybliżeniu podają także szybkość wykonywania zapytania. Wszystkie trzy wymienione miary są rejestrowane w dzienniku wolnych zapytań. Dlatego też przejrzenie tego dziennika jest jednym z najlepszych sposobów wykrycia zapytań, które analizują zbyt wiele danych.

Czas wykonywania zapytania

Jak wspomniano w rozdziale 2., standardowa funkcja rejestrowania wolnych zapytań w MySQL 5.0 oraz wcześniejszych wersjach posiada wiele ograniczeń, m.in. brakuje obsługi bardziej szczegółowego poziomu rejestrowania. Na szczęście, istnieją poprawki pozwalające na rejestrowanie i analizowanie wolnych zapytań z dokładnością wyrażaną w mikrosekundach. Poprawki wprowadzono w MySQL 5.1, ale można je zastosować we wcześniejszych wersjach serwera, jeśli trzeba. Należy pamiętać, aby nie kłaść zbyt dużego nacisku na czas wykonywania zapytania. Warto traktować go jak miarę obiektywną, która nie zachowuje spójności w różnych warunkach obciążenia. Inne czynniki — takie jak blokady silnika magazynu danych (blokady tabeli i rekordów), wysoki poziom współbieżności i używany osprzęt komputerowy — również mogą mieć istotny wpływ na czas wykonywania zapytania. Miara ta będzie użyteczna podczas wyszukiwania zapytań, które najbardziej wpływają na czas udzielenia odpowiedzi przez aplikację i najbardziej obciążają serwer, ale nie odpowie na pytanie, czy rzeczywisty czas udzielenia odpowiedzi jest rozsądny dla zapytania o podanym stopniu złożoności. (Czas wykonywania zapytania może być zarówno symptomem, jak i źródłem problemów, i nie zawsze jest oczywiste, z którym przypadkiem mamy do czynienia).

Rekordy przeanalizowane i rekordy zwrócone

Podczas analizowania zapytań warto pochylić się nad liczbą rekordów sprawdzanych przez zapytanie, ponieważ dzięki temu można poznać efektywność zapytań w wyszukiwaniu potrzebnych danych. Jednak, podobnie jak w przypadku czasu wykonywania zapytania, nie jest to doskonała miara w trakcie wyszukiwania błędnych zapytań. Nie wszystkie operacje dostępu do rekordów są takie same. Krótsze rekordy pozwalają na szybszy dostęp, a pobieranie rekordów z pamięci jest znacznie szybsze niż ich odczytywanie z dysku twardego.

W idealnej sytuacji liczba przeanalizowanych rekordów powinna być równa liczbie zwróconych rekordów, ale w praktyce rzadko ma to miejsce. Przykładowo podczas budowania rekordów w operacjach złączeń w celu wygenerowania każdego rekordu zbioru wynikowego serwer musi uzyskać dostęp do wielu innych rekordów. Współczynnik liczby rekordów przeanalizowanych do liczby rekordów zwróconych zwykle jest mały — powiedzmy między 1:1 i 10:1 — ale czasami może być większy o rząd wielkości.

Rekordy przeanalizowane i rodzaje dostępu do danych

Podczas zastanawiania się nad kosztem zapytania trzeba rozważyć także koszt związany ze znalezieniem pojedynczego rekordu w tabeli. Baza danych może używać wiele metod dostępu pozwalających na odszukanie i zwrócenie rekordu. Niektóre z nich wymagają przeanalizowania wielu rekordów, podczas gdy inne mogą mieć możliwość wygenerowania wyniku bez potrzeby analizowania jakiegokolwiek rekordu.

Rodzaj metody (lub metod) dostępu jest wyświetlany w kolumnie `type` danych wyjściowych polecenia `EXPLAIN`. Zakres stosowanych rodzajów dostępu obejmuje zarówno pełne skanowanie tabeli, jak i skanowanie indeksu, a także skanowanie zakresu, wyszukiwanie unikalnego indeksu oraz stałych. Każda z nich jest szybsza od poprzedniej, ponieważ wymaga odczytu mniejszej ilości danych. Czytelnik nie musi uczyć się na pamięć metod dostępu, ale powinien zrozumieć ogólną koncepcję skanowania tabeli, skanowania indeksu, dostępu do zakresu oraz dostępu do pojedynczej wartości.

Jeżeli używana metoda dostępu jest nieodpowiednia, wówczas najlepszym sposobem rozwiązania problemu zwykle będzie dodanie właściwego indeksu. Szczegółowe omówienie indeksów przedstawiono w poprzednim rozdziale. Teraz widać, dlaczego indeksy są tak ważne podczas optymalizacji zapytań. Indeksy pozwalają bazie danych MySQL na wyszukiwanie rekordów za pomocą efektywniejszych metod dostępu, które analizują mniejszą ilość danych.

Warto np. spojrzeć na proste zapytanie do przykładowej bazy danych `Sakila`:

```
mysql> SELECT * FROM sakila.film_actor WHERE film_id = 1;
```

Powyższe zapytanie zwróci dziesięć rekordów, a polecenie `EXPLAIN` pokazuje, że w celu wykonania zapytania MySQL stosuje metodę dostępu `ref` względem indeksu `idx_fk_film`:

```
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** Rekord 1. ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: const
         rows: 10
        Extra:
```

Dane wyjściowe polecenia EXPLAIN pokazują, że baza danych MySQL oszacowała na dziesięć liczbę rekordów, do których musi uzyskać dostęp. Innymi słowy, optymalizator wiedział, że wybrana metoda dostępu jest wystarczająca w celu efektywnego wykonania zapytania. Co się stanie, jeżeli dla zapytania nie zostanie znaleziony odpowiedni indeks? Serwer MySQL może wykorzystać mniej optymalną metodę dostępu, o czym można się przekonać, usuwając indeks i ponownie wydając to samo polecenie:

```
mysql> ALTER TABLE sakila.film_actor DROP FOREIGN KEY fk_film_actor_film;
mysql> ALTER TABLE sakila.film_actor DROP KEY idx_fk_film_id;
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
*************************** Rekord 1. ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5073
        Extra: Using where
```

Zgodnie z przewidywaniami, metoda dostępu została zmieniona na pełne skanowanie tabeli (ALL) i baza danych MySQL oszacowała, że musi przeanalizować 5073 rekordy, aby wykonać zapytanie. Ciąg tekstowy „Using where" w kolumnie Extra wskazuje, że serwer MySQL używa klauzuli WHERE do odrzucenia rekordów po ich odczytaniu przez silnik magazynu danych.

Ogólnie rzecz biorąc, MySQL może zastosować klauzulę WHERE na trzy wymienione niżej sposoby, od najlepszego do najgorszego.

- Zastosowanie warunków w operacji przeszukiwania indeksu w celu wyeliminowania niepasujących rekordów. To zachodzi na poziomie silnika magazynu danych.
- Użycie indeksu pokrywającego (ciąg tekstowy „Using index" w kolumnie Extra) w celu uniknięcia bezpośredniego dostępu do rekordu i odfiltrowanie niepasujących rekordów po pobraniu każdego wyniku z indeksu. To zachodzi na poziomie serwera, ale nie wymaga odczytywania rekordów z tabeli.
- Pobranie rekordów z tabeli, a następnie odfiltrowanie niepasujących (ciąg tekstowy „Using where" w kolumnie Extra). To zachodzi na poziomie serwera i wymaga, aby serwer odczytał rekordy z tabeli przed rozpoczęciem ich filtrowania.

Powyższy przykład pokazuje więc, jak ważne jest tworzenie właściwych indeksów. Dobre indeksy pomagają zapytaniom w wyborze lepszej metody dostępu, a tym samym powodują analizowanie jedynie potrzebnych rekordów. Jednak dodanie indeksu nie zawsze oznacza, że baza danych MySQL uzyska dostęp i zwróci tę samą liczbę rekordów. Poniżej jako przykład przedstawiono zapytanie używające funkcji agregującej COUNT()[2]:

```
mysql> SELECT actor_id, COUNT(*) FROM sakila.film_actor GROUP BY actor_id;
```

Powyższe zapytanie zwróci jedynie 200 rekordów, ale w celu zbudowania zbioru wynikowego musi ich odczytać tysiące. W takim zapytaniu indeks nie zredukuje liczby analizowanych rekordów.

[2] Więcej informacji na ten temat przedstawiono w podrozdziale „Optymalizacja zapytań COUNT()", znajdującym się w dalszej części rozdziału.

Niestety, baza danych MySQL nie wskaże programiście liczby rekordów, do których uzyskała dostęp podczas budowy zbioru wynikowego, informuje jedynie o ogólnej liczbie rekordów, z których skorzystała. Wiele tych rekordów mogłoby zostać wyeliminowanych za pomocą klauzuli `WHERE`, a tym samym nie brałoby udziału w budowaniu zbioru wynikowego. W poprzednim przykładzie po usunięciu indeksu z tabeli `sakila.film_actor` zapytanie sprawdzało każdy rekord tabeli, a klauzula `WHERE` odrzuciła wszystkie, poza dziesięcioma. A więc pozostawione dziesięć rekordów utworzyło zbiór wynikowy. Zrozumienie, ile rekordów serwer przeanalizuje i ile faktycznie zostanie użytych do zbudowania zbioru wynikowego, wymaga umiejętności wyciągania wniosków z zapytania.

Jeżeli programista stwierdzi, że w celu zbudowania zbioru wynikowego obejmującego względnie małą liczbę rekordów jest analizowana duża liczba rekordów, wówczas można wypróbować bardziej zaawansowane techniki, czyli:

- użycie indeksów pokrywających przechowujących dane, wtedy silnik magazynu danych nie musi pobierać pełnych rekordów (indeksy pokrywające zostały omówione w poprzednim rozdziale),
- zmianę schematu; można np. zastosować tabele podsumowań (omówione w poprzednim rozdziale),
- przepisanie skomplikowanego zapytania, aby optymalizator MySQL mógł wykonać je w sposób optymalny (temat ten został przedstawiony w dalszej części rozdziału).

Sposoby restrukturyzacji zapytań

Podczas optymalizacji problematycznych zapytań celem powinno być odnalezienie alternatywnych sposobów otrzymania pożądanego wyniku — choć niekoniecznie oznacza to otrzymanie takiego samego wyniku z bazy danych MySQL. Czasami zapytania udaje przekształcić się tak, aby uzyskać jeszcze lepszą wydajność. Jednak warto także rozważyć napisanie zapytania od nowa w celu otrzymania innych wyników, jeśli przyniesie to znaczące korzyści w zakresie wydajności. Być może programista będzie mógł ostatecznie wykonać to samo zadanie poprzez zmianę zarówno kodu aplikacji, jak i zapytania. W podrozdziale zostaną przedstawione techniki, które mogą pomóc w restrukturyzacji szerokiego zakresu zapytań, a także przykłady, kiedy można zastosować każdą z omówionych technik.

Zapytanie skomplikowane kontra wiele mniejszych

Oto jedno z najważniejszych pytań dotyczących projektu: „Czy bardziej pożądane jest podzielenie zapytania skomplikowanego na kilka prostszych?”. Tradycyjne podejście do projektu bazy danych kładzie nacisk na wykonanie maksymalnej ilości pracy za pomocą minimalnej możliwej liczby zapytań. Takie podejście było w przeszłości uznawane za lepsze z powodu kosztu komunikacji sieciowej oraz obciążenia na etapie przetwarzania zapytania i optymalizacji.

Jednak rada ta nie zawsze jest właściwa w przypadku bazy danych MySQL, ponieważ została ona zaprojektowana w celu efektywnej obsługi operacji nawiązywania i zamykania połączenia oraz szybkiego udzielania odpowiedzi na małe i proste zapytania. Nowoczesne sieci są również znacznie szybsze niż w przeszłości, co zmniejsza ich opóźnienie. Serwer MySQL

może wykonywać ponad 50000 prostych zapytań na sekundę, korzystając z przeciętnego osprzętu komputerowego, oraz ponad 2000 zapytań na sekundę poprzez pojedynczy port sieciowy o przepustowości gigabitu. Dlatego też wykonywanie wielu zapytań niekoniecznie musi być złym rozwiązaniem.

Czas udzielenia odpowiedzi poprzez sieć nadal jest stosunkowo długi w porównaniu do liczby rekordów, które MySQL może wewnętrznie przekazywać w ciągu sekundy. Wymienioną liczbę szacuje się na milion w ciągu sekundy w przypadku danych znajdujących się w pamięci. Zatem nadal dobrym pomysłem jest stosowanie minimalnej liczby zapytań pozwalającej na wykonanie zadania. Jednak czasami zapytanie może być bardziej efektywne po rozłożeniu na części i wykonaniu kilku prostych zapytań zamiast jednego złożonego. Nie należy się obawiać tego rodzaju sytuacji, najlepiej ocenić koszty, a następnie wybrać strategię wymagającą mniejszego nakładu pracy. Przykłady takiej techniki zostaną zaprezentowane w dalszej części rozdziału.

Mając to na uwadze, warto pamiętać, że używanie zbyt wielu zapytań jest błędem często popełnianym w projekcie aplikacji. Przykładowo niektóre aplikacje wykonują dziesięć zapytań pobierających pojedynczy rekord danych z tabeli, zamiast użyć jednego pobierającego dziesięć rekordów. Autorzy spotkali się z aplikacjami pobierającymi oddzielnie każdą kolumnę, czyli wykonującymi wielokrotne zapytania do każdego rekordu!

Podział zapytania

Innym sposobem podziału zapytania jest technika „dziel i rządź", w zasadzie oznaczająca to samo, ale przeprowadzana w mniejszych „fragmentach", które każdorazowo wpływają na mniejszą liczbę rekordów.

Usuwanie starych danych to doskonały przykład. Okresowe zadania czyszczące mogą mieć do usunięcia całkiem sporą ilość danych, a wykonanie tego za pomocą jednego ogromnego zapytania może na bardzo długi czas zablokować dużą ilość rekordów, zapełnić dziennik zdarzeń transakcji, zużyć wszystkie dostępne zasoby oraz zablokować małe zapytania, których wykonywanie nie powinno być przerywane. Podział zapytania `DELETE` i użycie zapytań o średniej wielkości może znacząco wpłynąć na zwiększenie wydajności oraz zredukować opóźnienie podczas replikacji tego zapytania. Przykładowo zamiast wykonywania przedstawionego poniżej monolitycznego zapytania:

```
mysql> DELETE FROM messages WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH);
```

warto wykonać poniższy pseudokod:

```
rows_affected = 0
do {
   rows_affected = do_query(
      "DELETE FROM messages WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH)
      LIMIT 10000")
} while rows_affected > 0
```

Usunięcie jednorazowo dziesięciu tysięcy rekordów jest zazwyczaj na tyle dużym zadaniem, aby spowodować efektywne wykonanie każdego zapytania, i jednocześnie na tyle krótkim, aby zminimalizować jego wpływ na serwer[3] (silniki magazynu danych obsługujące transakcje mogą osiągnąć lepszą wydajność podczas wykonywania mniejszych zapytań). Dobrym rozwiązaniem

[3] Narzędzie *mk-archiver* z pakietu Maatkit bardzo łatwo wykonuje takie zadania.

może być również zastosowanie pewnego rodzaju przerwy między poleceniami DELETE. W ten sposób następuje rozłożenie obciążenia w czasie oraz zredukowanie okresu czasu, przez który są nałożone blokady.

Podział złączeń

Wiele witryn internetowych o wysokiej wydajności stosuje *podział złączeń*, który polega na rozdzieleniu jednego złączenia obejmującego wiele tabel na kilka zapytań obejmujących jedną tabelę, a następnie wykonaniu złączenia w aplikacji. I tak zamiast poniższego zapytania:

```
mysql> SELECT * FROM tag
    ->   JOIN tag_post ON tag_post.tag_id=tag.id
    ->   JOIN post ON tag_post.post_id=post.id
    -> WHERE tag.tag='mysql';
```

można wykonać następujące:

```
mysql> SELECT * FROM  tag WHERE tag='mysql';
mysql> SELECT * FROM  tag_post WHERE tag_id=1234;
mysql> SELECT * FROM  post WHERE  post.id in (123,456,567,9098,8904);
```

Na pierwszy rzut oka wygląda to na marnotrawstwo, ponieważ zwiększono liczbę zapytań bez otrzymania innych wyników. Jednak tego rodzaju restrukturyzacja może w rzeczywistości przynieść wyraźne korzyści w zakresie wydajności.

- Buforowanie może być efektywniejsze. Wiele aplikacji buforuje „obiekty", które mapują bezpośrednio do tabel. W przedstawionym powyżej przykładzie aplikacja pominie pierwsze zapytanie, jeżeli obiekt ze znacznikiem `mysql` jest już buforowany. Jeżeli w buforze znajdą się posty o wartości identyfikatora `id` wynoszącej 123, 567 lub 9098, wtedy można usunąć je z listy `IN()`. Bufor zapytania także może skorzystać na takiej strategii. Jeśli częstym zmianom ulega tylko jedna tabela, podział złączenia może zredukować liczbę nieprawidłowości w buforze.
- W tabelach MyISAM wykonywanie jednego zapytania na tabelę znacznie efektywniej stosuje blokady tabel: zapytania blokują tabele po kolei i na względnie krótki okres czasu, zamiast jednocześnie zablokować wszystkie na dłuższy okres czasu.
- Przeprowadzanie złączeń w aplikacji znacznie ułatwia skalowalność bazy danych poprzez umieszczenie tabel w różnych serwerach.
- Same zapytania również mogą być efektywniejsze. W powyższym przykładzie użycie listy `IN()` zamiast złączenia pozwala serwerowi MySQL na sortowanie identyfikatorów rekordów i bardziej optymalne pobieranie rekordów, niż byłoby to możliwe za pomocą złączenia. Zostanie to szczegółowo omówione w dalszej części rozdziału.
- Istnieje możliwość zmniejszenia liczby nadmiarowych operacji dostępu do rekordów. Przeprowadzenie złączenia w aplikacji oznacza, że każdy rekord jest pobierany tylko jednokrotnie, podczas gdy złączenie w zapytaniu w zasadzie jest denormalizacją, która może wymagać wielokrotnego dostępu do tych samych danych. Z tego samego powodu restrukturyzacja taka może też zredukować ogólny poziom ruchu sieciowego oraz zużycie pamięci.
- W pewnej mierze technikę tę można potraktować jako ręczną implementację złączenia typu hash zamiast algorytmu zagnieżdżonych pętli używanych przez MySQL do przeprowadzenia złączenia. Takie złączenie typu hash może być efektywniejsze. (Strategie złączeń w MySQL zostały przeanalizowane w dalszej części rozdziału).

Podsumowanie. Kiedy przeprowadzanie złączeń w aplikacji może być efektywniejsze?

Przeprowadzanie złączeń w aplikacji może być efektywniejsze, gdy:

- buforowana i ponownie używana jest duża ilość danych z poprzednich zapytań,
- używanych jest wiele tabel MyISAM,
- dane są rozproszone na wielu serwerach,
- w ogromnych tabelach złączenia są zastępowane listami `IN()`,
- złączenie odwołuje się wielokrotnie do tej samej tabeli.

Podstawy wykonywania zapytań

Jeżeli programiście zależy na osiągnięciu wysokiej wydajności działania serwera MySQL, jedną z najlepszych inwestycji będzie poznanie sposobów, w jakie MySQL optymalizuje i wykonuje zapytania. Po zrozumieniu tego zagadnienia większość procesów optymalizacji zapytania stanie się po prostu kwestią wyciągania odpowiednich wniosków, a sama optymalizacja zapytania okaże się procesem logicznym.

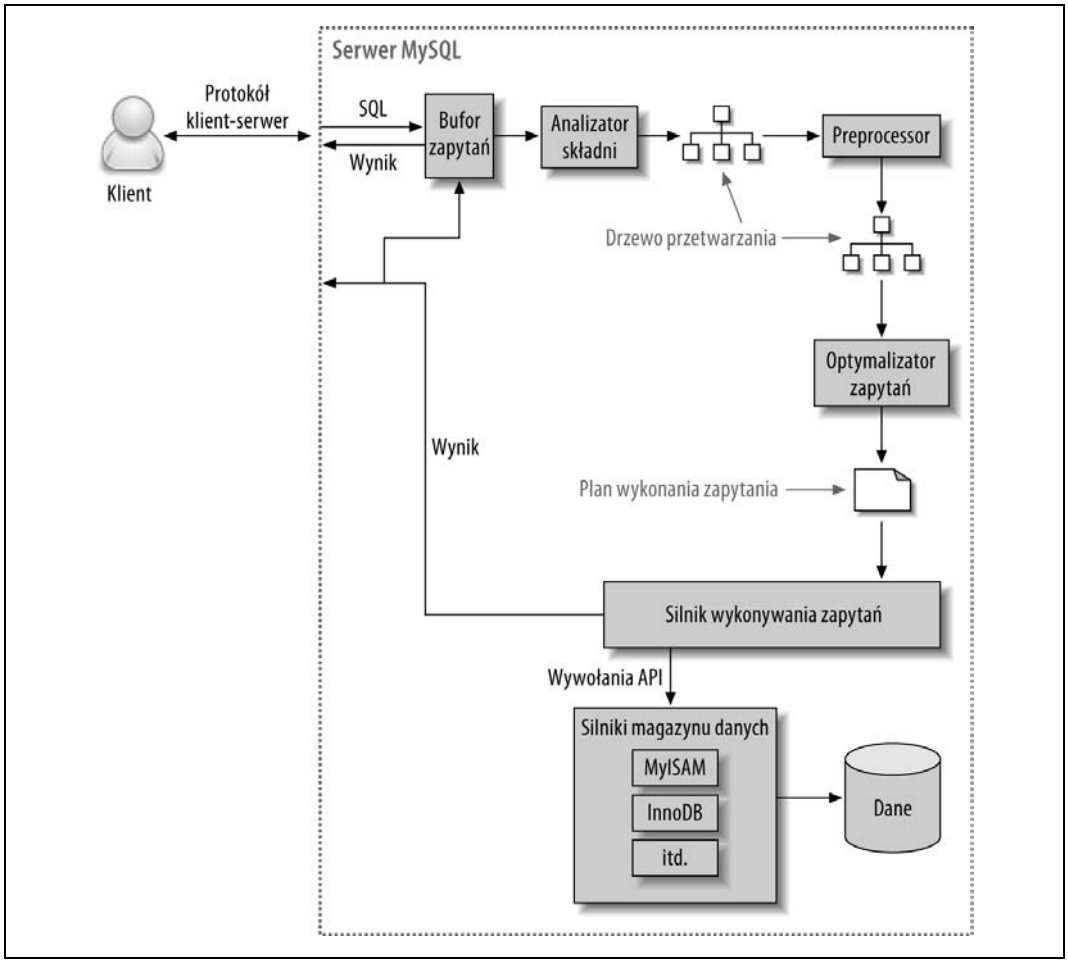
W poniższej analizie autorzy zakładają, że czytelnik zapoznał się z rozdziałem 2., w którym przedstawiono m.in. silniki wykonywania zapytań w MySQL i podstawy ich działania.

Na rysunku 4.1 pokazano ogólny sposób wykonywania zapytań przez MySQL.

Korzystając z rysunku, można zilustrować procesy zachodzące po wysłaniu zapytania do MySQL.

1. Klient wysyła polecenie SQL do serwera.
2. Serwer sprawdza bufor zapytań. Jeżeli dane zapytanie znajduje się w buforze, wyniki są pobierane z bufora. W przeciwnym razie polecenie SQL zostaje przekazane do kolejnego kroku.
3. Serwer analizuje, przetwarza i optymalizuje SQL na postać planu wykonania zapytania.
4. Silnik wykonywania zapytań realizuje plan poprzez wykonanie wywołań do API silnika magazynu danych.
5. Serwer zwraca klientowi wyniki zapytania.

Każdy z powyższych kroków wiąże się z pewnym poziomem złożoności, co będzie przeanalizowane w kolejnych podrozdziałach. Ponadto zostaną przedstawione stany, w których zapytanie znajduje się podczas realizacji poszczególnych kroków. Proces optymalizacji zapytania jest szczególnie złożony i jednocześnie najważniejszy do zrozumienia.



Rysunek 4.1. Ogólny sposób wykonywania zapytania w MySQL

Protokół klient-serwer MySQL

Chociaż nie jest konieczne zrozumienie wewnętrznych szczegółów protokołu klient-serwer MySQL, jednak trzeba zrozumieć jego działanie na wysokim poziomie. Protokół półdupleksowy, co oznacza, że w danej chwili serwer MySQL może albo wysyłać, albo odbierać komunikaty, ale nie jedno i drugie jednocześnie. Oznacza to także brak możliwości skrócenia komunikatu.

Protokół powoduje, że komunikacja MySQL jest prosta i szybka, ale równocześnie na pewne sposoby ją ogranicza. Z tego powodu brakuje kontroli przepływu — kiedy jedna strona wyśle komunikat, druga strona musi pobrać cały komunikat, zanim będzie mogła udzielić odpowiedzi. Przypomina to grę polegającą na rzucaniu piłki między uczestnikami: w danej chwili tylko jeden gracz ma piłkę, a więc inny gracz nie może rzucić piłką (wysłać komunikatu), zanim faktycznie jej nie otrzyma.

Klient wysyła zapytanie do serwera jako pojedynczy pakiet danych. To jest powód, dla którego konfiguracja zmiennej `max_packet_size` ma tak istotne znaczenie, gdy wykonywane są ogromne zapytania[4]. Po wysłaniu zapytania przez klienta piłka nie znajduje się już po jego stronie i może jedynie czekać na otrzymanie wyników.

Natomiast odpowiedź udzielana przez serwer, w przeciwieństwie do zapytania, zwykle składa się z wielu pakietów danych. Kiedy serwer udzieli odpowiedzi, klient musi otrzymać *cały* zbiór wynikowy. Nie może pobrać kilku rekordów, a następnie poprosić serwer o zaprzestanie wysyłania pozostałych. Jeżeli klientowi potrzebne jest jedynie kilka pierwszych rekordów ze zbioru wynikowego, to albo może poczekać na otrzymanie wszystkich pakietów wysłanych przez serwer i odrzucić niepotrzebne, albo w sposób nieeleganckі zerwać połączenie. Żadna z wymienionych możliwości nie jest dobrym rozwiązaniem i to kolejny powód, dla którego odpowiednie klauzule `LIMIT` mają tak istotne znaczenie.

Oto inny sposób przedstawienia tego procesu: kiedy klient pobiera rekordy z serwera, wtedy sądzi, że je *wyciąga*. Jednak w rzeczywistości to serwer MySQL *wypycha* rekordy podczas ich generowania. Klient jest jedynie odbiorcą wypchniętych rekordów, nie ma możliwości nakazania serwerowi, aby zaprzestał wysyłania rekordów. Używając innego porównania, można powiedzieć, że „klient pije z węża strażackiego". (Tak, to jest pojęcie techniczne).

Większość bibliotek nawiązujących połączenie z bazą danych MySQL pozwala na pobranie całego zbioru wynikowego i jego buforowanie w pamięci albo pobieranie poszczególnych rekordów, gdy będą potrzebne. Zazwyczaj zachowaniem domyślnym jest pobranie całego zbioru wynikowego i buforowanie go w pamięci. To jest bardzo ważne, ponieważ dopóki wszystkie rekordy nie zostaną dostarczone, dopóty serwer MySQL nie zwolni blokad oraz innych zasobów wymaganych przez dane zapytanie. Zapytanie będzie znajdowało się w stanie „Sending data" (stany zostaną omówione w kolejnym podrozdziale zatytułowanym „Stany zapytania"). Kiedy biblioteka klienta jednorazowo pobierze wszystkie rekordy, wtedy redukuje ilość pracy wykonywaną przez serwer: tzn. serwer może zakończyć wykonywanie zapytania i przeprowadzić czyszczenie po nim tak szybko, jak to możliwe.

Większość bibliotek klienckich pozwala na traktowanie zbioru wynikowego tak, jakby był pobierany z serwera. Jednak w rzeczywistości rekordy są pobierane z bufora w pamięci biblioteki. W większości sytuacji takie rozwiązanie sprawdza się doskonale, ale nie jest odpowiednie dla ogromnych zbiorów wynikowych, ponieważ ich pobranie zabiera dużo czasu oraz wymaga dużych ilości pamięci. Poprzez zakazanie bibliotece buforowania wyniku można użyć mniejszej ilości pamięci oraz szybciej rozpocząć pracę ze zbiorem wynikowym. Wadą takiego rozwiązania są blokady oraz inne zasoby serwera otwarte w czasie, kiedy aplikacja współdziała z biblioteką[5].

Warto spojrzeć na przykład w języku PHP. W poniższym kodzie pokazano, w jaki sposób najczęściej następuje wykonanie zapytania MySQL z poziomu PHP:

```
<?php
$link   = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_query('SELECT * FROM HUGE_TABLE', $link);
while ($row = mysql_fetch_array($result)) {
    // Dowolny kod przetwarzający wyniki zapytania.
}
?>
```

[4] Jeżeli zapytanie będzie zbyt duże, serwer odmówi przyjęcia kolejnych danych i nastąpi wygenerowanie błędu.

[5] Rozwiązaniem problemu może być opcja `SQL_BUFFER_RESULT`, która zostanie przedstawiona w dalszej części rozdziału.

Kod wydaje się wskazywać, że w pętli `while` rekordy są pobierane jedynie wtedy, gdy są potrzebne. Jednak w rzeczywistości za pomocą wywołania funkcji `mysql_query()` kod pobiera cały zbiór wynikowy i umieszcza go w buforze. Pętla `while` po prostu przechodzi przez poszczególne elementy bufora. Natomiast poniższy kod w ogóle nie buforuje wyników, ponieważ zamiast funkcji `mysql_query()` używa funkcji `mysql_unbuffered_query()`:

```
<?php
$link   = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_unbuffered_query('SELECT * FROM HUGE_TABLE', $link);
while ($row = mysql_fetch_array($result)) {
   // Dowolny kod przetwarzający wyniki zapytania.
}
?>
```

Języki programowania oferują różne sposoby uniknięcia buforowania. Przykładowo sterownik Perla `DBD:mysql` wymaga użycia atrybutu `mysql_use_result` w bibliotece języka C po stronie klienta (atrybutem domyślnym jest `mysql_buffer_result`). Poniżej przedstawiono przykład:

```
#!/usr/bin/perl
use DBI;
my $dbh = DBI->connect('DBI:mysql:;host=localhost', 'user', 'p4ssword');
my $sth = $dbh->prepare('SELECT * FROM HUGE_TABLE', { mysql_use_result => 1 });
$sth->execute();
while (my $row = $sth->fetchrow_array()) {
   # Dowolny kod przetwarzający wyniki zapytania.
}
```

Warto zwrócić uwagę, że wywołanie funkcji `prepare()` zakłada użycie wyniku zamiast jego buforowania. Można to również określić podczas nawiązywania połączenia, które spowoduje, że żadne polecenie nie będzie buforowane:

```
my $dbh = DBI->connect('DBI:mysql:;mysql_use_result=1', 'user', 'p4ssword');
```

Stany zapytania

Każde połączenie MySQL, czyli *wątek*, posiada stan wskazujący to, co dzieje się z nim w danej chwili. Istnieje kilka sposobów sprawdzenia tego stanu, ale najłatwiejszym pozostaje użycie polecenia `SHOW FULL PROCESSLIST` (stan jest wyświetlany w kolumnie `Command`). Wraz z postępem realizacji zapytania, czyli przechodzeniem przez cykl życiowy, stan zmienia się wielokrotnie, a samych stanów są dziesiątki. Podręcznik użytkownika MySQL jest odpowiednim źródłem informacji o wszystkich stanach, ale poniżej przedstawiono kilka z nich wraz z objaśnieniem znaczenia.

`Sleep`
: Wątek oczekuje na nowe zapytanie od klienta.

`Query`
: Wątek albo wykonuje zapytanie, albo odsyła klientowi wyniki danego zapytania.

`Locked`
: Wątek oczekuje na nałożenie blokady tabeli na poziomie serwera. Blokady, które są implementowane przez silnik magazynu danych, np. blokady rekordów w InnoDB, nie powodują przejścia wątku w stan `Locked`.

`Analyzing` *oraz* `statistics`
: Wątek sprawdza dane statystyczne silnika magazynu danych oraz optymalizuje zapytanie.

`Copying to tmp table [on disk]`
: Wątek przetwarza zapytanie oraz kopiuje wyniki do tabeli tymczasowej, prawdopodobnie ze względu na klauzulę `GROUP BY`, w celu posortowania lub spełnienia klauzuli `UNION`. Jeżeli nazwa stanu kończy się ciągiem tekstowym „on disk", wtedy MySQL konwertuje tabelę znajdującą się w pamięci na tabelę zapisaną na dysku twardym.

`Sorting result`
: Wątek sortuje zbiór wynikowy.

`Sending data`
: Ten stan może mieć kilka znaczeń: wątek może przesyłać dane między stanami zapytania, generować zbiór wynikowy bądź zwracać klientowi zbiór wynikowy.

Przydatna jest znajomość przynajmniej podstawowych stanów zapytania, aby można było złapać sens „po czyjej stronie jest piłka", czyli zapytanie. W przypadku bardzo obciążonych serwerów można zaobserwować, że niecodzienne lub zazwyczaj krótkotrwałe stany, np. `statistics`, zaczynają zabierać znaczące ilości czasu. Zwykle wskazuje to pewne nieprawidłowości.

Bufor zapytania

Przed rozpoczęciem przetwarzania zapytania MySQL sprawdza, czy dane zapytanie znajduje się w buforze zapytań, o ile został włączony. Operacja ta jest wyszukiwaniem typu hash, w którym ma znaczenie wielkość liter. Jeżeli zapytanie różni się od zapytania znalezionego w buforze nawet tylko o pojedynczy bajt, nie zostanie dopasowane i proces przetwarzania zapytania przejdzie do kolejnego etapu.

Jeżeli MySQL znajdzie dopasowanie w buforze zapytań, wówczas przed zwróceniem buforowanych wyników musi sprawdzić uprawnienia. Ta czynność jest możliwa bez przetwarzania zapytania, ponieważ MySQL wraz z buforowanym zapytaniem przechowuje tabelę informacyjną. Gdy uprawnienia są w porządku, MySQL pobiera z bufora przechowywany wynik zapytania i wysyła go klientowi, pomijając pozostałe etapy procesu wykonywania zapytania. To zapytanie nigdy nie będzie przetworzone, zoptymalizowane bądź wykonane.

Więcej informacji na temat bufora zapytań znajduje się w rozdziale 5.

Proces optymalizacji zapytania

Kolejny krok w cyklu życiowym zapytania powoduje zmianę zapytania SQL na postać planu wykonywania przeznaczoną dla silnika wykonywania zapytań. Krok ten ma kilka etapów pośrednich: analizowanie, przetwarzanie oraz optymalizację. Błędy (np. błędy składni) mogą być zgłoszone w dowolnym miejscu tego procesu. Autorzy w tym miejscu nie próbują udokumentować wnętrza bazy danych MySQL, a więc pozwolą sobie na pewną swobodę, np. opisywanie etapów oddzielnie, nawet jeśli często są ze sobą łączone w całość bądź częściowo, ze względu na wydajność. Celem autorów jest po prostu pomoc czytelnikowi w zrozumieniu, jak MySQL wykonuje zapytania oraz jak można utworzyć lepsze.

Analizator składni i preprocesor

Na początek *analizator* MySQL dzieli zapytanie na tokeny i na ich podstawie buduje „drzewo analizy". W celu interpretacji i weryfikacji zapytania analizator wykorzystuje gramatykę SQL bazy danych MySQL. Ten etap gwarantuje, że tokeny w zapytaniu są prawidłowe i znajdują się we właściwej kolejności. Ponadto następuje sprawdzenie pod kątem występowania błędów, takich jak ciągi tekstowe ujęte w cudzysłów, które nie zostały prawidłowo zakończone.

Następnie *preprocesor* weryfikuje otrzymane drzewo analizy pod kątem dodatkowej semantyki, której analizator nie mógł zastosować. Przykładowo preprocesor sprawdza istnienie tabel i kolumn, a także nazwy i aliasy, aby upewnić się, że odniesienie nie są dwuznaczne.

Kolejny etap to weryfikacja uprawnień przez preprocesor. Czynność zwykle jest bardzo szybka, chyba że serwer posiada ogromną liczbę uprawnień. (Więcej informacji na temat uprawnień i bezpieczeństwa znajduje się w rozdziale 12.).

Optymalizator zapytania

Na tym etapie drzewo analizy jest poprawne i przygotowane do tego, aby *optymalizator* przekształcił je na postać planu wykonywania zapytania. Zapytanie często może być wykonywane na wiele różnych sposobów, generując takie same wyniki. Zadaniem optymalizatora jest znalezienie najlepszej opcji.

Baza danych MySQL stosuje optymalizator kosztowy, co oznacza, że optymalizator próbuje przewidzieć koszt różnych wariantów planu wykonania i wybrać najtańszy. Jednostką kosztu jest odczytanie pojedynczej, losowo wybranej strony danych o wielkości czterech kilobajtów. Istnieje możliwość sprawdzenia oszacowanego przez optymalizator kosztu zapytania przeznaczonego do wykonania poprzez wyświetlenie wartości zmiennej `Last_query_cost`:

```
mysql> SELECT SQL_NO_CACHE COUNT(*) FROM sakila.film_actor;
+----------+
| count(*) |
+----------+
|     5462 |
+----------+
mysql> SHOW STATUS LIKE 'last_query_cost';
+-----------------+-------------+
| Variable_name   | Value       |
+-----------------+-------------+
| Last_query_cost | 1040.599000 |
+-----------------+-------------+
```

Powyższy wynik oznacza, że optymalizator oszacował liczbę losowo odczytywanych stron danych koniecznych do wykonania zapytania na 1040. Wynik jest obliczany na podstawie danych statystycznych: liczby stron w tabeli bądź indeksie, *liczebności* (liczby odmiennych wartości) indeksów, długości rekordów i kluczy oraz rozproszenia klucza. W trakcie obliczeń optymalizator nie bierze pod uwagę wpływu jakiegokolwiek mechanizmu buforowania — zakłada, że każdy odczyt będzie skutkował operacją I/O na dysku twardym.

Z wielu podanych niżej powodów optymalizator nie zawsze wybiera najlepszy plan.

- Dane statystyczne mogą być błędne. Serwer polega na danych statystycznych dostarczanych przez silnik magazynu danych, a one mogą znajdować się w zakresie od ściśle dokładnych aż do zupełnie niedokładnych. Przykładowo silnik InnoDB nie zawiera dokładnych danych statystycznych na temat liczby rekordów w tabeli, co jest związane z jego architekturą MVCC.

- Koszt metryczny nie zawsze dokładnie odpowiada rzeczywistemu kosztowi wykonania zapytania. Dlatego też nawet wtedy, kiedy dane statystyczne są dokładne, wykonanie zapytania może być mniej lub bardziej kosztowne, niż wynika to z obliczeń MySQL. W niektórych sytuacjach plan odczytujący większą liczbę stron może faktycznie być tańszy, np. gdy odczyt danych jest ciągły, ponieważ wtedy operacje I/O na dysku są szybsze, lub jeśli odczytywane strony zostały wcześniej buforowane w pamięci.
- Znaczenie optymalności dla MySQL nie musi pokrywać się z oczekiwaniami programisty. Programista prawdopodobnie dąży do osiągnięcia krótszego czasu wykonania zapytania, ale MySQL w rzeczywistości nie rozumie pojęcia „krótsze". Rozumie jednak pojęcie „koszt" i, jak wcześniej pokazano, oszacowanie kosztu nie zawsze jest nauką ścisłą.
- MySQL nie bierze pod uwagę innych zapytań wykonywanych w tym samym czasie, co jednak ma wpływ na szybkość wykonywania danego zapytania.
- MySQL nie zawsze wykorzystuje optymalizację na podstawie kosztu. Czasami po prostu stosuje się do reguł, np. takiej: „Jeśli w zapytaniu znajduje się klauzula `MATCH()` dopasowania pełnotekstowego, użyj indeksu `FULLTEXT`, o ile taki istnieje". Serwer wykona to nawet wtedy, kiedy szybszym rozwiązaniem będzie użycie innego indeksu oraz zapytanie innego rodzaju niż `FULLTEXT`, zawierające klauzulę `WHERE`.
- Optymalizator nie bierze pod uwagę kosztów operacji pozostających poza jego kontrolą, takich jak wykonanie procedur składowanych lub funkcji zdefiniowanych przez użytkownika.
- W dalszej części rozdziału zostanie pokazane, że optymalizator nie zawsze oszacowuje każdy możliwy plan wykonywania, a więc istnieje niebezpieczeństwo pominięcia planu optymalnego.

Optymalizator MySQL to bardzo skomplikowany fragment oprogramowania, który używa wielu optymalizacji w celu przekształcenia zapytania na postać planu wykonywania. Istnieją dwa rodzaje optymalizacji: *statyczna* i *dynamiczna*. *Optymalizacja statyczna* może być przeprowadzona po prostu przez badanie drzewa analizy. Przykładowo optymalizator może przekształcić klauzulę `WHERE` na odpowiadającą jej inną formę za pomocą reguł algebraicznych. Optymalizacja statyczna dotyczy wartości niezależnych, np. wartości stałej w klauzuli `WHERE`. Ten rodzaj optymalizacji może być przeprowadzony jednokrotnie i pozostanie ważny nawet wtedy, kiedy zapytanie zostanie ponownie wykonane z użyciem innych wartości. Optymalizację tę można traktować jak „optymalizację w trakcie kompilacji".

Optymalizacja dynamiczna, w przeciwieństwie do optymalizacji statycznej, bazuje na kontekście i może zależeć od wielu czynników, takich jak wartość w klauzuli `WHERE` lub liczba rekordów w indeksie. Ten rodzaj optymalizacji musi być przeprowadzany w trakcie każdego wykonywania zapytania. Optymalizację tę można więc traktować jako „optymalizację w trakcie wykonywania zapytania".

Różnica między nimi jest istotna podczas wykonywania przygotowanych poleceń lub procedur składowanych. Optymalizację statyczną MySQL może przeprowadzić tylko jednokrotnie, ale optymalizację dynamiczną musi powtarzać w trakcie każdego wykonywania zapytania. Czasami zdarza się również, że MySQL ponownie optymalizuje zapytanie już w trakcie jego wykonywania[6].

[6] Przykładowo sprawdzenie zakresu planu wykonywania ponownie określa indeksy dla każdego rekordu złączenia (`JOIN`). Ten plan wykonywania można zobaczyć, szukając ciągu tekstowego „range checked for each record" w kolumnie `Extra` danych wyjściowych polecenia `EXPLAIN`. Taki plan zapytania zwiększa także o jednostkę wartość zmiennej serwera o nazwie `Select_full_range_join`.

Poniżej przedstawiono kilka rodzajów optymalizacji, które MySQL może przeprowadzić.

Zmiana kolejności złączeń
: Tabele nie zawsze muszą być złączone w kolejności wskazanej w zapytaniu. Określenie najlepszej kolejność złączeń jest bardzo ważnym rodzajem optymalizacji. Temat ten został dokładnie omówiony w podrozdziale „Optymalizator złączeń", w tym rozdziale.

Konwersja klauzuli `OUTER JOIN` *na* `INNER JOIN`
: Klauzula `OUTER JOIN` niekoniecznie musi być wykonana jako `OUTER JOIN`. Pewne czynniki, np. klauzula `WHERE` i schemat tabeli, mogą w rzeczywistości powodować, że klauzula `OUTER JOIN` będzie odpowiadała klauzuli `INNER JOIN`. Baza danych MySQL rozpoznaje takie sytuacje i przepisuje złączenie, co pozwala na zmianę ustawień.

Zastosowanie algebraicznych odpowiedników reguł
: MySQL stosuje przekształcenia algebraiczne w celu uproszczenia wyrażeń i sprowadzenia ich do postaci kanonicznej. Baza danych może również zredukować zmienne, eliminując niemożliwe do zastosowania ograniczenia oraz warunki w postaci zdefiniowanych stałych. Przykładowo wyrażenie `(5=5 AND a>5)` zostanie zredukowane do zwykłego `a>5`. Podobnie `(a<b ANB b=c) AND a=5` stanie się wyrażeniem `b>5 AND b=c AND a=5`. Te reguły są bardzo użyteczne podczas tworzenia zapytań warunkowych, które zostaną omówione w dalszej części rozdziału.

Optymalizacja funkcji `COUNT()`, `MIN()` *oraz* `MAX()`
: Indeksy i kolumny akceptujące wartość `NULL` bardzo często mogą pomóc serwerowi MySQL w optymalizacji tych wyrażeń. Aby np. odnaleźć wartość minimalną kolumny wysuniętej najbardziej na lewo w indeksie B-Tree, MySQL może po prostu zażądać pierwszego rekordu indeksu. To może nastąpić nawet na etapie optymalizacji zapytania, a otrzymaną wartość serwer może potraktować jako stałą dla pozostałej części zapytania. Podobnie w celu znalezienia wartości maksymalnej w indeksie typu B-tree, serwer odczytuje ostatni rekord. Jeżeli serwer stosuje taką optymalizację, wówczas w danych wyjściowych polecenia `EXPLAIN` znajdzie się ciąg tekstowy „Select tables optimized away". Dosłownie oznacza to, że optymalizator usunął tabelę z planu wykonywania i zastąpił ją zmienną.

  Ponadto zapytania `COUNT(*)` bez klauzuli `WHERE` często mogą być optymalizowane w pewnych silnikach magazynu danych (np. MyISAM, który przez cały czas przechowuje dokładną liczbę rekordów tabeli). Więcej informacji na ten temat przedstawiono w podrozdziale „Optymalizacja zapytań COUNT()", znajdującym się w dalszej części rozdziału.

Określanie i redukowanie wyrażeń stałych
: Kiedy MySQL wykryje, że wyrażenie może zostać zredukowane na postać stałej, wtedy taka operacja będzie przeprowadzona w trakcie optymalizacji. Przykładowo zmienna zdefiniowana przez użytkownika może być skonwertowana na postać stałej, jeśli nie ulega zmianie w zapytaniu. Wyrażenia arytmetyczne to kolejny przykład.

  Prawdopodobnie największym zaskoczeniem jest fakt, że na etapie optymalizacji nawet wyrażenie uważane za zapytanie może być zredukowane na postać stałej. Jednym z przykładów jest funkcja `MIN()` w indeksie. Można to nawet rozciągnąć na wyszukiwanie stałej w kluczu podstawowym lub unikalnym indeksie. Jeżeli w takim indeksie klauzula `WHERE` stosuje warunek w postaci stałej, wówczas optymizator „wie", że MySQL może wyszukać wartość na początku zapytania. Wartość ta będzie traktowana jako stała w pozostałej części zapytania:

```
mysql> EXPLAIN SELECT film.film_id, film_actor.actor_id
    -> FROM sakila.film
    ->   INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id = 1;
+----+-------------+------------+-------+----------------+-------+------+
| id | select_type | table      | type  | key            | ref   | rows |
+----+-------------+------------+-------+----------------+-------+------+
|  1 | SIMPLE      | film       | const | PRIMARY        | const |    1 |
|  1 | SIMPLE      | film_actor | ref   | idx_fk_film_id | const |   10 |
+----+-------------+------------+-------+----------------+-------+------+
```

Powyższe zapytanie MySQL wykonuje w dwóch krokach, które odpowiadają dwóm rekordom danych wyjściowych. Pierwszym krokiem jest odszukanie pożądanego rekordu w tabeli `film`. Optymalizator MySQL wie, że to jest tylko jeden rekord, ponieważ kolumna `film_id` jest kluczem podstawowym. Poza tym, w trakcie optymalizacji zapytania indeks został już sprawdzony, aby przekonać się, ile rekordów będzie zwróconych. Ponieważ optymalizator znał ilość (wartość w klauzuli `WHERE`) używaną w zapytaniu, typ `ref` tej tabeli wynosi `const`.

W drugim kroku MySQL traktuje kolumnę `film_id` z rekordu znalezionego w pierwszym kroku jako znaną ilość. Optymalizator może przyjąć takie założenie, ponieważ wiadomo, że gdy zapytanie dotrze do drugiego kroku, otrzyma wszystkie wartości z poprzedniego kroku. Warto zwrócić uwagę, że typ `ref` tabeli `film_actor` wynosi `const`, podobnie jak dla tabeli `film`.

Inną sytuacją, w której można spotkać się z zastosowaniem warunku w postaci stałej, jest propagowanie wartości niebędącej stałą z jednego miejsca do innego, jeżeli występują klauzule `WHERE`, `USING` lub `ON` powodujące, że wartości są równe. W omawianym przypadku optymalizator przyjmuje, że klauzula `USING` wymusza, aby kolumna `film_id` miała taką samą wartość w każdym miejscu zapytania — musi być równa wartości stałej podanej w klauzuli `WHERE`.

Indeksy pokrywające
: Aby uniknąć odczytywania danych rekordów, MySQL może czasami użyć indeksu, ale indeks musi zawierać wszystkie kolumny wymagane przez zapytanie. Szczegółowe omówienie indeksów pokrywających przedstawiono w rozdziale 3.

Optymalizacja podzapytania
: Baza danych MySQL może skonwertować niektóre rodzaje podzapytań na bardziej efektywne, alternatywne formy, redukując je do wyszukiwań indeksu zamiast oddzielnych zapytań.

Wcześniejsze zakończenie zapytania
: MySQL może zakończyć przetwarzanie zapytania (lub etapu w zapytaniu), gdy tylko zostanie spełnione zapytanie albo jego etap. Oczywistym przykładem jest klauzula `LIMIT`, choć istnieje również kilka innych rodzajów wcześniejszego zakończenia zapytania. Jeżeli np. MySQL odkryje warunek niemożliwy do spełnienia, może przerwać wykonywanie całego zapytania. Taka sytuacja zachodzi w poniższym zapytaniu:

```
mysql> EXPLAIN SELECT film.film_id FROM sakila.film WHERE film_id = -1;
+----+...+------------------------------------------------------+
| id |...| Extra                                                |
+----+...+------------------------------------------------------+
|  1 |...| Impossible WHERE noticed after reading const tables |
+----+...+------------------------------------------------------+
```

Zapytanie zostało przerwane na etapie optymalizacji, ale w niektórych sytuacjach MySQL może przerwać wykonywanie zapytania również wcześniej. Serwer może wykorzystać ten rodzaj optymalizacji, kiedy silnik wykonywania zapytań stwierdzi, że musi pobrać zupełnie inne wartości lub wymagana wartość nie istnieje. Przedstawione poniżej przykładowe zapytanie ma wyszukać wszystkie filmy, w których nie ma aktorów[7]:

```
mysql> SELECT film.film_id
    -> FROM sakila.film
    ->   LEFT OUTER JOIN sakila.film_actor USING(film_id)
    -> WHERE film_actor.film_id IS NULL;
```

Powyższe zapytanie powoduje odrzucenie filmów, w których występują aktorzy. W każdym filmie może występować wielu aktorów, ale tuż po znalezieniu aktora następuje przerwanie przetwarzania bieżącego filmu i przejście do następnego. Dzieje się tak, ponieważ klauzula `WHERE` to informacja dla optymalizatora, że ma uniemożliwić wyświetlenie filmów, w których występują aktorzy. Podobny rodzaj optymalizacji, czyli „wartość odmienną lub nieistniejącą", można zastosować w określonych rodzajach zapytań `DISTINCT`, `NOT EXISTS()` oraz `LEFT JOIN`.

Propagowanie równości

Baza danych MySQL rozpoznaje, kiedy zapytanie zawiera dwie kolumny, które są jednakowe — np. w warunku `JOIN` — i propaguje użycie klauzuli `WHERE` na takich kolumnach. Warto spojrzeć na poniższe przykładowe zapytanie:

```
mysql> SELECT film.film_id
    -> FROM sakila.film
    ->   INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id > 500;
```

Serwer MySQL przyjmuje, że klauzula `WHERE` ma zastosowanie nie tylko względem tabeli `film`, ale również tabeli `film_actor`, ponieważ użycie klauzuli `USING` wymusiło dopasowanie tych dwóch kolumn.

Jeżeli byłby zastosowany inny serwer bazy danych, niewykonujący takiej czynności, programista mógłby zostać zachęcony do „udzielenia pomocy optymalizatorowi" poprzez ręczne podanie klauzuli `WHERE` dla obu tabel, np. w taki sposób:

```
... WHERE film.film_id > 500 AND film_actor.film_id > 500
```

W bazie danych MySQL jest to niepotrzebne. Taka modyfikacja powoduje, że zapytania stają się trudniejsze w obsłudze.

Porównania list IN()

W wielu serwerach baz danych `IN()` to po prostu synonim wielu klauzul `OR`, ponieważ pod względem logicznym obie konstrukcje są odpowiednikami. Nie dotyczy to bazy danych MySQL, która sortuje wartości w liście `IN()` oraz stosuje szybkie wyszukiwanie binarne w celu określenia, czy dana wartość znajduje się na liście. Jest to O(log *n*) w wielkości listy, podczas gdy odpowiednik serii klauzul `OR` to O(*n*) w wielkości listy (np. wyszukiwanie przeprowadzane jest znacznie wolniej w przypadku ogromnych list).

[7] Autorzy zgadzają się, że film bez aktorów jest czymś dziwnym. Jednak przykładowa baza danych Sakila „twierdzi", że w filmie *Slacker Liaisons* nie występują aktorzy. W opisie filmu można przeczytać „Dynamiczna opowieść o rekinie i studencie, który musi spotkać krokodyla w starożytnych Chinach".

Przedstawiona powyżej lista jest żałośnie niekompletna, ponieważ MySQL może przeprowadzić znacznie więcej rodzajów optymalizacji, niż zmieściłoby się w całym rozdziale. Lista powinna jednak pokazać stopień złożoności optymalizatora oraz trafność podejmowanych przez niego decyzji. Jeżeli czytelnik miałby zapamiętać tylko jedno z przedstawionej analizy, powinno to być zdanie: *Nie warto próbować być sprytniejszym od optymalizatora*. Taka próba może po prostu zakończyć się klęską bądź znacznym zwiększeniem stopniem skomplikowania zapytań, które nie przyniesie żadnych korzyści, a same zapytania staną się trudniejsze w obsłudze. Ogólnie rzecz biorąc, zadanie optymalizacji lepiej pozostawić optymalizatorowi.

Oczywiście, nadal istnieją sytuacje, w których optymalizator nie zapewni najlepszych wyników. Czasami programista ma wiedzę na temat danych, której nie ma optymalizator, np. wie, że gwarancja ich poprawności wynika z logiki aplikacji. Ponadto czasami optymalizator po prostu nie ma niezbędnej funkcjonalności, np. indeksów typu hash. Z kolei w innych przypadkach, jak już wspomniano, wskutek oszacowanych przez niego kosztów preferowany będzie plan wykonywania, który okaże się kosztowniejszy niż inne możliwości.

Jeżeli programista jest przekonany, że optymalizator nie wykonuje dobrze swojego zadania, i wie dlaczego, może spróbować mu pomóc. Niektóre dostępne możliwości obejmują dodanie wskazówki do zapytania, ponowne napisanie zapytania, przeprojektowanie schematu lub dodanie indeksów.

Dane statystyczne dotyczące tabeli i indeksu

Warto przypomnieć sobie różne warstwy w architekturze serwera MySQL, które zostały pokazane na rysunku 1.1. Warstwa serwera zawierająca optymalizator zapytań nie przechowuje danych statystycznych dotyczących danych i indeksów. To jest zadanie dla silników magazynu danych, ponieważ każdy silnik może przechowywać różne dane statystyczne (lub obsługiwać je w odmienny sposób). Niektóre silniki, np. Archive, w ogóle nie przechowują danych statystycznych!

Ponieważ serwer nie przechowuje danych statystycznych, optymalizator zapytań MySQL musi uzyskać od silnika dane statystyczne dotyczące tabel, które znajdują się w zapytaniu. Silnik może dostarczyć optymalizatorowi dane statystyczne, takie jak liczba stron w tabeli lub indeksie, liczebność tabel i indeksów, długość rekordów i kluczy oraz informacje o rozproszeniu klucza. Otrzymane dane statystyczne optymizator może wykorzystać podczas wyboru najlepszego planu wykonania zapytania. W kolejnych podrozdziałach pokazano, jak dane te wpływają na decyzje podejmowane przez optymalizator.

Strategia MySQL w trakcie wykonywania złączeń

Baza danych MySQL używa pojęcia „złączenie" w znacznie szerszym kontekście, niż można się spodziewać. Ogólnie rzecz ujmując, baza traktuje jak złączenie każde zapytanie — nie tylko zapytanie dopasowujące rekordy z dwóch tabel, ale wszystkie zapytania (łącznie z podzapytaniami, a nawet zapytaniami `SELECT` względem pojedynczej tabeli). W konsekwencji bardzo ważne jest, aby dokładnie zrozumieć, w jaki sposób serwer MySQL wykonuje złączenia.

Warto rozważyć przykład zapytania `UNION`. Serwer MySQL wykonuje klauzulę `UNION` jako serię zapytań, których wyniki są umieszczane w tabeli tymczasowej, a następnie ponownie z niej odczytywane. Według MySQL każde poszczególne zapytanie jest złączeniem — podobnie jak akt odczytania ich z wynikowej tabeli tymczasowej.

Na tym etapie strategia wykonywania złączeń przez MySQL jest prosta: każde złączenie jest traktowane jak zagnieżdżona pętla złączenia. Oznacza to, że baza danych MySQL wykonuje pętlę w celu wyszukania rekordu w tabeli, a następnie wykonuje zagnieżdżoną tabelę, szukając dopasowania rekordu w kolejnej tabeli. Proces jest kontynuowany aż do chwili znalezienia dopasowania rekordu w każdej tabeli złączenia. Następnym krokiem jest zbudowanie i zwrócenie rekordu z kolumn wymienionych na liście polecenia `SELECT`. Baza próbuje zbudować kolejny rekord poprzez znalezienie następnych dopasowanych rekordów w ostatniej tabeli. Jeżeli żaden nie zostanie znaleziony, wówczas baza wraca tą samą drogą do poprzedniej tabeli, szukając w niej kolejnych rekordów. Powrót trwa aż do chwili znalezienia dopasowanego rekordu w dowolnej tabeli. Wówczas następuje wyszukiwanie dopasowania w kolejnej tabeli itd.[8]

Proces wyszukiwania rekordów, sprawdzania kolejnej tabeli, a następnie powrotu może zostać zapisany w postaci zagnieżdżonych pętli w planie wykonywania — stąd nazwa „złączenia zagnieżdżonych pętli". Warto spojrzeć na poniższe proste zapytanie:

```
mysql> SELECT tbl1.col1, tbl2.col2
    -> FROM tbl1 INNER JOIN tbl2 USING(col3)
    -> WHERE tbl1.col1 IN(5,6);
```

Przy założeniu, że baza danych MySQL zadecyduje o złączeniu tabel w kolejności przedstawionej w zapytaniu, w poniższym pseudokodzie pokazano, jak baza danych MySQL mogłaby wykonać to zapytanie:

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row  = outer_iter.next
while outer_row
   inner_iter = iterator over tbl2 where col3 = outer_row.col3
   inner_row  = inner_iter.next
   while inner_row
      output [outer_row.col1, inner_row.col2]
      inner_row = inner_iter.next
   end
   outer_row = outer_iter.next
end
```

Powyższy plan wykonania ma zastosowanie zarówno do prostego zapytania pojedynczej tabeli, jak i zapytania obejmującego wiele tabel. Dlatego też nawet zapytania do pojedynczej tabeli mogą być uznawane za złączenia — złączenia w pojedynczych tabelach są prostymi operacjami, które składają się na bardziej złożone złączenia. Obsługiwane są również klauzule `OUTER JOIN`. Przykładowo zapytanie można zmienić na następującą postać:

```
mysql> SELECT tbl1.col1, tbl2.col2
    -> FROM tbl1 LEFT OUTER JOIN tbl2 USING(col3)
    -> WHERE tbl1.col1 IN(5,6);
```

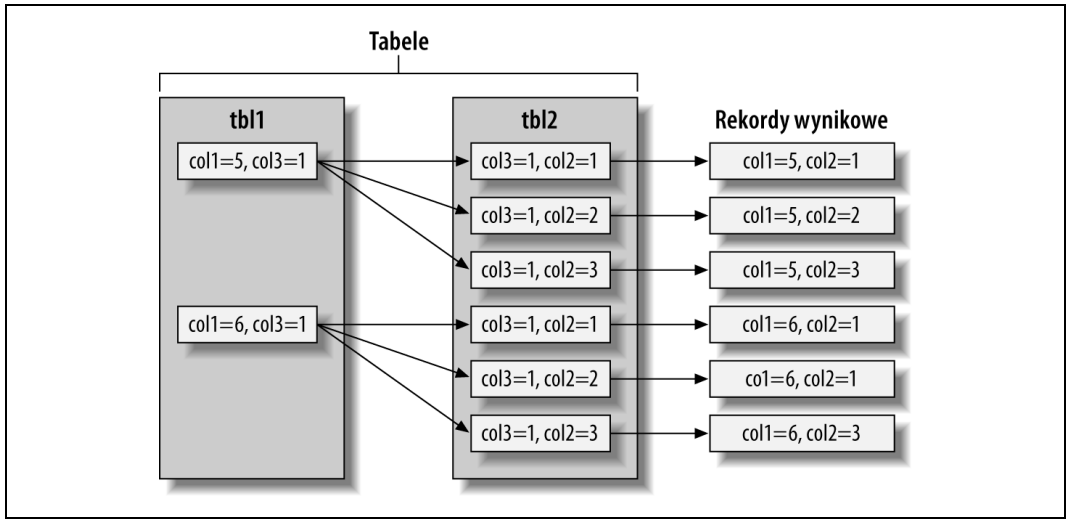
Poniżej znajduje się odpowiadający mu pseudokod, w którym zmienione fragmenty zostały pogrubione:

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row  = outer_iter.next
while outer_row
   inner_iter = iterator over tbl2 where col3 = outer_row.col3
   inner_row  = inner_iter.next
   if inner_row
```

[8] Jak to zostało pokazane w dalszej części rozdziału, wykonywanie zapytania MySQL nie jest takie proste. Istnieje wiele różnych optymalizacji komplikujących ten proces.

```
        while inner_row
           output [outer_row.col1, inner_row.col2]
           inner_row = inner_iter.next
        end
     else
        output [outer_row.col1, NULL]
     end
     outer_row = outer_iter.next
  end
```

Innym sposobem wizualizacji planu wykonania zapytania jest użycie tego, co osoby zajmujące się optymalizacją nazywają „wykres swim-lane". Na rysunku 4.2 pokazano wykres swim-lane dotyczący początkowego zapytania `INNER JOIN`. Wykres odczytuje się od lewej do prawej strony, od góry do dołu.



Rysunek 4.2. Wykres swim-lane pokazujący pobieranie rekordów za pomocą złączenia

W zasadzie serwer MySQL wykonuje każdy rodzaj zapytania w taki sam sposób. Przykładowo podzapytania w klauzuli `FROM` są wykonywane w pierwszej kolejności, a ich wyniki zostają umieszczone w tabeli tymczasowej[9]. Następnie tabela tymczasowa jest traktowana jak zwykła tabela (stąd nazwa „tabela pochodna"). W zapytaniach `UNION` baza danych MySQL także stosuje tabele tymczasowe, a wszystkie zapytania `RIGHT OUTER JOIN` są przepisywane na ich odpowiedniki `LEFT OUTER JOIN`. W skrócie mówiąc, MySQL zmusza każdy rodzaj zapytania do „wpasowania się" w przedstawiony plan wykonywania.

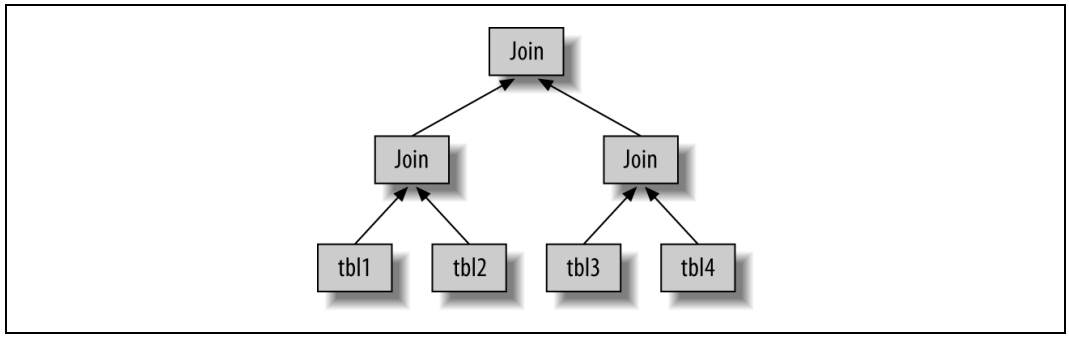
Jednak niemożliwe jest wykonanie w ten sposób każdego poprawnego zapytania SQL. Przykładowo zapytanie `FULL OUTER JOIN` nie może być wykonane za pomocą zagnieżdżonych pętli oraz powracania po dotarciu do tabeli, w której nie znaleziono dopasowanych rekordów, ponieważ zapytanie może rozpoczynać się od tabeli nieposiadającej pasujących rekordów. To wyjaśnia, dlaczego MySQL nie obsługuje zapytań `FULL OUTER JOIN`. Nadal wszystkie pozostałe zapytania mogą być wykonywane za pomocą zagnieżdżonych pętli, ale wynik takich operacji jest opłakany. Więcej informacji na ten temat znajduje się w dalszej części rozdziału.

[9] W tabeli tymczasowej nie ma indeksów i należy o tym pamiętać podczas tworzenia skomplikowanych złączeń względem podzapytań w klauzuli `FROM`. Dotyczy to również zapytań `UNION`.

Plan wykonywania

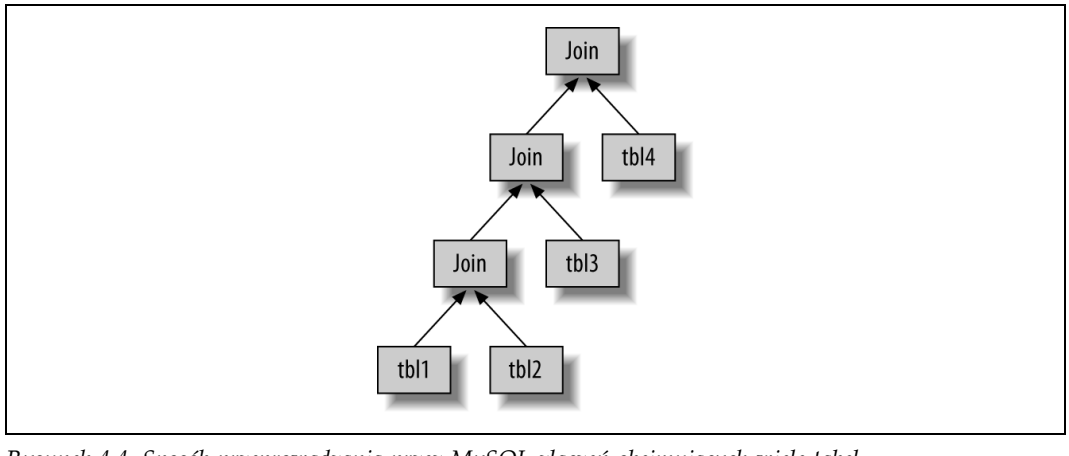
Baza danych MySQL nie generuje kodu bajtowego w celu wykonania zapytania, co ma miejsce w wielu innych bazach danych. Plan wykonania zapytania w rzeczywistości jest drzewem instrukcji, które silnik wykonywania zapytania realizuje w celu otrzymania wyniku zapytania. Plan ostateczny zawiera ilość informacji wystarczającą do zrekonstruowania początkowego zapytania. Jeżeli zapytanie jest wykonywane z użyciem polecenia `EXPLAIN EXTENDED` poprzedzonego przez `SHOW WARNINGS`, wówczas można zobaczyć zrekonstruowane zapytanie[10].

Każde zapytanie obejmujące więcej niż jedną tabelę może zostać przedstawione jako drzewo. Przykładowo zapytanie obejmujące operację złączenia czterech tabel można wykonać tak, jak pokazano na rysunku 4.3.



Rysunek 4.3. Jeden ze sposobów przeprowadzenia operacji złączenia na wielu tabelach

Naukowcy nazywają je *drzewem zrównoważonym*. Jednak nie jest to sposób, w jaki MySQL wykonuje zapytanie. Jak wspomniano w poprzednim podpunkcie, baza danych MySQL zawsze rozpoczyna wykonywanie zapytania od jednej tabeli i wyszukuje pasujące rekordy w kolejnej. Dlatego też plan wykonywania zapytania w MySQL zawsze przybiera postać *drzewa lewostronnie zagnieżdżonego*, co pokazano na rysunku 4.4.



Rysunek 4.4. Sposób przeprowadzania przez MySQL złączeń obejmujących wiele tabel

[10] Serwer generuje dane wyjściowe na podstawie planu wykonania zapytania. Dlatego też znajduje się w nich taka sama semantyka jak w zapytaniu początkowym, ale niekoniecznie ten sam tekst.

Optymalizator złączeń

Najważniejszą częścią optymalizatora zapytań MySQL jest *optymalizator złączeń*, który decyduje o najlepszej kolejności wykonywania zapytań obejmujących wiele tabel. Zazwyczaj operacje złączeń tabel można przeprowadzić w odmiennej kolejności, wciąż otrzymując te same wyniki. Optymalizator złączeń oszacowuje koszt różnych planów, a następnie stara się wybrać najtańszy i dający te same wyniki.

Poniżej przedstawiono zapytanie, którego tabele mogą zostać złączone w różnej kolejności bez zmiany otrzymanych wyników:

```
mysql> SELECT film.film_id, film.title, film.release_year, actor.actor_id,
    ->    actor.first_name, actor.last_name
    ->    FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    ->    INNER JOIN sakila.actor USING(actor_id);
```

Czytelnik prawdopodobnie myśli o kilku różnych planach wykonania zapytania. Przykładowo baza danych MySQL mogłaby rozpocząć od tabeli `film`, użyć indeksu obejmującego kolumny `film_id` i `film_actor` w celu wyszukania wartości `actor_id`, a następnie znaleźć rekordy w kluczu podstawowym tabeli `actor`. Takie rozwiązanie byłoby efektywne, nieprawdaż? Warto więc użyć polecenia `EXPLAIN` i przekonać się, w jaki sposób baza danych MySQL wykonuje to zapytanie:

```
*************************** Rekord 1. ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 200
        Extra:
*************************** Rekord 2. ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY,idx_fk_film_id
          key: PRIMARY
      key_len: 2
          ref: sakila.actor.actor_id
         rows: 1
        Extra: Using index
*************************** Rekord 3. ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.film_actor.film_id
         rows: 1
        Extra:
```

Jak widać, to nieco odmienny plan od zasugerowanego w poprzednim akapicie. Baza danych MySQL rozpoczyna od tabeli `actor` (wiemy o tym, ponieważ została wyświetlona w pierwszej grupie danych wyjściowych polecenia `EXPLAIN`), a następnie porusza się w odwrotnej kolejności. Czy takie rozwiązanie naprawdę jest efektywne? Warto to sprawdzić. Słowo kluczowe `STRAIGHT_JOIN` wymusza przeprowadzanie operacji złączeń w kolejności określonej przez zapytanie. Poniżej przedstawiono dane wyjściowe polecenia `EXPLAIN` dla zmodyfikowanego zapytania:

```
mysql> EXPLAIN SELECT STRAIGHT_JOIN film.film_id...\G
*************************** Rekord 1. ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra:
*************************** Rekord 2. ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: PRIMARY,idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: sakila.film.film_id
         rows: 1
        Extra: Using index
*************************** Rekord 3. ***************************
           id: 1
  select_type: SIMPLE
        table: actor
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 2
          ref: sakila.film_actor.actor_id
         rows: 1
        Extra:
```

To pokazuje, dlaczego MySQL stosuje odwrotną kolejność złączeń: dzięki temu serwer musi przeanalizować mniejszą ilość rekordów w pierwszej tabeli[11]. W obu przypadkach możliwe jest przeprowadzenie szybkich wyszukiwań indeksu w tabelach drugiej i trzeciej. Różnica dotyczy liczby wymienionych wyszukiwań indeksu, które muszą zostać wykonane.

- Użycie tabeli `film` jako pierwszej wymaga 951 prób w tabelach `film_actor` i `actor`, po jednej dla każdego rekordu w pierwszej tabeli.
- Jeżeli serwer jako pierwszą skanuje tabelę `actor`, w kolejnych tabelach będzie musiał wykonać tylko około dwustu wyszukiwań indeksu.

[11] A mówiąc dokładniej, baza danych MySQL nie próbuje zredukować liczby odczytywanych rekordów. Zamiast tego próbuje przeprowadzić optymalizację pozwalającą na odczyt mniejszej liczby stron. Liczba rekordów często może w przybliżeniu podać koszt zapytania.

Innymi słowy, odwrotna kolejność wykonywania złączeń będzie wymagała mniejszej liczby operacji powrotów oraz ponownego odczytu. Aby dwukrotnie sprawdzić wybór optymalizatora, autorzy wykonali dwie wersje zapytania i sprawdzili wartość zmiennej `Last_query_cost` dla każdego z nich. Oszacowany koszt zmodyfikowanego zapytania wyniósł 241, podczas gdy oszacowany koszt zapytania wymuszającego zachowanie kolejności złączeń wyniósł 1154.

To prosty przykład na to, jak optymalizator złączeń w MySQL może zmienić kolejność operacji w zapytaniu, aby jego wykonanie stało się tańsze. Zmiana kolejności operacji złączeń zwykle jest bardzo efektywną formą optymalizacji. Zdarzają się sytuacje, w których optymalizator nie wybiera optymalnego planu, ale wówczas można użyć słowa kluczowego `STRAIGHT_JOIN` oraz napisać zapytanie w kolejności uznawanej przez programistę za najlepszą. Takie sytuacje jednak występują bardzo rzadko. W większości przypadków optymalizator złączeń wygrywa z człowiekiem.

Optymalizator złączeń próbuje zbudować drzewo planu wykonania zapytania o najniższym możliwym do zaakceptowania koszcie. Kiedy będzie to możliwe, analizuje wszystkie potencjalne kombinacje poddrzew, a rozpoczyna od wszystkich planów obejmujących jedną tabelę.

Niestety, operacja złączeń *n* tabel będzie miała *n!* (silnia) kombinacji kolejności złączeń do przeanalizowania. Współczynnik ten nosi nazwę *przestrzeni przeszukiwana* dla wszystkich możliwych planów zapytania i zwiększa się bardzo szybko. Złączenie dziesięciu tabel może być przeprowadzone na maksymalnie 3628800 różnych sposobów! Kiedy przestrzeń przeszukiwania za bardzo się rozrasta, wtedy optymalizacja zapytania może wymagać zbyt dużej ilości czasu. Serwer zatrzymuje więc wykonywanie pełnej analizy. W zamian stosuje skróty, np. szukanie „zachłanne", kiedy liczba tabel przekroczy wartość graniczną ustaloną przez zmienną `optimizer_search_depth`.

Baza danych MySQL dysponuje zgromadzonymi w czasie lat badań i eksperymentów wieloma algorytmami heurystycznymi, które są używane do przyśpieszenia działania fazy optymalizacji. Wprawdzie jest to korzystne, ale równocześnie oznacza, że serwer MySQL może (sporadycznie) pominąć plan optymalny i wybrać nieco mniej optymalny, ponieważ nie będzie próbował przeanalizować każdego możliwego do wykonania planu.

Zdarzają się sytuacje, gdy zmiana kolejności w zapytaniu jest niemożliwa. Optymalizator złączeń może wykorzystać ten fakt do zmniejszenia przestrzeni przeszukiwania poprzez eliminację określonych rozwiązań. Klauzula `LEFT JOIN` jest dobrym przykładem, ponieważ składa się ze skorelowanych podzapytań (więcej informacji na ten temat znajduje się w dalszej części rozdziału). Wiąże się to z faktem, że wyniki jednej tabeli zależą od danych otrzymanych z innej. Tego rodzaju zależności pomagają optymalizatorowi złączeń w redukcji przeszukiwanej przestrzeni poprzez eliminację pewnego rodzaju rozwiązań.

Optymalizacja sortowania

Sortowanie wyników może być kosztowną operacją, stąd często można poprawić wydajność poprzez unikanie sortowania bądź sortowanie mniejszej liczby rekordów.

Wykorzystanie indeksów podczas sortowania zostało przedstawione w rozdziale 3. Kiedy baza danych MySQL nie może użyć indeksu w celu zbudowania posortowanego wyniku, wówczas samodzielnie musi posortować rekordy. Operację można przeprowadzić w pamięci lub na dysku twardym, ale proces zawsze nosi nazwę *sortowania pliku*, nawet jeśli w rzeczywistości nie używa pliku.

Jeżeli wartości przeznaczone do sortowania mieszczą się w buforze sortowania, MySQL może przeprowadzić sortowanie całkowicie w pamięci za pomocą *szybkiego sortowania*. Jeśli MySQL nie może wykonać sortowania w pamięci, wtedy operacja jest przeprowadzana na dysku poprzez sortowanie wartości fragmentami. Proces wykorzystuje szybkie sortowanie w celu posortowania każdego fragmentu, a każdy posortowany fragment jest dołączany do wyniku.

Istnieją dwa algorytmy sortowania pliku.

Dwuprzebiegowy (stary)
: Odczytuje wskaźniki rekordów oraz kolumny `ORDER BY`, sortuje je, a następnie skanuje posortowaną listę i ponownie odczytuje rekordy w celu utworzenia danych wyjściowych. Algorytm dwuprzebiegowy może być bardzo kosztowny, ponieważ rekordy są odczytywane z tabeli dwukrotnie, a drugi odczyt powoduje wykonanie dużej ilości losowych operacji I/O. Proces jest szczególnie kosztowny w silniku MyISAM, który do pobrania każdego rekordu używa wywołań systemowych (ponieważ MyISAM polega na buforze systemu operacyjnego przechowującego dane). Jednak w trakcie sortowania przechowywana jest minimalna ilość danych. Tak więc rekordy są sortowane całkowicie w pamięci, a przechowywanie mniejszej ilości danych i ponowny odczyt rekordów w celu wygenerowania wyniku końcowego może okazać się tańszym rozwiązaniem.

Jednoprzebiegowy (nowy)
: Odczytuje wszystkie kolumny wymagane przez zapytanie, sortuje je według kolumn `ORDER BY`, a następnie skanuje posortowaną listę i wyświetla dane wyjściowe wskazanych kolumn.

  Ten algorytm sortowania jest dostępny jedynie w MySQL 4.1 i nowszych. Może być znacznie efektywniejszy zwłaszcza dla ogromnych zbiorów danych opierających się na operacjach I/O, ponieważ unika dwukrotnego odczytywania rekordów z tabel i zastępuje losowe operacje I/O bardziej ciągłymi operacjami I/O. Jednak potencjalnie może używać dużej ilości miejsca, ponieważ przechowuje wszystkie pożądane kolumny z każdego rekordu, a nie tylko kolumny wymagane do posortowania rekordów. Oznacza to, że mniejsza ilość zbiorów elementów zmieści się w buforze sortowania, a samo sortowanie pliku będzie musiało wykonać więcej operacji łączenia wyników sortowania.

MySQL może na potrzeby sortowania pliku wykorzystywać znacznie więcej przestrzeni tymczasowej, niż można przypuszczać, bo przy sortowaniu każdego zbioru elementów alokuje rekord o stałej długości. Te rekordy są na tyle olbrzymie, aby pomieścić największy możliwy zbiór danych, łącznie z każdą kolumną `VARCHAR` o pełnej długości. Ponadto podczas stosowania kodowania UTF-8 serwer MySQL alokuje trzy bajty dla każdego znaku. Autorzy spotkali się z kiepsko zoptymalizowanymi schematami, które powodowały, że przestrzeń tymczasowa używana podczas sortowania była wielokrotnie większa niż wielkość całej tabeli na dysku twardym.

Podczas sortowania złączeń baza danych MySQL może w trakcie wykonywania zapytania przeprowadzić sortowanie pliku na dwóch etapach. Jeżeli klauzula `ORDER BY` odnosi się jedynie do kolumn w pierwszej tabeli złączenia, MySQL może przeprowadzić sortowanie tej tabeli, a następnie wykonać operację złączenia. W takim przypadku dane wyjściowe polecenia `EXPLAIN` zawierają w kolumnie `Extra` ciąg tekstowy „Using filesort". W przeciwnym razie baza danych MySQL musi przechowywać wynik zapytania w tabeli tymczasowej, a następnie

przeprowadzić na niej sortowanie pliku po zakończeniu operacji złączenia. W takim przypadku dane wyjściowe polecenia `EXPLAIN` zawierają w kolumnie `Extra` ciąg tekstowy „Using temporary; Using filesort". Jeżeli w zapytaniu znajduje się klauzula `LIMIT`, będzie zastosowana po operacji sortowania pliku, a więc tabela tymczasowa może być ogromna, natomiast operacja sortowania pliku bardzo kosztowna.

Więcej informacji na temat dostrajania serwera pod kątem sortowania pliku oraz wpływu używanego przez serwer algorytmu przedstawiono w rozdziale 6., w podrozdziale „Optymalizacja sortowania pliku".

Silnik wykonywania zapytań

W wyniku etapu analizy i optymalizacji powstaje plan wykonania zapytania, który silnik wykonywania zapytań MySQL stosuje w celu przetworzenia zapytania. Plan jest strukturą danych, a nie wykonywalnym kodem bajtowym, jak ma to miejsce w wielu innych bazach danych.

Faza wykonania zapytania, w przeciwieństwie do fazy optymalizacji, zwykle nie jest tak skomplikowana: MySQL po prostu podąża za instrukcjami znajdującymi się w planie wykonania zapytania. Większość operacji w planie polega na wywołaniu metod zaimplementowanych przez interfejs silnika magazynu danych, znany również pod nazwą *API procedury obsługi*. Każda tabela w zapytaniu jest przedstawiana jako egzemplarz procedury obsługi. Jeżeli np. tabela występuje w zapytaniu trzykrotnie, serwer utworzy trzy egzemplarze procedury obsługi. Chociaż nie było to eksponowane wcześniej, trzeba powiedzieć, że MySQL w rzeczywistości tworzy egzemplarze procedury obsługi wcześniej, w fazie optymalizacji. Optymalizator wykorzystuje je w celu pobrania informacji dotyczących tabel, np. nazw kolumn oraz danych statystycznych indeksu.

Interfejs silnika magazynu danych ma dużą liczbę funkcji, ale do wykonania większości zapytań wymaga jedynie około dziesięciu operacji typu „bloku budulcowego". Przykładowo istnieje operacja służąca do odczytania pierwszego rekordu z indeksu oraz inna operacja do odczytania kolejnego rekordu z indeksu. To wystarczy w zapytaniu, które przeprowadza skanowanie indeksu. Taka uproszczona metoda wykonywania zapytania jest możliwa dzięki architekturze silnika magazynu danych MySQL, ale jednocześnie nakłada pewne ograniczenia w zakresie optymalizacji, co zostało opisane wcześniej.

Nie wszystko jest operacją procedury obsługi. Przykładowo to serwer zarządza blokadami tabel. Procedura obsługi może implementować własne blokowanie niższego poziomu, tak jak w przypadku InnoDB i jego blokowania na poziomie rekordu, ale to nie zastępuje własnej implementacji blokowania znajdującej się w serwerze. Jak wyjaśniono w rozdziale 1., wszystko, co jest współdzielone przez wszystkie silniki magazynu danych, zostało zaimplementowane w serwerze, np. funkcje daty i godziny, widoki i wyzwalacze.

W celu wykonania zapytania serwer po prostu powtarza instrukcje aż do chwili, gdy nie będzie kolejnych rekordów do przeanalizowania.

Zwrot klientowi wyników zapytania

Ostatnim krokiem w trakcie wykonywania zapytania jest zwrot wyników klientowi. Nawet zapytania niezwracające zbioru wynikowego też udzielają klientowi odpowiedzi, która zawiera informacje o zapytaniu, np. o liczbie rekordów, których dotyczyło zapytanie.

Jeżeli zapytanie można buforować, na tym etapie baza danych MySQL umieści jego wyniki w buforze zapytania.

Serwer generuje i wysyła wyniki w sposób przyrostowy. Warto w tym miejscu przypomnieć sobie omówioną wcześniej metodę wielu złączeń. Kiedy baza danych MySQL przetworzy ostatnią tabelę i z powodzeniem wygeneruje rekord, może i powinna wysłać ten rekord klientowi. Takie rozwiązanie niesie ze sobą dwie korzyści: umożliwia serwerowi uniknięcie konieczności przechowywania rekordu w pamięci oraz oznacza, że klient będzie otrzymywał wyniki tak szybko, jak to możliwe[12].

Ograniczenia optymalizatora zapytań MySQL

Podejście MySQL, określane mianem „wszystko jest złączeniem zagnieżdżonych pętli", stosowane podczas wykonywania zapytań nie jest idealnym rozwiązaniem optymalizacji każdego rodzaju zapytań. Na szczęście, istnieje tylko niewielka liczba sytuacji, w których optymalizator zapytań MySQL się nie sprawdza. Zazwyczaj możliwe jest wówczas ponowne napisanie zapytań, aby działały znacznie efektywniej.

Informacje przedstawione w tym podrozdziale mają zastosowanie do wersji serwera MySQL dostępnych w trakcie pisania książki, czyli do wersji 5.1 MySQL. Niektóre z omówionych zapytań prawdopodobnie zostaną złagodzone lub całkowicie usunięte w przyszłych wersjach serwera. Część została już poprawiona i umieszczona w wersjach, które nie są jeszcze oficjalnie dostępne. W kodzie źródłowym MySQL 6 wprowadzono pewną liczbę optymalizacji podzapytań, a nad wieloma kolejnymi trwają prace.

Podzapytania skorelowane

MySQL czasami kiepsko optymalizuje podzapytania. Najtrudniej poddają się optymalizacji podzapytania `IN()` w klauzuli `WHERE`. Przeanalizujmy przykładowe zapytanie mające za zadanie znalezienie wszystkich filmów w tabeli `sakila.film` bazy danych `Sakila`, w których występuje aktorka Penelope Guiness (`actor_id=1`). Naturalne wydaje się utworzenie następującego podzapytania:

```
mysql> SELECT * FROM sakila.film
    -> WHERE film_id IN(
    ->    SELECT film_id FROM sakila.film_actor WHERE actor_id = 1);
```

[12] Jeżeli trzeba, programista może wpłynąć na takie zachowanie bazy danych, stosując wskazówkę związaną z `SQL_BUFFER_RESULT`. Więcej informacji na ten temat przedstawiono w podrozdziale „Wskazówki dotyczące optymalizatora zapytań", znajdującym się w dalszej części rozdziału.

Kuszące wydaje się założenie, że baza danych MySQL wykona powyższe zapytanie na odwrót, poprzez wyszukanie listy wartości `actor_id` i zastosowanie ich w liście `IN()`. Wcześniej wspomniano, że lista `IN()` w zasadzie jest bardzo szybka, a więc można spodziewać się, że zapytanie zostanie zoptymalizowane np. w następujący sposób:

```
-- SELECT GROUP_CONCAT(film_id) FROM sakila.film_actor WHERE actor_id = 1;
-- Result: 1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,832,939,970,980
SELECT * FROM sakila.film
WHERE film_id
IN(1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,832,939,970,980);
```

Niestety, mamy do czynienia z sytuacją odwrotną. MySQL próbuje „pomóc" podzapytaniu poprzez wepchnięcie do niego korelacji z zewnętrznej tabeli, żeby efektywniej mogło wyszukiwać rekordy. Nowo napisane zapytanie jest więc następujące:

```
SELECT * FROM sakila.film
WHERE EXISTS (
   SELECT * FROM sakila.film_actor WHERE actor_id = 1
AND film_actor.film_id = film.film_id);
```

Po powyższej modyfikacji podzapytanie wymaga kolumny `film_id` z zewnętrznej tabeli `film` i nie może być wykonane w pierwszej kolejności. Dane wyjściowe polecenia `EXPLAIN` przedstawiają wynik jako `DEPENDENT SUBQUERY` (można wykonać polecenie `EXPLAIN EXTENDED` w celu dokładnego zobaczenia, jak zapytanie zostało napisane na nowo):

```
mysql> EXPLAIN SELECT * FROM sakila.film ...;
+----+--------------------+------------+--------+-----------------------+
| id | select_type        | table      | type   | possible_keys         |
+----+--------------------+------------+--------+-----------------------+
|  1 | PRIMARY            | film       | ALL    | NULL                  |
|  2 | DEPENDENT SUBQUERY | film_actor | eq_ref | PRIMARY,idx_fk_film_id |
+----+--------------------+------------+--------+-----------------------+
```

Zgodnie z danymi wyjściowymi polecenia `EXPLAIN`, baza danych MySQL przeprowadzi skanowanie tabeli `film` i wykona podzapytanie względem każdego znalezionego rekordu. W przypadku małej tabeli nie przełoży się to na zbyt duże obniżenie wydajności, ale jeśli zewnętrzna tabela będzie ogromna, spadek wydajność będzie katastrofalny. Na szczęście, nowe zapytanie można bardzo łatwo napisać, np. z użyciem klauzuli `JOIN`:

```
mysql> SELECT film.* FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE actor_id = 1;
```

Innym dobrym rozwiązaniem optymalizacyjnym jest ręczne wygenerowanie listy `IN()` poprzez wykonanie podzapytania jako oddzielnego zapytania z funkcją `GROUP_CONCAT()`. Czasami będzie to szybsze rozwiązanie niż użycie klauzuli `JOIN`.

Baza danych MySQL była mocno krytykowana za ten szczególny rodzaj planu wykonywania podzapytania. Chociaż niewątpliwie wymaga on poprawienia, krytycy często mylili dwa różne elementy: kolejność wykonywania oraz buforowanie. Wykonanie zapytania na odwrót jest jedną z form jego optymalizacji, natomiast buforowanie wyniku zapytania to inna forma. Samodzielne ponowne napisanie zapytania daje kontrolę nad obydwoma aspektami. Przyszłe wersje MySQL powinny zapewniać lepszą optymalizację tego rodzaju zapytań, choć nie jest to zadanie łatwe do wykonania. Występują znacznie gorsze przypadki związane z dowolnym planem wykonania zapytania, łącznie z zastosowaniem odwrotnego planu wykonania zapytania, o którym sądzi się, że będzie prostszy do optymalizacji.

Kiedy podzapytanie skorelowanie jest dobre?

Baza danych MySQL nie zawsze błędnie optymalizuje podzapytania skorelowane. Jeżeli czytelnik usłyszy poradę, aby zawsze ich unikać, nie należy jej słuchać! W zamian warto przeprowadzić testy wydajności i samodzielnie podjąć decyzję. W niektórych sytuacjach podzapytanie skorelowane jest całkiem rozsądnym lub nawet optymalnym sposobem otrzymania wyniku. Warto spojrzeć na poniższy przykład:

```
mysql> EXPLAIN SELECT film_id, language_id FROM sakila.film
    -> WHERE NOT EXISTS(
    ->    SELECT * FROM sakila.film_actor
    ->    WHERE film_actor.film_id = film.film_id
    ->)\G
*************************** Rekord 1. ***************************
           id: 1
  select_type: PRIMARY
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra: Using where
*************************** Rekord 2. ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: film_actor
         type: ref
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: film.film_id
         rows: 2
        Extra: Using where; Using index
```

Standardową poradą dla takiego zapytania jest zastosowanie w nim klauzuli `LEFT OUTER JOIN` zamiast użycia podzapytania. Teoretycznie, baza danych MySQL powinna zastosować w obu przypadkach taki sam plan wykonania zapytania. Warto to sprawdzić:

```
mysql> EXPLAIN SELECT film.film_id, film.language_id
    -> FROM sakila.film
    ->    LEFT OUTER JOIN sakila.film_actor USING(film_id)
    -> WHERE film_actor.film_id IS NULL\G
*************************** Rekord 1. ***************************
           id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 951
        Extra:
*************************** Rekord 2. ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
```

```
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
      key_len: 2
          ref: sakila.film.film_id
         rows: 2
        Extra: Using where; Using index; Not exists
```

Plany są niemal identyczne, ale występują między nimi pewne różnice.

- Zapytanie `SELECT` względem tabeli `film_actor` jest rodzaju `DEPENDENT SUBQUERY` w jednym zapytaniu, natomiast `SIMPLE` — w drugim. Różnica po prostu odzwierciedla składnię, ponieważ pierwsze zapytanie używa podzapytania, a drugie nie. Pod względem operacji procedury obsługi nie ma między nimi zbyt dużej różnicy.
- W kolumnie `Extra` drugiego zapytania tabela `film` nie jest opisana ciągiem tekstowym „Using where". To nie ma znaczenia: klauzula `USING` użyta w drugim zapytaniu i tak oznacza to samo, co klauzula `WHERE`.
- W kolumnie `Extra` drugiego zapytania tabela `film_actor` jest opisana ciągiem tekstowym „Not exists". To jest przykład działania algorytmu wczesnego zakończenia zapytania omówionego we wcześniejszej części rozdziału. Oznacza to, że baza danych MySQL chciała użyć nieistniejącej optymalizacji w celu uniknięcia odczytu więcej niż jednego rekordu z indeksu `idx_fk_film` tabeli `film_actor`. Odpowiada to funkcji `NOT EXISTS()` skorelowanego podzapytania, ponieważ zatrzymuje przetwarzanie bieżącego rekordu tuż po znalezieniu dopasowania.

Tak więc teoretycznie MySQL wykona powyższe zapytania niemal identycznie. W rzeczywistości przeprowadzenie testów wydajności to jedyny sposób stwierdzenia, które z wymienionych rozwiązań jest szybsze. Autorzy przeprowadzili testy wydajności na obu zapytaniach, korzystając ze standardowych ustawień. Wyniki zostały przedstawione w tabeli 4.1.

Tabela 4.1. Zapytanie NOT EXISTS kontra LEFT OUTER JOIN

Zapytanie	Wynik w liczbie zapytań na sekundę (QPS)
podzapytanie `NOT EXISTS`	360 QPS
`LEFT OUTER JOIN`	425 QPS

A zatem podzapytanie jest nieco wolniejsze!

Jednak nie zawsze tak bywa. Zdarzają się sytuacje, gdy podzapytanie może być szybsze. Przykładowo może sprawdzać się doskonale, kiedy trzeba będzie po prostu wskazać rekordy z jednej tabeli dopasowane do rekordów w drugiej. Chociaż brzmi to jak doskonały opis złączenia, nie zawsze jest nim. Przedstawione poniżej złączenie zaprojektowane w celu wyszukania każdego filmu, w którym występuje aktor, zwróci powielone rekordy, ponieważ w niektórych filmach występuje wielu aktorów:

```
mysql> SELECT film.film_id FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id);
```

Aby wyeliminować powielone rekordy, trzeba użyć klauzul `DISTINCT` lub `GROUP BY`:

```
mysql> SELECT DISTINCT film.film_id FROM sakila.film
    ->    INNER JOIN sakila.film_actor USING(film_id);
```

Czy to naprawdę jest próba przyśpieszenia zapytania i czy wynika z kodu SQL? Operator EXISTS wyraża logiczną koncepcję „ma dopasowanie", bez tworzenia powielających się rekordów, oraz unika operacji GROUP BY lub DISTINCT, które mogłyby wymagać tabeli tymczasowej. Poniżej przedstawiono nowo napisane zapytanie zawierające podzapytanie zamiast złączenia:

```
mysql> SELECT film_id FROM sakila.film
    ->     WHERE EXISTS(SELECT * FROM sakila.film_actor
    ->     WHERE film.film_id = film_actor.film_id);
```

Także w tym przypadku autorzy przeprowadzili testy wydajności sprawdzające obie strategie. Wyniki zostały przedstawione w tabeli 4.2.

Tabela 4.2. Zapytanie EXISTS kontra INNER JOIN

Zapytanie	Wynik w liczbie zapytań na sekundę (QPS)
INNER JOIN	185 QPS
podzapytanie EXISTS	325 QPS

Powyższy przykład pokazuje, że podzapytanie zostało wykonane szybciej niż operacja złączenia.

Ten długi przykład został zaprezentowany w celu ilustracji dwóch aspektów. Po pierwsze, nie należy zważać na kategoryczne porady dotyczące podzapytań. Po drugie, trzeba przeprowadzać testy wydajności mające na celu potwierdzenie założeń dotyczących planów zapytania i szybkości ich wykonywania.

Ograniczenia klauzuli UNION

Baza danych MySQL może czasami „wciągnąć" do środka warunki z zewnętrznej klauzuli UNION, gdzie będą mogły zostać użyte w celu ograniczenia liczby wyników lub włączenia dodatkowych optymalizacji.

Jeżeli programista sądzi, że dowolne z poszczególnych zapytań połączonych klauzulą UNION może odnieść korzyści z użycia klauzuli LIMIT lub stanie się podmiotem działania klauzuli ORDER BY po połączeniu z innymi zapytaniami, wówczas klauzulę LIMIT trzeba umieścić wewnątrz każdej części konstrukcji UNION. Jeżeli np. dwie ogromne tabele są powiązane klauzulą UNION, a wynik zostaje ograniczony do dwudziestu rekordów za pomocą klauzuli LIMIT, wtedy baza danych MySQL będzie przechowywała obie ogromne tabele w tabeli tymczasowej, a następnie pobierze z niej dwadzieścia rekordów. Można uniknąć takiej sytuacji poprzez umieszczenie klauzuli LIMIT wewnątrz każdego zapytania tworzącego konstrukcję UNION.

Optymalizacja połączonych indeksów

Wprowadzone w MySQL 5.0 algorytmy łączenia indeksów pozwalają bazie danych MySQL na używanie w zapytaniu więcej niż tylko jednego indeksu na tabelę. Wcześniejsze wersje MySQL mogły używać tylko pojedynczego indeksu. Dlatego też kiedy pojedynczy indeks nie był wystarczająco dobry, aby poradzić sobie z wszystkimi ograniczeniami w klauzuli WHERE, baza danych MySQL często wybierała opcję skanowania tabeli. Przykładowo tabela film_actor ma indeks obejmujący kolumnę film_id oraz indeks obejmujący kolumnę actor_id, ale żaden z nich nie jest dobrym wyborem dla dwóch warunków klauzuli WHERE w poniższym zapytaniu:

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1;
```

We wcześniejszych wersjach MySQL powyższe zapytanie spowodowałoby skanowanie tabeli, o ile nie zostałoby napisane od nowa w postaci dwóch zapytań połączonych klauzulą UNION:

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor WHERE actor_id = 1
    -> UNION ALL
    -> SELECT film_id, actor_id FROM sakila.film_actor WHERE film_id = 1
    ->    AND actor_id <> 1;
```

Jednak w bazie danych MySQL 5.0 i nowszych zapytanie to może użyć obu indeksów równocześnie, skanując je i łącząc wyniki. Istnieją trzy odmiany algorytmu: unia dla warunków OR, część wspólna dla warunków AND oraz unia części wspólnych dla połączenia dwóch wymienionych. Przedstawione poniżej zapytanie używa unii operacji skanowania dwóch indeksów, co można zobaczyć podczas analizy kolumny Extra:

```
mysql> EXPLAIN SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1\G
*************************** Rekord 1. ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: index_merge
possible_keys: PRIMARY,idx_fk_film_id
          key: PRIMARY,idx_fk_film_id
      key_len: 2,2
          ref: NULL
         rows: 29
        Extra: Using union(PRIMARY,idx_fk_film_id); Using where
```

Baza danych MySQL może wykorzystać tę technikę w skomplikowanych klauzulach WHERE. Dlatego też w niektórych zapytaniach kolumna Extra będzie wyświetlała zagnieżdżone operacje. Mechanizm zazwyczaj działa bardzo dobrze, ale czasami przeprowadzane przez algorytm buforowanie, sortowanie i operacje łączenia używają dużej ilości zasobów procesora i pamięci. Dotyczy to zwłaszcza sytuacji, gdy nie wszystkie indeksy zapewniają wysoki poziom selektywności, a więc równoczesne skanowanie zwraca dużą ilość rekordów poddawanych operacji łączenia. Warto sobie w tym miejscu przypomnieć, że optymalizator nie wlicza tego do kosztu — po prostu optymalizuje liczbę losowych odczytów stron. W ten sposób koszt zapytania jest „niedoszacowany" i wykonanie takiego zapytania może być wolniejsze niż zwykłe skanowanie tabeli. Ogromne zużycie zasobów pamięci i mocy obliczeniowej procesora może mieć także wpływ na równocześnie wykonywane inne zapytania, ale efekt ten nie będzie widoczny, kiedy zapytanie jest wykonywane w izolacji. To kolejny powód skłaniający do projektowania realistycznych testów wydajności.

Jeżeli zapytanie jest wykonywane znacznie wolniej z powodu tego ograniczenia optymalizatora, można sobie poradzić, wyłączając określone indeksy za pomocą polecenia IGNORE INDEX lub po prostu wracając do starej taktyki z użyciem klauzuli UNION.

Szerzenie równości

W niektórych przypadkach szerzenie równości może nieść ze sobą nieoczekiwane koszty. Przykładowo warto rozważyć ogromną listę IN() obejmującą kolumnę, która wg informacji optymalizatora będzie równa określonym kolumnom w innych tabelach z powodu klauzul WHERE, ON lub USING ustanawiających równość kolumn wobec siebie.

Optymalizator będzie „współdzielił" listę poprzez skopiowanie jej do odpowiednich kolumn we wszystkich powiązanych tabelach. Zwykle jest to pomocne, ponieważ daje optymalizatorowi zapytań oraz silnikowi wykonywania więcej możliwości w zakresie miejsca faktycznego przeprowadzenia sprawdzenia listy `IN()`. Kiedy jednak lista jest ogromna, skutkiem może być mniej efektywna optymalizacja i wolniejsze wykonanie zapytania. W trakcie pisania tej książki nie było dostępne żadne wbudowane w bazę rozwiązanie pozwalające na ominięcie tego problemu — w takim przypadku trzeba po prostu zmodyfikować kod źródłowy. (Problem jednak nie występuje u większości użytkowników).

Wykonywanie równoległe

Baza danych MySQL nie potrafi wykonywać pojedynczego zapytania równocześnie na wielu procesorach. Taka funkcja jest oferowana przez niektóre serwery baz danych, ale nie przez MySQL. Autorzy wspominają, aby nie poświęcać zbyt dużej ilości czasu na próbę odkrycia, jak w MySQL uzyskać wykonywanie zapytania równocześnie na wielu procesorach.

Złączenia typu hash

Podczas pisania książki baza danych MySQL nie przeprowadzała prawdziwych złączeń typu hash — wszystko pozostaje złączeniem w postaci zagnieżdżonej pętli. Jednak złączenia typu hash można emulować za pomocą indeksów typu hash. Jeżeli używany silnik magazynu danych jest inny niż Memory, trzeba także emulować indeksy typu hash. Więcej informacji na ten temat przedstawiono w sekcji „Budowa własnych indeksów typu hash", znajdującej się w rozdziale 3.

Luźne skanowanie indeksu

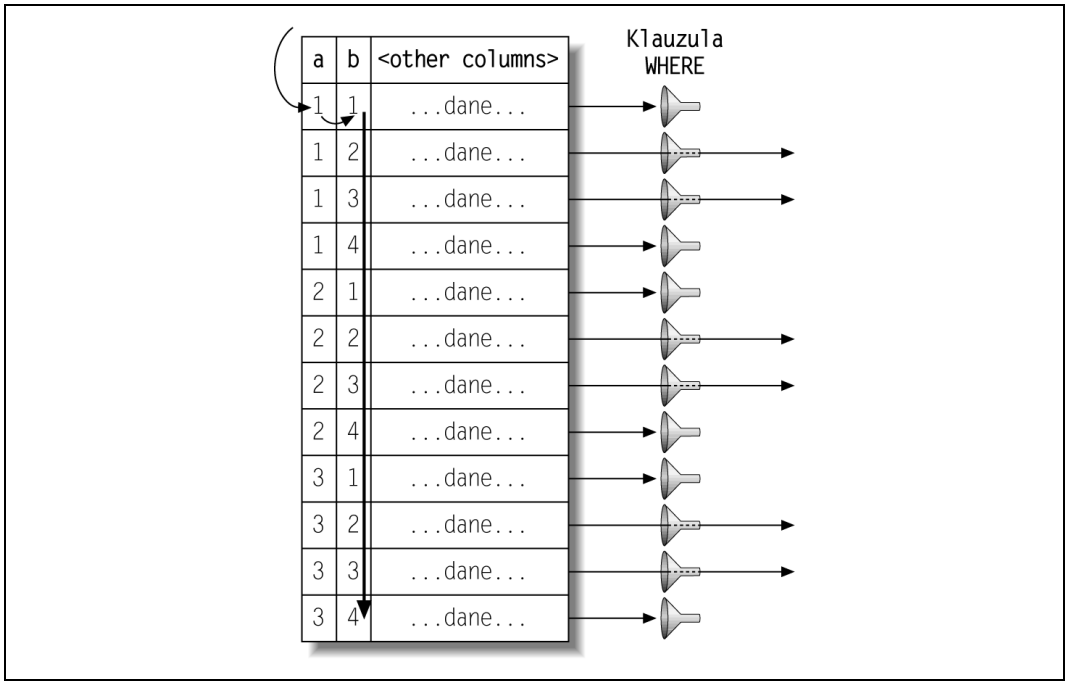
Baza danych MySQL znana jest z tego, że nie umożliwia przeprowadzenia luźnego skanowania indeksu, które polega na skanowaniu nieprzylegających do siebie zakresów indeksu. Ogólnie rzecz biorąc, skanowanie indeksu w MySQL wymaga zdefiniowania punktu początkowego oraz końcowego w indeksie nawet wtedy, jeśli kilka nieprzylegających do siebie rekordów w środku jest naprawdę pożądanych w danym zapytaniu. Baza danych MySQL będzie skanowała cały zakres rekordów wewnątrz zdefiniowanych punktów końcowych.

Przykład pomoże w zilustrowaniu tego problemu. Zakładamy, że tabela zawiera indeks obejmujący kolumny `(a, b)`, a programista chce wykonać poniższe zapytanie:

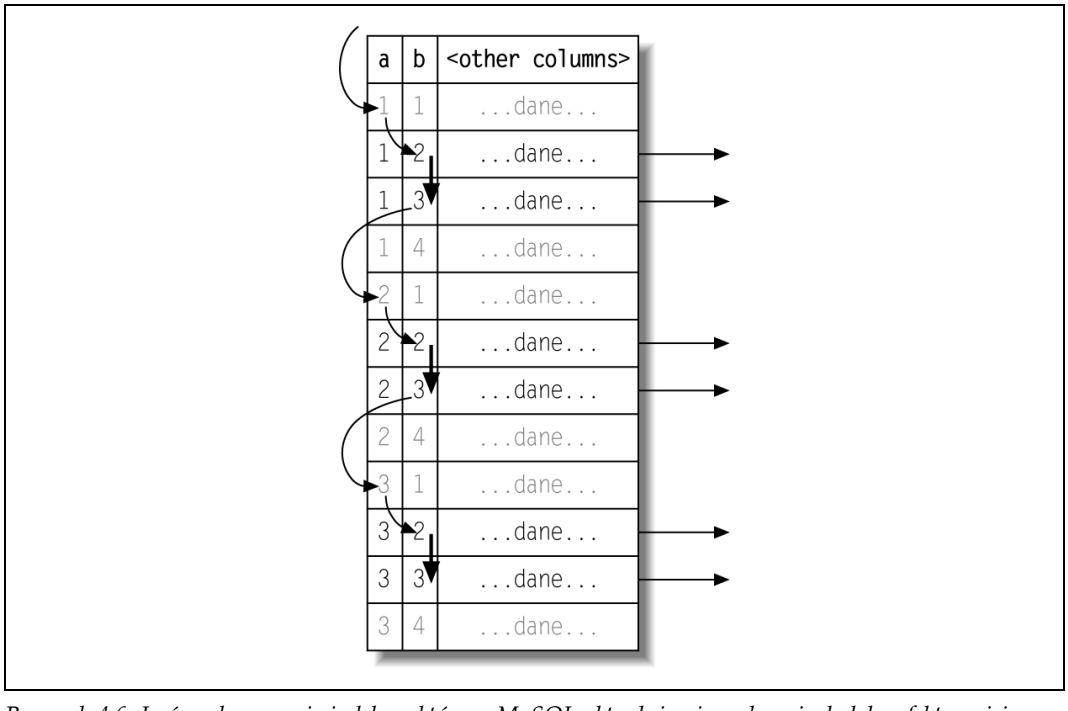
```
mysql> SELECT ... FROM tbl WHERE b BETWEEN 2 AND 3;
```

Ponieważ indeks rozpoczyna się od kolumny `a`, ale klauzula `WHERE` zapytania nie zawiera kolumny `a`, baza danych MySQL przeprowadzi skanowanie tabeli oraz za pomocą klauzuli `WHERE` wyeliminuje nieprzylegające do siebie rekordy, co pokazano na rysunku 4.5.

Bardzo łatwo można dostrzec, że istnieją szybsze sposoby wykonania tego zapytania. Struktura indeksu (ale nie API silnika magazynu danych MySQL) pozwala na wyszukanie początku każdego zakresu wartości, skanowanie aż do końca zakresu, a następnie przejście na początek kolejnego zakresu. Na rysunku 4.6 pokazano, jak taka strategia mogłaby wyglądać, gdyby została zaimplementowana w MySQL.



Rysunek 4.5. Baza danych MySQL skanuje całą tabelę w celu wyszukania rekordów



Rysunek 4.6. Luźne skanowanie indeksu, którego MySQL aktualnie nie wykonuje, byłoby efektywniejszym sposobem wykonania omawianego zapytania

Warto zwrócić uwagę na brak klauzuli WHERE, która stała się zbędna, ponieważ sam indeks pozwala na pomijanie niepotrzebnych rekordów. (Przypominamy ponownie, że baza danych MySQL nie ma jeszcze takich możliwości).

Jest to, co prawda, uproszczony przykład i przedstawione zapytanie można łatwo zoptymalizować poprzez dodanie innego indeksu. Jednak istnieje wiele sytuacji, gdy dodanie innego indeksu nie stanowi rozwiązania. Jednym z takich przypadków jest zapytanie, które zawiera warunek zakresu względem pierwszej kolumny indeksu oraz warunek równości względem drugiej kolumny indeksu.

Począwszy od MySQL w wersji 5.0, operacja luźnego skanowania indeksu jest możliwa w pewnych ściśle określonych sytuacjach, np. w zapytaniach wyszukujących wartości maksymalną i minimalną w zgrupowanym zapytaniu:

```
mysql> EXPLAIN SELECT actor_id, MAX(film_id)
    -> FROM sakila.film_actor
    -> GROUP BY actor_id\G
*************************** Rekord 1. ***************************
           id: 1
  select_type: SIMPLE
        table: film_actor
         type: range
possible_keys: NULL
          key: PRIMARY
      key_len: 2
          ref: NULL
         rows: 396
        Extra: Using index for group-by
```

Informacja „Using index for group-by" wyświetlona w danych wyjściowych polecenia EXPLAIN wskazuje na zastosowanie luźnego skanowania indeksu. To jest dobry rodzaj optymalizacji w tym specjalnym przypadku, ale równocześnie nie jest to ogólnego przeznaczenia luźne skanowanie indeksu. Lepiej byłoby, gdyby informacja miała postać „loose index probe".

Dopóki baza danych MySQL nie będzie obsługiwała ogólnego przeznaczenia luźnego skanowania indeksu, obejściem problemu jest zastosowanie stałej bądź listy stałych dla pierwszych kolumn indeksu. W zaprezentowanym w poprzednim rozdziale studium przypadku indeksowania przedstawiono kilka przykładów osiągnięcia dobrej wydajności za pomocą takich zapytań.

Funkcje MIN() i MAX()

Baza danych MySQL nie może zbyt dobrze zoptymalizować pewnych zapytań wykorzystujących funkcje MIN() lub MAX(). Oto przykład takiego zapytania:

```
mysql> SELECT MIN(actor_id) FROM sakila.actor WHERE first_name = 'PENELOPE';
```

Ponieważ nie ma indeksu obejmującego kolumnę first_name, powyższe zapytanie spowoduje przeprowadzenie skanowania tabeli. Jeżeli MySQL skanuje klucz podstawowy, teoretycznie może zatrzymać skanowanie po odczytaniu pierwszego dopasowanego rekordu, gdyż klucz podstawowy jest w kolejności ściśle rosnącej i każdy następny rekord będzie miał większą wartość actor_id. Jednak w omawianym przypadku MySQL przeskanuje całą tabelę, o czym można się przekonać po sprofilowaniu zapytania. Rozwiązaniem problemu jest usunięcie funkcji MIN() i napisanie zapytania z użyciem klauzuli LIMIT, np. następująco:

```
mysql> SELECT actor_id FROM sakila.actor USE INDEX(PRIMARY)
    -> WHERE first_name = 'PENELOPE' LIMIT 1;
```

Taka strategia bardzo często działa doskonale w innej sytuacji, kiedy baza danych MySQL wybrała skanowanie większej liczby rekordów niż potrzeba. Puryści mogą uznać, że tego rodzaju zapytanie oznacza brak zrozumienia SQL. Z reguły programista informuje serwer, *co* chce uzyskać, a serwer określa, *jak* pobrać te dane. W omawianym zapytaniu programista informuje serwer MySQL, *jak* wykonać dane zapytanie. Dlatego też z zapytania nie wynika jasno, że *szukane* dane to wartość minimalna. To prawda, ale czasami trzeba poświęcić zasady w imię uzyskania większej wydajności.

Równoczesne wykonywanie poleceń SELECT i UPDATE w tej samej tabeli

Baza danych MySQL nie pozwala na wykonywanie polecenia `SELECT` względem tabeli, na której jednocześnie jest wykonywane polecenie `UPDATE`. Naprawdę nie jest to ograniczenie wynikające z optymalizatora, ale wiedza o sposobie wykonywania zapytań przez MySQL może pomóc w obejściu tego problemu. Poniżej przedstawiono przykład niedozwolonego zapytania, mimo że jest standardowym kodem SQL. Zapytanie powoduje uaktualnienie każdego rekordu liczbą podobnych rekordów znajdujących się w tabeli:

```
mysql> UPDATE tbl AS outer_tbl
    ->   SET cnt = (
    ->      SELECT count(*) FROM tbl AS inner_tbl
    ->      WHERE inner_tbl.type = outer_tbl.type
    ->   );
ERROR 1093 (HY000): You can't specify target table 'outer_tbl' for update in FROM
clause
```

Aby obejść to ograniczenie, można wykorzystać tabelę pochodną, ponieważ MySQL potraktuje ją jak tabelę tymczasową. W ten sposób faktycznie zostaną wykonane dwa zapytania: pierwsze to `SELECT` w podzapytaniu, drugie obejmuje wiele tabel `UPDATE` z połączonymi wynikami tabeli oraz podzapytania. Podzapytanie będzie otwierało i zamykało tabelę przed otworzeniem jej przez zewnętrzne zapytanie `UPDATE`, a więc całe zapytanie będzie mogło zostać wykonane:

```
mysql> UPDATE tbl
    ->    INNER JOIN(
    ->       SELECT type, count(*) AS cnt
    ->       FROM tbl
    ->       GROUP BY type
    ->    ) AS der USING(type)
    -> SET tbl.cnt = der.cnt;
```

Optymalizacja określonego rodzaju zapytań

W podrozdziale zostaną przedstawione wskazówki dotyczące optymalizacji określonego rodzaju zapytań. Większość tych zagadnień została szczegółowo omówiona w innych częściach książki, ale autorzy chcieli utworzyć listę najczęściej spotykanych problemów optymalizacji, do której można łatwo powracać.

Większość wskazówek przedstawionych w podrozdziale jest uzależniona od wersji serwera i może być nieaktualna w przyszłych wersjach MySQL. Nie ma żadnego powodu, aby pewnego dnia sam serwer nie uzyskał możliwości przeprowadzania niektórych bądź wszystkich z wymienionych optymalizacji.

Optymalizacja zapytań COUNT()

Funkcja agregująca `COUNT()` i sposób optymalizacji zapytań wykorzystujących tę funkcję to prawdopodobnie jeden z dziesięciu najbardziej niezrozumiałych tematów w MySQL. Liczba błędnych informacji na ten temat, które autorzy znaleźli w Internecie, jest większa, niż można sądzić.

Przed zagłębieniem się w zagadnienia optymalizacji bardzo ważne jest zrozumienie, jak działa funkcja `COUNT()`.

Jakie jest działanie funkcji COUNT()?

`COUNT()` to funkcja specjalna działająca na dwa odmienne sposoby: zlicza *wartości* oraz *rekordy*. Wartość jest wyrażeniem innym niż `NULL` (ponieważ `NULL` oznacza brak wartości). Jeżeli w nawiasie zostanie podana nazwa kolumny lub inne wyrażenie, funkcja `COUNT()` obliczy, ile razy podane wyrażenie ma wartość. Dla wielu osób będzie to bardzo mylące, co po części wynika z faktu, że same wartości `NULL` są mylące. Jeżeli czytelnik musi nauczyć się, jak działa SQL, warto sięgnąć pod dobrą książkę omawiającą podstawy SQL. (Internet niekoniecznie jest dobrym źródłem informacji na ten temat).

Inna forma funkcji `COUNT()` po prostu oblicza liczbę rekordów w wyniku. Jest to sposób działania bazy danych MySQL, kiedy wie, że wyrażenie umieszczone w nawiasie nigdy nie będzie `NULL`. Najbardziej oczywistym przykładem jest polecenie `COUNT(*)` będące specjalną formą funkcji `COUNT()`. Nie powoduje ona rozszerzenia znaku wieloznacznego * na pełną listę kolumn w tabeli, jak można by tego oczekiwać. Zamiast tego całkowicie ignoruje kolumny i zlicza rekordy.

Jednym z najczęściej popełnianych błędów jest podawanie w nawiasie nazw kolumn, kiedy programista chce, aby funkcja zliczyła rekordy. Gdy trzeba obliczyć liczbę rekordów w wyniku, wtedy *zawsze* należy użyć funkcji `COUNT(*)`. Taka postać jasno wskazuje intencje programisty i pozwala uniknąć kiepskiej wydajności.

Mity dotyczące MyISAM

Często popełnianym błędem jest przeświadczenie, że silnik MyISAM jest wyjątkowo szybki podczas wykonywania zapytań `COUNT()`. Wprawdzie jest szybki, ale tylko w wyjątkowej sytuacji: kiedy stosujemy funkcję `COUNT(*)` bez klauzuli `WHERE`, która po prostu zlicza liczbę rekordów w całej tabeli. Baza danych MySQL może zoptymalizować to zapytanie, ponieważ silnik magazynu danych zawsze otrzymuje informację, ile rekordów znajduje się w tabeli. Jeżeli w MySQL określono, że `col` nigdy nie przyjmie wartości `NULL`, wówczas można również zoptymalizować wyrażenie `COUNT(col)` poprzez wewnętrzną konwersję wyrażenia na `COUNT(*)`.

Silnik MyISAM nie ma żadnych magicznych optymalizacji dotyczących zliczania rekordów, kiedy zapytanie używa klauzuli `WHERE` lub dla bardziej ogólnych przypadków zliczania wartości zamiast rekordów. W określonym zapytaniu może działać szybciej niż inne silniki magazynu danych, ale nie musi. Wszystko zależy od wielu czynników.

Prosta optymalizacja

Czasami można wykorzystać zalety optymalizacji MyISAM w postaci `COUNT(*)` do zliczenia wszystkiego, z wyjątkiem małej liczby rekordów, które zostały doskonale zindeksowane. W przedstawionym poniżej przykładzie użyto standardowej bazy danych `World` w celu pokazania, jak można efektywnie znaleźć liczbę miast, których identyfikator `ID` jest większy niż `5`. Takie zapytanie można zapisać następująco:

```
mysql> SELECT COUNT(*) FROM world.City WHERE ID > 5;
```

Jeżeli powyższe zapytanie zostanie sprofilowane za pomocą polecenia `SHOW STATUS`, można przekonać się, że zapytanie przeskanowało 4079 rekordów. Po odwróceniu warunków i odjęciu liczby miast, których identyfikator `ID` ma wartość mniejszą lub równą `5` od ogólnej liczby miast, liczba analizowanych rekordów spada do pięciu:

```
mysql> SELECT (SELECT COUNT(*) FROM world.City) - COUNT(*)
    -> FROM world.City WHERE ID <= 5;
```

Powyższa wersja zapytania odczytuje mniejszą ilość rekordów, ponieważ w trakcie fazy optymalizacji podzapytanie jest zamieniane na stałą, o czym można przekonać się, przeglądając dane wyjściowe polecenia `EXPLAIN`:

```
+----+-------------+-------+...+------+-----------------------------+
| id | select_type | table |...| rows | Extra                       |
+----+-------------+-------+...+------+-----------------------------+
|  1 | PRIMARY     | City  |...|    6 | Using where; Using index    |
|  2 | SUBQUERY    | NULL  |...| NULL | Select tables optimized away |
+----+-------------+-------+...+------+-----------------------------+
```

Często pojawiające się pytanie na listach dyskusyjnych i kanałach IRC dotyczy tego, jak pobrać liczbę odmiennych wartości w tej samej kolumnie za pomocą tylko jednego zapytania, ograniczając w ten sposób ogólną liczbę wymaganych zapytań. Załóżmy np., że programista chce utworzyć pojedyncze zapytanie, które zlicza ilość elementów w kilku kolorach. Nie można użyć klauzuli `OR` (np. `SELECT COUNT(color = 'blue' OR color = 'red') FROM items;`), ponieważ programista nie chce oddzielić różnych liczników od odmiennych kolorów. Kolorów nie można także umieścić w klauzuli `WHERE` (np. `SELECT COUNT(*) FROM items WHERE color = 'blue' AND color = 'red';`), ponieważ kolory są wzajemnie wykluczające się. Poniżej przedstawiono zapytanie rozwiązujące ten problem:

```
mysql> SELECT SUM(IF(color = 'blue', 1, 0)) AS blue,
SUM(IF(color = 'red', 1, 0))   ->AS red FROM items;
```

Poniżej znajduje się kolejne rozwiązanie przedstawionego problemu, ale zamiast funkcji `SUM()` zastosowano w zapytaniu funkcję `COUNT()`, przy upewnieniu się, że wyrażenie nie będzie miało wartości, kiedy kryteria będą fałszywe:

```
mysql> SELECT COUNT(color = 'blue' OR NULL) AS blue, COUNT(color = 'red' OR NULL)
    -> AS red FROM items;
```

Bardziej skomplikowana optymalizacja

Ogólnie rzecz biorąc, zapytania `COUNT()` są trudne do optymalizacji, ponieważ z reguły muszą obliczać dużą liczbę rekordów (np. dostęp do ogromnej ilości danych). Jedyną inną opcją jest optymalizacja wewnątrz samego serwera MySQL, tak aby używał indeksu pokrywającego. Takie rozwiązanie przedstawiono w rozdziale 3. Jeżeli i to okaże się niewystarczające, trzeba będzie wprowadzić zmiany w architekturze aplikacji. Warto rozważyć użycie tabel

podsumowania (również omówione w rozdziale 3.) oraz zewnętrzny system buforowania, taki jak *memcached*. Czytelnik prawdopodobnie stanie przed dobrze znanym dylematem: „szybko, dobrze i prosto — wybierz dwa dowolne".

Optymalizacja zapytań typu JOIN

Temat ten jest w rzeczywistości poruszany w całej książce, ale w tym miejscu autorzy uważają za stosowne, by wspomnieć o kilku aspektach.

- Należy się upewnić, że indeksy obejmują kolumny używane w klauzulach `ON` lub `USING`. Więcej informacji na temat indeksowania przedstawiono w podrozdziale „Podstawy indeksowania", znajdującym się w rozdziale 3. Podczas dodawania indeksów warto rozważyć użycie kolejności stosowanej w złączeniach. Jeżeli za pomocą kolumny `c` złączane są tabele `A` i `B`, a optymalizator postanowi o złączeniu tabel w kolejności `B, A`, wówczas nie trzeba indeksować kolumny w tabeli `B`. Nieużywane indeksy stanowią dodatkowe obciążenie. Ogólnie rzecz biorąc, indeksy warto dodać jedynie do drugiej tabeli i zastosować przy tym kolejność użytą w złączeniu, o ile indeksy nie są potrzebne do jeszcze innych zadań.
- Warto podjąć próbę upewnienia się, że każde wyrażenie `GROUP BY` i `ORDER BY` odnosi się do kolumn z pojedynczej tabeli. W ten sposób baza danych MySQL będzie mogła spróbować użycia indeksu podczas wykonywania tej operacji.
- Należy zachować ostrożność podczas uaktualniania serwera MySQL, ponieważ składnia złączeń, kolejność operatorów oraz inne zachowania mogły ulec zmianie. Czasami to, co było zwykłym złączeniem, może stać się innym produktem, innym rodzajem złączenia zwracającym odmienne wyniki bądź nawet już niewłaściwą składnią.

Optymalizacja podzapytań

Najważniejszą wskazówką dotyczącą podzapytań, jakiej można udzielić, jest zalecenie, aby — gdy tylko jest to możliwe — stosować złączenia, przynajmniej w bieżących wersjach MySQL. Temat został dokładnie omówiony na początku rozdziału.

Podzapytania są przedmiotem intensywnych prac zespołu zajmującego się optymalizatorem. Dlatego też przyszłe wersje MySQL mogą posiadać więcej możliwości w zakresie optymalizacji podzapytań. Dopiero wówczas okaże się, które optymalizacje znajdą się w wersji finalnej oraz jaką przyniosą różnicę. Udzielona w tym miejscu wskazówka, że „warto stosować złączenia", nie musi być aktualna w przyszłości. Wraz z upływem czasu serwer staje się coraz „sprytniejszy" i sytuacje, w których programista musi wskazywać sposób rozwiązania danego problemu zamiast oczekiwanych wyników, będą coraz rzadsze.

Optymalizacja zapytań typu GROUP BY i DISTINCT

W wielu przypadkach baza danych MySQL optymalizuje te dwa rodzaje zapytań bardzo podobnie. W rzeczywistości, kiedy trzeba, podczas procesu optymalizacji po prostu przeprowadza wewnętrzną konwersję między nimi. Dla obu rodzajów zapytań bardzo korzystne są indeksy, a więc będzie to najważniejsza droga prowadząca do ich optymalizacji.

Kiedy nie można użyć indeksów, MySQL ma dwa rodzaje strategii stosowania GROUP BY: albo użycie tabeli tymczasowej, albo sortowanie plików w celu wykonania grupowania. Każda z tych strategii jest efektywna w określonych zapytaniach. Za pomocą SQL_BIG_RESULT oraz SQL_SMALL_RESULT można wymusić na optymalizatorze wybór danej metody.

Jeżeli trzeba zgrupować złączenie poprzez wartość pochodzącą z przeszukiwanej tabeli, wtedy zwykle efektywniejszym sposobem grupowania jest przeszukanie identyfikatora tabeli zamiast wartości. Przedstawione jako przykład poniższe zapytanie nie jest tak efektywne, jak mogłoby być:

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
    -> FROM sakila.film_actor
    ->    INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY actor.first_name, actor.last_name;
```

Zapytanie będzie efektywniejsze po zapisaniu w postaci:

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
    -> FROM sakila.film_actor
    ->    INNER JOIN sakila.actor USING(actor_id)
    -> GROUP BY film_actor.actor_id;
```

Grupowanie pod względem kolumny actor.actor_id może być efektywniejsze niż grupowanie pod względem kolumny film_actor.actor_id. Należy przeprowadzić profilowanie i (lub) testy wydajności, aby zobaczyć, jak to wygląda dla używanych danych.

Zapytanie wykorzystuje fakt, że imię i nazwisko aktora jest uzależnione od wartości pola actor_id, a więc zwróci te same wyniki. Jednak nie zawsze będzie możliwe beztroskie wybranie niezgrupowanych kolumn i otrzymanie tych samych wyników. Opcja konfiguracyjna serwera o nazwie SQL_MODE może nawet uniemożliwiać taki krok. Alternatywą jest użycie funkcji MIN() lub MAX() kiedy wiadomo, że wartości w grupie będą odmienne, ponieważ zależą od zgrupowanych kolumn. Ewentualnie, jeśli nie ma znaczenia, która wartość będzie otrzymana, można wykonać zapytanie:

```
mysql> SELECT MIN(actor.first_name), MAX(actor.last_name), ...;
```

Puryści mogą spierać się, że grupowanie nastąpiło względem niewłaściwego elementu, i będą mieli rację. Niepożądane funkcje MIN() lub MAX() są znakiem, że struktura zapytania jest niewłaściwa. Jednak czasami jedynym celem jest to, aby serwer MySQL wykonywał zapytanie SQL tak szybko, jak to możliwe. Puryści będą usatysfakcjonowani po zapisaniu powyższego zapytania w następującej postaci:

```
mysql> SELECT actor.first_name, actor.last_name, c.cnt
    -> FROM sakila.actor
    ->    INNER JOIN (
    ->       SELECT actor_id, COUNT(*) AS cnt
    ->       FROM sakila.film_actor
    ->       GROUP BY actor_id
    ->    ) AS c USING(actor_id) ;
```

Czasami koszt utworzenia i wypełnienia tabeli tymczasowej wymaganej przez podzapytanie jest bardzo wysoki w porównaniu z kosztem lekkiego wypaczenia teorii relacyjności. Należy pamiętać, że tabela tymczasowa utworzona przez podzapytanie nie ma indeksów.

Ogólnie rzecz biorąc, kiepskim pomysłem jest wybór niezgrupowanych kolumn w zgrupowanym zapytaniu, ponieważ wyniki będą niedeterministyczne oraz łatwo mogą ulec zmianie, jeśli nastąpi modyfikacja indeksu bądź optymalizator zadecyduje o użyciu odmiennej

strategii. Większość tego rodzaju zapytań, z którymi spotkali się autorzy, było czystym przypadkiem (ponieważ serwer nie zgłaszał zastrzeżeń) lub wynikiem lenistwa, a nie celowym projektem dotyczącym optymalizacji. Znacznie lepiej zachować jasność. Autorzy zalecają ustawienie zmiennej konfiguracyjnej serwera o nazwie `SQL_MODE` w taki sposób, aby zawierała `ONLY_FULL_GROUP_BY`. Dzięki temu serwer wyświetli komunikat błędu i nie pozwoli programiście na utworzenie błędnego zapytania.

Baza danych MySQL automatycznie ustala kolejność zgrupowanych zapytań, uwzględniając kolumny w klauzuli `GROUP BY`, o ile programista wyraźnie nie zastosuje klauzuli `ORDER BY`. Jeżeli programista nie przykłada wagi do kolejności i zauważy zastosowanie operacji sortowania pliku, może użyć klauzuli `ORDER BY NULL` wyłączającej automatyczne sortowanie. Po klauzuli `GROUP BY` można także dodać opcjonalne słowa kluczowe `DESC` lub `ASC` powodujące ułożenie wyników w pożądanym kierunku względem kolumn klauzuli.

Optymalizacja zapytań typu GROUP BY WITH ROLLUP

Pewną odmianą zgrupowanych zapytań jest nakazanie bazie danych MySQL przeprowadzenia superagregacji wewnątrz wyników. W tym celu można wykorzystać klauzulę `WITH ROLLUP`, ale takie rozwiązanie może nie zapewnić tak dobrej optymalizacji, jakiej oczekuje programista. Za pomocą polecenia `EXPLAIN` warto sprawdzić przyjętą przez serwer metodę wykonania, zwracając uwagę na to, czy grupowanie zostało przeprowadzone za pomocą operacji sortowania pliku czy użycia tabeli tymczasowej. Warto również usunąć klauzulę `WITH ROLLUP` i zobaczyć, czy zostanie wykorzystana taka sama metoda grupowania. Istnieje możliwość wymuszenia użycia wskazanej metody grupowania poprzez zastosowanie wskazówek przedstawionych we wcześniejszej części rozdziału.

Czasami przeprowadzenie superagregacji w aplikacji jest dużo bardziej efektywnym rozwiązaniem, nawet jeśli oznacza konieczność pobrania z serwera znacznie większej liczby rekordów. W klauzuli `FROM` można także zastosować zagnieżdżone podzapytanie lub użyć tabeli tymczasowej przechowującej wyniki pośrednie.

Najlepszym rozwiązaniem może być przeniesienie funkcjonalności `WITH ROLLUP` do kodu aplikacji.

Optymalizacja zapytań typu LIMIT i OFFSET

Zapytania z klauzulami `LIMIT` i `OFFSET` są często stosowane w systemach implementujących stronicowanie niemal zawsze w połączeniu z klauzulą `ORDER BY`. Pomocne będzie posiadanie indeksu obsługującego tę kolejność, ponieważ w przeciwnym razie serwer będzie musiał wykonać dużą liczbę operacji sortowania pliku.

Częstym problemem jest wysoka wartość przesunięcia. Jeżeli zapytanie ma postać `LIMIT 10000, 20`, wygeneruje 10020 rekordów, a następnie odrzuci pierwsze 10000, co jest bardzo kosztowne. Przy założeniu, że wszystkie strony są używane z podobną częstotliwością, zapytanie będzie przeciętnie skanowało połowę tabeli. W celu optymalizacji takich zapytań można albo ograniczyć liczbę stron wyświetlanych w widoku stronicowania, albo spowodować, by wysoka wartość przesunięcia była znacznie bardziej efektywna.

Jedną z prostych technik poprawy wydajności jest przeprowadzanie przesunięcia w indeksie pokrywającym zamiast na pełnych rekordach. Później wynik można połączyć z pełnymi rekordami i pobrać dodatkowe wymagane kolumny. Takie rozwiązanie jest znacznie bardziej efektywne. Warto przeanalizować poniższe zapytanie:

```
mysql> SELECT film_id, description FROM sakila.film ORDER BY title LIMIT 50, 5;
```

Jeżeli tabela będzie ogromna, wówczas lepsza postać powyższego zapytania jest następująca:

```
mysql> SELECT film.film_id, film.description
    -> FROM sakila.film
    ->    INNER JOIN (
    ->       SELECT film_id FROM sakila.film
    ->       ORDER BY title LIMIT 50, 5
    ->    ) AS lim USING(film_id);
```

Takie rozwiązanie sprawdza się, pozwala bowiem serwerowi na analizę tak małej ilości danych, jaka jest możliwa w indeksie, bez konieczności uzyskania dostępu do rekordów. Następnie, po znalezieniu pożądanych rekordów przeprowadzane jest złączenie z pełną tabelą w celu pobrania innych kolumn rekordów. Podobna technika ma zastosowanie w przypadku złączeń z klauzulami `LIMIT`.

Czasami można również skonwertować ograniczenie na postać zapytania pozycyjnego, które serwer może wykonać jako skanowanie pewnego zakresu indeksu. Przykładowo po wcześniejszym przeliczeniu i zindeksowaniu kolumny `position` zapytanie można napisać na nowo w takiej postaci:

```
mysql> SELECT film_id, description FROM sakila.film
    -> WHERE position BETWEEN 50 AND 54 ORDER BY position;
```

Ranking danych powoduje powstanie tego samego problemu, ale zapytania zwykle dokładają jeszcze klauzulę `GROUP BY`. Niemal na pewno programista będzie musiał wcześniej obliczyć i przechowywać ranking danych.

Jeżeli naprawdę trzeba zoptymalizować system stronicowania, prawdopodobnie należy zastosować przygotowane wcześniej podsumowania. Alternatywą jest użycie złączeń względem nadmiarowych tabel, które będą zawierały jedynie klucz podstawowy oraz kolumny wymagane przez klauzulę `ORDER BY`. Można także użyć mechanizmu Sphinx. Więcej informacji na jego temat znajduje się w dodatku C.

Optymalizacja za pomocą opcji SQL_CALC_FOUND_ROWS

Inną powszechnie stosowaną techniką wyświetlania stronicowanych wyników jest dodanie opcji `SQL_CALC_FOUND_ROWS` do zapytania z klauzulą `LIMIT`. W ten sposób będzie wiadomo, ile rekordów zostałoby zwróconych, gdyby w zapytaniu nie było klauzuli `LIMIT`. Można odnieść wrażenie, że to pewnego rodzaju „magia", kiedy serwer przewiduje liczbę rekordów, które mógłby znaleźć. Niestety, serwer tego nie robi, tzn. nie potrafi obliczyć liczby rekordów, których faktycznie nie znajduje. Wymieniona opcja po prostu nakazuje serwerowi wygenerowanie i odrzucenie pozostałej części zbioru wynikowego, zamiast zakończyć działanie po znalezieniu pożądanej liczby rekordów. Działanie to jest bardzo kosztowne.

Lepszym rozwiązaniem jest konwersja programu stronicowania na łącze „następny". Przy założeniu, że na stronie znajdzie się dwadzieścia wyników wyszukiwania, zapytanie powinno ograniczyć za pomocą klauzuli `LIMIT` liczbę rekordów do 21, a następnie wyświetlić jedynie 20.

Jeżeli w zbiorze wynikowym znajduje się 21. rekord, oznacza on obecność kolejnej strony, a tym samym możliwość wygenerowania łącza „następny".

Inna możliwość to pobranie i buforowanie większej liczby rekordów niż potrzeba — powiedzmy 1000 — a następnie pobieranie ich z bufora i umieszczanie na kolejnych stronach. Ta strategia pozwala aplikacji „wiedzieć", jak duży jest zbiór wynikowy. Jeżeli będzie mniejszy niż 1000 rekordów, wówczas aplikacja może obliczyć, ile powinna wygenerować łączy. Natomiast dla większego zbioru wynikowego może po prostu wyświetlić komunikat „Znaleziono ponad 1000 wyników". Obie opisane strategie są znacznie efektywniejsze od nieustannego generowania całego zbioru wynikowego i odrzucania jego większości.

Jeśli nie można użyć powyższych strategii, zastosowanie oddzielnego zapytania `COUNT(*)` w celu określenia liczby rekordów może być szybsze niż wykorzystanie opcji `SQL_CALC_FOUND_ROWS`, o ile możliwe jest użycie indeksu pokrywającego.

Optymalizacja klauzuli UNION

Baza danych MySQL zawsze wykonuje zapytania `UNION` poprzez utworzenie tabeli tymczasowej i umieszczenie w niej wyników operacji `UNION`. Serwer MySQL nie potrafi zastosować na zapytaniach `UNION` tak wielu optymalizacji, ilu mógłby użyć programista. Dlatego też może się zdarzyć, że będzie trzeba udzielić optymalizatorowi pomocy i ręczne „wciągnąć" klauzule `WHERE`, `LIMIT`, `ORDER BY` oraz inne warunki (np. kopiując je z zewnętrznego zapytania do każdego zapytania `SELECT` w konstrukcji `UNION`).

Zawsze trzeba używać klauzuli `UNION ALL`, chyba że nie jest wymagane usunięcie przez serwer powielających się rekordów. Jeżeli słowo kluczowe `ALL` zostanie pominięte, MySQL dołączy do tabeli tymczasowej opcję `distinct`, która powoduje używanie pełnych rekordów w celu ustalenia unikalności. To jest kosztowne. Należy mieć na uwadze, że słowo kluczowe `ALL` nie powoduje wyeliminowania tabeli tymczasowej. Baza danych MySQL zawsze będzie umieszczała wyniki w tabeli tymczasowej, a następnie ponownie je odczytywała, nawet jeśli nie będzie to naprawdę konieczne (np. kiedy wyniki mogłyby zostać bezpośrednio zwrócone klientowi).

Opcje dotyczące optymalizatora zapytań

W bazie danych MySQL umieszczono kilka opcji optymalizatora, które można wykorzystać w celu nadzorowania planu wykonywania zapytania, jeśli programista nie jest zadowolony z wyborów dokonanych przez optymalizator MySQL. Poniżej przedstawiono opcje oraz wskazano sytuacje, w których korzystne jest ich użycie. Odpowiednią opcję należy umieścić w modyfikowanym zapytaniu; będzie efektywna tylko w tym konkretnym zapytaniu. Składnię każdej opcji można znaleźć w podręczniku użytkownika MySQL. Niektóre z nich są zdecydowanie powiązane z wersją serwera bazy danych. Oto dostępne opcje.

`HIGH_PRIORITY` *oraz* `LOW_PRIORITY`
: Opcje informują MySQL, w jaki sposób ustalać priorytet polecenia względem innych poleceń, które próbują uzyskać dostęp do tych samych tabel.

  Opcja `HIGH_PRIORITY` informuje MySQL, że polecenia `SELECT` mają być wykonywane przed wszelkimi innymi poleceniami, które mogą oczekiwać na możliwość nałożenia blokad, czyli chcą zmodyfikować dane. W efekcie polecenia `SELECT` zostaną umieszczone

na początku kolejki, zamiast oczekiwać na swoją kolej. Ten modyfikator można również zastosować względem poleceń INSERT, co po prostu spowoduje zniwelowanie efektu globalnego ustawienia serwera o nazwie LOW_PRIORITY.

Opcja LOW_PRIORITY jest przeciwieństwem poprzedniej: powoduje umieszczenie polecenia na samym końcu kolejki oczekiwania, jeśli, oczywiście, są inne polecenia, które chcą uzyskać dostęp do tabel — nawet gdy inne polecenia zostały wydane później. Przypomina to nadmiernie uprzejmą osobę stojącą przed drzwiami restauracji: dopóki ktokolwiek inny będzie czekał przed restauracją, uprzejma osoba będzie przymierała głodem. Opcję tę można zastosować względem poleceń SELECT, INSERT, UPDATE, REPLACE oraz DELETE.

Opcje są efektywne w silnikach magazynu danych obsługujących blokowanie na poziomie tabeli, ale nigdy nie powinno się ich stosować w InnoDB bądź innych silnikach zapewniających bardziej szczegółową kontrolę blokowania i współbieżności. Należy zachować szczególną ostrożność podczas używania ich w silniku MyISAM, ponieważ mogą wyłączyć możliwość przeprowadzania operacji jednoczesnego wstawiania danych, a tym samym znacznie zmniejszyć wydajność.

Często opcje HIGH_PRIORITY oraz LOW_PRIORITY są źródłem pewnego zamieszania. Nie powodują zarezerwowania dla zapytań większej bądź mniejszej ilości zasobów, aby „pracowały ciężej" lub „nie pracowały tak ciężko". Wymienione opcje po prostu wpływają na sposób kolejkowania przez serwer zapytań, które oczekują na uzyskanie dostępu do tabeli.

DELAYED
: Opcja jest przeznaczona do używania z poleceniami INSERT i REPLACE. Pozwala poleceniu na natychmiastowe zwrócenie danych oraz umieszczenie wstawianych rekordów w buforze, a następnie wstawienie ich razem do tabeli, gdy tylko będzie dostępna. Taka opcja jest najbardziej użyteczna podczas rejestrowania zdarzeń oraz w podobnych aplikacjach, w których trzeba wstawić duże ilości rekordów bez wstrzymywania klienta oraz bez powodowania operacji I/O w przypadku każdego zapytania. Istnieje kilka ograniczeń związanych z tą opcją, np. opóźnione operacje wstawiania nie są zaimplementowane we wszystkich silnikach magazynu danych. Poza tym, funkcja LAST_INSERT_ID() nie działa z opóźnionymi operacjami wstawiania.

STRAIGHT_JOIN
: Opcja ta pojawia się albo tuż za słowem kluczowym SELECT w poleceniu SELECT, albo między dwiema łączonymi tabelami w każdym innym poleceniu. Pierwszy sposób użycia wymusza, aby wszystkie tabele w zapytaniu były złączane z zachowaniem kolejności ich przedstawienia w zapytaniu. Drugi sposób wymusza zachowanie kolejności podczas złączenia dwóch tabel, między którymi znajduje się ta opcja.

  Opcja STRAIGHT_JOIN jest użyteczna, w sytuacji kiedy baza danych MySQL nie wybiera dobrej kolejności złączenia lub optymalizator wymaga dużej ilości czasu na podjęcie decyzji dotyczącej stosowanej kolejności złączenia. W tym drugim przypadku wątek spędza dużo czasu w stanie „Statistics", a dodanie wymienionej opcji powoduje ograniczenie optymalizatorowi przestrzeni wyszukiwania.

  Kolejność wybraną przez optymalizator można poznać za pomocą danych wyjściowych polecenia EXPLAIN. Należy napisać nowe zapytanie w tej kolejności i dodać opcję STRAIGHT_JOIN. Jest to bardzo dobry pomysł, przynajmniej tak długo, jak długo ustalona kolejność nie skutkuje kiepską wydajnością w niektórych klauzulach WHERE. Jednak po uaktualnieniu serwera MySQL należy ponownie przejrzeć takie polecenia, ponieważ mogą pojawić się nowe rodzaje optymalizacji, które będą unieważniane przez opcję STRAIGHT_JOIN.

`SQL_SMALL_RESULT` *oraz* `SQL_BIG_RESULT`
: Opcje te są przeznaczone dla poleceń `SELECT`. Informują optymalizator, jak i kiedy używać tabel tymczasowych oraz sortować zapytania `GROUP BY` i `DISTINCT`. Opcja `SQL_SMALL_RESULT` informuje optymalizator, że zbiór wynikowy będzie mały i może zostać umieszczony w zindeksowanej tabeli tymczasowej w celu uniknięcia sortowania dla grupowania. Z kolei opcja `SQL_BIG_RESULT` wskazuje, że zbiór wynikowy będzie ogromny i lepszym rozwiązaniem jest użycie tabel tymczasowych na dysku wraz z sortowaniem.

`SQL_BUFFER_RESULT`
: Opcja ta nakazuje optymalizatorowi umieszczenie wyników w tabeli tymczasowej oraz zwolnienie blokad tabeli tak wcześnie, jak tylko będzie to możliwe. To zupełnie inna opcja od buforowania po stronie klienta, przedstawionego w podrozdziale „Protokół klient-serwer MySQL", we wcześniejszej części rozdziału. Buforowanie po stronie serwera może być bardzo użyteczne, kiedy nie jest stosowane buforowanie po stronie klienta, pozwala bowiem uniknąć zużywania ogromnych ilości pamięci po stronie klienta i nadal powoduje bardzo szybkie zwalnianie blokad. Jednak oznacza również wykorzystanie pamięci serwera zamiast klienta.

`SQL_CACHE` *oraz* `SQL_NO_CACHE`
: Opcje te informują serwer, że dane zapytanie jest lub nie jest kandydatem do buforowania w buforze zapytań. Szczegółowe informacje na temat używania tych opcji zostały przedstawione w następnym rozdziale.

`SQL_CALC_FOUND_ROWS`
: Opcja nakazuje MySQL obliczenie pełnego zbioru wynikowego, gdy stosowana jest klauzula `LIMIT`, choć zwracane będą jedynie rekordy wskazane przez tę klauzulę. Za pomocą funkcji `FOUND_ROWS()` istnieje możliwość pobrania całkowitej liczby znalezionych rekordów. (Warto powrócić do podrozdziału „Optymalizacja za pomocą opcji SQL_CALC_FOUND_ROWS" we wcześniejszej części rozdziały, aby przypomnieć sobie, dlaczego nie powinno używać się tej opcji).

`FOR UPDATE` *oraz* `LOCK IN SHARE MODE`
: Opcje nadzorują blokady w poleceniach `SELECT`, ale jedynie w przypadku silników magazynu danych obsługujących blokowanie na poziomie rekordu. Wymienione opcje pozwalają na nałożenie blokad na dopasowanych rekordach. Taka możliwość może być użyteczna, gdy programista chce zablokować rekordy, o których wiadomo, że będą później uaktualniane, ewentualnie wtedy, kiedy chce uniknąć eskalacji blokad i po prostu nakładać blokady na wyłączność tak szybko, jak będzie to możliwe.

  Opcje nie są potrzebne w zapytaniach `INSERT ... SELECT`, które w MySQL 5.0 domyślnie umieszczają blokady odczytu na rekordach źródłowych. (Istnieje możliwość wyłączenia takiego zachowania, ale nie jest to dobry pomysł — wyjaśnienie znajduje się w rozdziałach 8. i 11.). Baza danych MySQL 5.1 może znieść to ograniczenie w pewnych warunkach.

  W czasie pisania tej książki jedynie silnik InnoDB obsługiwał te opcje i było jeszcze zbyt wcześnie, aby stwierdzić, czy inne silniki magazynu danych oferujące blokady na poziomie rekordu będą w przyszłości je obsługiwały. Podczas używania opcji w InnoDB należy zachować ostrożność, ponieważ mogą doprowadzić do wyłączenia pewnych optymalizacji, np. stosowania indeksu pokrywającego. Silnik InnoDB nie może zablokować rekordów na wyłączność bez uzyskania dostępu do klucza podstawowego, który jest miejscem rekordu przechowującym informacje dotyczące wersji.

`USE INDEX`, `IGNORE INDEX` *oraz* `FORCE INDEX`
: Opcje informują optymalizator, które indeksy mają być używane bądź ignorowane podczas wyszukiwania rekordów w tabeli (np. w trakcie decydowania o kolejności złączenia). W bazie danych MySQL 5.0 i wcześniejszych wymienione opcje nie wpływają na to, które indeksy będą stosowane przez serwer podczas operacji sortowania i grupowania. W MySQL 5.1 składnia może pobierać opcjonalną klauzulę `FOR ORDER BY` lub `FOR GROUP BY`.

  Opcja `FORCE INDEX` jest taka sama jak `USE INDEX`, ale informuje optymalizator, że skanowanie tabeli jest wyjątkowo kosztowne w porównaniu ze skanowaniem indeksu, nawet jeśli indeks nie jest zbyt użyteczny. Opcje można zastosować, kiedy programista uważa, że optymalizator wybiera niewłaściwy indeks bądź z jakiegokolwiek powodu mają być wykorzystane zalety płynące z użycia indeksu, np. jawne ustalenie kolejności bez użycia klauzuli `ORDER BY`. Przykład takiego rozwiązania pokazano w podrozdziale „Optymalizacja zapytań typu LIMIT i OFFSET", we wcześniejszej części rozdziału, podczas omawiania efektywnego pobierania wartości minimalnej za pomocą klauzuli `LIMIT`.

W bazie danych MySQL 5.0 i nowszych występują jeszcze pewne zmienne systemowe, które mają wpływ na działanie optymalizatora. Oto one.

`optimizer_search_depth`
: Zmienna wskazuje optymalizatorowi, w jaki sposób wyczerpująco analizować plany częściowe. Jeżeli wykonywanie zapytania w stanie „Statistics" zabiera bardzo dużo czasu, można spróbować obniżyć wartość tej zmiennej.

`optimizer_prune_level`
: Zmienna, domyślnie włączona, pozwala optymalizatorowi na pomijanie określonych planów na podstawie liczby przeanalizowanych rekordów.

Obie opisane zmienne nadzorują skracanie działania optymalizatora. Takie skróty są cenne pod względem wydajności w skomplikowanych zapytaniach, ale mogą spowodować, że serwer „przeoczy" optymalne plany w imię efektywności. To jest powód, dla którego zmiana tych opcji czasami ma sens.

Zmienne zdefiniowane przez użytkownika

Bardzo łatwo zapomnieć o zmiennych zdefiniowanych przez użytkownika w MySQL, ale mogą one stanowić technikę o potężnych możliwościach, służącą do tworzenia efektywnych zapytań. Zmienne takie działają szczególnie dobrze w przypadku zapytań odnoszących korzyści z połączenia logik proceduralnej i relacyjnej. Czysto relacyjne zapytania traktują wszystko jak nieuporządkowane zbiory, którymi serwer w pewien sposób manipuluje, wszystkimi jednocześnie. Baza danych MySQL stosuje nieco bardziej pragmatyczne podejście. Można to uznać za wadę, ale równocześnie może okazać się zaletą, gdy programista nauczy się korzystać z tej możliwości. A zmienne zdefiniowane przez użytkownika mogą dodatkowo pomóc.

Zmienne zdefiniowane przez użytkownika są tymczasowymi magazynami dla wartości, które będą zachowane aż do zakończenia połączenia z serwerem. Definicja zmiennej polega po prostu na przypisaniu jej wartości za pomocą poleceń `SET` lub `SELECT`[13].

[13] W niektórych sytuacjach przypisanie można wykonać za pomocą zwykłego znaku równości (=). Autorzy uważają jednak, że lepiej unikać dwuznaczności i zawsze stosować wyrażenie :=.

```
mysql> SET @one       := 1;
mysql> SET @min_actor := (SELECT MIN(actor_id) FROM sakila.actor);
mysql> SET @last_week := CURRENT_DATE-INTERVAL 1 WEEK;
```

Zmienną można zastosować w miejsce wyrażenia, np.:

```
mysql> SELECT ... WHERE col <= @last_week;
```

Zanim zostaną przedstawione zalety zmiennych zdefiniowanych przez użytkownika, warto spojrzeć na pewne ich cechy i wady, aby przekonać się, kiedy *nie można* z nich skorzystać.

- Uniemożliwiają buforowanie zapytania.
- Nie można ich użyć w sytuacji, gdy wymagany jest literał bądź identyfikator, np. nazwa tabeli lub kolumny albo w klauzuli `LIMIT`.
- Zmienne te są przywiązane do połączenia, a więc nie można ich użyć w trakcie komunikacji między połączeniami.
- Jeżeli stosowana jest pula połączeń bądź trwałe połączenie, mogą spowodować oddziaływanie na siebie pozornie wyizolowanych fragmentów kodu.
- W wersjach wcześniejszych niż MySQL 5.0 rozróżniają wielkość liter. Należy więc zachować ostrożność i wystrzegać się problemów związanych ze zgodnością kodu.
- Nie można jawnie zadeklarować rodzaju zmiennej, a punkt, w którym wybierany jest rodzaj dla niezdefiniowanej zmiennej, jest odmienny w różnych wersjach MySQL. Najlepszym wyjściem jest początkowe przypisanie wartości `0` zmiennym, które mają być używane z liczbami całkowitymi, `0.0` dla liczb zmiennoprzecinkowych oraz `''` (pusty ciąg tekstowy) dla ciągów tekstowych. Rodzaj zmiennej ulega zmianie po przypisaniu jej wartości. Ustalanie typu zmiennej zdefiniowanej przez użytkownika jest w serwerze MySQL przeprowadzane dynamicznie.
- W niektórych sytuacjach optymalizator może pozbyć się zmiennych, uniemożliwiając im wykonanie zadań zaplanowanych przez programistę.
- Kolejność przypisania (a wręcz godzina przypisania) może być niedeterministyczna i zależeć od planu wykonania zapytania wybranego przez optymalizator. Jak czytelnik przekona się w dalszej części rozdziału, wyniki mogą być bardzo mylące.
- Operator przypisania `:=` ma niższe pierwszeństwo od wszelkich pozostałych operatorów. Należy więc zachować szczególną ostrożność i stosować nawiasy, jasno określając kolejność.
- Niezdefiniowane zmienne nie powodują wygenerowania błędu składni. Bardzo łatwo popełnić błąd, nawet nie zdając sobie z tego sprawy.

Jedną z najważniejszych funkcji zmiennych jest fakt, że zmiennej można przypisać wartość, a następnie zastosować wartość otrzymaną w wyniku operacji przypisania. Innymi słowy, przypisanie jest *L-wartością*. Poniżej przedstawiono przykład jednoczesnego obliczenia i wyświetlenia „liczby rekordów" dla danego zapytania:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS rownum
    -> FROM sakila.actor LIMIT 3;
+----------+--------+
| actor_id | rownum |
+----------+--------+
|        1 |      1 |
|        2 |      2 |
|        3 |      3 |
+----------+--------+
```

Przedstawiony przykład zdecydowanie nie jest interesujący, ponieważ pokazuje, że powielono klucz podstawowy tabeli. Jednak wciąż ma swoje zastosowanie: jednym z nich jest ranking. W kolejnym przykładzie zaprezentowano zapytanie zwracające dziesięciu aktorów, którzy wystąpili w największej liczbie filmów. Użyta zostanie kolumna rankingu nadająca aktorowi tę samą pozycję, jeśli został zaangażowany. Trzeba rozpocząć od zapytania wyszukującego aktorów oraz liczbę filmów, w których wystąpili:

```
mysql> SELECT actor_id, COUNT(*) as cnt
    -> FROM sakila.film_actor
    -> GROUP BY actor_id
    -> ORDER BY cnt DESC
    -> LIMIT 10;
+----------+-----+
| actor_id | cnt |
+----------+-----+
|      107 |  42 |
|      102 |  41 |
|      198 |  40 |
|      181 |  39 |
|       23 |  37 |
|       81 |  36 |
|      106 |  35 |
|       60 |  35 |
|       13 |  35 |
|      158 |  35 |
+----------+-----+
```

Następnie należy dodać ranking, który powinien być taki sam dla wszystkich aktorów występujących w 35 filmach. W tym celu zostaną użyte trzy zmienne: pierwsza śledząca bieżącą pozycję w rankingu, druga przechowująca poprzednią liczbę filmów, w których wystąpił aktor, oraz trzecia przechowująca bieżącą liczbę filmów, w których wystąpił aktor. Pozycja w rankingu zostanie zmieniona wraz ze zmianą liczby filmów, w których zaangażowano danego aktora. Poniżej przedstawiono pierwsze podejście do utworzenia takiego zapytania:

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
mysql> SELECT actor_id,
    ->    @curr_cnt := COUNT(*) AS cnt,
    ->    @rank     := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
    ->    @prev_cnt := @curr_cnt AS dummy
    -> FROM sakila.film_actor
    -> GROUP BY actor_id
    -> ORDER BY cnt DESC
    -> LIMIT 10;
+----------+-----+------+-------+
| actor_id | cnt | rank | dummy |
+----------+-----+------+-------+
|      107 |  42 |    0 |     0 |
|      102 |  41 |    0 |     0 |
...
```

Ups! Zarówno ranking, jak i licznik nigdy nie przekraczają wartości zero. Dlaczego tak się stało?

Nie możliwe jest udzielenie jednej odpowiedzi na takie pytanie. Problem może być bardzo prosty i sprowadzać się do błędnie zapisanej nazwy zmiennej (w tym przypadku jednak tak nie jest) lub nieco bardziej skomplikowany. W omawianym przykładzie dane wyjściowe polecenia `EXPLAIN` pokazują, że używana jest tabela tymczasowa oraz sortowanie pliku. Dlatego też zmienne są obliczane w zupełnie innym czasie, niż jest to oczekiwane.

Jest to ten rodzaj tajemniczego zachowania, którego można często doświadczyć w MySQL podczas używania zmiennych zdefiniowanych przez użytkownika. Usuwanie takich błędów

może być trudne, ale naprawdę opłacalne. Utworzenie rankingu w MySQL zwykle wymaga algorytmu równania kwadratowego, np. zliczania różnych aktorów, którzy występowali w większej liczbie filmów. Rozwiązanie z użyciem zmiennej zdefiniowanej przez użytkownika może być algorytmem liniowym — całkiem spore usprawnienie.

W omawianym przypadku łatwym rozwiązaniem jest dodanie do zapytania innego poziomu tabel tymczasowych za pomocą podzapytania w klauzuli `FROM`:

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
    -> SELECT actor_id,
    ->    @curr_cnt := cnt AS cnt,
    ->    @rank     := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
    ->    @prev_cnt := @curr_cnt AS dummy
    -> FROM (
    ->    SELECT actor_id, COUNT(*) AS cnt
    ->    FROM sakila.film_actor
    ->    GROUP BY actor_id
    ->    ORDER BY cnt DESC
    ->    LIMIT 10
    -> ) as der;
+----------+-----+------+-------+
| actor_id | cnt | rank | dummy |
+----------+-----+------+-------+
|      107 |  42 |    1 |    42 |
|      102 |  41 |    2 |    41 |
|      198 |  40 |    3 |    40 |
|      181 |  39 |    4 |    39 |
|       23 |  37 |    5 |    37 |
|       81 |  36 |    6 |    36 |
|      106 |  35 |    7 |    35 |
|       60 |  35 |    7 |    35 |
|       13 |  35 |    7 |    35 |
|      158 |  35 |    7 |    35 |
+----------+-----+------+-------+
```

Większość problemów dotyczących zmiennych zdefiniowanych przez użytkownika wiąże się z przypisywaniem im wartości i odczytywaniem ich na różnych etapach zapytania. Przykładowo przewidywalnie nie będzie działało przypisanie zmiennej w poleceniu `SELECT` a odczytanie w klauzuli `WHERE`. Wydaje się, że przedstawione poniżej zapytanie zwróci po prostu jeden rekord, ale to błędne wyobrażenie:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
    -> FROM sakila.actor
    -> WHERE @rownum <= 1;
+----------+------+
| actor_id | cnt  |
+----------+------+
|        1 |    1 |
|        2 |    2 |
+----------+------+
```

Wynika to z faktu, że polecenia `WHERE` i `SELECT` znajdują się na różnych etapach procesu wykonywania zapytania. Stanie się to bardziej oczywiste po dodaniu kolejnego etapu wykonywania zapytania przez dołączenie klauzuli `ORDER BY`:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
    -> FROM sakila.actor
    -> WHERE @rownum <= 1
    -> ORDER BY first_name;
```

Powyższe zapytanie zwróci każdy rekord z tabeli, ponieważ klauzula ORDER BY dodaje operację sortowania pliku, a klauzula WHERE jest obliczana przed operacją sortowania. Rozwiązaniem problemu jest przypisanie oraz odczytanie wartości *na tym samym* etapie procesu wykonywania zapytania:

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum AS rownum
    -> FROM sakila.actor
    -> WHERE (@rownum := @rownum + 1) <= 1;
+----------+--------+
| actor_id | rownum |
+----------+--------+
|        1 | 1      |
+----------+--------+
```

Pytanie konkursowe: „Co się stanie, jeśli do powyższego zapytania ponownie zostanie dodana klauzula ORDER BY?". Warto ją dodać i się przekonać. Jeżeli wyniki nie są zgodne z oczekiwaniami, dlaczego tak się stało? Co można powiedzieć na temat poniższego zapytania, w którym klauzula ORDER BY zmienia wartość zmiennej, a klauzula WHERE ją oblicza?

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, first_name, @rownum AS rownum
    -> FROM sakila.actor
    -> WHERE @rownum <= 1
    -> ORDER BY first_name, LEAST(0, @rownum := @rownum + 1);
```

Odpowiedź na większość nieoczekiwanych zachowań związanych ze zmiennymi definiowanymi przez użytkownika można znaleźć po wykonaniu polecenia EXPLAIN i wyszukaniu informacji „Using where", „Using temporary" lub „Using filesort" w kolumnie Extra.

W ostatnim przykładzie zademonstrowano inną użyteczną sztuczkę — umieszczenie przypisania w funkcji LEAST(). Jej wartość będzie więc efektywnie maskowana i nie wypaczy wyniku działania klauzuli ORDER BY (jak wcześniej napisano, wartością zwrotną funkcji LEAST() zawsze jest 0). Sztuczka ta jest bardzo użyteczna, gdy programista chce dokonać przypisania zmiennej jedynie dla jej efektów ubocznych, pozwala bowiem na ukrycie wartości zwrotnej oraz pomaga w uniknięciu dodatkowych kolumn, takich jak dummy, pokazanych we wcześniejszym przykładzie. Funkcje GREATEST(), LENGTH(), ISNULL(), NULLIF(), COALESCE() oraz IF() również można wykorzystać w tym celu, samodzielnie oraz w połączeniu, ponieważ mają zachowania specjalne. Przykładowo funkcja COALESCE() zatrzymuje obliczanie argumentów, gdy tylko jeden z nich będzie miał zdefiniowaną wartość.

Przypisanie zmiennej można umieścić we wszystkich rodzajach poleceń, nie tylko w poleceniach SELECT. W rzeczywistości to jeden z najlepszych sposobów użycia dla zmiennych definiowanych przez użytkownika. Programista może np. przepisać kosztowne zapytania, takie jak obliczenia rankingu z podzapytaniami, na postać tanich jednoprzebiegowych poleceń UPDATE.

Jednak uzyskanie pożądanego zachowania może być trochę trudne. Czasami optymalizator decyduje się na uznanie w czasie kompilacji zmiennych za stałe i odmawia przeprowadzenia przypisania. Umieszczenie operacji przypisania wewnątrz funkcji, takiej jak LEAST(), zwykle pomaga rozwiązać ten problem. Innym sposobem jest sprawdzenie, czy zmienna ma zdefiniowaną wartość przed wykonaniem zawierającego ją polecenie. Czasami takie zachowanie jest pożądane, czasami nie.

Przy odrobinie eksperymentów zmienne definiowane przez użytkownika można wykorzystać do wykonywania różnych ciekawych operacji. Poniżej przedstawiono kilka pomysłów.

- Obliczanie wartości całkowitych i przeciętnych.
- Emulacja funkcji `FIRST()` oraz `LAST()` w zgrupowanych zapytaniach.
- Przeprowadzanie operacji matematycznych na wyjątkowo ogromnych liczbach.
- Redukcja całej tabeli na postać pojedynczej wartości hash typu MD5.
- „Odpakowanie" przykładowej wartości zapakowanej po przekroczeniu przez nią ustalonych granic.
- Emulacja odczytu i zapisu kursorów.

Podczas uaktualniania MySQL należy zachować ostrożność

Jak wcześniej wspomniano, próba przechytrzenia optymalizatora MySQL z założenia nie jest dobrym pomysłem. Ogólnie rzecz biorąc, wymaga wykonania większej ilości pracy i zwiększa koszty obsługi, a przynosi jedynie minimalne korzyści. Dotyczy to zwłaszcza operacji uaktualniania bazy danych MySQL, ponieważ opcje zastosowane przez programistę w optymalizatorze mogą uniemożliwić zastosowanie nowych strategii wprowadzonych przez kolejną wersję optymalizatora.

Optymalizator MySQL stosuje indeksy jako ruchomy cel. Nowe wersje MySQL zmieniają sposoby używania istniejących indeksów i należy dostosować praktyki korzystania z indeksów, gdy tylko nowa wersja serwera będzie dostępna. Przykładowo wspomniano, że MySQL 4.0 i wcześniejsze wersje mogą używać tylko jednego indeksu na tabelę w zapytaniu, ale MySQL 5.0 i nowsze wersje mogą zastosować strategię łączenia indeksów.

Oprócz dużych zmian okazjonalnie wprowadzanych w optymalizatorze zapytań MySQL, każde kolejne wydanie serwera zwykle zawiera wiele drobniejszych zmian. Zazwyczaj dotyczą drobnych elementów, np. warunków, kiedy indeks zostanie wykluczony, oraz pozwalają MySQL na przeprowadzenie optymalizacji przypadków specjalnych.

Chociaż w teorii brzmi to całkiem dobrze, w praktyce niektóre zapytania działają *gorzej* po przeprowadzeniu uaktualnienia. Jeżeli programista używał danej wersji przez długi okres czasu, prawdopodobnie zapytania zostały dostrojone do niej, niezależnie od tego, czy programista zdaje sobie z tego sprawę, czy nie. Wprowadzone optymalizacje mogą nie funkcjonować w nowszych wersjach serwera bądź prowadzić do zmniejszenia wydajności.

Jeżeli programiście szczególnie zależy na wysokiej wydajności, należy przeprowadzać testy wydajności obejmujące faktyczny poziom obciążenia. Testy trzeba przeprowadzić na serwerze projektowym przy użyciu nowej wersji oprogramowania i jeszcze przed aktualizacją serwerów produkcyjnych. Ponadto przed przeprowadzeniem uaktualnienia należy zapoznać się z listą wprowadzonych zmian w oprogramowaniu oraz listą znanych błędów w nowej wersji serwera. Podręcznik użytkownika MySQL zawiera listę znanych poważnych błędów przedstawioną w dostępnej postaci.

Większość uaktualnień MySQL przynosi lepszą ogólną wydajność, autorzy nie sugerują, że mogłoby być inaczej. Jednak zawsze należy zachować ostrożność.