Praca magisterska

Franciszek Słupski

Brak

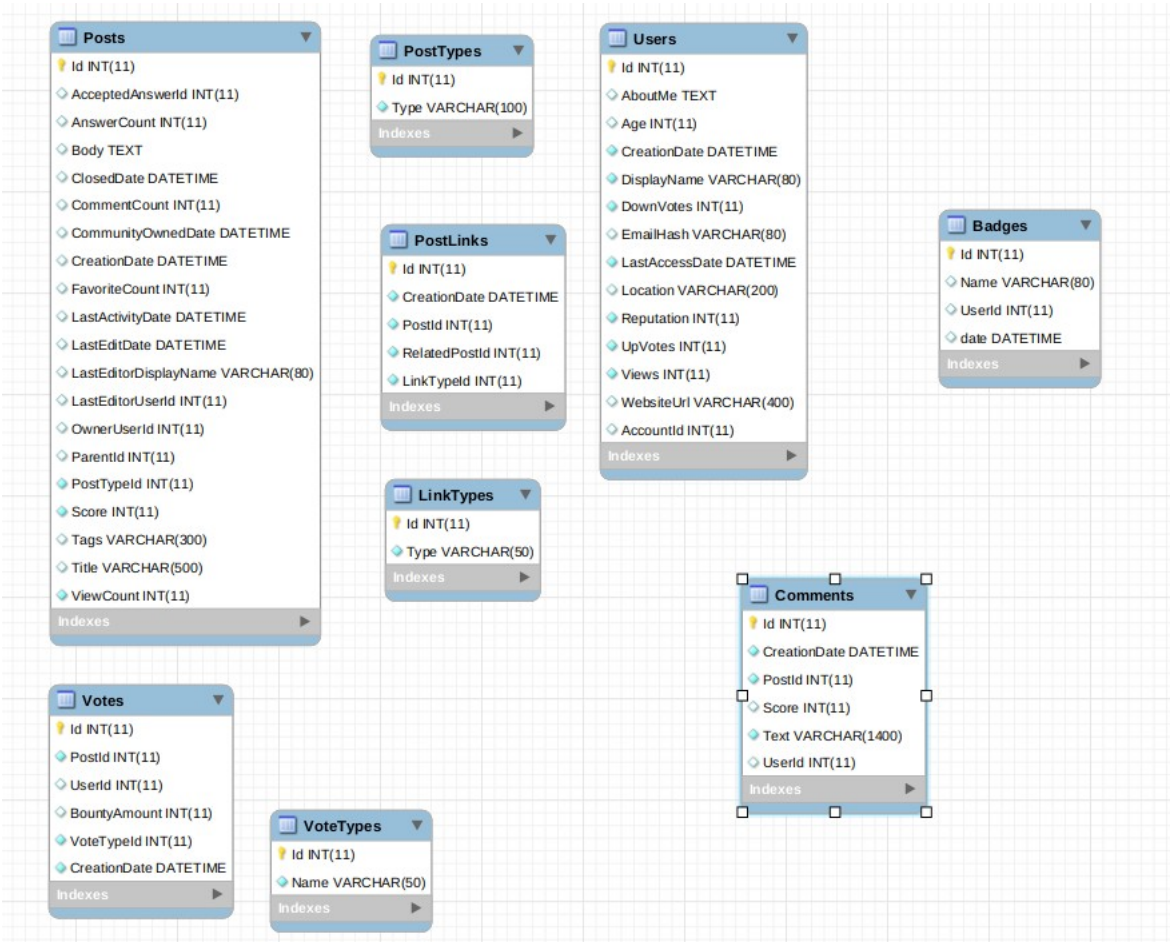
1 Wstęp

1.1 Testowa baza danych

Aby przedstawić techniki optymalizacji zawarte w pracy na rzeczywistych przykładach, wykorzystałem bazę danych udostępnioną przez portal stackoverflow.com. Baza zawiera w granicach 50 Gb danych zebranych w latach 2008-2013. Archiwum po zaimportowaniu do serwera MySQL nie zawiera klucz głównych, kluczy obcych, indeksów. Początkowy schemat bazy danych jest przedstawiony na rysunku 1.

1.2 Porównywanie zapytań

W rozdziale zostanie przedstawione działanie polecenia EXPLAIN. Polecenie to umożliwia uzyskanie informacji o planie wykonania zapytania. Jest podstawowym sposobem określenia, w jaki sposób MySQL wykonuje zapytania. Analiza wyników EXPLAIN jest zdecydowanie bardziej użyteczna od mierzenia czasów zapytań. Na czas wykonania zapytania może mieć wpływ kilka czynników, które wprowadzą nas w błąd podczas badania wydajności danego zapytania. Pierwszym z nich jest cache zapytań. Przeprowadzając testy zapytania przy włączonym buforze zapytań, może zdarzyć się, że



Rysunek 1: Schemat bazy danych stackoverflow

rezultat zapytania zostanie zwrócony z tego właśnie bufora. Doprowadzi to do sytuacji, kiedy nawet najbardziej niewydajne zapytania będą zwracane w ułamki sekund. Problem ze zwracaniem wyników z bufora zapytań możemy rozwiązać poprzez wyłączenie bufora zapytań lub dodanie modyfikatora SQL_NO_CACHE do zapytań. Drugim czynnikiem zaburzającym mierzenie czasów wykonania zapytań jest bufor MySQL. MySQL stara się przechowywać w pamięci często używane dane, dla przykładu indeksy. Jeżeli wykonujemy zapytanie dla tabeli, której indeks nie znajduje się w pamięci. Serwer pobiera indeks z dysku, co trwa. Następnie poprawiamy zapytanie w celu poprawienia wydajności i wykonujemy, aby sprawdzić, czy nasze działanie poprawiło wydajność. Tym razem cały indeks znajduje się w pamięci i zapytanie wykonuje się wielokrotnie szybciej. Mierząc jedynie czasy wykonania obu zapytań, możemy dojść do fałszywego wniosku, że drugie zapytanie jest zdecydowanie bardziej wydajne, nawet jeżeli w rzeczywistości nasze działanie doprowadziło do pogorszenia wydajności zapytania. W takim przypadku dobrym rozwiązaniem wydaje się kilkukrotne mierzenie czasów, obliczenie średniej i na tej podstawie porównywanie wyników. Dodatkowo nasz serwer rzadko kiedy jest całkowicie odcięty od świata. Bardzo często będziemy testować wydajność zapytań na tabelach, które są w równocześnie modyfikowane w tle. Dla przykładu jeżeli testujemy zapytanie na tabeli, na której wykonywane są w tym momencie operacje zapisu, nasze wyniki będą zwracane w czasach, które będzie nam bardzo trudno interpretować. Dodatkowo na czasy wykonywania zapytań wpływ może mieć aktualne obciążenie serwera, co nie ułatwia pracy przy porównywaniu wyników. Jak widzimy aby skutecznie porównywać wydajność zapytań nie powinniśmy opierać się jedynie na czasie ich wykonania.

1.3 Polecenie EXPLAIN

Polecenie EXPLAIN będzie jedną z podstawowych metod porównywania wydajności zapytań stosowaną w tej pracy, dlatego w tym podroździale przed-

stawie podstawy używania tego polecenia. Funkcja EXPLAIN to główny sposób określania, w jaki sposób optymalizator decyduje o sposobie wykonania zapytania.

Aby użyć polecenia EXPLAIN, wystarczy poprzedzić słowa kluczowe takie jak SELECT,INSERT,UPDATE,DELETE słowem EXPLAIN. Spowoduje to, że zamiast wykonania samego zapytania, sewer zwróci informacje o planie wykonania zapytania. Rezultat polecenia EXPLAIN zawiera po jednym rekordzie dla każdej tabeli użytej w zapytaniu.

1.3.1 Wyniki polecenia EXPLAIN

Aby przedstawić wyniki polecenia EXPLAIN na rzeczywistych przykładach wykonałem kilka zapytań EXPLAIN na bazie StackOverflow. Dla porządku uznajmy, że zapytania są ponumerowane względem kolejności ich występowania w rozdziale.

```
SELECT u.DisplayName, c.CreationDate, c.‘Text‘ FROM  Comments c left
join Users u on c.UserId = u.Id where c.PostId = 875;
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	c	NULL	ALL	NULL	NULL	NULL	NULL	22155797	10.00	Using where
2	1	SIMPLE	u	NULL	eq_ref	PRIMARY	PRIMARY	4	stackOverflowMedium.c.UserId	1	100.00	NULL

Rysunek 2: Przykład 1

```
SELECT p.Body from Posts p where p.Id = 875 union
select c.‘Text‘ from Comments c where c.PostID = 875;
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	PRIMARY	p	NULL	const	PRIMARY	PRIMARY	4	const	1	100.00	NULL
2	2	UNION	c	NULL	ALL	NULL	NULL	NULL	NULL	22155797	10.00	Using where
3	NULL	UNION RESULT	<union1,2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	NULL	Using temporary

Rysunek 3: Przykład 2

```
SELECT * from Comments where UserId = (select id from Users where
DisplayName = 'Jarrod Dixon');
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	PRIMARY	Comments	NULL	ref	user_post_idx	user_post_idx	5	const	446	100.00	Using where
2	2	SUBQUERY	Users	NULL	ALL	NULL	NULL	NULL	NULL	2270672	10.00	Using where

Rysunek 4: Przykład 3

```
SELECT * FROM Comments where UserID in ( select UserId from Posts
group by UserId having count(*) > 10);
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	PRIMARY	Comments	NULL	ref	user_post_idx	user_post_idx	5	const	446	100.00	Using where
2	2	SUBQUERY	Users	NULL	ALL	NULL	NULL	NULL	NULL	2270672	10.00	Using where

Rysunek 5: Przykład 4

```
SELECT * FROM Comments where UserID in ( select UserId from Posts
group by UserId having count(*) > 10);
```

#	id	select_type	table	partitions	type	possit	key	key_len	ref	rows	filtered	Extra
1	1	PRIMARY	Comments	NULL	ALL	NULL	NULL	NULL	NULL	22155797	100.00	Using where
2	2	DEPENDENT SUBQUERY	Posts	NULL	index	NULL	PRIMARY	4	NULL	17189835	100.00	Using index; Using temporary; Using filesort

Rysunek 6: Przykład 5

```
SELECT * FROM Posts  WHERE OwnerUserId in (select id from Users where
Reputation>1000 UNION SELECT UserId FROM Comments WHERE Score >10)
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	PRIMARY	Posts	NULL	ALL	NULL	NULL	NULL	NULL	17189835	100.00	Using where
2	2	DEPENDENT SUBQUERY	Users	NULL	eq_ref	PRIMARY	PRIMARY	4	func	1	33.33	Using where
3	3	DEPENDENT UNION	Comments	NULL	ref	user_post_idx	user_post_idx	5	func	23	33.33	Using where
4	NULL	UNION RESULT	<union2,3>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	NULL	Using temporary

Rysunek 7: Przykład 6

```
SELECT * FROM Comments WHERE UserId = (select @var1 from Users where
DisplayName = 'Jarrod Dixon');
```

#	id	select_type	table	partitions	type	possit	key	key_len	ref	rows	filtered	Extra
1	1	PRIMARY	Comments	NULL	ALL	NULL	NULL	NULL	NULL	22155797	100.00	Using where
2	2	UNCACHEABLE SUBQUERY	Users	NULL	ALL	NULL	NULL	NULL	NULL	2270672	10.00	Using where

Rysunek 8: Przykład 7

```
SELECT * FROM Comments limit 10;
```

#	id	select_type	table	partitions	type	possit	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Comments	NULL	ALL	NULL	NULL	NULL	NULL	22155797	100.00	NULL

Rysunek 9: Przykład 8

Kolumna ID

Kolumna id zawiera numer zapytania, którego dotyczy. W przypadku zapytań z podzapytaniami, podzapytania w dyrektywie FROM oraz zapytań z słowem kluczowym JOIN podzapytania numerowane są najczęściej względem ich występowania w zapytaniu. Kolumna ID może przyjąć również wartość NULL, w przypadku polecenia UNION (przykład 2).

Kolumna select_type

Kolumna select_type pokazuje, czy rekord jest prostym, czy złożonym zapytaniem SELECT. Wartość **SIMPLE** oznacza, że zapytanie nie zawiera podzapytań, oraz nie używa klauzuli UNION.

Jeżeli natomiast zapytanie zawiera podzapytania lub wykorzystuje klauzulę UNION, to rekord dla kolumny select_type przyjmie wartość **PRIMARY** (przykład 2). Jeżeli rekord dotyczy podzapytania oznaczonego jako PRIMARY, to będzie oznaczony jako **SUBQUERY** (przykład 3). Jako **UNION** zostaną oznaczone zapytania, które są drugim i kolejnym zapytaniem w klauzuli UNION. Pierwsze zapytanie zostanie oznaczone tak samo,

jakby było wykonywane jako zwykłe zapytanie SELECT (przykład 2). **DERIVED** oznacza, że zapytanie jest umieszczone jako podzapytanie w klauzuli FROM, jest wykonywane rekurencyjnie i wyniki są umieszczane w tabeli tymczasowej. Wartość **UNION RESULT** oznacza wiersz, jako polecenie SELECT użyte do pobrania wyników z tabeli tymczasowej użytej przy poleceniu UNION (przykład 2). Jeśli polecenie SELECT zależy od danych znajdujących się w podzapytaniu lub znajdujących się w wyniki klazuli UNION, to zostanie oznaczone odpowiednio jako **DEPENDENT SUBQUERY** (przykład 5) lub **DEPENDENT UNION** (Przykład 6). Dodatkowo w przypadku, jeżeli wynik zwracany jest z *zmaterializowanego widoku (eng. materialized view)*, zapytanie zostanie oznaczone jako **MATERIALIZED**. W przykładzie 7, który jest oczywiście nonsensowny, ale dobrze obrazuje sytuację, kiedy jako select_type otrzymamy wartość **UNCACHABLE_SUBQUERY**, która oznacza, że coś w podzapytaniu uniemożliwiło jego buforowanie. Analogiczną sytuację mamy, jeżeli wiersz zostanie oznaczony jako **UNCACHABLE_UNION**, ale w tym przypadku niemożliwe jest oczywiście buforowanie wyników polecenia UNION.

Kolumna table

Kolumna *table* w większości przypadków zawiera nazwę tabeli lub jej alias, do której odnosi się dany wiersz wyniku polecenia *EXPLAIN*. W przypadku gdy zapytanie dotyczy tabel tymczasowych możemy zobaczyć np. table: <union1,2> (przykład 2), co oznacza, że zapytanie dotyczy tabeli tymczasowej stworzonej na podstawie polecenia *UNION* na tabelach z wierszy o id 1 oraz 2. Odczytując kolejno wartości kolumny *table* możemy dowiedzieć się, w jakiej kolejności optymalizator MySQL zdecydował się ułożyć zapytania.

Kolumna Type

Kolumna *Type* informuje o tym, w jaki sposób MySQL będzie przetwarzał wiersze w tabeli. Poniżej przedstawiono najważniejsze metody dostępu do

danych, w kolejności od najgorszej do najlepszej.

ALL

Wartość *ALL* informuje o tym, że serwer musi przeskanować całą tabelę w celu odnalezienia rekordów. Istnieją jednak wyjątki takie, jak w przykładzie 8, w którym polecenie *EXPLAIN* pokazuje, że będzie wykonywany pełny skan tabeli, a w rzeczywistości dzięki użyciu polecenia *LIMIT* zapytanie będzie wymagało jedynie 10 rekordów.

index

MySQL skanuje wszystkie wiersze w tabeli, ale może wykonać to w porządku w jakim jest przechowywane w indeksie, dzięki czemu unika sortowania. Największą wadą jest konieczność odczytu całej tabeli w kolejności indeksu, co zazwyczaj oznacza pobieranie danych z dysku w losowej kolejności. Jeżeli w kolumnie *extra* jest dodatkowo zawarta informacja "Using Indexóznacza to, że MySQL wykorzystuje idenks pokrywający (opisany w dalszej części pracy) i nie wymaga odczytywania indeksów z dysku.

range

Wartość *range* oznacza ograniczone skanowanie zakresu. Takie skanowanie rozpoczyna się od pewnego miejsca indeksu, dzięki czemu nie musimy przechodzić przez cały indeks. Skanowanie indeksu powodują zapytania zawierające klauzulę *BETWEEN* lub *WHERE* z $<$ lub $>$. Wady są takie same jak przy rodzaju *index*

index_subquery

TODO przyklad

unique_subquery

TODO przyklad

index merge

TODO przyklad

fulltext

TODO przyklad

ref

Jest to wyszukiwanie, w którym MySQL musi przeszukać jedynie indeks w celu znalezienia rekordu opowiadającego pojedyńczej wartości. Przykładem takiego zapytania może być wyszukiwanie numerów postów danego użytkownika w tabeli *Comments* zawierającej indeks typu *BTREE* na kolumnach *UserId* oraz *PostId*.

```
SELECT PostId FROM Comments WHERE UserId = 10;
```

Dodatkowo odmianą dostępu *ref* jest dostępd *ref_or_null*, który oznacza, że wymagany jest dodatkowy dostęp w celu sprawdzenia wartości NULL.

eq_ref

Jest to najlepsza możliwa forma złączenia. Oznacza, że każdy wiersz z pierwszej tabeli jest dopasowywany do pierwszego zwróconego wyniku w drugiej tabeli i nie jest wymagane dalsze przeszukiwanie. Z tego rodzaju złączeniem mamy do czynienia, jeżeli wszystkie kolumny używane do złączenia są kluczem głównym lub indeksem ŃOT NULL UNIQUE". Przykładem takiego zapytania jest złączenie wszystkich komentarzy z postami bazując na kluczu głównym Id z tabeli Posts.

```
SELECT * FROM Comments c JOIN Posts p ON c.PostId = p.id;
```

const

Przeważnie występuje w przypadku użycia w klauzuli WHERE wartości z

indeksu głównego. Wtedy MySQL musi znaleźć tylko jedną wartość, która jest kluczem. Dla przykładu w bazie StackOverflow może to być zapytanie pobierające komentarz bazując na Id.

```
EXPLAIN SELECT * FROM Comments WHERE id = 93;
```

NULL

Oznacza, że serwer nie wymaga dostepu do tabeli czy indeksu i może zwrócić wartość już podczas fazy optymalizacji. Przykładem takiego zapytania może być zwrócenie minimalnej wartości z indeksu tabeli.

```
SELECT MIN(UserId) FROM Comments;
#Tabela Comments zawiera indeks BTREE na kolumnie UserID
```

Kolumna Possible_keys

Komulna possible_keys zawiera listę indeksów, które optymalizator brał pod uwagę podczas tworzenia planu wykonania zapytania. Lista tworzona jest na początku procesu optymalizacji zapytania.

Kolumna key

Kolumna *key* sygnalizuje, który indeks został wybrany do optymalizacji dostępu do tabeli.

Kolumna key_len

Wartość oznacza jaki jest rozmiar bajtów użytego indeksu. W przypadku, kiedy zostanie wykorzystana jedynie część kolumn indeksu, wtedy wartość *key_len* będzie odpowiednio mniejsza. Istotny jest fakt, że rozmiar jest zawsze maksymalnym rozmiarem zindeksowanych kolumn, a nie rzeczywistą liczbą bajtów danych używanych przez tabelę.

Kolumna ref

Kolumna pokazuje, które kolumny z innych tabel lub zmienne z innych tabel zostaną wykorzystane do wyszukania wartości w indeksie podanym w kolumnie *key*. W przykładzie 1, widzimy, że do przeszukania indeksu tabeli Posts została wykorzystana kolumna UserId z tabeli Comments (alias c). Wartość *const* oznacza, że do przeszukania wartości została wykorzystana stała podana np. w klauzuli WHERE (Przykład 2).

Kolumna rows

Kolumna wskazuje oszacowaną liczbę wierszy, które MySQL będzie musiał odczytać w celu znalezienia szukanych rekordów. Istotne jest to, że jest to liczba przeszukiwanych rekordów na danym poziomie zagnieżdenia pętli planu złączenia. To znaczy, że nie jest to całkowita liczba rekordów, a jedynie liczba rekordów w jednej pętli złączenia danej tabeli. W przypadku złączenia sumaryczna liczba przeszukiwanych nie jest sumą wartości z wszystkich wierszy, a iloczynem wartości z wierszy biorących udział w złączeniu. W przykładzie 9, łączna suma wierszy, które muszą zostać przeszukane nie wynosi 1574

```
SELECT * FROM Posts p JOIN PostTypes pt ON p.PostTypeId = pt.Id;
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	pt	NULL	ALL	NULL	NULL	NULL	NULL	7	100.00	NULL
2	1	SIMPLE	p	NULL	ALL	NULL	NULL	NULL	NULL	15748463	10.00	Using where; Using join buffer (Block Nested Loop)

Rysunek 10: Przykład 9

Dodatkowo należy wziąć pod uwagę, że są to jedynie szacunkowe wartości, które w praktyce mogą być zupełnie nie prawidłowe. Ponadto optymalizator podczas szacowania wartości w kolumnie *rows* nie bierze pod uwagę klauzli *LIMIT*.

Kolumna filtered

Kolumna *filtered* pojawia się jedynie podczas użycia polecenia *EXPLAIN EXTENDED*. Wskazuje na wartość oszacowaną przez optymalizator, która informuje ile rekordów może zostać odfiltrowane za pomocą klauzuli WHERE. W przykładzie 4 optymalizator MySQL oszacował, że jedynie 10 procent użytkowników napisało w sumie więcej niż 10 komentarzy.

Kolumna extra

Kolumna *extra* zawiera informacje, których nie udało się zamieścić w pozostałych kolumnach. Poniżej przedstawione zostanie kilka najważniejszych informacji, które mogą znaleźć się w tej kolumnie.

- 'Using index' - MySQL użyje indeksu pokrywającego zamiast dostępu do tabeli.
- 'Using where' - oznacza, że MySQL przeprowadzi filtrowanie danych już po wyciągnięciu danych z tabeli. Często jest to informacja, którą warto przemyśleć pod kątem stworzenia nowego (lub modyfikacji istniejących) indeksów.
- 'Using temporary' - do sortowania wyników używana jest tabela tymczasowa.
- 'Using filesort' - sortowanie nie może wykorzystywać indeksu, więc wiersze są sortowane za pomocą jednego z algorytmów sortowania.

2 Architektura MySQL

2.1 Obsługa połączeń i wątków

Serwer MySQL oczekuje na połączenia klientów na wielu interfejsach sieciowych:

- jeden wątek obsługuje połączenia TCP/IP (standardowo port 3306)
- w systemach UNIX, ten sam wątek obsługuje połączenia poprzez pliki gniazda
- w systemie Windows osobny wątek obsługuje połączenia komunikacji międzyprocesorowej
- w każdym systemie operacyjnym, dodatkowy interfejs sieciowy może obsługiwać połączenia administracyjne. Do tego celu może być wykorzystywany osobny wątek lub jeden z wątków menadżera połączeń.

Jeżeli dany system operacyjny nie wykorzystuje połączeń na innych wątkach, osobne wątki nie są tworzone.

Maksymalna ilość połączeń zdefinowana jest poprzez zmienną systemową max_connections, który domyślnie przyjmuje wartość 151. Dodaktowo MySQL jedno połączenie rezerwuje dla użytkownika z uprawnieniami SUPER lub CONNECTION_ADMIN. Taki użytkownik otrzyma połączenie nawet w przypadku braku dostępnych połączeń w głównej puli.

Do każdego klienta łączącego się do bazy MySQL przydzielany jest osobny wątek wewnątrz procesu serwera, który odpowiada za przeprowadzenie autentykacji oraz dalszą obsługę połączenia. Co ważne, nowy wątek tworzony jest jedynie w ostateczności. Jeżeli to możliwe, menadżer wątków stara się przydzielić wątek do połączenia, z puli dostępnych w pamięci podręcznej wątków.

3 Bufor zapytań

Bufor zapytań przechowuje gotowe odpowiedzi serwera dla poleceń SELECT. Jeżeli wynik danego zapytania znajduje się w buforze zapytań, serwer może zwrócić wynik bez konieczności dalszej analizy.

Proces wyszukiwania zapytania w buforze wykorzystuje funkcję skrótu. Dla każdego zapytania tworzony jest hash, który pozwala w prosty sposób zweryfikować, czy dane zapytanie znajduje się w buforze. Co ważne, hash uwzględnia wielkość liter, co prowadzi do sytuacji, gdzie dwa zapytania różniące się jedynie wielkością liter nie zostaną uznane za jednakowe.

Jeżeli tabela, z której pobierane są dane poprzez polecenie SELECT zostanie zmodyfikowana, wszystkie zapytania odnoszące się do takiej tabeli zostają usunięte z bufora. Dodatkowo bufor zapytań nie przechowuje zapytań uznanych, za niederministyczne. Przykładowo wszystkie polecenia pobierające aktualną datę, użytkownika itp nie zostaną dodane do bufora zapytań. Co istotne nawet w przypadku zapytania niederministycznego, serwer oblicza funkcję skrótu dla zapytania i próbuje dopasować odpowienie zapytanie z tabeli bufora. Dzieje się tak ze względu na fakt, że analiza zapytania odbywa się dopiero po przeszukaniu bufora i na etapie przeszukiwania bufora, serwer nie ma informacji o tym czy zapytanie jest deterministyczne. Jedynym filtrem, który weryfikuje zapytanie przed przeszukaniem bufora, jest sprawdzenie czy polecenie rozpoczyna się od liter SEL.

Jeżeli polecenie SELECT składa się z wielu podzapytań, ale nie znajduje się w tabeli bufora, to żadne z nich nie zostanie pobrane, ponieważ bufor zapytań działa na podstawie całego polecenia SELECT.

4 Indeksy w MySQL

Indeks jest strukturą danych zwiększającą szybkość operacji wyszukiwania na tabeli. Poprawne stosowanie indeksów jest krytyczne dla zachowania dobrej wydajności bazy danych. Najprostszą analogią pozwalającą zrozumieć działanie indeksu w bazie danych jest porównanie go z indeksem znajdującym się w książce. Zakładając, że książka nie zawiera indeksu, wyszukiwanie konkretnego słowa lub tematu w najgorszym wypadku wymaga przewertowania całej książki. Z tego powodu w książkach stosuje się indeksy, które zawierają

kluczowe słowa użyte w książce. Indeks taki zawiera listę słów oraz stron, na których słowa te występują. Dzięki temu wyszukanie konkretnego słowa wymaga jedynie sprawdzenia numeru strony w indeksie. Jest to szczególnie przydatne przy książkach zawierających dużą liczbę stron. Podobnie jest z tabelami w bazie danych. Przy tabelach o niewielkiej ilości wierszy, wyszukanie konkretnego rekordu trwa krótki nawet przy niestosowaniu indeksów. Indeksy stają się jednak kluczowe wraz ze wzrostem zbioru danych.

4.1 Rodzaje indeksów

W MySQL istnieje wiele rodzajów indeksów, które są implementowane w wartstwie silnika bazy danych, dlatego też nie każdy rodzaj indeksu jest obsługiwany przez wszystkie silniki. W tym rozdziale omówię tylko najpopularniejsze z nich.

4.1.1 Indeksy typu B-Tree

Indeks typu B-Tree jest zdecydowanie najczęściej stosowanym typem indeksu w bazach MySQL i jest domyślnie stosowany podczas tworzenia nowego indeksu, dlatego to właśnie jemu poświęce zdecydowaną wiekszość rozdziału dotyczącego indeksowania.

Struktura

Indeks typu B-Tree zbudowany jest na bazie struktury B-Drzewa. B-Drzewo jest drzewiastą strukturą danych przechowującą dane wraz z kluczami posortowanymi w pewnej kolejności. Każdy węzeł drzewa może posiadać od M+1 do 2M+1 dzieci, za wyjątkiem korzenia, który od 0 do 2M+1 potomków, gdzie M jest nazywany rzędem drzewa. Dzięki temu maksymalna wysokość drzewa zawierającego n kluczy wynosi $\log\_M\_n$. Takie właściwości sprawiają, że operacje wyszukiwania są złożoności asymptotycznej O($\log\_Mn$). Chcać być dokładnym, należy wspomnieć, że MySQL do

zapisu indeksów stosuje strukurę B+Drzewa, która jest szczególnym przypadkiem B-Drzewa i zawiera dane jedynie w liściach. Zastosowanie struktury B+Drzewa sprawia, że liście z danymi znajdują się w jednakowej odległości od korzenia drzewa. Wysoki rząd oznacza niską wysokość drzewa, to z kolei sprawia, że zapytanie wymaga mniejszej ilości operacji odczytu z dysku. Ma to fundamentalne znaczenie, ponieważ dane zapisane są na dyskach twardych, których czasy dostępu są dużo większe niż do pamięci RAM. Dla przykładu, załóżmy, że dana jest tabela zawierająca bilion wierszy, oraz indeks, którego rząd wynosi 64. Operacja wyszukania na danej tabeli wykorzystująca indeks będzie wymagać średnio tylu operacji odczytu, jaka jest wysokość drzewa przechowującego indeksy. Wysokość drzewa obliczamy ze wzoru $\log Mn$,gdzie M jest rzędem drzewa równym 64, a n oznacza ilość wierszy. W takim przypadku będziemy potrzebować zaledwie 5 odczytów danych z dysku. Dodatkowo silnik InnoDB nie przechowuje referencji do miejsca w pamięci, w którym znajdują się dane, ale odwołuje się do rekordów poprzez klucz podstawowy, który jednoznacznie identyfikuje każdy wiersz w tabeli.Dzięki temu zmiana fizycznego położenia rekordu nie wymusza aktualizacji indeksu. Indeksy mogą być zakładane zarówno na jedną jak i wiele kolumn. W przypadku indeksu wielokolumnowego, węzły sortowane są w pierwszej kolejności względem pierwszej kolumny indeksu. W następnej kolejności węzły z równymi wartościami pierwszej kolumny, sortowane są względem drugiej itd. Kolejność kolumn jest ustalana na podstawie kolejności podczas polecenia tworzenia indeksu.

Zastosowanie indeksu typu B-Tree

Aby przedstawić działanie indeksu typu B-Tree na rzeczyczywistym przykładzie przygotowałem dwie tabele. Pierwszą jest tabela *Comments* z bazy danych stackoverflow. Drugą tabelą jest *Init_Comments*, która jest kopią tabeli Comments i nie zawiera klucza głównego oraz indeksów. Dla tabeli *Comments_idx* za pomocą polecania

```
    CREATE INDEX user_post_idx
ON Comments(userId,last_name);
```

utworzyłem indeks typu B-Tree na dwóch kolumnach *first_UserId* oraz *last_PostId*.

Dopasowanie pełnego indeksu

Załóżmy, że w tabeli *Comments* chcemy wyszukać wszystkie komentarze użytkownika o id 1200 do postu o id 910331.

Najpierw wykonamy zapytania na tabeli nie zawierającej indeksów. *employees*.

```
SELECT * FROM Init_Comments where UserId = 1200 and PostId = 910331;
```

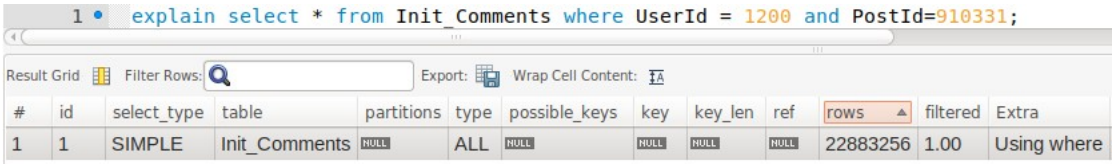
Zapytanie zwróciło wynik w 13,7 sekundy.

Następnie analogiczne zapytanie wykonałem na tabeli *Comments* zawierającej indeks na obu kolumnach. *employees_idx*.

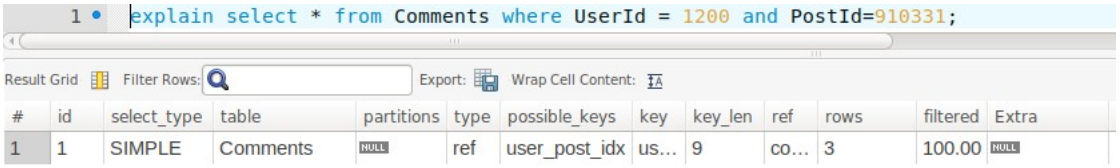
```
SELECT * FROM Comments where UserId = 1200 and PostId = 910331;
```

Tym razem zapytanie zwróciło wyniki w 0,013 sekundy. Tym razem serwer nie skanował całej tabeli. Z czego wynika różnica w czasie wykonania obu zapytań? Wykorzystując polecenie EXPLAIN dla obu zapytań otrzymujemy ciekawe dane. Rysunek 2 przestawia wynik polecenia EXPLAIN dla pierwszego zapytania, natomiast Rysunek 3 wynik polecania EXPLAIN dla drugiego zapytania. Polecenie EXPLAIN wyjaśnia, że pierwsze zapytania nie będzie korzystać z indeksów, dlatego w kolumnie rows widzimy, że serwer MySQL będzie musiał przeskanować wszystkie 23 miliony wierszy z tabeli *init_Comments*. Drguie zapytanie korzysta z indeksu z naszego indeksu. Tym razem serwer będzie musiał przeskanować jedynie 3 wiersze tabeli Comments.

Dopasowanie pełnego indeksu ma miejsce wtedy, kiedy w klauzuli *where* uwzględnimy wszystkie kolumny, na które założony jest indeks.



Rysunek 11: EXPLAIN 1



Rysunek 12: EXPLAIN 2

Dopasowanie prefiksu znajdującego się najbardziej na lewo Dopasowanie prefiksu znajdujcego się najbardziej na lewo może pomóc w wyszukaniu wsystkich komentarzy użytkownika. Załóżmy, że chcemy znaleźć wszystkie komentarze użytkownika o id 1200. W tym celu przygotowujemy dwa zapytania. Pierwsze na tabeli *Init_Comments*, drugie na tebeli *Comments* zawierającej indeks typu B-Tree, który założyliśmy wcześniej.

```
select * from Init_Comments where UserId = 1200;

select * from Comments where UserId = 1200;
```

Pierwsze zapytanie zostało wykonane w czasie 12,9 sekundy, natomiast drugie wymagało jedynie 0,0044 sekundy. Ponownie sprawdźmy rezultat polecenia EXPLAIN na obu zapytaniach. W pierwszym zapytaniu serwer po raz kolejny musiał przeszukać wszystkie wiersze w tabeli. Drugie zapytanie wymagało przeszukania 209 wierszy, dlatego że tym razem zapytanie było mniej selektywne niż przy dopasowaniu pełnego indeksu.

Dopasowanie zakresu wartości Dopasowanie zakresu wartości oznacza wyszukiwanie wartości w danym przedziale. W naszym przypadku może

```
1 • explain select * from Init_Comments where UserId = 1200;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Init_Comments	NULL	ALL	NULL	NULL	NULL	NULL	22883256	10.00	Using where

Rysunek 13: EXPLAIN 3

```
1 • explain select * from Comments where UserId = 1200;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Comments	NULL	ref	user_post_idx	user_post_idx	5	const	209	100.00	NULL

Rysunek 14: EXPLAIN 4

to być wyszukiwanie wszystkich komentarzy użytkowników o identyfikatorach z przedziału od 1990 do 2000.

Ponownie wykonujemy dwa zapytania. Pierwsze na tabeli bez indeksu, drugie na tabeli z indeksem.

```
select * from Init_Comments where UserId >1990 and UserId <2000;
```

Tym razem pierwsze zapytanie trwało 1.068 sekundy. Drugie zapytanie wykonujemy na tabeli Comments zawierającej indeksy.

```
select * from Comments where UserId >1990 and UserId <2000;
```

Następnie sprawdzamy wynik polecenia EXPLAIN dla obu zapytań. Przy pierwszym zapytaniu, kolejny raz MySQL przeskanował całą tabelę Init_Comments. Drugie natomiast wymagało przeskanowania jedynie wierszy, które zostały zwrócone jako rezultat zapytania.

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Init_Comments	NULL	ALL	NULL	NULL	NULL	NULL	22402608	11.11	Using where

Rysunek 15: EXPLAIN 5

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Comments	NULL	ra…	user_post_idx	user_post_idx	5	NULL	114	100.00	Using Inde…

Rysunek 16: EXPLAIN 6

PREFIX INDEX będą wymagały przeszukania całej tabeli. Dodatkowo wyszukiwanie za pomocą prefiksu nie będzie optymalne w przypadku indeksu wielokolumnowego dla wszystkich kolumn za wyjątkiem pierwszej. Jest to bezpośrednim następstwem budowy indeksów typu B-Tree i wynika z faktu sortowania kluczy względem pierwszej kolumny.

Zapytania dotyczące jedynie indeksów Zapytania dotyczące jedynie indeksów to zapytania, które wykorzystują jedynie wartości indeksu, a nie do rekordów bazy danych.

Indeks jednokolumnowy zawiera klucze posortowane zgodnie z wartościami kolumny, na której założony został indeks. W przypadku indeksów wielokolumnowych węzły sortowane są w kolejności kolumn w indeksie. Zakładając, że indeks został założony na kolumnach k1,k2 oraz k3, dane w pierwszej kolejności zostaną posortowane zgodnie z wartościami kolumny k1. Następnie rekordy z równą wartością kolumny k1 zostaną posortowane zgodnie z wartościami kolumny k2. Analogicznie rekordy z równą wartością kolumny k1 oraz k2 zostaną posortowane zgodnie z wartościami kolumny k3. Zrozumienie tej zasady jest kluczowe do poprawnego korzystania z indeksów typu B-Tree. Taka struktura powoduje, że taki indeks jest użyteczny tylko w przypadku, gdy wyszukiwanie używa znajdujego się najbardziej na lewo prefiksu indeksu. Kolejnym zastosowaniem indeksu typu B-Tree jest wyszukiwanie na podstawie prefiksu kolumny. Przykładem takiego zapytania może być wyszukiwanie wszystkich pracowników, których nazwiska rozpoczynają się od litery K (zakładamy, że tabela posiada indeks na kolumnie nazwisko). Istotnym jest fakt, że indeks staje się nieprzydany przy wyszukiwa-

niu na podstawie suffixu lub środkowej wartości. Następnym przypadkiem, w którym indeks typu B-Tree przyśpiesza zapytanie, jest wyszukiwanie na podstawie zakresu wartości. Dla tabeli z indeksem typu B-Tree założonym na kolumnie k, indeks może posłużyć do efektywnego wyszukania wartości z przedziału wartości tej kolumny. Zastosowanie struktury B-Tree powoduje, że sortowanie wyników zapytania względem indeksy jest zdecydowanie bardziej wydajne.

4.1.2 Indeksy tupu hash

Indeksy typu hash są dostępne jedynie dla tabel typu memory i są domyślnie ustawianymi indeksami dla takich tabel. Indeksy typu hash opierają się na funkcji skrótu liczonej na wartościach indeksowanych kolumn. Dla każdego rekordu takiej tabeli liczona jest krótka sygnatura, na podstawie wartości klucza wiersza. Podczas wyszukiwania wartości na podstawie kolumn indeksowanych tego typu kluczem obliczana jest funkcja skrótu dla klucza, a następnie wyszukuje w indeksie odpowiadających wierwszy. Możliwe jest, że do jednej wartości funkcji skrótu dopasowane zostanie więcej niż jeden różny wiersz. Takie zachowanie wynika bezpośrednio z zasady działania funkcji skrótu, która nie zapewnia unikatowości dla różnych wartości dla zbioru danych wejściowych. Niemniej taka sytuacja nie należy do częstych i nawet wtedy operacja wyszukiwania na podstawie indeksu typu hash jest bardzo wydajna, ponieważ serwer w najgorszym wypadku musi odczytać zaledwie kilka wierszy z tabeli. Podstawową wadą indeksu typu hash jest konieczność wyszukiwania na podstawie pełnej wartości klucza. Wynika to z tego, że funkcja skrótu wyliczona na podstawie niepełnego zbioru danych, nie ma korelacji z wartością funkcji wyliczonej na pełnym kluczu. Dodatkowo indeksy hash nie optymalizują operacji sortowania, ponieważ wartości funkcji f1, f2 skrótu dla dwóch rekordów x1 oraz x2, gdzie x1 jest mniejsze od x2 nie zapewniają, że f1 będzie mniejsze od f2.

5 Praktyczne problemy

W tej sekcji zostaną przedstawione przypadki zastosowania teoretycznej wiedzy wraz z praktycznymi przykładami.

5.1 Sortowanie wyników

Czasami zdarza się, że chcemy, aby wyniki zapytania były posortowane według pewnej kolejności. Jest to oczywiście pewien dodatkowy nakład, który serwer MySQL musi wykonać podczas wykonania zapytania. W tym podrozdziale pokażę, co zrobić, aby ta operacja nie wypłyneła drastycznie na wydajność naszego zapytania.

Podstawą optymalizacji sortowania jest używanie indeksów typu B-Tree, co wynika bezpośrednio z faktu, że indeks jest posortowany względem jego kolumn. Aby przedstawić działanie indeksów na realnych przykładach przygotowałem do tego bazę StackOverflow. Z bazy usunąłem wszystkie indeksy oraz klucze główne założone na wykorzystywanych w przykładach tabelach, aby nie wpływały one na prezentowane przykłady.

MySQL może użyć indeksu do sortowania wyników w następujących przypadkach.

Najlepszym z możliwych scenariuszy wykorzystania indeksu do sortowania danych jest przypadek, kiedy kolumny użyte do sortowania odpowiadają indeksowi, a kolumny, które chcemy zwrócić jako wynik zapytania są podzbiorem kolumn indeksu. Weźmy tabelę Users, na którą założymy indeks typu BTREE jak poniżej.

```
CREATE INDEX Rank_idx ON Users(Reputation, UpVotes);
```

Teraz wykonajmy zapytanie:

```
EXPLAIN SELECT Reputation,UpVotes FROM Users ORDER BY Reputation,
UpVotes;
```

W takim przypadku poleceni EXPLAIN zwróci w kolumnie EXTRA informację: "Using filesort", co oznacza, że do sortowania wartości użyty został indeks znajdujący się w kolumnie key, czyli indeks, który przed chwilą stworzyliśmy.

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Users	NULL	index	NULL	Rank_Idx	8	NULL	2475228	100.00	Using index

Jeżeli w wyniki chcemy otrzymać jedynie kolumnę *Reputation*, to MySQL wciąż będzie wykorzystywał indeks do sortowania wyników, ponieważ spełnia to warunek zawierania się kolumn rezulatu zapytania w zbiorze kolumn indeksu. Sprawdźmy teraz, co się stanie jeżeli do klauzuli WHERE dodamy kolejną kolumnę:

```
ALTER TABLE Users ADD PRIMARY KEY (Id);
EXPLAIN SELECT Id, Reputation, UpVotes FROM Users ORDER BY Reputation,
UpVotes;
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Users	NULL	ALL	NULL	NULL	NULL	NULL	2466330	100.00	Using filesort

Widzimy, że tym razem MySQL nie wykorzystał indeksu, ale pobrał wszystkie dane i posortował wykorzystując jeden z dostępnych w MySQL algorytmów sortowania. Co ciekawe, nie zawsze musi się tak stać. MySQL na etapie analizy wykonania sprawdza, czy wydajniejsze będzie dla niego sortowanie wyników na podstawie pobranych danych, czy może, jeżeli sortujemy dane względem jednego z indeksów na tabeli, pobrać ten indeks i wykorzystać do wydajniejszego sortowania. Dodajmy teraz klucz główny dla tabeli Users i sprawdźmy, co się stanie jeżeli umieścimy go jako jedną z kolumn wyniku naszego zapytania.

```
ALTER TABLE Users ADD PRIMARY KEY (Id);
EXPLAIN SELECT Id, Reputation, UpVotes FROM Users ORDER BY Reputation,
UpVotes;
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Users	NULL	Index	NULL	Rank_Idx	8	NULL	2466330	100.00	Using index

Wynik polecenia EXPLAIN jest interesujący. Przypomnijmy sobie zatem, w jaki sposób MySQL przechowuje dane w indeksie. Każdy wpis w indeksie musi wskazywać na konkretny wiersz naszej tabeli. W niektórych systemach baz danych indeks wskazuje na wiersz poprzez jego miejsce na dysku. W MySQL jednak, indeks wskazuje na wiersz poprzez klucz główny tabeli. Właśnie z faktu, że klucz główny jest integralną częścią indeksu, skorzystał w tym przypadku optymalizator MySQL.

Kolejnym często używanym zapytaniem jest pobranie wszystkich kolumn z tabeli, ale posortanie ich według określonych kolumn. Weźmy następujące zapytanie:

```
EXPLAIN SELECT u.* FROM Users u ORDER BY u.UpVotes, u.Reputation;
```

Tym razem MySQL znów najprawdopodobniej nie użyje indeksu do posortowania danych. Oczywiście nadal może zdecydować, że efektywniejszym będzie dodatkowe pobranie indeksu i wykorzystanie go do sortowania danych.

Przeanalizujmy teraz następne zapytanie.

```
EXPLAIN SELECT * FROM Users WHERE Reputation = 1 ORDER BY UpVotes;
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Users	NULL	ref	Rank_Idx	Rank_Idx	4	c...	1233165	100.00	NULL

Tym razem MySQL znów wykorzystał indeks, do posortowania wyników. W jaki sposób to zrobił? Skorzystał z faktu, że indeksie są posortowane względem kolumn Reputation, a w przypadku, kiedy wartość Reputation jest równa, względem kolumny UpVote, co odpowiada wartości ORDER BY. Sprawdźmy co się stanie, jeżeli delikatnie zmodyfikujemy zapytanie do postaci:

```
EXPLAIN SELECT * FROM Users WHERE Reputation > 1000 ORDER BY UpVotes;
```

W tym przypadku nie ma jednoznaczej odpowiedzi na pytanie, w jaki sposób MySQL posortuje dane. Optymalizator MySQL musi podjąć decyzję, czy warunki w klauzuli WHERE są wystarczająco selektywne, czy może pobranie indeksu i na jego podstawie przeprowadzenie sortowania będzie efektywniejsze.