

# Praca magisterska

Franciszek Słupski

Brak

## Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
1.1	Testowa baza danych . . . . .	2
<b>2</b>	<b>Porównywanie zapytań</b>	<b>3</b>
<b>3</b>	<b>Architektura MySQL</b>	<b>14</b>
3.1	Obsługa połączeń i wątków . . . . .	14
<b>4</b>	<b>Bufor zapytań</b>	<b>15</b>
<b>5</b>	<b>Optymalizator MySQL</b>	<b>16</b>
5.1	Dane statystyczne dla optymalizatora . . . . .	18
5.2	Plan wykonania zapytania . . . . .	18
5.3	Optymalizator złączeń . . . . .	19
5.4	Konfiguracja optymalizatora złączeń . . . . .	21
5.5	Konfiguracja statystyk tabel InnoDB dla optymalizatora . . .	25
5.5.1	Trwałe statystyki . . . . .	25
<b>6</b>	<b>Skalowalność i wysoka dostępność</b>	<b>27</b>
6.1	Terminologia . . . . .	28

6.1.1	Skalowalność a wydajność . . . . .	28
6.1.2	Teoria CAP . . . . .	28
6.1.3	Rodzaje skalowalności . . . . .	29
6.2	Skalowanie wertykalne . . . . .	29
6.3	Skalowanie horyzontalne . . . . .	30
6.3.1	Mechanizm replikacji . . . . .	30
6.4	Partycjonowanie funkcjonalne . . . . .	32
6.5	MySQL Cluster . . . . .	32
<b>7</b>	<b>Indeksy w MySQL</b>	<b>33</b>
7.1	Indeksy typu B-Tree . . . . .	33
7.2	Indeksy typu hash . . . . .	39
<b>8</b>	<b>Praktyczne problemy</b>	<b>40</b>
8.1	Sortowanie wyników . . . . .	40

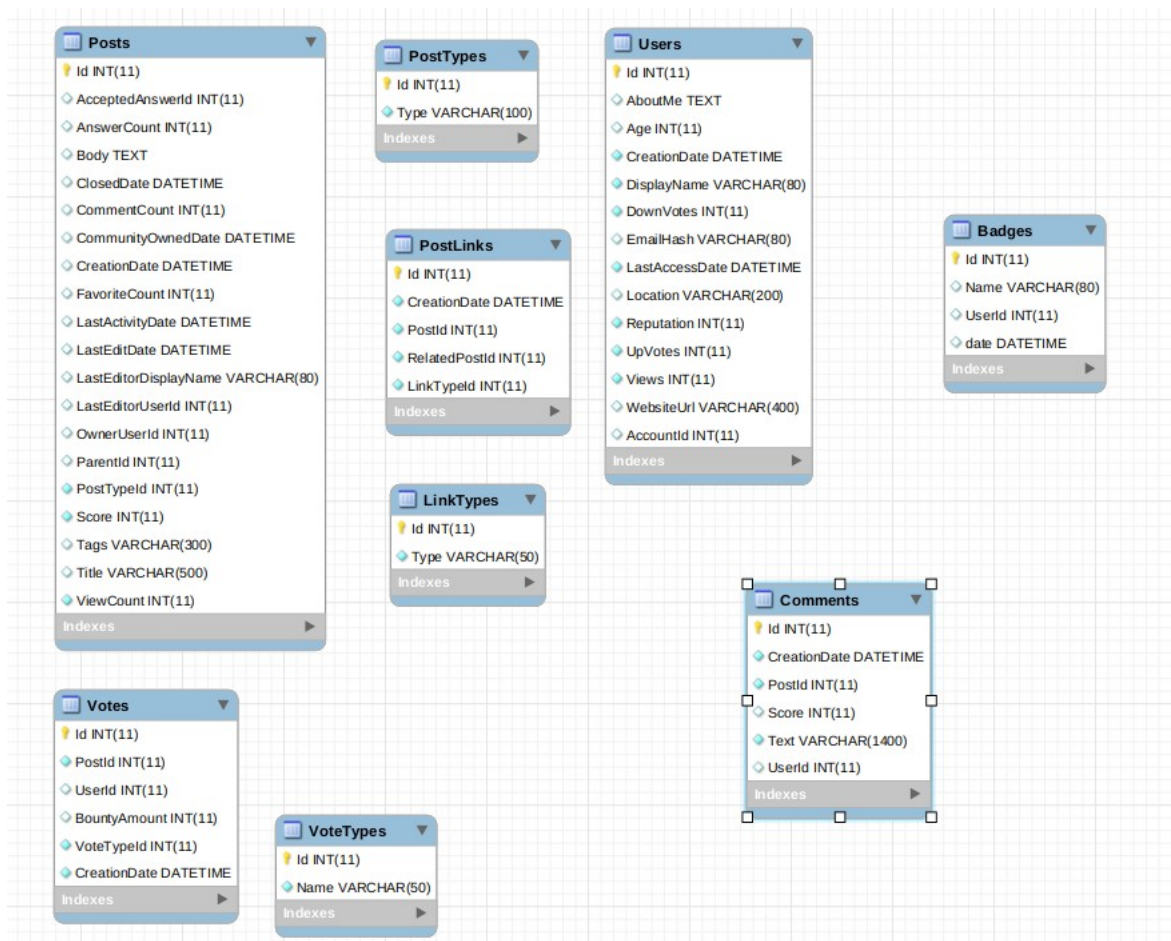
# 1 Wstęp

## 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.

## 2 Porównywanie zapytań

W rozdziale zostanie przedstawione działanie polecenia EXPLAIN, które pozwala uzyskać informacje o planie wykonania zapytania. Jest podstawowym



Rysunek 1: Schemat bazy danych stackoverflow

sposobem określenia, w jaki sposób MySQL wykonuje zapytania. Analiza wyników EXPLAIN jest zdecydowanie bardziej miarodajna od mierzenia czasów zapytań. Na czas wykonania zapytania mogą mieć wpływ zewnętrzne czynniki, 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 rezultat zapytania zostanie zwrócony błyskawicznie z bufora zapytań. Doprowadzi to do sytuacji, kiedy nawet najbardziej niewydajne zapytania będą zwracane w ułamkach sekundy. 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, przykładowo indeksy lub nawet często pobierane dane. Jeżeli wykonujemy zapytanie dla tabeli, której indeks nie znajduje się w pamięci. Serwer pobiera indeks z dysku, co trwa. Następnie poprawiamy zapytanie w celu zwiększenia wydajności i wykonujemy, aby sprawdzić, czy nasze działanie wpłynęło na wydajność. Tym razem cały indeks znajduje się w pamięci i zapytanie wykonuje się wielokrotnie szybciej. Mierzając jedynie czasy wykonania obu zapytań, możemy dojść do fałszywego wniosku, że drugie zapytanie jest wydajniejsze, nawet jeżeli w rzeczywistości nasze działanie doprowadziło do pogorszenia wydajności. 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. Przykładowo jeżeli testujemy zapytanie na tabeli, na której równocześnie wykonywane są operacje zapisu, czasy wykonania naszego zapytania nie będą miarodajne. Na czasy wykonywania zapytań wpływać może również aktualne obciążenie serwera, co nie ułatwia pracy przy porównywaniu wyników. Jak widzimy aby skutecznie porówny-

wać efektywność zapytań nie powinniśmy opierać się jedynie na czasie ich wykonania.

**Polecenie EXPLAIN** Polecenie EXPLAIN będzie jedną z głównych metod porównywania wydajności zapytań stosowaną w tej pracy, dlatego w tym podrozdziale przedstawie podstawy stosowania tego polecenia. Funkcja EXPLAIN to główny sposób określania, jaki sposób wykonania zapytania wybrany na etapie jego optymalizacji przez optymalizator MySQL.

Aby użyć polecenia EXPLAIN, wystarczy poprzedzić słowa kluczowe takie jak SELECT, INSERT, UPDATE, DELETE poleceniem EXPLAIN. Spowoduje to, że zamiast wykonania zapytania, serwer zwróci informacje o planie wykonania. Rezultat polecenia EXPLAIN zawiera po jednym rekordzie dla każdej tabeli użytej w zapytaniu. Kolejność wierszy w wyniku zapytania odpowiada kolejności, w jakiej MySQL będzie wykonywał zapytania. Pierwszym zapytaniem wykonanym przez MySQL będzie zapytanie z ostatniego wiersza.

**Wyniki polecenia EXPLAIN** Żeby zademonstrować 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	ALL	NULL	NULL	NULL	NULL	22155797	100.00	Using where
2	2	DEPENDENT SUBQUERY	Posts	NULL	Index	NULL	title_idx	1503	NULL	16575372	100.00	Using index; Using temporary

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	<small>NULL</small>	ALL	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	17189835	100.00	Using where
2	2	DEPENDENT SUBQUERY	Users	<small>NULL</small>	eq_ref	PRIMARY	PRIMARY	4	func	1	33.33	Using where
3	3	DEPENDENT UNION	Comments	<small>NULL</small>	ref	user_post_idx	user_post_idx	5	func	23	33.33	Using where
4	<small>NULL</small>	UNION RESULT	<union2,3>	<small>NULL</small>	ALL	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	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	<small>NULL</small>	ALL	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	22155797	100.00	Using where
2	2	UNCACHEABLE SUBQUERY	Users	<small>NULL</small>	ALL	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	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	<small>NULL</small>	ALL	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	22155797	100.00	<small>NULL</small>

Rysunek 9: Przykład 8

```
SELECT * FROM Users WHERE Id BETWEEN 1 AND 100 AND
id NOT IN (SELECT OwnerUserId FROM Posts WHERE Score >100);
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	PRIMARY	Users	<small>NULL</small>	range	PRIMARY	PRIMARY	4	<small>NULL</small>	79	100.00	Using where
2	2	DEPENDENT SUBQUERY	Posts	<small>NULL</small>	index_subquery	owner_idx	owner_idx	5	func	22	33.33	Using where

Rysunek 10: Przykład 9

```
SELECT * FROM Comments WHERE UserId = 20500 OR id = 20500;
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Comments	<small>NULL</small>	index_merge	PRIMARY,user_post_idx	user_post_idx...	5,4	<small>NULL</small>	2	100.00	Using sort_union(user_post_idx,PRIMARY); Using

Rysunek 11: Przykład 10

**Kolumna #** Wartości w kolumnie # określają kolejność, w jakiej MySQL będzie odczytywał tebele. Jako pierwsza odczytywana jest tabela z najmniejszą wartością.

### 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 wyniku klauzuli UNION, to zostanie oznaczone odpowiednio jako **DEPENDENT SUBQUERY** (przykład 5) lub **DEPENDENT UNION** (Przykład 6). Dodat-



kowo 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 jednak nadal konieczność odczytu całej tabeli. Co więcej, dane z dysku pobierane są z adresów, których kolejność wynika z użytego indeksu. Adresy te nie muszą zajmować na dysku ciągłych obszarów, a to oznacza, że czas odczytu danych może znacznie wydłużyć się. Jeżeli w kolumnie *extra* jest dodatkowo zawarta informacja "Using Index" oznacza to, że MySQL wykorzystuje indeks pokrywający (opisany w dalszej części pracy) i nie wymaga odczytywania innych danych z dysku – do wykonania zapytania wystarczają dane umieszczone w indeksie.

### **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**

Tego typu zapytanie zostało przedstawione w przykładzie 9, w którym podzapytanie korzysta z nieunikalnego indeksu, jest wykonane przed głównym zapytaniem i jego wartości są przekazane do niego jako stałe.

### **unique\_subquery**

Analogicznie do *index\_subquery*, ale tym razem z użyciem klucza głównego lub indeksu *UNIQUE NOT NULL*.

### **index\_merge**

Czasami jeden indeks nie wystarczy do efektywnego wykonania zapytania. Rozważmy przykład 10. Na tabeli *Comments* mamy założone dwa różne

indeksy, obejmujące obie kolumny występujące w zapytaniu. Użycie tylko jednego indeksu nie poprawiłoby efektywności zapytania, ponieważ nadal serwer MySQL musiałby przeprowadzić pełny skan tabeli. Dlatego od wersji 5.0 optymalizator może zdecydować się na złączenie kilku indeksów, dla efektywniejszego wykonania zapytania. Decyzja o tym, czy łączyć indeksy często zapada na podstawie rozmiaru tabeli. Przy tabelach niewielkich rozmiarów operacja złączenia może być kosztowniejsza niż pełny skan tabeli, ale przy dużych tabelach, przykładowo takich jak *Comments* złączenie znacząco przyspiesza wykonanie zapytania.

### **fulltext**

Wartość *fulltext* oznacza, że wykorzystane zostało wyszukiwanie pełnotekstowe, opisane w dalszej części pracy.

### **ref**

Jest to wyszukiwanie, w którym MySQL musi przeszukać jedynie indeks w celu znalezienia rekordu opowiadającego pojedynczej 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 dostęp *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 z tabeli odczytywany jest tylko jeden wiersz dla każdej kombinacji wierszy z poprzednich tabel. Z tego rodzaju złączeniem mamy do czynienia, jeżeli wszystkie kolumny

używane do złączenia są kluczem głównym lub indeksem "NOT 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 wystarczy jednokrotne przeszukanie indeksu, a na znalezionym liściu indeksu dostępne są już wszystkie dane z wiersza tabeli. 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 skanowania całej tabeli lub 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**

Kolumna 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 do zapisu wiersza w tabeli.

### 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 może też przyjąć wartość *func*, co oznacza, że wartość użyta do wyszukania jest wynikiem obliczenia pewnej funkcji (przykład 9).

### Kolumna `rows`

Kolumna wskazuje oszacowaną liczbę wierszy, które MySQL będzie musiał odczytać w celu znalezienia szukanych rekordów. Wartość może znacząco odbiegać od rzeczywistej liczby wierszy, które zostaną odczytane podczas wykonania zapytania. Istotne jest to, że jest to liczba przeszukiwanych rekordów na danym poziomie zagnieżdżenia 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 15748463.

```
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 12: 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

. 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. Przed wersją 8.0, aby kolumna filtered była umieszczona w wynikach zapytania należało wykorzystać polecenie EXPLAIN EXTENDED.

### 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 dopiero po wczytaniu danych z tabeli. Często jest to informacja, która może sugerować zmianę lub stworzenia nowego indeksu bądź całego zapytania.
- 'Using temporary' - do sortowania wyników używana jest tabela tymczasowa.

- 'Using filesort' - sortowanie nie może skorzystać z istniejących indeksów (nie ma odpowiedniego optymalnego indeksu), więc wiersze są sortowane za pomocą jednego z algorytmów sortowania.

## 3 Architektura MySQL

### 3.1 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 zdefiniowana jest poprzez zmienną systemową max\_connections, który domyślnie przyjmuje wartość 151. Dodatkowo 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.

## 4 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 niedeterministyczne. Przykładowo wszystkie polecenia pobierające aktualną datę, użytkownika itp nie zostaną dodane do bufora zapytań. Co istotne nawet w przypadku zapytania niedeterministycznego, 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.



## 5 Optymalizator MySQL

Zasadniczo każde zapytanie SQL skierowane do bazy danych MySQL może zostać zrealizowane na wiele różnych sposobów. Optymalizator jest fragmentem oprogramowania serwera MySQL, który odpowiada za wybranie najefektywniejszego sposobu wykonania zapytania (plan wykonania zapytania). Proces ten ma kilka etapów. W pierwszej kolejności analizator MySQL dzieli zapytanie na tokeny i z nich tworzy "drzewo analizy". Na tym etapie przeprowadzana jest jednocześnie analiza składni zapytania. Następnym krokiem jest *preprocessing*, w którego trakcie sprawdzane są między innymi: nazwy kolumn i tabel, a także nazwy i aliasy, aby zapewnić, że nazwy użyte w zapytaniu nie są np. dwuznaczne. Na kolejnym etapie weryfikowane są uprawnienia. Czynność ta może zajmować szczególnie dużo czasu, jeżeli serwer posiada ogromną liczbę uprawnień. Po zakończeniu etapu *preprocessingu* drzewo analizy jest poprawne i gotowe do tego, aby optymalizator przekształcił je do postaci planu wykonania.

W MySQL stosowany jest optymalizator kosztowy, co oznacza, że optymalizator szacuje koszt wykonania dla wariantów planu wykonania i wybiera ten z najmniejszym kosztem. Jednostką kosztu jest odczytanie pojedynczej, losowo wybranej strony danych o wielkości czterech kilobajtów. Wartość kosztu jest wyliczana na podstawie danych statystycznych, dlatego optymalizator wcale nie musi wybrać optymalnego planu. Istnieją dwa rodzaje optymalizacji: *statyczna* i *dynamiczna*. Optymalizacja *statyczna* przeprowadzana jest tylko raz i jest niezależna od wartości używanych w zapytaniu. To oznacza, że przeprowadzona raz będzie aktualna, nawet jeżeli zapytanie będzie wykonywane z różnymi wartościami. Natomiast optymalizacja *dynamiczna* bazuje na kontekście, w którym wykonywane jest zapytanie i jest przeprowadzana za każdym razem, kiedy polecenie jest wykonywane. Optymalizacja *dynamiczna* opiera się na wielu parametrach, takich jak: wartości w klauzuli WHERE czy liczba wierszy w indeksie, które znane są dopiero w momencie

wykonania konkretnego zapytania.

Poniżej przedstawione zostało kilka przykładowych typów optymalizacji, które może wykonać moduł optymalizatora.

- **Zmiana kolejności złączeń.** Podczas wykonywania zapytania tabele nie zawsze są łączone w takiej kolejności jak w zapytaniu. Zagadnienie jest dokładniej opisane w podrozdziale dotyczącym optymalizatora złączeń.
- **Zamiana OUTER JOIN na INNER JOIN.** OUTER JOIN nie zawsze musi być wykonywany jako OUTER JOIN. Niektóre czynniki takie jak warunki w klauzuli WHERE czy schemat bazy danych mogą spowodować, że OUTER JOIN będzie równoznaczne złączeniu INNER JOIN.
- **Przekształcenia algebraiczne.** Optymalizator przeprowadza transformacje algebraiczne takie jak: redukcja stałych, eliminowanie nieosiągalnych warunków czy stałych. Przykładowo warunek  $(2=2 \text{ AND } a>2)$  może zostać przekształcony do postaci  $(a>2)$ . Podobnie warunek  $(a<b \text{ AND } b=c \text{ AND } a=5)$  może być przekształcony do  $(b>5 \text{ AND } b=c \text{ AND } a=5)$ .
- **Optymalizacja funkcji MIN(), MAX().** Serwer już na etapie optymalizacji zapytania może uznać wartości zwracane przez funkcje jako stałe dla reszty zapytania. W niektórych przypadkach optymalizator może nawet pominąć tabelę w planie wykonania zapytania, jeżeli jedyną wartością pobieraną z tabeli jest wynik funkcji MIN() lub MAX(). W takim przypadku w danych wyjściowych polecenia EXPLAIN znajdzie się ciąg tekstowy "Select tables optimized away". Na poniższym przykładzie widzimy, że kolumna *ref* dla pierwszego wiersza jest wartością "const", czyli najmniejsza wartość id z tabeli Users została potraktowana jako stała.

```
EXPLAIN SELECT * FROM Comments WHERE UserId = (SELECT MIN(id)
FROM Users);
```

#	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	57	100.00	Using where
2	2	SUBQUERY	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	Select tables optimized away

- **Optymalizacja funkcji COUNT().** Wynik funkcji COUNT(\*) bez klauzuli WHERE w niektórych silnikach (np. MyISM), również mogą zostać potraktowane jako stała, ale nie dotyczy to najpopularniejszego obecnie w MySQL silnika InnoDB. **Optymalizacja stałej tabeli**
- **Stale tabele.** *Stala tabela* jest to tabela, która zawiera co najwyżej jeden wiersz lub warunek zawarty w klauzuli WHERE odnosi się do wszystkich kolumn klucza głównego, albo do indeksu UNIQUE NOT NULL. W takim przypadku MySQL może zwrócić wartość jeszcze przed wykonaniem zapytania i potraktować jako stałą dla dalszej części zapytania.

## 5.1 Dane statystyczne dla optymalizatora

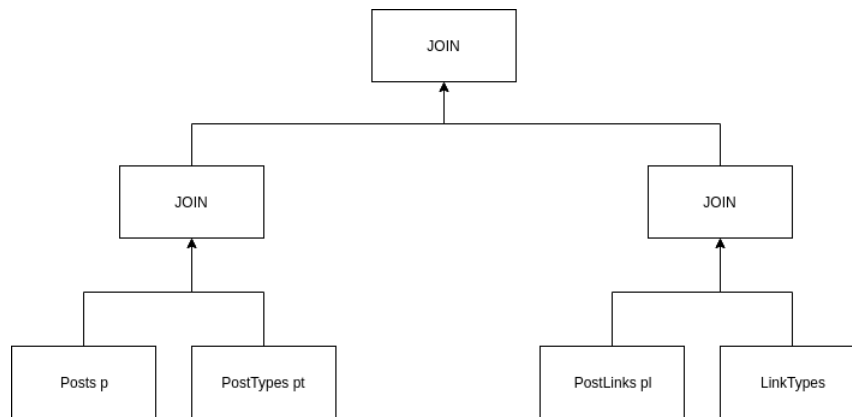
Przechowywaniem danych statystycznych jest zadaniem silników bazy danych. Z tego powodu w zależności od użytego silnika przechowywane wartości statystyczne mogą być różne. Przykładowo silnik MyISM przechowuje informację o aktualnej liczbie rekordów w tabeli, a InnoDB takiej informacji nie przechowuje, natomiast niektóre silniki, np. Archive, wcale nie przechowują danych statystycznych.

## 5.2 Plan wykonania zapytania

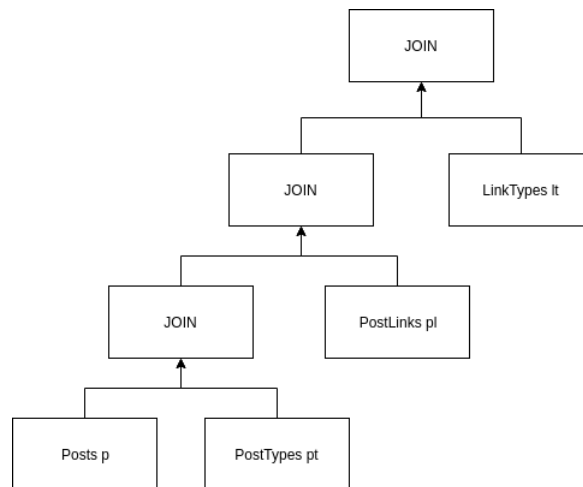
Wynikiem optymalizacji jest plan wykonania zapytania. Plan wykonania jest zapisywany w postaci drzewa instrukcji, które kolejno wykonywane doprowadzą do zwrócenia wyniku zapytania.

```
SELECT * FROM Posts p LEFT JOIN PostTypes pt ON p.PostTypeId = pt.Id
LEFT JOIN PostLinks pl ON p.Id = pl.PostId LEFT JOIN LinkTypes lt
on pl.LinkTypeId = lt.id WHERE PostID = 9;
```

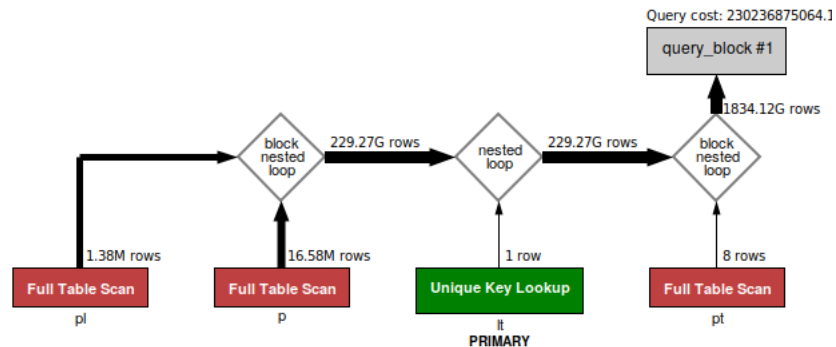
Gdybyśmy mieli wyobrazić sobie sposób łączenia tabel w MySQL, zapewne przedstawilibyśmy to tak, jak na poniższym schemacie.



W praktyce drzewo instrukcji przybiera postać *drzewa lewostronnie zagnieżdżonego*, co pokazano na rysunku poniżej.



Wywołując polecenie EXPLAIN dla naszego zapytania, używając klienta MySQL Workbench możemy wyświetlić graficzną postać odpowiadającą drzewu instrukcji.



Widzimy, że drzewo otrzymane jako wynik polecenia EXPLAIN jest drzewem lewostronnie zagnieżdżonym. Widzimy też, że optymalizator zdecydował się zamienić kolejność złączeń, aby zminimalizować koszt wykonania zapytania.

### 5.3 Optymalizator złączeń

Większość operacji złączeń można wykonać na wiele różnych sposobów, uzyskując ten sam wynik. Zamiana kolejności jest bardzo skuteczną formą optymalizacji zapytań. Rozważmy teraz następujące przykładowe zapytanie:

```
SELECT u.Id, p.Id, c.Id, pt.'Type' FROM Users u INNER JOIN Posts
p ON u.Id = p.OwnerUserId INNER JOIN Comments c ON c.UserId = u.Id
INNER JOIN PostTypes pt ON pt.Id = p.PostTypeId WHERE u.Id = 4;
```

Wykonując polecenie z prefiksem EXPLAIN otrzymujemy następujący wynik:

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	u	<a href="#">NULL</a>	const	PRIMARY	PRIMARY	4	const	1	100.00	Using index
2	1	SIMPLE	pt	<a href="#">NULL</a>	ALL	<a href="#">NULL</a>	<a href="#">NULL</a>	<a href="#">NULL</a>	<a href="#">NULL</a>	8	100.00	<a href="#">NULL</a>
3	1	SIMPLE	p	<a href="#">NULL</a>	ref	owner_idx	owner_idx	5	const	182	10.00	Using where
4	1	SIMPLE	c	<a href="#">NULL</a>	ref	user_post_idx	user_post_idx	5	const	322	100.00	Using index

Następnie modyfikujemy zapytanie dodając słowo kluczowe STRAIGHT\_JOIN, aby wymusić kolejność złączeń taką jak w zapytaniu.

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	u	HULL	const	PRIMARY	PRIMARY	4	const	1	100.00	Using index
2	1	SIMPLE	p	HULL	ref	owner_idx	owner_idx	5	const	182	100.00	HULL
3	1	SIMPLE	c	HULL	ref	user_post_idx	user_post_idx	5	const	322	100.00	Using index
4	1	SIMPLE	pt	HULL	ALL	HULL	HULL	HULL	HULL	8	12.50	Using where

Widzimy, że oba rezultaty polecenia są niemalże identyczne, jedyną różnicą jest kolejność dokonywanych złączeń. Sprawdźmy teraz, jaki jest koszt wykonania obu zapytań. Koszt pierwszego zapytania, którego kolejność została zamieniona na etapie optymalizacji, wynosi 6510. Koszt drugiego zapytania wynosi 66868! Zamiana kolejności złączeń zmniejszyła koszt dziesięciokrotnie. W kolejnym kroku włączyłem profilowanie zapytań za pomocą komendy:

```
SET PROFILING = 1;
```

, wykonałem 10 zapytań dla każdego wariantu i policzyłem średni czas, jaki serwer MySQL spędzał na etapie "executing", czyli etapie faktycznego wykonywania zapytania. Dla zapytania z kolejnością wybraną przez optymalizator otrzymałem wartość 0.04 sekundy, natomiast przy kolejności wybranej przez nas w zapytaniu wartość ta wynosiła już 0.28 sekundy. Powyższy eksperyment pokazuje, że zamiana kolejności złączeń jest niezwykle skuteczną formą optymalizacji zapytań i może prowadzić do wielokrotnego zmniejszenia kosztu zapytania. Oczywiście nadal może się zdarzyć sytuacja, kiedy zapytanie z wymuszoną kolejnością załączeń będzie wydajniejsze, ponieważ optymalizator MySQL nie zawsze może sprawdzić wszystkie potencjalne kombinacje złączeń, ale w zdecydowanej większości przypadków optymalizator złączeń wygrywa z człowiekiem.

## 5.4 Konfiguracja optymalizatora złączeń

Optymalizator złączeń stara się wygenerować plan zapytań o najniższym możliwym koszcie. W idealnym przypadku optymalizator może zweryfikować wszystkie potencjalne kombinacje złączeń. Niestety operacja łączenia

dla  $n$  tabel będzie miała  $n!$  możliwych kombinacji. Oznacza to, że dla dziesięciu tabel złączenia mogą zostać przeprowadzone na 3628800 różnych sposobów i gdyby optymalizator zdecydował się przetestować każdy dostępny scenariusz, kompilacja mogłaby zająć wiele godzin, a nawet dni. Do zdefiniowania, jak wiele planów powinien przetestować optymalizator służy opcja *optimizer\_search\_depth*. Na ogół im niższa wartość zmiennej, tym szybciej optymalizator zwróci plan wykonania, ale zmniejsza się też prawdopodobieństwo optymalności planu. Wartość 0 oznacza, że MySQL dla każdego zapytania dobierze odpowiednią (zdaniem optymalizatora) przestrzeń przeszukiwania.

Aby pokazać wpływ parametru *optimizer\_search\_depth* przygotowałem następujący kod sql, który tworzy dwie tabele, a następnie wypełnia je losowymi danymi.

```
CREATE TABLE 'lecturers'
(
  'id' INT(11) NOT NULL AUTO_INCREMENT,
  PRIMARY KEY ('id')
);
CREATE TABLE 'students'
(
  'id' INT(11) NOT NULL AUTO_INCREMENT,
  'lecturer_id' INT(11) NOT NULL,
  'value' SMALLINT(6) NOT NULL,
  PRIMARY KEY ('id'),
  INDEX 'lecturer_id' ('lecturer_id'),
  INDEX 'value' ('value')
);
INSERT INTO 'lecturers' VALUES (1), (2);

delimiter ;;
```

```

CREATE PROCEDURE fill_tables()
BEGIN
DECLARE i int DEFAULT 0;
WHILE i <= 1000 DO
INSERT INTO 'students' ('id', 'lecturer_id', 'value') VALUES (0,
1, i);
INSERT INTO 'students' ('id', 'lecturer_id', 'value') VALUES (0,
2, i);
SET i = i + 1;
END WHILE;
END;;
delimiter ;

```

```
CALL fill_tables();
```

W kolejnym kroku wykonałem wielokrotnie następujące zapytanie:

```

SELECT COUNT(*) FROM table_parent AS p WHERE 1
AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 1 LIMIT 1)
AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 2 LIMIT 1)
AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 3 LIMIT 1)
AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 4 LIMIT 1)
AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 5 LIMIT 1)
AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 6 LIMIT 1)
AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 7 LIMIT 1)

```



```

AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 8 LIMIT 1)
AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 9 LIMIT 1)
AND EXISTS (SELECT 1 FROM students AS s WHERE s.lecturer_id = p.id
AND s.value = 10 LIMIT 1)

```

zmieniając parametr *optimizer\_search\_depth*, dla każdej wartości zmiennej zapisałem średni czas wykonania polecenia EXPLAIN. Wyniki umieściłem w poniższej tabeli.

optimizer_search_depth	czas [sekund]
0	1.8
1	0.0011
2	0.0019
3	0.0059
5	0.6
10	15
20	22
30	27
62	36

Widzimy, że wraz ze wzrostem wartości parametru wzrastał czas wykonywania zapytania. Kolejnym krokiem było włączenie profilowania i sprawdzenie, na który z etapów serwer spędził najwięcej czasu. Wyniki zostały posortowane według czasu i na poniższym zrzucie ekranu widzimy profil zapytania dla wartości 62.

#	Status	Duration
1	statistics	36.301148
2	executing	0.000234

Profilowanie zapytania wskazuje nam, że przez większość czasu zapytania, serwer starał się zebrać dane, które pozwolą mu wybrać optymalny plan wykonania zapytania. Obserwacja pokazuje, że dla pewnych zapytań, próba

wybrania optymalny rozwiązania może skończyć się gigantycznym wydłużeniem czasu zapytania. Co ciekawe nawet dla wartości 0 optymalizacja okazał się nieefektywna, co prowadzi do wniosku, że czasami jedynym rozwiązaniem w przypadku, kiedy serwer zbyt dużą ilość czasu spędza na szukaniu optymalnego planu, jest ręczna zmiana wartości parametru `optimizer_search_depth`.

Drugim atrybutem, który służy do konfiguracji optymalizatora złączeń jest `optimizer_prune_level`. Parametr decyduje o tym, czy optymalizator może wykorzystać heurystyki do wybrania optymalnego planu. Jeżeli ta opcja zostanie włączona, optymalizator może pominąć niektóre plany, bazując na pewnych heurystykach. Dokumentacja MySQL wskazuje, że relatywnie rzadko dochodzi do sytuacji, kiedy optymalizator pomija optymalny plan wykonania, a włączenie tej opcji może znacznie przyspieszyć proces optymalizacji. Dlatego też domyślną wartością jest 1, co oznacza, że serwer może bazować na heurystykach. Aby pokazać wpływ parametru na wydajność zapytania, wykorzystałem jeszcze raz zapytanie do wygenerowanej tabeli *students*. Najpierw ustawiłem wartość parametru `optimizer_search_depth=0`, a następnie `optimizer_prune_level=0`. Z poprzedniego eksperymentu wiemy, że to zapytanie powinno wykonać się w czasie mniej więcej 1.8 sekundy, ale tym razem wymagało średnio ok. 3.8 sekundy, czyli ponad 2 razy więcej czasu. Następnie ustawiłem wartość `optimizer_search_depth=62` i ponownie zmierzyłem średni czas wykonania zapytania. Tym razem zapytanie trwało średnio 121 sekund, co oznacza prawie czterokrotny spadek wydajności w stosunku do `optimizer_prune_level=1`. Dlatego też zalecane jest pozostawienie domyślnej wartości `optimizer_prune`, chyba że mamy pewności pominięcia optymalnego planu zapytania.

## 5.5 Konfiguracja statystyk tabel InnoDB dla optymalizatora

Dla tabeli InnoDB zbierane są dwa rodzaje statystyk: trwałe i nietrwałe. , natomiast nietrwałe są czyszczone po każdym restarcie oraz po niektórych operacjach.

### 5.5.1 Trwałe statystyki

Trwałe statystyki są przechowywane niezależnie od restartów serwera MySQL, dlatego zapisywane są na dysku twardym. Od wersji MySQL 5.6.6 trwałe statystyki są domyślnie włączone dla wszystkich tabel InnoDB, ale można je wyłączyć dla wszystkich tabel poprzez ustawienie parametru *innodb\_stats\_persistent* = OFF, lub poprzez ustawienie *STAST\_PERSISTENT* = 0 dla wybranej tabeli.

**Konfiguracja automatycznego przeliczania statystyk** Standardowo przeliczenie trwałych statystyk dla tabeli ma miejsce, jeżeli zmodyfikowane zostanie więcej niż 10 % wierszy w tabeli. Jeżeli chcemy wyłączyć automatyczne kalkulowanie, możemy ustawić parametr *innodb\_stats\_auto\_recalc* = false. Należy pamiętać o tym, że przeliczanie statystyk odbywa się asynchronicznie, to znaczy serwer nie wykonuje kalkulacji natychmiastowo po zmodyfikowaniu 10 % wierszy, ale próbuje znaleźć optymalny czas. Jeżeli chcemy wymusić przeliczenie możemy wywołać polecenie ANALYZE TABLE. Jeżeli dodajemy indeks do tabeli lub kolumna jest usuwana, przeliczenie odbywa się automatycznie.

**Obliczanie statystyk** Dane statystyczne dotyczące tabeli są obliczane na podstawie pewnej grupy losowo wybranych wierszy. Domyślnie skanowane jest 20 stron, ale wartość ta może być zmieniona poprzez parametr *innodb\_stats\_persistent\_sample\_pages* lub parametr *STATS\_SAMPLE\_PAGES*

dla konkretnej tabeli. Kiedy należy rozważyć zmianę liczby stron?

- **Wartości statystyczne odbiegają od rzeczywistych.**

Aby sprawdzić dokładność statystyk dla wybranego indeksu możemy porównać dane statystyczne znajdujące się w tabeli `innodb_index_stats` i porównać z liczbą rzeczywistych unikatowych wartości indeksu. Sprawdźmy dokładność danych statystycznych dla indeksów tabeli *Posts* i różnych liczb skanowanych stron. Tabela *Posts* zawiera dwa indeksy: `owner_idx` (`owner_id`) oraz `favorite_idx` (`FavoriteCount`, `Score`). W tym celu przygotujemy cztery zapytania:

Dla `owner_idx`:

```
SELECT count(DISTINCT OwnerUserId) from Posts; #liczba unikalnych
wartości indeksu
```

```
SELECT stat_value FROM mysql.innodb_index_stats WHERE database_name='stackOve
AND table_name = 'Posts' AND index_name = 'owner_idx' AND
stat_name = 'n_diff_pfx01'; # oszacowana liczba unikalnych wartości
indeksu
```

Dla `favorite_idx`:

```
SELECT count(DISTINCT FavoriteCount, Score) from Posts;
```

```
SELECT stat_value FROM mysql.innodb_index_stats WHERE database_name='stackOve
AND table_name = 'Posts' AND index_name = 'favorite_idx' AND
stat_name = 'n_diff_pfx01';
```

W poniższej tabeli zamieszczone są wyniki eksperymentu.

sample pages	owneridx	favorite_idx	ANALYZE TABLE czas [s]
1	3597380	8368	0.04
20	1364542	159689	0.06
400	1443184	22930	0.4
8000	1435072	23311	4.3
16000	1435072	23311	7
rzeczywista wartość	1435072	23311	

Jak widzimy, wraz ze wzrostem liczby stron użytych do analizy wzrasta dokładność statystyk, ale rośnie czas przeprowadzania analizy tabeli. Można też zauważyć, że dla domyślnej wartości parametru, oszacowana wartość unikatowych wartości indeksu favorite\_idx diametralnie różni się od rzeczywistej, co może doprowadzić do wyboru nieoptymalnego indeksu na etapie optymalizacji. W takiej sytuacji dobrym wyborem może być zwiększenie wartości parametru.

- **Zbyt długi czas zbierania statystyk dla tabeli**  
Eksperyment pokazał również, że przy wysokich wartościach parametru, serwer MySQL spędza dużo czasu na obliczaniu statystyk dla tabeli, co może prowadzić, szczególnie przy często zmieniających się tabelach, do wysokiego wykorzystania zasobów, szczególnie operacji odczytów z dysku.

## 6 Skalowalność i wysoka dostępność

Rozdział rozpoczne od wyjaśnienia kilku terminów i teorii, które będą przydatne podczas dalszego rozważania problemów wydajności i wysokiej dostępności MySQL.

## 6.1 Terminologia

### 6.1.1 Skalowalność a wydajność

Rozpoczynając ten rozdział najważniejszym zadaniem jest wyjaśnienie różnicy pomiędzy skalowalnością, a wydajnością. Termin *wydajność* w informatyce dotyczy ilości danych przetwarzanych w czasie. W przypadku baz danych termin ten może dotyczyć na przykład: ilości jednoczesnych połączeń do bazy danych, liczbie zapytań wykonywanych na sekundę lub rozmiar odczytywanych danych. Natomiast *skalowalność* oznacza możliwość aplikacji do zwiększenia wydajności. Skalowalny system to taki, w którym w najgorszym przypadku wzrost kosztów wynikający ze zwiększenia zasobów jest liniowy do wzrostu wydajności, jakie te zasoby zapewniają. Innymi słowy możliwy jest system, który jest bardzo wydajny, ale bardzo słabo skalowany. Możliwy jest także system niewydajny, który jest bardzo dobrze skalowany, dlatego istotnym jest, żeby nie mylić tych pojęć.

### 6.1.2 Teoria CAP

Autorem teorii CAP jest Eric Brewer, który przypisał bazę danych trzy własności:

- Spójność (eng. *Consistency*), oznaczająca, że odpytując dowolny działający węzeł, zawsze otrzymamy takie same dane.
- Dostępność (eng. *Availability*), określająca możliwość zapisywania i odczytywania danych nawet w przypadku awarii dowolnego węzła.
- Odporność na podział (eng. *Partition Tolerance*), pozwalająca na rozproszenie niewrażliwe na awarię.

Istotą teorii CAP jest stwierdzenie, że baza danych może spełniać co najwyżej dwie spośród trzech wymienionych wyżej własności. Konkluzją z powyższego stwierdzenia jest fakt nieistnienia idealnej bazy danych i każda z nich

jest pewnym kompromisem, kładącym nacisk na pewne własności, kosztem innych.

### **6.1.3 Rodzaje skalowalności**

Skalowaniem wertykalnym nazywamy zwiększanie wydajności pojedynczego serwera. Oznacza to wymianę serwera na taki o potężniejszych możliwościach. Wadą takiego rozwiązania jest ograniczenie możliwości pojedynczego serwera. Dodatkowo koszt zwiększenia możliwości pojedynczego serwera powyżej pewnej wartości przestaje być liniowy i każde zwiększenie wydajności serwera, wiąże się z wielokrotnie większym kosztem.

Drugim rodzajem skalowalności jest skalowalność horyzontalna, która polega na zwiększaniu liczby serwerów, przy zachowaniu ich wydajności. Takie rozwiązanie jest zdecydowanie tańsze i teoretycznie nie posiada limitu wydajności. Niestety wraz ze wzrostem liczby serwerów, rosną też problemy z zachowaniem spójności pomiędzy poszczególnymi węzłami.

## **6.2 Skalowanie wertykalne**

Zgodnie z tym co napisałem we wstępie rozdziału, skalowaniem pionowym nazywamy zwiększenie wydajności w ramach jednej monolitycznej maszyny. Takie rozwiązanie jest skuteczne tylko do pewnego momentu. Później takie skalowanie będzie się wiązało z szybkim dużym wzrostem kosztów, nawet przy małym wzroście możliwości serwera. Ostatecznie osiągniemy limit możliwości jednej maszyny i zwiększenie wydajności nie będzie dalej możliwe. Dodatkowo sam serwer MySQL ma problemy z wykorzystywaniem większej ilości procesorów i dysków twardych. Kolejnym utrudnieniem jest fakt, że MySQL nie potrafi obsłużyć pojedynczego zapytania na kilku procesorach, co jest sporym utrudnieniem przy dużych zapytaniach. Z powodu wyżej wymienionych ograniczeń skalowania pionowego, nie jest ono zalecane, jeżeli aplikacja ma zapewnić wysoką skalowalność.

## 6.3 Skalowanie horyzontalne

Skalowanie horyzontalne można zrealizować na kilka sposobów, które przedstawię w kolejnych podrozdziałach.

### 6.3.1 Mechanizm replikacji

Najpopularniejszym sposobem skalowania horyzontalnego jest realizacja architektury *master-slave*. Rozwiązanie polega na podziale serwerów bazodanowych na jeden serwer nadrzędny (*master*) oraz wiele serwerów podrzędnych (*slave*). Replikacja danych opiera się na bardzo następującej zasadzie. Serwer nadrzędny realizuje wszystkie operacje modyfikujące dane, równocześnie rejestrując każdą operację, która wykonał. Następnie każdy z serwerów podrzędnych odczytuje rejestr operacji (bin-log) i wykonuje zapytania. Efektem tej pracy jest wiele instancji bazy danych zawierających identyczne dane. Co ciekawe możliwe jest takie skonfigurowanie mechanizmu replikacji, żeby replikowane były dane z tylko niektórych schematów, lub nawet pojedynczych tabel. Taki podział ma wiele zalet. Przede wszystkim wyraźnie zwiększone zostaje wydajność operacji odczytu, ponieważ mogą być realizowane równolegle przez wiele instancji serwera MySQL. Dodatkowo redundancja danych na wielu instancjach sprawia, że system staje się bardziej odporny na utratę danych spowodowaną awarią sprzętową. W przypadku awarii na którymkolwiek z serwerów, dane na innych instancjach umożliwią przywrócenie stanu sprzed awarii. Dodatkowo zmniejsza się zapotrzebowanie na tworzenie kopii zapasowych danych, co jest szczególnie przydatne przy dużych rozmiarach danych, ponieważ operacje tworzenia kopii zapasowych mogą być znaczącym obciążeniem dla serwera. Dodatkowo operację backupu danych można wykonać na serwerze *slave* nie obciążając serwera głównego. Replikacja danych w trybie *master-slave* może odbywać się w jednym z następujących trybów:

- **SBR** (statement-based replication) - Najprostsza z metod. Zapisywane są tylko zapytania modyfikujące dane. SBR jest pierwszym typem



replikacji stosowanym w MySQL. Tryb ten może jednak prowadzić do braku spójności danych na różnych serwerach *slave*. Wyobraźmy sobie zapytanie, które zawiera funkcję `RAND()` lub funkcje odwołujące się do aktualnego czasu. Takie zapytania wykonane na różnych serwerach mogą zmodyfikować w taki sposób, że będą one niespójne.

- **RBR** (row-based replication) - logowane są wyniki zapytań modyfikujących dane, czyli informacja o tym, który rekord i w jaki sposób został zmodyfikowany. Jest to domyślny typ replikacji w MySQL 8. W porównaniu do metody SBR jest niestety wolniejsza i dodatkowo zwiększa się ilość przesyłanych danych pomiędzy serwerem master i slave.
- **MFL** - połączenie dwóch powyższych metod. W tym trybie domyślnie używana jest metoda SBR, ale w niektórych sytuacjach wykorzystywana jest metoda RBR.

Rozwiązanie polegające na replikacji danych do serwerów podległych idealnie sprawdzi się w aplikacjach, które przechowują ograniczoną wielkość danych, oraz zdecydowana większość operacji, to operacje odczytu. Taka realizacja skalowania poziomego może nie sprawdzić się w systemach przechowujących ogromne ilości danych, ponieważ wymaga przechowywania danych w rozmiarze wielokrotnie większym niż baza, wymaga powielania buforów, co może skutkować bardzo wysokimi kosztami serwerów, a nawet być niemożliwe, jeżeli rozmiar danych przekracza możliwości pojedynczego węzła. Kolejnym przypadkiem, którego nie rozwiązuje mechanizm replikacji jest system, w którym większość operacji to operacje modyfikujące dane. W takim scenariuszu serwer nadrzędny stanie się wąskim gardłem, a dużą część obciążenia serwerów będzie stanowiła replikacja pomiędzy serwerem nadrzędnym i serwerami podrzędnymi, co dodatkowo zmniejszy możliwości serwera nadrzędnego, który będzie przyjmował praktycznie całe obciążenie systemu. Reasumując skalowanie horyzontalne za pomocą mechanizmu replikacji idealnie sprawdzi się, zdecydowaną większością operacji, są operacje odczytu, a

rozmiar danych jest ograniczony.

## 6.4 Partycjonowanie funkcjonalne

Funkcje aplikacji można podzielić na pewne podgrupy, które nie łączą się z pozostałymi, na poziomie zapytań SQL. Przykładowo serwis społecznościowy *Facebook* umożliwia odczytywanie postów innych użytkowników oraz dokonywanie zakupów w sekcji *Marketplace*. Jeżeli użytkownik przegląda aktualne oferty w dziale *Marketplace*, to zapytania kierowane o bazę danych, będą odpytywać jedynie kilka tabel związanych z zakupami. Jeżeli w tym samym czasie inny użytkownik przegląda posty użytkowników, zapytania nie będą dotyczyć table związanych z zakupami. Oczywiście przykład, który przedstawiłem powyżej, może i z pewnością jest rozwiązany za pomocą podziału na osobne serwisy jeszcze na poziomie aplikacji, ale zakładając, że nasza aplikacja nie została podzielona na osobne serwisy i łączy się z jedną bazą danych. W takiej sytuacji obciążenie serwera MySQL jest sumą obciążeń związanych z postami i zakupami. W takim przypadku skutecznym rozwiązaniem jest podział danych pojedynczej aplikacji na zestaw tabel, które nigdy nie są ze sobą łączone. Oczywiście takiego partycjonowania danych nie można przeprowadzać w nieskończoność, ponieważ nie istnieje skończony zbiór tabel, które możemy w taki sposób podzielić. Dodatkowo w ramach każdej z grup jesteśmy ograniczeni możliwościami skalowania pionowego bazy danych. Wadą jest też zwiększona złożoność samej aplikacji, która musi obsłużyć kilka źródeł danych.

## 6.5 MySQL Cluster

//TODO dopisać. Książka strona 271 + link do dokumentacji + przykłady

## 7 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, wyszukiwanie konkretnego rekordu trwa krótki nawet przy niestosowaniu indeksów. Indeksy stają się jednak kluczowe wraz ze wzrostem zbioru danych. W MySQL istnieje wiele rodzajów indeksów, które są implementowane w warstwie silnika bazy danych, dlatego też nie każdy rodzaj indeksu jest obsługiwany przez wszystkie silniki. W tym rozdziale omówię tylko najpopularniejsze z nich.

### 7.1 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 poświęcę zdecydowaną większość 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 klu-

czami 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  $O(\log_M n)$ . Chcąc być dokładnym, należy wspomnieć, że MySQL do zapisu indeksów stosuje strukturę 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, założmy, 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  $\log_M n$ , 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 rzeczywistym 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 polecenia

```
CREATE INDEX user_post_idx
ON Comments(UserId,PostId);
```

utworzyłem indeks typu B-Tree na dwóch kolumnach *first\_UserId* oraz *last\_PostId*.

### Dopasowanie pełnego indeksu

Założmy, 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;
```

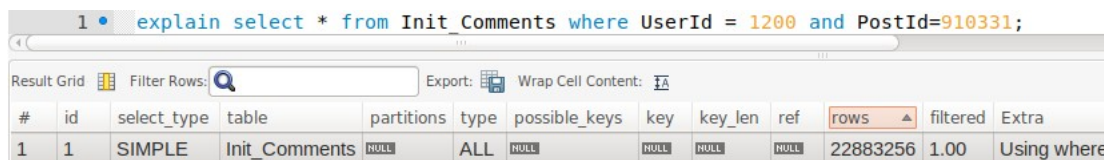
Zapytanie zwróciło wynik w 13,7 sekundy.

Następnie analogiczne zapytanie wykonałem na tabeli *Comments* zawierającej indeks na obu kolumnach. *employees\_idx*.

```
SELECT * FROM Comments WHERE UserId = 1200 AND PostId = 910331;
```

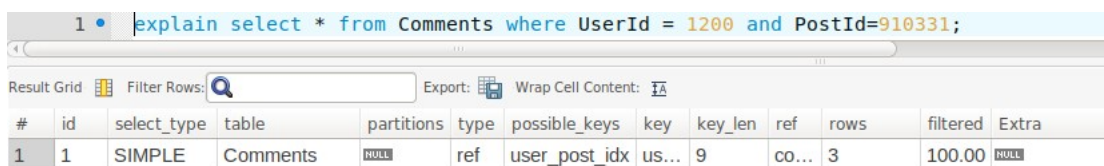
Tym razem zapytanie zwróciło wyniki w 0,013 sekundy. Tym razem serwer nie skanował całej tabeli. Z czego wynika różnica w czasie wykonania obu zapytań? Wykorzystując polecenie EXPLAIN dla obu zapytań otrzymujemy ciekawe dane. Rysunek 2 przedstawia wynik polecenia EXPLAIN dla pierwszego zapytania, natomiast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

indeksu. Tym razem serwer będzie musiał przeskanować jedynie 3 wiersze tabeli Comments.



#	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	1.00	Using where

Rysunek 13: EXPLAIN 1



#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Comments	NULL	ref	user_post_idx	us...	9	co...	3	100.00	NULL

Rysunek 14: EXPLAIN 2

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

**Dopasowanie prefiksu znajdującego się najbardziej na lewo** Dopasowanie prefiksu znajdującego się najbardziej na lewo może pomóc w wyszukiwaniu wszystkich 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 tabeli *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 sprawdzimy 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.

#	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 15: EXPLAIN 3

#	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 16: EXPLAIN 4

**Dopasowanie zakresu wartości** Dopasowanie zakresu wartości oznacza wyszukiwanie wartości w danym przedziale. W naszym przypadku może 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 17: 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 18: 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 znajdującego 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 nieprzydatny przy wyszukiwaniu na podstawie suffixu lub środkowej wartości. Następnym przypadkiem, w którym indeks typu B-Tree przyspiesza 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 indeksu jest zdecydowanie bardziej wydajne.

## 7.2 Indeksy typu 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 wierszy. 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 koniecz-

ność 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  $f_1$ ,  $f_2$  skrótu dla dwóch rekordów  $x_1$  oraz  $x_2$ , gdzie  $x_1$  jest mniejsze od  $x_2$  nie zapewniają, że  $f_1$  będzie mniejsze od  $f_2$ .

## 8 Praktyczne problemy

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

### 8.1 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 wypłynęła 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 pod-

zbiorem 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 index", 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	<small>NULL</small>	index	<small>NULL</small>	Rank_idx	8	<small>NULL</small>	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 rezultatu zapytania w zbiorze kolumn indeksu. Sprawdźmy teraz, co się stanie jeżeli do klauzuli WHERE dodamy kolejną kolumnę:

```
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	<small>NULL</small>	index	<small>NULL</small>	Rank_idx	8	<small>NULL</small>	2466330	100.00	Using index

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	ALL	NULL	NULL	NULL	NULL	2466330	100.00	Using filesort

Wynik polecenia EXPLAIN jest interesujący. Przypomnijmy sobie zatem, w jaki sposób MySQL przechowuje dane w indeksie, jeżeli tabela posiada klucz podstawowy. W takim przypadku wiersze w liściach indeksu są identyfikowane za pomocą wartości kluczy głównych. W naszym przypadku wiersze w indeksie są identyfikowane na podstawie kolumny *id*, co oznacza, że indeks zawiera wszystkie kolumny użyte w zapytaniu.

Kolejnym często używanym zapytaniem jest pobranie wszystkich kolumn z tabeli, ale posortowanie 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 jednoznacznej 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.