

Uniwersytet im. Adama Mickiewicza w Poznaniu
Wydział Matematyki i Informatyki



Optymalizacja wydajności baz danych MySQL

Database optimization in MySQL

Franciszek Słupski

Nr albumu: 452122

Kierunek: Informatyka

Praca magisterska
napisana pod kierunkiem
prof. dr hab. Marka Wisły

Poznań, 2020

Poznań, dnia

OŚWIADCZENIE

Ja, niżej podpisany student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt. Audyty sieci teleinformatycznych z wykorzystaniem narzędzia Nessus napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem/ tej rozprawy lub jej części od innych osób. Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej. Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego, stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[.....]* - wyrażam zgodę na udostępnienie mojej pracy w czytelni Archiwum UAM

[.....]* - wyrażam zgodę na udostępnienie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

*Należy wpisać TAK w przypadku wyrażenia zgody na udostępnianie pracy w czytelni archiwum UAM, NIE w przypadku braku zgody. Niewypełnienie pola oznacza brak zgody na udostępnienie pracy.

.....
(czytelny podpis studenta)

Streszczenie

Celem niniejszej pracy jest analiza zagadnienia optymalizacji bazy danych na przykładzie MySQL. Relacyjne bazy danych są obecnie jednym z najpopularniejszych sposobów przechowywania trwałych danych w informatyce. Praca zawiera zbiór teoretycznych zagadnień związanych z wydajnym stosowaniem bazy danych MySQL poparty wieloma przykładami. Badania i przykłady zostały przygotowane w taki sposób, aby ich powtórzenie przez czytelnika pracy było możliwie proste i wygodne. Dodatkowo większość przykładów nie jest specyficzna jedynie dla baz danych MySQL i techniki optymalizacji oraz zasada działania wielu elementów jest bardzo podobna do konkurencyjnych relacyjnych baz danych. Na końcu pracy przedstawiono rozwiązanie kilku często spotykanych problemów i wątpliwości dotyczących efektywnego używania bazy danych MySQL.

Słowa kluczowe: baza danych, relacyjna baza danych, MySQL, indeks bazodanowy, optymalizator MySQL, skalowalność

Abstract

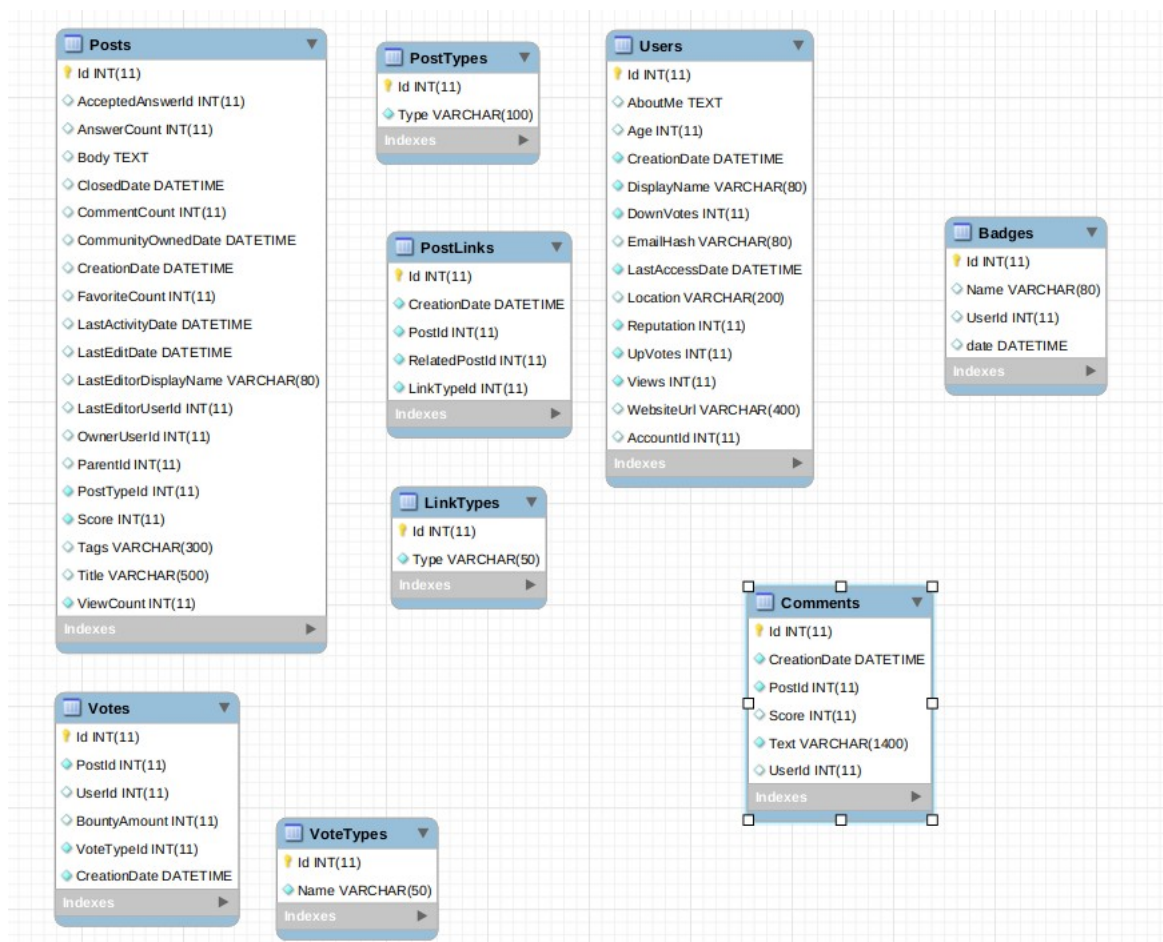
This thesis describes problem of database optimization on the example of MySQL. Relational databases are one of the most popular ways of storing persistent data in IT today. Thesis contains a set of theoretical issues related to the efficient use of the MySQL database, supported by many examples. The research and examples have been prepared in such a way that their repetition by the reader of the work is as simple and convenient as possible. In addition, most examples are not specific to MySQL databases and optimization techniques, and the principle of operation of many elements is very similar to competing relational databases. At the end of the thesis, a solution to some common problems and doubts regarding the effective use of the MySQL database was presented.

Key words: database, relational database, MySQL, database index, MySQL optimizer, scalability

Spis treści

1	Wstęp	10
1.1	Testowa baza danych	10
2	Porównywanie zapytań	11
2.1	Polecenie EXPLAIN	12
2.2	Wyniki polecenia EXPLAIN	12
3	Architektura MySQL	22
3.1	Obsługa połączeń i wątków	22
3.2	Bufor zapytań	22
3.3	InnoDB Buffer Pool	23
3.4	Mechanizm Multiversion Concurrency Control	24
4	Silniki magazynu danych	25
4.1	Krótką charakterystyka podstawowych silników.	25
4.2	Porównanie silników	29
5	Optymalizator MySQL	38
5.1	Możliwości optymalizatora	38
5.2	Dane statystyczne dla optymalizatora	40
5.3	Plan wykonania zapytania	40
5.4	Optymalizator złączeń	41
5.5	Konfiguracja optymalizatora złączeń	43
5.6	Konfiguracja statystyk tabel InnoDB	46
6	Skalowalność i wysoka dostępność	49
6.1	Terminologia	49
6.2	Architektura master-slave	50
6.3	Architektura multi-master	52
6.4	Partycjonowanie funkcjonalne	53

6.5	Data-sharding	54
6.6	InnoDB Cluster	55
6.7	Podsumowanie	56
7	Partycjonowanie Tabel	58
7.1	Metody partycjonowania	58
7.2	Przypadki użycia	60
7.3	Podsumowanie	61
8	Indeksy	62
8.1	Indeksy typu B-Tree	62
8.2	Indeksy typu hash	67
9	Praktyczne problemy	69
9.1	Sortowanie wyników	69
9.2	Przechowywanie wartości NULL czy pustej wartości	71
9.3	Indeks na wielu kolumnach czy wiele indeksów na jednej	73
9.4	Sztuczny czy naturalny klucz główny	74
9.5	Procedury składowane	74
9.6	Monitorowanie niewydajnych zapytań	74
	Bibliografia	75



Rysunek 1: Schemat bazy danych stackoverflow

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 kluczy 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 i jest podstawową metodą określenia sposobu wykonywania zapytań przez serwer MySQL. Analiza wyników polecenia 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 bufor 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 nieproporcjonalnie szybko. 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, a taka operacja wymaga dodatkowego nakładu na pobranie danych do pamięci, co skutkuje wydłużeniem czasu wykonania zapytania. Wykonując kolejne zapytanie, może okazać się, że pomimo pogorszenia jego wydajności, zostanie ono wykonane w krótszym czasie (ze względu na różnicę w czasie dostępu do pamięci i dysku twardego). Mierzac 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. Problem ten można rozwiązać poprzez, obliczenie średniego czasu i na jego podstawie porównywanie wyników. Takie rozwiązanie jednak wciąż nie gwarantuje deterministycznego charakteru naszego porównania.

Dodatkowo baza danych rzadko kiedy jest całkowicie odcięta od świata. Z reguły testowanie wydajności będzie odbywać się dla tabeli, które są w jednocześnie modyfikowane przez inne połączenia. Przykładowo jeżeli testujemy zapytanie na tabeli, na której w tym samym czasie wykonywane są operacje zapisu; porównanie czasów wykonania zapytań nie musi być miarodajne.

Na czasy wykonywania zapytań wpływać może również aktualne obciążenie serwera. Wyniki czasów wykonania zapytań będą wyraźnie zależeć od aktualnego poziomu wykorzystania zasobów bazy danych.

Po przeanalizowaniu powyższych ograniczeń metody analizy wydajności zapytań,

można stwierdzić, że operowanie tylko na takiej metodzie jest obarczone wyraźnym błędem i nie powinno być jedynym sposobem porównywania wydajności.

2.1 Polecenie EXPLAIN

Polecenie EXPLAIN będzie jedną z głównych metod porównywania wydajności zapytań stosowaną w tej pracy, dlatego w tym podrozdziale przedstawiono podstawowy jego stosowania. Język SQL jest językiem deklaratywnym. Decyzję o sposobie przechowywania i pobrania danych pozostawia się systemowi zarządzania bazą danych. Funkcja EXPLAIN służy do określenia sposobu, w jaki baza danych wykona zapytanie.

Aby użyć polecenia EXPLAIN, należy poprzedzić słowa kluczowe takie jak SELECT, INSERT, UPDATE, DELETE poleceniem EXPLAIN. Spowoduje to, że zamiast wykonania zapytania, baza danych zwróci informacje o planie jego wykonania. Rezultat polecenia EXPLAIN zawiera po jednym rekordzie dla każdej tabeli użytej w zapytaniu, chociaż czasami może zawierać również tabele stworzone przez serwer w pamięci. Kolejność wierszy w wyniku zapytania odpowiada kolejności, w jakiej MySQL będzie je wykonywał. Pierwszym zapytaniem wykonanym przez MySQL będzie zapytanie z ostatniego wiersza.

2.2 Wyniki polecenia EXPLAIN

W celu zademonstrowania wyników polecenia EXPLAIN na rzeczywistych przykładach, wykonano polecenie EXPLAIN dla kilku zapytań na bazie StackOverflow. 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	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

```
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	NULL	range	PRIMARY	PRIMARY	4	NULL	79	100.00	Using where
2	2	DEPENDENT SUBQUERY	Posts	NULL	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	NULL	index_merge	PRIMARY,user_post_idx	user_post_idx...	5,4	NULL	2	100.00	Using sort_union(user_post_idx,PRIMARY); Using where

Rysunek 11: Przykład 10

Kolumna

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

Kolumna ID

Kolumna id zawiera numer zapytania, którego dotyczy. W przypadku zapytań z podzapytaniem, podzapytania w dyrektywie FROM oraz zapytań ze słowem kluczowym JOIN, 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` informuje o rodzaju wykonywanego zapytania `SELECT`. Poniżej przedstawiono wartości, które mogą zostać zwrócone dla tej kolumny.

- **SIMPLE** Wartość `SIMPLE` oznacza, że zapytanie nie zawiera podzapytań, oraz nie używa złączeń (klauszula `UNION`).
- **PRIMARY** Jeżeli zapytanie zawiera podzapytania lub wykorzystuje złączenie, to rekord dla kolumny `select_type` przyjmie wartość `PRIMARY` (przykład 2).
- **SUBQUERY** Jeżeli rekord dotyczy podzapytania oznaczonego jako `PRIMARY`, to zostanie oznaczony jako `SUBQUERY` (przykład 3)
- **UNION** Jako `UNION` zostaną oznaczone zapytania, które są drugim i kolejnym zapytaniem operacji złączenia tabel. 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.
- **UNION RESULT** Oznacza wiersz, w którym polecenie `SELECT` zostało użyte do pobrania wyników z tabeli tymczasowej użytej przy poleceniu `UNION` (przykład 2).
- **DEPENDENT SUBQUERY** Jeśli polecenie `SELECT` zależy od danych znajdujących się w podzapytaniu (przykład 5).
- **DEPENDENT UNION** Jeśli polecenie `SELECT` zależy od danych znajdujących się w tabeli tymczasowej będącej wynikiem złączenia (przykład 6).
- **MATERIALIZED** Jeżeli wynik zwracany jest ze *zmaterializowanego widoku* (eng. *materialized view*). Widok zmaterializowany jest obiektem bazy danych zawierającym rezultat zapytania.
- **UNCACHABLE_SUBQUERY**. Oznaczający, że zapytanie nie może być buforowane (przykład 7).
- **UNCACHABLE_UNION**. Oznaczający, że wynik złączenia tabel nie może zostać buforowany.

Kolumna *table*

Kolumna *table* w większości przypadków zawiera nazwę tabeli lub jej alias, do której odnosi się dany wiersz wyniku polecenia *EXPLAIN*. 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 roz-

poczyna 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 wykonania 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 liście 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. Jest to liczba przeszukiwanych rekordów na danym poziomie zagnieżdżenia pętli planu złączenia. Oznacza to, 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

Podczas szacowania wartości w kolumnie `rows` optymalizator 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 przedstawiono 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.
- 'Using

Z opisu możliwości polecenia `EXPLAIN` zawartego w tym rozdziale wynika, że jego użycie niesie więcej informacji od mierzenia czasów wykonywanych operacji. Dzięki szczegółowym informacjom na temat planu wykonania zapytania (termin opisany szerzej w rozdziale dotyczącym Optymalizatora MySQL) możliwe jest dokładniejsze zdefiniowanie przyczyn braku wydajności, jak i porównanie wydajności dwóch zapytań. Oczywiście nie można pominąć faktu, że dane przedstawione jako wynik analizy są zebrane

na podstawie pewnych statystyk, które nie zawsze muszą być zbieżne z rzeczywistością. Z tego powodu mierzenie czasów może być skutecznym dopełnieniem analizy planu wykonania zapytania.

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.

3.2 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 odpowiedź zapytanie z tabeli bufora. Jest to spowodowane tym, 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`.

W MySQL 8.0 bufor zapytań został usunięty z serwera, ale wciąż jest dostępny w rozwiązaniach takich jak *ProxySQL*. W takiej architekturze bufor znajduje się jeszcze przed serwerem MySQL. Przykład takiej architektury znajduje się w rozdziale dotyczącym skalowania horyzontalnego.

3.3 InnoDB Buffer Pool

InnoDB Buffer Pool jest przestrzenią pamięci przydzieloną dla serwera MySQL, w której przechowywane są często używane dane oraz indeksy. Dzięki temu możliwy jest szybszy dostęp do często używanych danych.

Dane w buforze przechowywane są w postaci listy podzielonej na 2 części (nową oraz starą). Domyślnie nowa podlista zawiera 5/8 długości całej listy, a punkt ten jest nazywany środkiem listy. Jeżeli strona jest odczytywana po raz pierwszy, trafia na koniec starej części (środek listy), a ostatni element ze starej części jest usuwany. Jeżeli w kolejnym odczycie strona zostanie pobrana ze starej podlisty, trafia na koniec nowej części. W efekcie tego w nowej podliście przechowywane są najczęściej używane dane i dzięki zastosowanemu podziałowi jest mniejsze prawdopodobieństwo usunięcia najczęściej odczytywanych stron przez dodanie strony, która prawdopodobnie zostanie odczytana tylko raz.

3.4 Mechanizm Multiversion Concurrency Control

Mechanizm MVCC (Multiversion Concurrency Control) jest implementacją mechanizmu optymistycznej wersji kontroli współbieżności. MVCC nie jest używany jedynie w bazach danych MySQL, ale jest ogólnie stosowany w większość nowoczesnych relacyjnych bazach danych.

Podstawową ideą mechanizmu jest utrzymanie stałego obrazu danych dla transakcji. To oznacza, że z punktu widzenia transakcji stan bazy danych jest niezmienny, niezależnie od czasu jej trwania. Taka właściwość jest możliwa dzięki zastosowaniu dwóch dodatkowych ukrytych kolumn dla każdego wiersza: pierwsza kolumna oznacza numer wersji jego utworzenia, natomiast druga kolumna numer wersji, w której został usunięty. Numer wersji jest zwiększany z rozpoczęciem każdej kolejnej transakcji i jest unikatowy dla każdej z nich.

W ramach transakcji odczytywane mogą być jedynie wiersze z numerem wersji równym lub mniejszym od wersji bieżącej transakcji oraz numer wersji usunięcia wiersza musi być większy od aktualnej wersji transakcji.

Numer wersji mogą być modyfikowane w następujących przypadkach.

- **INSERT** Nowe rekordy zapisywane są z numerem wersji transakcji, w której zostały dodane, oraz niezdefiniowaną wersją usunięcia.
- **DELETE** Numer wersji usunięcia zapisywany jest jako wersja transakcji, w której rekord został usunięty.
- **UPDATE** Tworzona jest kopia rekordu z numerem wersji transakcji, w której został zmodyfikowany. Dodatkowo wersja oznacza moment usunięcia starego rekordu.

Dzięki temu odczytywanie danych nie będzie wymagało blokowania nawet na poziomie rekordu. Zapewnienie spójności wewnątrz transakcji odbywa się poprzez sprawdzenie dwóch powyższych warunków, zamiast blokowania rekordu na czas odczytu. Wiersze muszą być blokowane jedynie na czas modyfikowania danych.

Wadą mechanizmu MVCC jest konieczność przechowywania zwiększonej ilości danych. Dodatkowo mechanizm nie działa ze wszystkimi poziomami izolacji. Poziom izolacji `READ_UNCOMMITTED` jest sprzeczny z jego ideą, ponieważ na tym poziomie izolacji, wewnątrz transakcji dostępne są dane równolegle modyfikowane przez pozostałe transakcje. Podobnie poziom izolacji `SERIALIZABLE`, który blokuje wiersze, aż do momentu zakończenia transakcji nie będzie wykorzystywał mechanizmu MVCC.

4 Silniki magazynu danych

Niniejszy rozdział poświęcony jest omówieniu zagadnień związanych z silnikami bazy danych. Silnik bazy danych jest częścią bazy danych odpowiedzialną za wykonywanie kodu SQL, czyli wykonywanie operacji na danych. SQL jest językiem deklaratywnym. Klient podając polecenie SQL, opisuje warunku, jakie musi spełnić końcowe rozwiązanie, a nie szczegółową implementację. Z tego wynika, że to silnik bazy danych odpowiedzialny jest za dostarczenie implementacji pozwalającej na wykonywanie kodu SQL. Silnik dodatkowo definiuje sposób przechowywania danych oraz zbiór operacji, które możemy na nich wykonać.

Architektura MySQL umożliwia korzystanie z wielu różnych silników. Silnik bazy danych wybierany jest per tabela, co oznacza, że w ramach pojedynczej bazy danych można używać różnych silników.

4.1 Krótka charakterystyka podstawowych silników.

W tym podrozdziale nie zostały przedstawione szczegółowe opisy silników dostępnych w MySQL, raczej ich główne charakterystyki, zalety oraz ograniczenia. Część z silników pominięto ze względu na ich marginalną popularność oraz zastosowanie. W zdecydowanej większości przypadków aktualnie najodpowiedniejszym silnikiem jest InnoDB, aczkolwiek każdy z silników opisanych w tym rozdziale ma pewne zalety i w specyficznych przypadkach przedstawionych w tym podrozdziale, warto rozważyć użycie silnika alternatywnego do InnoDB.

MyISAM

MyISAM był domyślnym silnikiem składowania danych do wersji 5.4 (włącznie). Każda tabela przechowywana jest w dwóch plikach na dysku twardym. Dane przechowywane są w pliku z rozszerzeniem **.MYD (MYData)** natomiast w drugim pliku (**.MYI(MYIndex)**) składowane są indeksy. Poniżej przedstawiono kilka cech tego silnika bazy danych.

- **Brak wsparcia dla transakcji.** Z tego powodu MyISAM nie powinien być używany do tabel, dla których istotnym wymaganiem jest zapewnienie integralności danych.
- **Obsługa indeksów B-tree oraz Geospatial.**

- **Blokady tabeli.** W momencie wykonywania operacji dodającej dane do tabeli jest ona blokowana na cały czas wykonywania operacji (również dla operacji odczytujących dane). Sprawia to, że w przypadku dużej liczby operacji modyfikujących dane - wydajność bazy danych wyraźnie spada.
- **Brak obsługi mechanizmu kluczy obcych**
- **Mechanizm kompresji danych.** Silnik umożliwia kompresowanie danych w celu optymalizacji ilości miejsca potrzebnego do przechowywania danych z tabeli. Taka operacja sprawia, że skompresowane dane są dostępne jedynie do odczytu, a ich modyfikacja jest zablokowana i wymaga rozpakowania danych. Tabele MyISAM można kompresować i dekompresować za pomocą mechanizmu *myisam-pack*.
- **Buforowanie indeksów.** Silnik MyISAM buforuje jedynie indeksy.
- **Obsługa statystyk.**

W czasie pisania tej pracy silnik MyISAM nie był już rozwijany, dlatego raczej odradzane jest jego stosowanie w nowszych wersjach serwera MySQL.

InnoDB

Od wersji 5.5 InnoDB jest domyślnym silnikiem bazy danych MySQL. Mechanizm InnoDB uzupełniono o funkcje, których brakowało w MyISAM i obecnie jest zdecydowanie najpopularniejszym wyborem. Domyślnie dane przechowywane są w pojedynczych plikach, ale możliwe być również przechowywane w wielu plikach. Strukturę plików bazy *StackOverflow*, która dla wszystkich tabel używa silnika InnoDB, przedstawiono na rysunku 13. Poniżej przedstawiono podstawowe charakterystyki silnika.

Rysunek 13: Pliki silnika InnoDB testowej bazy danych *StackOverflow*

```
root@frank-ThinkPad-T440p:/var/lib/mysql/stackOverflowMedium# ls -lh
razem 35G
-rw-r----- 1 mysql mysql 504M gru 15 2019 Badges.ibd
-rw-r----- 1 mysql mysql 9,4G paź 8 2019 Comments.ibd
-rw-r----- 1 mysql mysql 112K gru 16 2019 lecturers.ibd
-rw-r----- 1 mysql mysql 112K paź 8 2019 LinkTypes.ibd
-rw-r----- 1 mysql mysql 88M gru 15 2019 PostLinks.ibd
-rw-r----- 1 mysql mysql 23G gru 28 15:16 Posts.ibd
-rw-r----- 1 mysql mysql 112K paź 8 2019 PostTypes.ibd
-rw-r----- 1 mysql mysql 288K gru 16 2019 students.ibd
-rw-r----- 1 mysql mysql 288K gru 16 2019 table_child.ibd
-rw-r----- 1 mysql mysql 112K gru 16 2019 table_parent.ibd
-rw-r----- 1 mysql mysql 380M gru 15 2019 Users.ibd
-rw-r----- 1 mysql mysql 2,0G paź 8 2019 Votes.ibd
-rw-r----- 1 mysql mysql 112K paź 8 2019 VoteTypes.ibd
root@frank-ThinkPad-T440p:/var/lib/mysql/stackOverflowMedium#
```

- **Wsparcie dla transakcji.** Silnik wspiera transakcje oraz wszystkie cztery poziomy izolacji modelu *ANSI*. Poziom izolacji transakcji określa zasady widoczności pomiędzy współbieżnymi transakcjami. Dzięki wsparciu dla wszystkich czterech izolacji silnik InnoDB spełnia wymagania stawiane aplikacją wymagającym zapewnienie integralności danych.
- **Wsparcie dla indeksów.** InnoDB wspiera najważniejsze indeksy takie jak: B-tree, Hash, Spatial.
- **Blokowanie dostępu na poziomie rekordów.** Dostęp do tabel InnoDB jest blokowany za pomocą mechanizmu MVCC (Multi-Versioned Concurrency Control). Blokowane są pojedyncze rekordy, zamiast całej tabeli. Wprowadzenie tej zmiany znacząco zwiększyło wydajność równoległych operacji modyfikujących dane w tabeli.
- **Wsparcie dla kluczy obcych.**
- **Buforowanie danych oraz indeksów.** Silnik InnoDB może buforować nie tylko indeksy, ale również dane.
- **Nieskompresowane indeksy.** Silnik InnoDB nie kompresuje indeksów, co prowadzi do zwiększenia zużycia przestrzeni dyskowej.
- **Wsparcie dla partycjonowania.** Szerzej opisane w podrozdziale dotyczącym partycjonowania.
- **Obsługa statystyk.**

CSV Storage Engine

Silnik CSV Storage Engine przechowuje dane tabeli w plikach tekstowych w formacie csv z wartościami rozdzielonymi przecinkami. Ten silnik może być przydatny, jeżeli chcemy nasze dane przechowywać w formacie csv. Posiada wiele ograniczeń, dlatego nie jest zalecane jego stosowanie, o ile nie zależy nam na przechowywaniu danych tabeli w formacie CSV. Poniżej przedstawiono podstawowe ograniczenia tego silnika.

- **Brak wsparcia dla indeksów i kluczy obcych.**
- **Brak wsparcia dla transakcji.**
- **Brak możliwości przechowywania wartości *null*.**
- **Brak wsparcia dla partycjonowania.**

Memory

Silnik Memory przechowuje wszystkie dane w pamięci, a nie na dysku twardym. Te dane są ulotne i zostają usunięte w momencie restartu serwera (struktura tabeli zostaje zachowana). Z powodu przechowywania w pamięci są o rząd wielkości szybsze od standardowych silników baz danych, ale ze względu na swoją ulotność nie powinny przechowywać istotnych danych dla aplikacji. Poniżej przedstawiono podstawowe własności tabeli Memory.

- **Wsparcie dla indeksów** Tabele Memory obsługują indeksy Hash oraz B-tree. Domyślnym indeksem jest indeks typ Hash.
- **Blokowanie na poziomie tabeli.** Podobnie jak tabele MyISAM, w momencie modyfikowania danych blokowana jest cała tabela.
- **Brak obsługi typów TEXT oraz BLOB.** Tabele nie obsługują typów TEXT. Przechowywanie tekstu możliwe jest w kolumnach VARCHAR o stałej zdefiniowanej wielkości, co prowadzi do marnotrawienia pamięci.
- **Brak danych statystycznych indeksu.** Tabele MEMORY nie przechowują statystyk dotyczących indeksów, co czasami może skutkować wybraniem nieodpowiedniego indeksu przez optymalizator zapytań i w efekcie pogorszenie wydajności zapytań.
- **Brak wsparcia dla transakcji.**

Zastosowanie silnika MEMORY warto rozważyć w następujących sytuacjach.

- Pamięć podręczna dla często odczytywanych danych, która jest wczytywana w momencie startu serwera.
- Buforowanie wyników agregowanych danych z często wykonywanych zapytań.
- Przechowywanie wyników pośrednich z zapytań.
- MySQL używa tabeli Memory do wewnętrznego przetwarzania zapytań wymagających tabeli tymczasowych do przechowywania wyników pośrednich.

Silnik Archive

Archive jest silnikiem służącym do przechowywania dużej ilości nieindeksowanych danych, które są rzadko pobierane.

Podstawowe cechy tego silnika zostały przedstawione poniżej.

- **Brak wsparcia dla transakcji.**
- **Możliwe wykonywanie jedynie operacji INSERT, REPLACE oraz SELECT.** W tabelach Archive niemożliwe jest usuwanie i modyfikowanie istniejących krotek.
- **Brak wsparcia dla indeksów.**
- **Blokowanie na poziomie tabeli.**
- **Kompresowanie danych.** Każdy wstawiony rekord jest automatycznie kompresowany za pomocą *zlib*, dlatego tabele Archive wymagają zdecydowanie mniej miejsca od tabel InnoDB lub MyISAM.

4.2 Porównanie silników

Przechowywanie danych

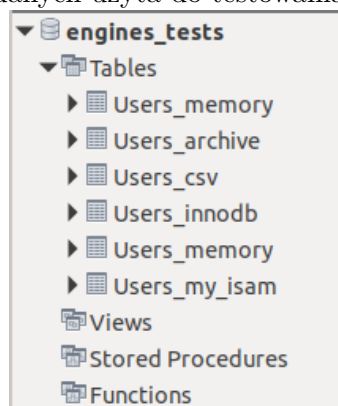
W celu porównania sposobu przechowywania danych na dysku przygotowano testową bazę danych, zawierającą pięć tabel będących niemalże kopią tabeli *Users* z bazy testowej, z których każda wykorzystuje inny silnik. Jedyną zmianą jest brak kolumny *AboutMe*, która została usunięta ze względu na brak wsparcia dla kolumn TEXT w silniku MEMORY. Do tworzenia tabel wykorzystano polecenia z poniższego listingu, a następnie zaimportowano dane z bazy *Stackoverflow*. Ponieważ nie wszystkie silniki wspierają przechowywanie wartości NULL, zamieniono domyślne wartości NULL na wartość 0 lub pusty tekst (w zależności od typu danych).

```
CREATE TABLE 'Users' (  
  'Id' int NOT NULL,  
  'Age' int NOT NULL DEFAULT 0,  
  'CreationDate' datetime NOT NULL,  
  'DisplayName' varchar(80) NOT NULL,  
  'DownVotes' int NOT NULL,  
  'EmailHash' varchar(80) NOT NULL DEFAULT '',  
  'LastAccessDate' datetime NOT NULL,  
  'Location' varchar(200) NOT NULL DEFAULT '',  
  'Reputation' int NOT NULL,  
  'UpVotes' int NOT NULL,  
  'Views' int NOT NULL,  
  'WebsiteUrl' varchar(400) NOT NULL DEFAULT '',
```

```
'AccountId' int NOT NULL DEFAULT 0,
PRIMARY KEY ('Id') -- w tabelach, które wspierają klucze główne
) ENGINE=<nazwa silnika>;
```

Klucze główne zostały usunięte w przypadku silników, które ich nie wspierają. Ostatecznie tabela zawiera około 2,5 miliona wierszy.

Rysunek 14: Baza danych użyta do testowania silników baz danych

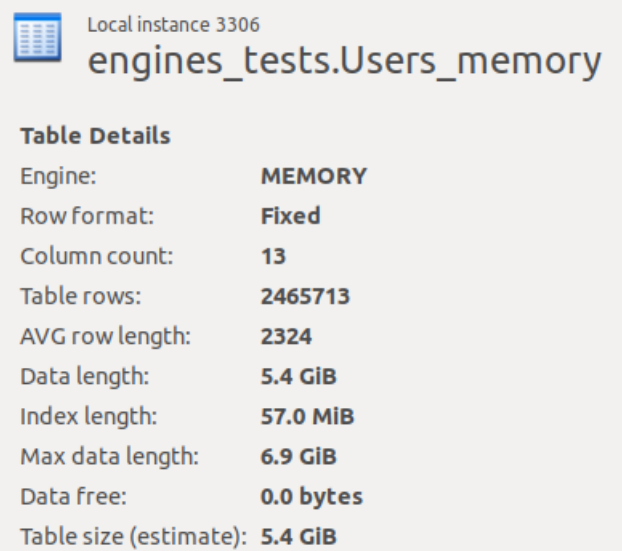


Rysunek 15: Pliki z danymi użytkowników dla testowych silników.

```
root@franek-ThinkPad-T440p:/var/lib/mysql/engines_tests# ls -lh
razem 748M
-rw-r----- 1 mysql mysql 15K cze 21 00:01 @0023sql@002d5deb_9_1347.sdi
-rw-r----- 1 mysql mysql 14K cze 20 23:28 Users_archive_1343.sdi
-rw-r----- 1 mysql mysql 67M cze 20 23:27 Users_archive.ARZ
-rw-r----- 1 mysql mysql 14K cze 20 23:26 Users_csv_1341.sdi
-rw-r----- 1 mysql mysql 35 cze 21 04:43 Users_csv.CSM
-rw-r----- 1 mysql mysql 256M cze 20 23:26 Users_csv.CSV
-rw-r----- 1 mysql mysql 252M cze 20 23:16 Users_innodb.ibd
-rw-r----- 1 mysql mysql 14K cze 20 23:56 Users_memory_1346.sdi
-rw-r----- 1 mysql mysql 11K cze 20 23:20 Users_my_isam_1338.sdi
-rw-r----- 1 mysql mysql 150M cze 20 23:18 Users_my_isam.MYD
-rw-r----- 1 mysql mysql 25M cze 21 02:28 Users_my_isam.MYI
```

Na rysunku 15 przedstawiono pliki tabel testowej bazy danych. Pod względem optymalizacji ilości miejsca na dysku zdecydowanie najlepiej wypada silnik Archive, który potrzebuje jedynie 67 MB dla danych użytkowników. Na drugim miejscu pod tym względem plasuje się silnik MyISAM. Silniki InnoDB oraz CSV wymagają praktycznie takiej samej przestrzeni dyskowej; w granicach 250 MB. Silnik MEMORY na dysku twardym przechowuje jedynie strukturę tabeli, ale do sprawdzenia ilości użytej pamięci możemy użyć narzędzia *MySQL Workbench*. Silnik MEMORY wymaga prawie 5.5 Gb pamięci. Wynika to w dużej mierze z tego, że silnik MEMORY dla kolumn VARCHAR zawsze rezerwuje rozmiar wynikający z maksymalnej wartości, nawet jeżeli nie jest ona

Rysunek 16: Informacje dotyczące tabeli `Users_memory` w *MySQL Workbench*. *StackOverflow*



Local instance 3306	
engines_tests.Users_memory	
Table Details	
Engine:	MEMORY
Row format:	Fixed
Column count:	13
Table rows:	2465713
AVG row length:	2324
Data length:	5.4 GiB
Index length:	57.0 MiB
Max data length:	6.9 GiB
Data free:	0.0 bytes
Table size (estimate):	5.4 GiB

wykorzystana.

Wymaganie transakcyjności

W przypadku tabel, które wymagają użycia transakcji, jedynym możliwym wyborem jest silnik InnoDB.

Operacje odczytu klucz-wartość

Do testowania użyto narzędzia *sysbench*, które posiada wbudowany mechanizm ułatwiający testowanie baz danych. Za pomocą poniższych poleceń rzygotowano testową bazę danych udostępnioną przez *sysbench*.

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=root --mysql-db=test  
--table_size=2000000 --range_selects=off --mysql_storage_engine=  
<nazwa silnika> /usr/share/sysbench/oltp_read_only.lua prepare
```

Baza danych zawiera 2 miliony rekordów, tabela domyślnie przyjmuje nazwę *sbtest1*.

Rysunek 17: Struktura tabeli *sbtest1*.

#	Field	Type	Null	Key	Default	Extra
1	id	int	NO	PRI	HULL	auto_increment
2	k	int	NO	MUL	0	
3	c	char(120)	NO			
4	pad	char(60)	NO			

Aby wykonać test polecenie *prepare* należy zamienić na polecenie *run*. Przy takiej konfiguracji wykonywane jest następujące zapytanie:

```
SELECT c FROM sbtest1 WHERE id=?
```

Rysunek 18: Przykładowe statystyki testu wydajności izolowanych operacji odczytu.

```

Initializing worker threads...

Threads started!

SQL statistics:
  queries performed:
    read:                182680
    write:                0
    other:               36536
    total:              219216
  transactions:         18268 (1826.29 per sec.)
  queries:              219216 (21915.49 per sec.)
  ignored errors:       0 (0.00 per sec.)
  reconnects:           0 (0.00 per sec.)

General statistics:
  total time:           10.0008s
  total number of events: 18268

Latency (ms):
  min:                  0.36
  avg:                  0.55
  max:                  7.48
  95th percentile:     0.77
  sum:                  9975.63

Threads fairness:
  events (avg/stddev):  18268.0000/0.00
  execution time (avg/stddev): 9.9756/0.00

```

-	MyISAM	InnoDB	Memory	Archive	CSV
średni czas [ms]	0.54	0.55	0.38	powyżej minuty	powyżej minuty

Z powyższej tabeli wynika, że w przypadku zapytań używających pełnego klucza głównego, silnik MEMORY jest najwydajniejszy. Dobry wynik wynika z faktu zastosowania indeksu HASH na kolumnie Id. Silniki InnoDB oraz MyISAM prezentują podobną wydajność w przypadku zapytań używających indeksy (w tym przypadku indeksy BTree). Silniki Archive oraz CSV wyraźnie odstają w tym zestawieniu ze względu na brak obsługi kluczy głównych.

Symultaniczne operacje odczytu z wykorzystaniem klucza głównego oraz operacji zapisu.

Do przygotowania testowych zestawów danych wykorzystano następujące skrypty:

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=root --mysql-db=test  
--table_size=2000000 --num-threads=12 --range_selects=off --mysql_storage_engine=<nazwa silnika>  
/usr/share/sysbench/oltp_read_write.lua prepare
```

Wykonanie testu analogicznie jak w poprzednich przypadkach wykonano, zastępując słowo kluczowe *prepare* na *run*.

Na rysunku 19 przedstawiono wyniki testu. W przypadku tego testu operacje odczytu wykonywane były równolegle z operacjami zapisu do tabeli.

Rysunek 19: Przykładowe statystyki testu symultanicznych odczytów i zapisów.

```

Running the test with following options:
Number of threads: 12
Initializing random number generator from current time

Initializing worker threads...

Threads started!

SQL statistics:
  queries performed:
    read:                2840
    write:               1136
    other:               568
    total:              4544
  transactions:         284   (27.27 per sec.)
  queries:              4544  (436.31 per sec.)
  ignored errors:       0     (0.00 per sec.)
  reconnects:           0     (0.00 per sec.)

General statistics:
  total time:           10.4126s
  total number of events: 284

Latency (ms):
  min:                  61.81
  avg:                  432.72
  max:                  752.31
  95th percentile:     634.66
  sum:                  122891.32

Threads fairness:
  events (avg/stddev):  23.6667/0.94
  execution time (avg/stddev): 10.2409/0.11

```

-	MyISAM	InnoDB	Memory	Archive	CSV
średni czas [ms]	432.7	75.5	425.1	powyżej minuty	powyżej minuty

Wyniki testu przedstawione w tabeli powyżej wskazują, że najwydajniejszym silnikiem w przypadku równoległych operacji odczytu i modyfikacji danych w środowisku wielowątkowym okazał się silnik InnoDB, na co wpływ ma zastosowanie mechanizmu blokowania pojedynczych rekordów, zamiast całej tabeli zastosowany w pozostałych.

Wyszukiwanie danych z zakresu.

Kolejny test symuluje operację wyszukiwania danych za pomocą zakresu. Do jego przygotowania wykorzystano następujące polecenie.

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=root --mysql-db=test  
--table_size=2000000 --mysql_storage_engine=<nazwa silnika> --num-threads=12  
/usr/share/sysbench/select_random_ranges.lua prepare
```

W związku z tym, że domyślnym indeksem dla tabeli MEMORY jest indeks HASH, a nie B-tree, na tej tabeli dodatkowo utworzono indeks typu B-Tree, wykorzystując następujące polecenie.

```
CREATE INDEX k_12 on sbtest1(k) USING btree;
```

-	MyISAM	InnoDB	Memory	Archive	CSV
średni czas [ms]	1.23	1.25	0.51	26687	51400

Wyniki testu pokazują, że przy tego typu operacjach najwydajniejsze są silniki wykorzystujące indeksy typu B-Tree, które pozwalają w optymalny sposób wyszukiwać za pomocą zakresu. Silnik *MEMORY* okazał się najszybszy ze względu na przechowywanie danych bezpośrednio w pamięci.

Operacje zapisu.

Kolejny test prezentuje wydajność operacji zapisu w środowisku wielowątkowym. Do jego wykonania użyto następującego polecenia:

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=root --mysql-db=test  
--table_size=2000000 --mysql_storage_engine=<nazwa silnika> --num-threads=12  
/usr/share/sysbench/oltp_insert.lua prepare
```

Wyniki testu przedstawiono w poniższej tabeli:

-	MyISAM	InnoDB	Memory	Archive	CSV
średni czas [ms]	107.8	37.9	104.1	17.1	108.9

W teście najszybszy okazał się silnik *Archive*, który został zaprojektowany właśnie do wydajnego zapisywania danych. Znaczny wpływ na dużą wydajność ma fakt braku indeksów. Dzięki temu serwer nie spędza dodatkowego czasu na aktualizacji indeksu podczas wstawiania nowych krotek. Istotna jest również niemal trzykrotnie większej wydajność operacji zapisu w tabelach *InnoDB* w porównaniu do *MyISAM* i *MEMORY*. Jest to wynikiem zastosowania blokowania na poziomie rekordu.

Operacje aktualizacji danych z wykorzystaniem indeksu.

Kolejny test przedstawia wydajność operacji aktualizacji danych z wykorzystaniem indeksu w środowisku wielowątkowym. Do jego przygotowania użyto następującego polecenia.

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=root --mysql-db=test
--table_size=2000000 --num-threads=12 --mysql-storage_engine=
<nazwa silnika> /usr/share/sysbench/oltp_update_index.lua prepare
```

Wyniki testu zamieszczono w poniższej tabeli.

-	MyISAM	InnoDB	Memory	Archive	CSV
średni czas [ms]	109.1	54.9	105.9	Brak wsparcia	Brak indeksów

Test jest miarodajny jedynie dla trzech silników wspierających indeksy. Podstawową przyczyną powodującą niemal dwukrotnie większą wydajność operacji modyfikujących dane w silniku *InnoDB*, jest mechanizm blokowania na poziomie rekordów.

Operacje aktualizacji danych bez wykorzystania indeksów.

Kolejny test również przedstawia wydajność operacji aktualizacji danych w środowisku wielowątkowym, ale bez wykorzystania indeksów. Do wykonania testu użyto polecenia:

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=root --mysql-db=test
--table_size=2000000 --num-threads=12 --mysql-storage_engine=
<nazwa silnika> /usr/share/sysbench/oltp_update_non_index.lua prepare
```

-	MyISAM	InnoDB	Memory	Archive	CSV
średni czas [ms]	107.6	46.9	107.1	Brak wsparcia	67627.1

Testy potwierdzają, że w środowisku wielowątkowym przy operacjach modyfikujących dane najwydajniejszy jest silnik *InnoDB*, dzięki blokowaniu na poziomie rekordów.

Operacje usuwania danych

W kolejnym teście sprawdzono wydajność operacji usuwania danych w środowisku wie

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=root --mysql-db=test
--table_size=2000000 --num-threads=12 --mysql-storage_engine=<nazwa tabeli>
/usr/share/sysbench/oltp_delete.lua prepare
```

-	MyISAM	InnoDB	Memory	Archive	CSV
średni czas [ms]	312.5	43.3	106.1	Brak wsparcia	62342.1

Silnik InnoDB jest najwydajniejszym rozwiązaniem ze względu na mechanizm blokowania na poziomie rekordów.

Podsumowanie

W tym rozdziale przedstawiono podstawowe własności najpopularniejszych dostępnych obecnie silników MySQL. W zdecydowanej większości przypadków, które możemy spotkać we współczesnych aplikacjach, korzystających z baz danych najlepszym rozwiązaniem jest silnik InnoDB. W przypadku tabel, do których dane są jedynie zapisywane, ale nie odczytywane, dobrym wyborem jest silnik *Archive*, który zapewnia najlepszą wydajność operacji zapisu, oraz zdecydowanie niższe zużycie przestrzeni dyskowej od pozostałych silników. Wyniki testów wydajnościowych potwierdzają, że silnik *MyISAM* nie powinien być obecnie stosowany. Silnik *InnoDB* będący jego następcą jest bardziej wydajny, a dodatkowo zawiera mechanizmy, których brakuje *MyISAM*. Silnik *MEMORY* w środowisku wielowątkowym nie jest bardziej wydajny od *InnoDB*, a dodatkowo zużywa wiele pamięci. Silnik *CSV* jest niewydajny i ubogi w funkcje, dlatego według autora nie powinien być używany w produkcyjnych zastosowaniach.

5 Optymalizator MySQL

Zasadniczo każde zapytanie SQL skierowane do bazy danych MySQL może zostać zrealizowane na wiele różnych sposobów. Optymalizator jest częścią 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 (nazywane również *drzewem 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 są np. jednoznaczne. Na kolejnym etapie weryfikowane są uprawnienia. Czynność ta może zajmować szczególnie dużo czasu, jeżeli serwer posiada bardzo dużą 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. Oznacza to, 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. W efekcie 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 zapytania.

5.1 Możliwości optymalizatora

W tym podrozdziale przedstawiono kilka przykładowych optymalizacji, które może wykonać moduł serwer MySQL.

- **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.

Rysunek 20: Pliki silnika InnoDB testowej bazy danych *StackOverflow*

#	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

- **Zamiana OUTER JOIN na INNER JOIN.** Nawet jeżeli w zapytaniu użyjemy klauzuli OUTER JOIN, złączenie 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 rysunku 20 widzimy, że kolumna *ref* dla pierwszego wiersza jest wartość "const", czyli najmniejsza wartość id z tabeli *Users* została potraktowana jako stała.

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

- **Optymalizacja funkcji COUNT().** Wyniki 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 taka, 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.

5.2 Dane statystyczne dla optymalizatora

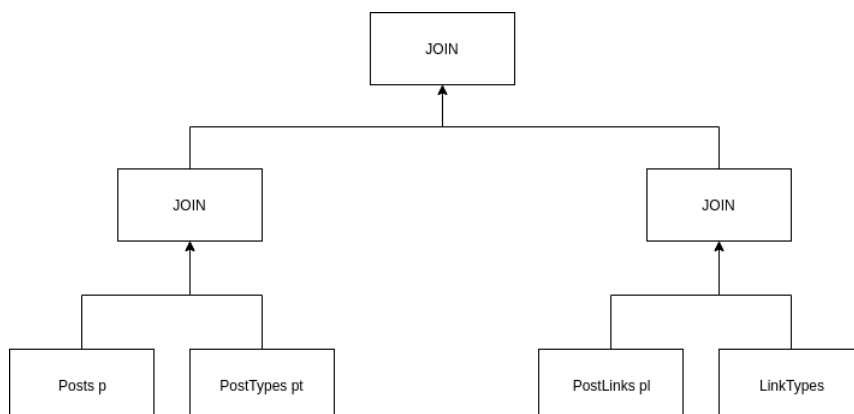
Przechowywaniem danych statystycznych jest zadaniem silników bazy danych. Z tego powodu w zależności od użytego silnika przechowywane wartości statystyczne mogą być różne. Przykładowo silnik MyISM przechowuje informacje o aktualnej liczbie rekordów w tabeli, a InnoDB takiej informacji nie przechowuje,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

5.3 Plan wykonania zapytania

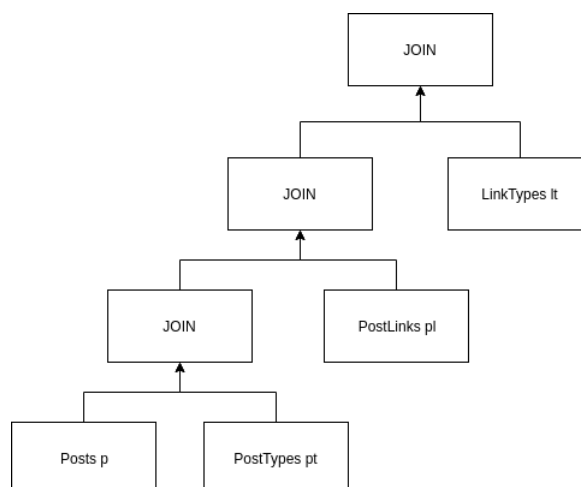
Wynikiem optymalizacji jest plan wykonania zapytania. Jest on zapisywany w postaci drzewa instrukcji, które kolejno wykonywane doprowadzą do zwrócenia wyniku.

```
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;
```

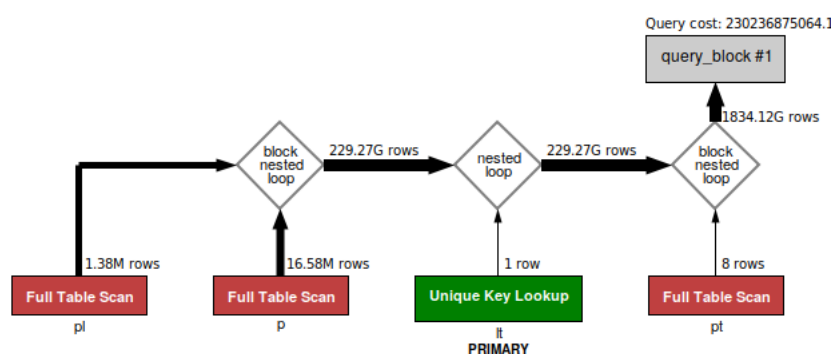
Sposób łączenia tabel moglibyśmy przedstawić jak na poniższym schemacie.



W rzeczywistości drzewo instrukcji przybiera postać drzewa lewostronnie zagnieżdżonego. Cechą charakterystyczną tych drzew jest to, że sekwencje operacji łączenia są wykonywane w sposób potokowy. Przykład takiego drzewa jest przedstawiony na poniższym rysunku.



Analizując wyniki polecenia EXPLAIN dla powyższego zapytania, używając aplikacji *MySQL Workbench* możliwe jest wyświetlenie graficznej postaci odpowiadającej drzewu instrukcji.



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

5.4 Optymalizator złączeń

Większość operacji złączeń można wykonać na różne sposoby, uzyskując ten sam wynik. Zamiana kolejności jest bardzo skuteczną formą optymalizacji zapytań. W tym podrozdziale rozważono następujące 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	<small>NULL</small>	const	PRIMARY	PRIMARY	4	const	1	100.00	Using index
2	1	SIMPLE	pt	<small>NULL</small>	ALL	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	8	100.00	<small>NULL</small>
3	1	SIMPLE	p	<small>NULL</small>	ref	owner_idx	owner_idx	5	const	182	10.00	Using where
4	1	SIMPLE	c	<small>NULL</small>	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	<small>NULL</small>	const	PRIMARY	PRIMARY	4	const	1	100.00	Using index
2	1	SIMPLE	p	<small>NULL</small>	ref	owner_idx	owner_idx	5	const	182	100.00	<small>NULL</small>
3	1	SIMPLE	c	<small>NULL</small>	ref	user_post_idx	user_post_idx	5	const	322	100.00	Using index
4	1	SIMPLE	pt	<small>NULL</small>	ALL	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	<small>NULL</small>	8	12.50	Using where

Wyniki polecenia są niemalże identyczne, jedyną różnicą jest kolejność dokonywanych złączeń. Koszt ostatniego zapytania można sprawdzić wykonując następujące polecenie:

```
SHOW STATUS LIKE 'Last_Query_Cost';
```

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 włączono profilowanie zapytań za pomocą komendy:

```
SET PROFILING = 1;
```

Wykonano 10 zapytań dla każdego wariantu i policzono średni czas, jaki serwer MySQL spędzał na etapie "executing", czyli etapie faktycznego wykonywania zapytania. Dla zapytania z kolejnością wybraną przez optymalizator średnia wartość wyniosła 0.04 sekundy, natomiast przy wymuszonej kolejności łączy w zapytaniu wartość ta wyniosła 0.28 sekundy.

Powyższy eksperyment pokazuje, że zamiana kolejności złączeń jest bardzo skuteczną formą optymalizacji i może prowadzić do wielokrotnego zmniejszenia kosztu zapytania. Oczywiście nadal może się zdarzyć sytuacja, kiedy zapytanie z wymuszoną kolejnością złączeń będzie wydajniejsze, ponieważ optymalizator MySQL nie zawsze może sprawdzić wszystkie potencjalne kombinacje złączeń, ale w zdecydowanej większości przypadków optymalizator złączeń okazuje się skuteczniejszy od człowieka.

5.5 Konfiguracja optymalizatora złączeń

Optymalizator złączeń stara się wygenerować plan zapytań o najniższym możliwym koszcie. W idealnym przypadku optymalizator zweryfikowałby wszystkie możliwe 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 określenia, 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* przygotowano 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 wykonano 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 zapisano średni czas wykonania polecenia EXPLAIN. Wyniki zamieszczono 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

Wraz ze wzrostem wartości parametru wzrastał czas wykonywania polecenia SQL. W kolejnym kroku włączono profilowania i sprawdzono, na który z etapów serwer spędził najwięcej czasu. Wyniki posortowano według czasu. Na poniższym zrzucie ekranu umieszczono profil zapytania dla wartości 62.

#	Status	Duration
1	statistics	36.301148
2	executing	0.000234

Profilowanie zapytania wskazuje, że przez większość czasu zapytania, serwer zbierał dane, które pozwolą mu wybrać optymalny plan wykonania zapytania. Okazało się, że dla pewnych zapytań, próba wybrania optymalnego rozwiązania może skończyć się gigantycznym wydłużeniem czasu zapytania. Co ciekawe nawet dla wartości 0 optymalizacja okazała 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, wykorzystano jeszcze raz zapytanie do wygenerowanej tabeli *Students*. Najpierw ustawiono wartość parametru *optimizer_search_depth*=0, a następnie *optimizer_prune_level*=0. Z poprzed-

niego 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 dłużej. Następnie ustawiono wartość `optimizer_search_depth=62` i ponownie zmierzono 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 zalecane jest pozostawienie domyślnej wartości `optimizer_prune`, chyba że mamy pewności pominięcia optymalnego planu zapytania.

5.6 Konfiguracja statystyk tabel InnoDB

Dla tabeli InnoDB zbierane są dwa rodzaje statystyk: trwałe i nietrwałe. Trwałe zapisywane są na dysku twardym i przechowywane są pomiędzy restartami serwera, nietrwałe (ulotne) są usuwane po każdym restarcie.

Trwałe statystyki

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, należy 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 wywołujemy 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 dla konkretnej tabeli poprzez parametr `innodb_stats_persistent_sample_pages` lub parametr `stats_sample_pages`. Zmianę wartości parametru powinna zostać rozpatrzone w następujących przypadkach:

- **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.

Sprawdzono 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 przygotowano cztery zapytania. Dwa dla *owner_idx*:

```
SELECT count(DISTINCT OwnerUserId) from Posts; #liczba unikalnych wartości
indeksu
SELECT stat_value FROM mysql.innodb_index_stats WHERE database_name=
'stackOverflowMedium' AND table_name = 'Posts' AND index_name = 'owner_idx'
AND
stat_name = 'n_diff_pfx01';
```

Oraz dwa dla *favorite_idx*:

```
SELECT count(DISTINCT FavoriteCount, Score) from Posts;
SELECT stat_value FROM mysql.innodb_index_stats WHERE database_name=
'stackOverflowMedium'
AND table_name = 'Posts' AND index_name = 'favorite_idx' AND
stat_name = 'n_diff_pfx01';
```

W poniższej tabeli zamieszczono 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	

Wraz ze wzrostem liczby stron użytych do analizy wzrasta dokładność statystyk, ale rośnie czas przeprowadzania analizy tabeli. 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 będzie zwiększenie wartości parametru.

- **Zbyt długi czas zbierania statystyk dla tabeli**

Analizując wyniki tabeli można stwierdzić, 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 rozpoczyna się od wyjaśnienia terminów i teorii, które będą przydatne podczas dalszego rozważania zagadnień wydajności, skalowalności i wysokiej dostępności MySQL.

6.1 Terminologia

Skalowalność a wydajność

Celem tego podrozdziału 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: ilości jednoczesnych połączeń do bazy danych, liczbie zapytań wykonywanych na sekundę lub rozmiar odczytywanych danych. 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.

Możliwy jest system, który jest bardzo wydajny, ale bardzo słabo skalowany. Możliwy jest także system niewydajny, który jest bardzo dobrze skalowany, dlatego istotnym jest, aby nie mylić tych pojęć.

Teoria CAP

Autorem teorii CAP jest Eric Brewer, który przypisał bazą 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 nieistnienie idealnej bazy danych i każda z nich jest pewnym kompromisem, kładącym nacisk na pewne własności, kosztem innych.

Skalowanie wertykalne i horyzontalne

Skalowaniem wertykalnym nazywamy zwiększenie wydajności w ramach pojedynczego serwera. Zwiększenie wydajności polega na dodawaniu kolejnych zasobów takich jak pamięć czy procesor. Takie rozwiązanie jest skuteczne tylko do pewnego momentu. Wraz z wzrostem wydajności, rozwiązanie to będzie się wiązało z dużym wzrostem kosztów, nawet przy małym wzroście możliwości serwera. Ostatecznie osiągniemy limit możliwości jednej maszyny (serwera) 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 zapytaniach, które mocno obciążają procesor maszyny. Z powodu wyżej wymienionych ograniczeń skalowania wertykalnego, nie nadaje się ono, jeżeli aplikacja ma zapewnić wysoką skalowalność.

W modelu skalowalności horyzontalnej zwiększenie wydajności odbywa się przez dodanie kolejnych serwerów, przy zachowaniu wydajności pojedynczych serwerów. W kolejnych podrozdziałach zostaną przedstawione różne sposoby realizacji modelu skalowania horyzontalnego. 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 serwerami (węzłami).

6.2 Architektura master-slave

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*). Operacje modyfikujące dane wykonywane są na serwerze nadrzędnym, a operacje odczytu mogą być wykonywane na każdym z serwerów. Najczęściej operacje odczytu wykonywane są tylko na serwerach nadrzędnych celem odciążenia serwera *master*. Replikacja danych pomiędzy serwerem nadrzędnym, a podrzędnymi opiera się na następującej zasadzie. Serwer nadrzędny obsługuje wszystkie operacje, które modyfikują dane (zapis, odczyt, aktualizacja), równocześnie rejestrując każdą operację, która wykonał. Następnie każdy z serwerów podrzędnych odczytuje rejestr operacji i aktualizuje swój stan. Efektem tej pracy jest wiele instancji bazy danych zawierających identyczne dane. 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 wyda-

ność 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. Co więcej, operację *backupu* danych można wykonać na serwerze *slave*, co dodatkowo odciąża serwer główny. 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 dane w taki sposób, że będą one niespójne, ponieważ funkcja `RAND()` może zwrócić różne wyniki na różnych serwerach.
- **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. Jego wadą w stosunku do metody SBR jest mniejsza szybkość oraz większa ilość danych przesyłanych pomiędzy serwerem nadrzędnym i podrzędnymi.
- **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 wszystkich danych w każdym z węzłów, co może skutkować bardzo wysokimi kosztami finansowymi utrzymania serwerów, a nawet być niemożliwe, jeżeli rozmiar danych przekracza możliwości pojedynczego węzła.

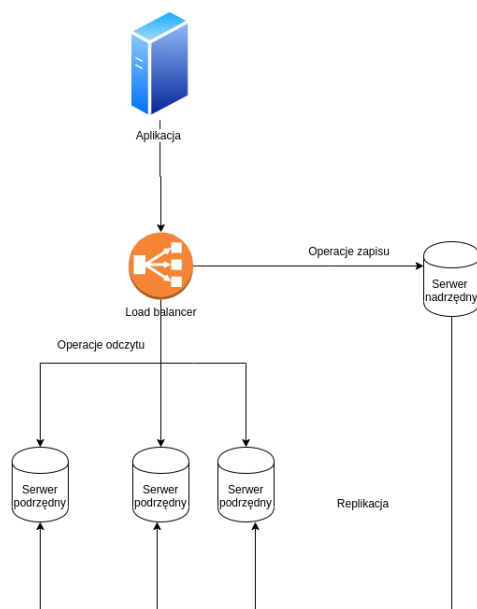
Kolejnym przypadkiem, którego nie rozwiązuje mechanizm replikacji jest system, w którym większość operacji to operacje modyfikujące dane. W takim scenariuszu serwer nadrzędny stanie się wąskim gardłem, a dodatkowo dużą część obciążenia serwerów będzie stanowiła replikacja pomiędzy serwerem nadrzędnym i serwerami podrzędnymi,

co dodatkowo zmniejszy wydajność operacji zapisów na serwerze nadrzędnym, który część zasobów będzie przeznaczał na przygotowanie plików replikacji.

Reasumując skalowanie horyzontalne za pomocą mechanizmu replikacji *master-slave* idealnie sprawdzi się, gdy zdecydowaną większością operacji są operacje odczytu, a rozmiar danych jest ograniczony.

Skalowanie z zastosowaniem replikacji *master-slave* jest często stosowane razem z *load balancerem* (przykładowo ProxySQL), który odpowiada za balansowanie obciążenia operacji odczytu pomiędzy serwerami podrzędnymi oraz zapewnia przeźroczystość dostępu do danych z punktu widzenia aplikacji, która ma wrażenie komunikowania się z tylko jednym serwerem. Przykład takiej architektura przedstawiono na poniższym diagramie.

Rysunek 21: Przykładowa architektura master-slave z load balancerem



6.3 Architektura multi-master

Multi-master, to architektura w której kilka serwerów pełni rolę serwera nadrzędnego (*master*). W takim modelu każdy z serwerów przechowuje pełny zestaw danych oraz może modyfikować je w dowolnym momencie, które następnie są propagowane do pozostałych serwerów. Węzły odpowiadają także za rozwiązywanie potencjalnych konfliktów, które powstały w wyniku równoległych zmian na kilku instancjach. W MySQL realizacją takiej architektury jest replikacja grupową. Replikacja grupowa zapewnia

spójność danych pomiędzy serwerami oraz dostępność nawet w przypadku awarii jednego z węzłów. Co istotne, jeżeli jeden z węzłów będzie nieaktywny, klienci z aktywnym połączeniem do tego serwera muszą zostać przełączeni na inny aktywny serwer. Replikacja grupowa nie posiada takiego mechanizmu, dlatego przełączanie ruchu pomiędzy serwerami master powinno być obsługiwane na poziomie aplikacji, a najlepiej poprzez umieszczenie *load balancera* pomiędzy aplikacją i serwerami należącymi do grupy.

Wykonanie zmiany w danych na jednym z serwerów, informacja o zmianie przesyłana jest do pozostałych węzłów w postaci plików *binlog*.

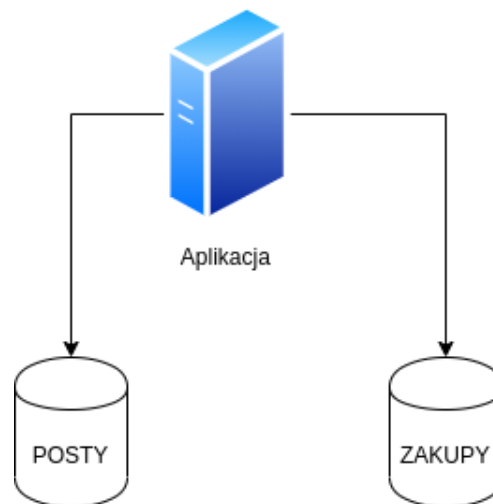
Replikacja grupowa jest modelem replikacji optymistycznej. W tym podejściu zakłada się, że większość operacji modyfikacji danych nie będzie powodowała problemów w spójności danych, dlatego nie stosuje się blokad w dostępie do danych, które nie zostały jeszcze zatwierdzone przez wszystkie serwery. Takie założenie może prowadzić do sytuacji, kiedy klient odpytujący dwa różne serwery może otrzymać różne wyniki (brak silnej spójności danych), ale współbieżne modyfikacje są kosztem, który został poniesiony w celu osiągnięcia większej wydajności i skalowalności grupy. W przypadku wystąpienia konfliktów pomiędzy transakcjami, potencjalne konflikty rozwiązywane są po zmodyfikowaniu danych. Z tego powodu ważne jest zachowanie dyscypliny po stronie aplikacji, żeby transakcje operujące na tych samych wierszach w miarę możliwości wykonywane były w ramach pojedynczej transakcji. Istotnym problemem może być wykorzystanie autoinkrementowanych kluczy głównych. Jeżeli dwa lub więcej serwerów będzie przydzielało klucze główne w dokładnie taki sam sposób, doprowadzi to do naruszenia kluczy podstawowych, dlatego bardzo ważne jest uważne ustawienie strategii generowania autoinkrementowanych kluczy podstawowych.

Architektura *multi-master* będzie szczególnie przydatna, kiedy potrzebujemy zapewnić skalowalność operacji zapisu. Dodatkowo realizacja w postaci grupy serwerów zapewnia wygodne skalowanie poprzez elastyczne dodawanie lub odejmowanie węzłów, które zostaną automatycznie dołączone lub odłączone w sposób przeźroczysty dla aplikacji. Jedną z głównych wad realizacji architektury *master-slave* w MySQL jest brak silnej spójności danych, który jednak w zdecydowanej większości sytuacji nie powinien stanowić istotnego problemu. Do wad z pewnością należy zaliczyć również większą złożoność systemu oraz bardziej skomplikowany proces konfiguracji w porównaniu do architektury *master-slave*.

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* umożli-

liwia odczytywania postów innych użytkowników oraz dokonywanie zakupów w sekcji *Marketplace*. Jeżeli użytkownik przegląda aktualne oferty w dziale *Marketplace*, to zapytania kierowane do bazy danych, będą odpytywać jedynie kilka tabel związanych z zakupami. Jeżeli w tym samym czasie inny użytkownik przegląda posty użytkowników, zapytania nie będą dotyczyć tabel związanych z zakupami. Oczywiście przykład, który przedstawiłem powyżej, może być 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 obsługiwać kilka źródeł danych.

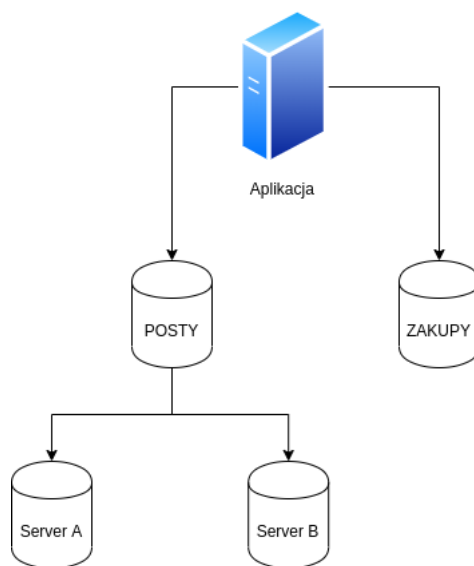


6.5 Data-sharding

Data-sharding wykorzystuje fakt, że rekordy w ramach pojedynczej tabeli są niezależne od siebie. Jako przykład przeanalizowano tabelę *Comments* z testowej bazy *StackOverflow*. Wiersze w tabeli *Comments* są logicznie powiązane z wierszami w tabeli *Posts*, ale pomiędzy poszczególnymi wierszami w tabeli *Comments* należącymi do innych postów, nie ma relacji. W naszej testowej bazie danych przechowujemy 25 milionów komentarzy. Z obsługą takiej ilości danych, baza danych nie powinna mieć problemów. Niestety z czasem tabela zawierająca komentarze może rozrosnąć się do rozmiarów, których ob-

służenie w pojedynczej bazie danych może być problematyczne. Rozwiązaniem tego problemu może być podział pojedynczej tabeli na kilka serwerów. Przykładowo posty wraz z komentarzami o parzystym *PostId* przechowywać na serwerze A, a nieparzyste na serwerze B. Dzięki temu, dwukrotnie zmniejszyliśmy liczbę zapytań do pojedynczej bazy danych, ilość wymaganego miejsca do przechowania postów i komentarzy, rozmiary buforów i indeksów. Wadą takiego rozwiązania jest konieczność obsługi sterowania zapytań do konkretnej bazy na poziomie aplikacji oraz wzrost złożoności logiki aplikacji i systemu bazy danych. Przykładowo chcąc pobrać najnowsze dziesięć postów, musimy wykonać dwa osobne zapytania, pobrać część redundantnych danych i na poziomie aplikacji dokonać wyboru dziesięciu najnowszych postów.

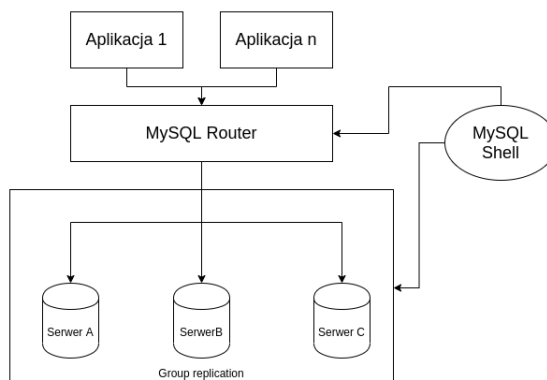
Głównymi przesłankami skłaniającymi do zastosowania tego sposobu skalowalności jest konieczność przechowywania danych w rozmiarach przekraczających możliwości jednego serwera lub skalowanie operacji zapisów, którego nie da się dokonać w ramach replikacji z jednym serwerem nadrzędnym. Sharding danych bardzo dobrze sprawdza się przy jednoczesnym zastosowaniu partycjonowania funkcjonalnego. Przykład zastosowania *data-sharding* wraz z partycjonowaniem funkcjonalnym przedstawiono na poniższym schemacie.



Rysunek 22: Przykładowa architektura partycjonowania funkcjonalnego

6.6 InnoDB Cluster

InnoDB Cluster jest dostępną natywnie kombinacją kilku technologii umożliwiających stworzenie wysoko dostępnej bazy danych w klastrze. Składa się z trzech podstawowych



Rysunek 23: Architektura InnoDB Cluster

elementów:

- **Group replication** czyli rozwiązania, które zostało opisane w poprzednich rozdziałach (architektura multi-master oraz master-slave).
- **MySQL Shell**, który jest klientem umożliwiającym zarządzanie serwerami MySQL za pomocą języków *Java Script*, *Python* lub SQL. Za jego pomocą możemy konfigurować klastre, między innymi poprzez dodawanie lub usuwanie węzłów, tworzenie nowych instancji serwerów, a nawet modyfikowanie danych na poszczególnych serwerach.
- **MySQL Router** będący punktem styku pomiędzy aplikacjami, a instancjami serwerów w klastrze. Router odpowiada za kierowanie ruchem do poszczególnych węzłów jednocześnie uniezależnia klientów od zmian wewnątrz klastra.

Rozwiązanie InnoDB Cluster posiada wszystkie ograniczenia replikacji grupowej przedstawionej w poprzednich sekcjach. Zaletą tego rozwiązania jest wygodniejsza konfiguracja, dzięki możliwości wykorzystania *MySQL shell* w którym możemy administrować naszym klastrem nawet wykorzystując takie języki jak *Java script* czy *Python*. Ważne jest to, że MySQL InnoDB Cluster może działać tylko z tabelami InnoDB, dlatego, żeby użyć klastra należy upewnić się, że wszystkie tabele spełniają ten wymóg.

6.7 Podsumowanie

W tym rozdziale przedstawiono kilka architektur możliwych do zrealizowania w MySQL. Przedstawione rozwiązania pokazują, że rozwiązanie wielu zagadnień związanych

z optymalizacją może być rozwiązane na poziomie tworzenia architektury bazy danych. Poprzez zastosowanie odpowiedniej architektury możliwe jest zniwelowanie problemów, które występują przy wykorzystaniu pojedynczego serwera bazy danych.

7 Partycjonowanie Tabel

MySQL wspiera partycjonowanie horyzontalne, które polega na dzieleniu tabeli na zbiór mniejszych fizycznych partycji w których przechowywane są dane. W rzeczywistości każda z partycji jest osobną tabelą zawierającą zarówno dane, jak i indeksy, ale z punktu widzenia klienta serwera MySQL, wszystkie partycje widoczne są jako pojedyncza tabela. Zasada działania procesu partycjonowania jest w pewien sposób podobna do indeksowania; oba podejścia służą do wskazania przybliżonego miejsca składowania danych, co ogranicza ilość danych, do których wymagany jest dostęp. Poniżej wymienię podstawowe właściwości tabel partycjonowanych:

- rekord danych może być przechowywany w tylko jednej partycji.
- nie ma możliwości przenoszenia danych pomiędzy partycjami, zmieniając wartość kolumny, która jest używana przez funkcję partycjonowania. W takim przypadku należy usunąć dane z jednej partycji i wstawić do drugiej.
- partycjonowane dane mogą być fizycznie rozproszone.
- wszystkie partycje muszą używać jednakowego silnika.
- partycja może należeć tylko do jednej tabeli.
- tabele partycjonowane nie działają z kluczami zewnętrznymi.
- tabele partycjonowane nie wspiera wyszukiwania pełnotekstowego.

7.1 Metody partycjonowania

MySQL używa funkcji partycjonowania w celu ustalenia partycji, do której powinien zostać wstawiony rekord. W tym podrozdziale przedstawię kilka metod partycjonowania wspieranych przez MySQL.

Range partitioning

Polega na podziale danych na podstawie pewnych wartości zakresów. Przykładowo użytkownicy o wzroście do 160 cm trafiają do partycji x1, użytkownicy o wzroście 160-180 cm do partycji x2, a użytkownicy o wzroście powyżej 180 cm do partycji x3. Żeby przedstawić wykorzystanie tabel partycjonowanych zakresowe, przygotowano tabele *Votes_partitioned* będącą kopią tabeli *Votes*, a następnie dokonano partycjonowania tabeli.

Rysunek 24: Pliki z partycjami tabeli *Votes_Partitioned*

```

root@franek-ThinkPad-T440p:/var/lib/mysql/stackoverflowMedium# ls -lh | grep Votes_partitioned
-rw-r----- 1 mysql mysql 112K lip 19 23:34 Votes_partitioned#p#p10.ibd
-rw-r----- 1 mysql mysql 44M lip 19 23:34 Votes_partitioned#p#p1.ibd
-rw-r----- 1 mysql mysql 164M lip 19 23:34 Votes_partitioned#p#p2.ibd
-rw-r----- 1 mysql mysql 244M lip 19 23:34 Votes_partitioned#p#p3.ibd
-rw-r----- 1 mysql mysql 416M lip 19 23:34 Votes_partitioned#p#p4.ibd
-rw-r----- 1 mysql mysql 636M lip 19 23:34 Votes_partitioned#p#p5.ibd
-rw-r----- 1 mysql mysql 840M lip 19 23:34 Votes_partitioned#p#p6.ibd
-rw-r----- 1 mysql mysql 112K lip 19 23:34 Votes_partitioned#p#p7.ibd
-rw-r----- 1 mysql mysql 112K lip 19 23:34 Votes_partitioned#p#p8.ibd
-rw-r----- 1 mysql mysql 112K lip 19 23:34 Votes_partitioned#p#p9.ibd

```

```
CREATE TABLE Votes_partitioned SELECT * FROM Votes;
```

```

ALTER TABLE Votes_partitioned PARTITION BY RANGE(YEAR (creationDate)) (
PARTITION p1 VALUES LESS THAN(2009),
PARTITION p2 VALUES LESS THAN(2010),
PARTITION p3 VALUES LESS THAN(2011),
PARTITION p4 VALUES LESS THAN(2012),
PARTITION p5 VALUES LESS THAN(2013),
PARTITION p6 VALUES LESS THAN(2014),
PARTITION p7 VALUES LESS THAN(2015),
PARTITION p8 VALUES LESS THAN(2016),
PARTITION p9 VALUES LESS THAN(2017),
PARTITION p10 VALUES LESS THAN(2018)
);

```

Na rysunku 24 przedstawiono, w jaki sposób zapisane są kolejne partycje na dysku twardym.

List partitioning

Ten typ partycjonowania jest bardzo podobny co do zasady do *Range partitioning*. Rozdział dokonywany jest na podstawie przynależności do pewnego zbioru danych. Przykładowo mieszkańcy Polski i Niemiec trafiają do partycji p1, mieszkańcy Włoch, Francji i Hiszpani do partycji p2, natomiast mieszkańcy Wielkiej Brytani i Irlandii do partycji p3. Z racji analogicznej zasady działania, ta metoda sprawdza się dobrze w dokładnie takich samych przypadkach, jak *range partitioning*. Różnicą jest sposób wskazania, w jaki sposób dane dzielone są partycje. W *List partitioning* dane dzielone są na podstawie przynależności do pewnego określonego zbioru elementów, a nie na podstawie przynależności do zakresu.

Rysunek 25:

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	v	NULL	ALL	NULL	NULL	NULL	NULL	52801175	1.11	Using where

Rysunek 26:

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	v	p6	ALL	NULL	NULL	NULL	NULL	18515779	1.11	Using where

Key partitioning

Bardzo podobne do poprzedniej metody, natomiast funkcja mieszająca wybierana jest przez serwer MySQL. Przykładowo dla *NDB Cluster* używana jest funkcja MD5.

Hash partitioning

Partycjonowanie dokonywane jest na podstawie wyniku pewnej funkcji mieszającej (*hash function*) podanej przez użytkownika.

7.2 Przypadki użycia

Założmy, że użytkownik chce pobrać liczbę ocen "Spam" w roku 2013. Pole *CreationDate* nie jest indeksowane. Przygotowano następujące zapytanie dla dwóch tabel przygotowanych wcześniej:

```
SELECT count(id) FROM Votes_partitioned v WHERE v.VoteTypeId = 12 AND v.CreationDate
BETWEEN '2013-01-01' AND '2013-12-31';
SELECT count(id) FROM Votes v WHERE v.VoteTypeId = 12 AND v.CreationDate
BETWEEN '2013-01-01' AND '2013-12-31';
```

W pierwszym kroku porównano wyniki polecenia EXPLAIN dla obu zapytań. Na rysunku 26 przedstawiono wynik dla tabeli partycjonowanej, a na 25 dla niepartycjonowanej. Jak wynika z analizy, już na etapie optymalizacji baza danych wybiera partycje w której będzie wyszukiwał rekordy. Z tego powodu liczba rekordów do przeszukania jest niemal trzykrotnie mniejsza dla tabeli *Votes_partitioned*. Czasy wykonania zapytania dla obu tabel to kolejno: 10.03 sekund dla tabeli partycjonowanej i 17.58 dla niepartycjonowanej.

Kolejnym przypadkiem użycia jest usuwanie dużej ilości danych. Przykładem będzie zapytanie, za pomocą którego użytkownik chce usunąć wszystkie oceny starsze

niż 1 stycznia 2009, czyli de facto dane z partycji *p1*. Przygotowano dwa następujące zapytania:

```
SET SQL_SAFE_UPDATES = 0;  
DELETE FROM Votes v WHERE v.CreationDate < '2009-01-01';  
ALTER TABLE Votes_partitioned DROP partition p1;
```

Usuwanie danych z tabeli *Votes* trwało 60.903 sekundy, natomiast drugie jedynie 0.073 sekundy. Przedstawiona analiza prowadzi do konkluzji, że szczególnie dla bardzo dużych ilości danych usuwanie całych partycji będzie zdecydowanie szybsze niż usuwanie wielu pojedynczych wierszy.

7.3 Podsumowanie

Zdaniem autora partycjonowanie powinno być jednym z ostatnich etapów optymalizacji i być wykonywane głównie dla tabel z ogromnymi ilościami danych, ponieważ szereg ograniczeń wynikających z użycia tabel partycjonowanych, jak i dodatkowy nakład pracy wynikający z konieczności utrzymywania partycji (przykładowo dodawanie kolejnych partycji w kolejnych latach, jeżeli data jest używana do partycjonowania) mogą nie być warte pewnego wzrostu wydajności, który można również uzyskać, stosując pozostałe metody optymalizacji opisane w tej pracy.

8 Indeksy

Indeks jest strukturą danych służącą do zwiększenia wydajności wyszukiwania danych w tabeli. Poprawne stosowanie indeksów jest krytyczne dla zachowania dobrej wydajności bazy danych. Najprostszą i zarazem najpopularniejszą analogią pozwalającą zrozumieć działanie indeksów bazodanowych jest indeks znajdujący się najczęściej na końcu książki, służący do wygodnego wyszukiwania interesujących nas zagadnień. Zakładając, że książka nie zawiera indeksu, wyszukiwanie konkretnego słowa lub tematu w najgorszym wypadku wymaga przewertowania wszystkich stron. Z tego powodu w książkach stosuje się indeksy, które zawierają kluczowe słowa użyte w książce. Indeks taki zawiera listę słów w kolejności alfabetycznej oraz stron, na których one 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 jest wydajne nawet przy niestosowaniu indeksów. Indeksy stają się jednak kluczowe wraz ze wzrostem rozmiaru 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 omówione zostaną najpopularniejsze z nich.

8.1 Indeksy typu B-Tree

Indeks typu B-Tree jest zdecydowanie najczęściej stosowanym typem indeksu w bazach MySQL i jest domyślnie wybierany przez serwer MySQL podczas tworzenia nowego indeksu. Dlatego właśnie jemu poświęcono zdecydowaną większość tego rozdziału.

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 $O(\log_M n)$. 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. Przykładowo 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 liczbę 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 przygotowano 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);
```

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

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

Założmy, że celem jest wyszukanie wszystkich komentarzy użytkownika o id 1200 do postu o id 910331 w tabeli *Comments*.

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


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

Zapytanie zwraca wyniki w 13,7 sekundy.

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

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

Dla drugiego zapytania baza danych zwraca wyniki w 0,013 sekundy. Wykorzystując polecenie EXPLAIN dla obu zapytań, otrzymano wyniki przedstawione na rysunkach 27 oraz 28. Rysunek 27 przedstawia wynik polecenia EXPLAIN dla pierwszego zapytania, natomiast Rysunek 28 wynik polecenia EXPLAIN dla drugiego zapytania. Z danych wynika, że pierwsze zapytanie nie będzie korzystać z indeksów, dlatego wartość w kolumnie *Rows* wskazuje, że serwer MySQL dokona skanowania wszystkich wierszy tabeli *init_Comments*. Drugie zapytanie korzysta z indeksu z utworzonego indeksu. W tym przypadku serwer musi przeskanować jedynie 3 wiersze tabeli *Comments*. Różnica w czasie wykonania obu zapytań wynika z użycia indeksu w drugim zapytaniu.

#	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 27: 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 28: EXPLAIN 2

Dopasowanie prefiksu znajdującego się najbardziej na lewo

Dopasowanie prefiksu znajdującego się najbardziej na lewo ma miejsce wtedy, kiedy wyszukiwanie bazuje na pierwszych kolumnach indeksu. Dopasowanie prefiksu znajdującego się najbardziej na lewo może pomóc w wyszukaniu wszystkich komentarzy użytkownika. Założono, że celem jest znalezienie wszystkich komentarzy użytkownika o id 1200. W tym celu przygotowano dwa zapytania. Pierwsze na tabeli *Init_Comments*, drugie na tabeli *Comments* zawierającej indeks typu B-Tree, który założono 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. Analizując wyniki polecenia EXPLAIN dla obu zapytań można stwierdzić, że w pierwszym zapytaniu serwer przeszukuje wszystkie wiersze w tabeli. Drugie zapytanie wymaga 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	<i>NULL</i>	ALL	<i>NULL</i>	<i>NULL</i>	<i>NULL</i>	<i>NULL</i>	22883256	10.00	Using where

Rysunek 29: EXPLAIN 3

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

Rysunek 30: EXPLAIN 4

Dopasowanie zakresu wartości

Dopasowanie zakresu wartości oznacza wyszukiwanie wartości w danym przedziale. W tabeli *StackOverflow* jest to przykładowo wyszukiwanie wszystkich komentarzy użytkowników o identyfikatorach z przedziału od 1990 do 2000.

Ponownie wykonano 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 przeanalizowano wynik polecenia EXPLAIN dla obu zapytań. Przy pierwszym zapytaniu MySQL przeskanował całą tabelę Init_Comments. Drugie natomiast wymagało przeskanowania jedynie wierszy, które zostały zwrócone jako rezultat zapytania. Efektywne wyszukiwanie za pomocą zakresu jest możliwe dzięki temu, że dane w indeksie przechowywane są w kolejności względem id użytkownika.

#	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 31: 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 32: EXPLAIN 6

Zapytania dotyczące jedynie indeksów

Taki indeks często nazywany jest indeksem pokrywającym. Zapytania dotyczące jedynie indeksów to zapytania, które wykorzystują jedynie wartości indeksu, więc nie muszą odwoływać się do rekordów bazy danych. Dzięki temu baza danych nie musi pobierać rekordów z dysku, a może pobrać je z indeksu.

Podsumowanie

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 nieprzydany 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 indeksy jest zdecydowanie bardziej wydajne.

8.2 Indeksy typu hash

Indeksy typu hash są dostępne jedynie dla tabel silnika *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 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. Stąd wynika, że największą zaletą indeksów typu *hash* w stosunku do indeksów *B-Tree* jest czas wyszukiwania dowolnego wiersza w tabeli jest stały niezależnie od liczby wierszy.

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.

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 . Dodatkowo z racji ograniczonego zbioru wartości funkcji hashującej, mogą występować problemy ze skalowaniem w przypadku dużych zbiorów danych. Po przekroczeniu pewnej liczby wierszy należy zwiększyć rozmiar klucza indeksu i ponownie obliczyć funkcję dla wszystkich wierszy w tabeli.

Indeksy typu SPATIAL

W wersji 8.0 MySQL wprowadził wsparcie dla indeksów przestrzennych nazywanych *SPATIAL INDEX* i bazujących na strukturze R-Tree. Sktuktura R-Tree jest rozwinięciem idei B-drzewa na większą liczbę wymiarów. Podobnie jak w B-drzewie operacja wyszukiwania danych jest operacją o złożoności asymptotycznej $O(\log M n)$, gdzie M jest rzędem drzewa. Poniżej przedstawiono przykład tworzenia tabeli zawierającej dane geograficzne oraz indeksu przestrzennego na jednej z kolumn. W pierwszym kroku stworzono tabelę, wykorzystując następujące polecenie:

```
CREATE TABLE shops (  
  location GEOMETRY NOT NULL SRID 4326,  
  name VARCHAR(32) NOT NULL);
```

Aby utworzyć indeks na kolumnie, musi być ona oznaczona jako *NOT NULL* i wskazany zostać układ współrzędnych, w naszym przypadku WGS 84 (EPSG:4326) identyfikujący punkty na podstawie szerokości i długości geograficznej. Następnie na kolumnie *location* stworzono indeks przestrzenny.

```
CREATE SPATIAL INDEX location_idx ON shops (location);
```

Chcąc utworzyć indeks na kolumnie, która nie ma zdefiniowanego układu współrzędnych, serwer utworzy go, ale użytkownik dostanie ostrzeżenie, że indeks nie będzie nigdy używany. Indeksy możemy zakładać nie tylko na punkty; możemy użyć innych geometrii, między innymi: linii, wielokątów czy kolekcji punktów. Ten podrozdział nie będzie przedstawiał szczegółowych zastosowań indeksów przestrzennych, autor chciał jedynie pokazać, że jest to bardzo ciekawe udogodnienie w przypadku używania danych geograficznych w bazie danych.

9 Praktyczne problemy

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

9.1 Sortowanie wyników

Założmy, że chcemy uzyskać wyniki zapytania posortowane według pewnej kolejności. Jest to oczywiście pewien dodatkowy nakład, który serwer MySQL musi wykonać podczas wykonania zapytania.

Podstawą optymalizacji sortowania jest używanie indeksów typu B-Tree, ponieważ indeks jest posortowany względem jego kolumn. Aby przedstawić działanie indeksu na rzeczywistych przykładach, użyto bazy StackOverflow. Z bazy usunięto wszystkie indeksy oraz klucze główne założone na wykorzystywanych w przykładach tabelach, aby nie wpływały one na prezentowane przykłady.

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

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

Następnie wykonano następujące zapytanie.

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

Dla tego zapytania polecenie EXPLAIN zwróci w kolumnie EXTRA informację: "Using index", co oznacza, że do sortowania wartości użyto indeksu znajdującego się w kolumnie key, czyli indeksu, który został stworzony przed wykonaniem zapytania.

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Users	<i>NULL</i>	index	<i>NULL</i>	Rank_idx	8	<i>NULL</i>	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. W kolejnym kroku sprawdzono, co się stanie, jeżeli do klauzuli WHERE zostanie dodana kolejna kolumna.

```
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

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.

Następnie dodano klucz główny dla tabeli Users i sprawdzono, co się stanie, jeżeli zostanie umieszczony jako jedna z kolumn wyniku 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. Wtedy wiersze w liściach indeksu są identyfikowane za pomocą wartości kluczy głównych. W rozważanym przypadku wiersze w indeksie są identyfikowane na podstawie kolumny *id*, co oznacza, że indeks zawiera wszystkie kolumny użyte w zapytaniu. Dzięki temu aby uzyskać wynik serwer nie musi odwoływać się do dysku, aby pobrać dane, ponieważ wszystkie dane użyte w zapytaniu znajdują się w indeksie. Jest to przykład wykorzystania indeksu pokrywającego.

Kolejnym często używanym zapytaniem jest pobranie wszystkich kolumn z tabeli, ale sortowanie 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, ponieważ kolejność kolumn użytych do sortowania danych nie jest identyczny jak w indeksie.

W kolejnym przykładzie przeanalizowano następujące 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, ponieważ kolumny w indeksie są posortowane względem kolumn Reputation, a w przypadku, kiedy wartość Reputation jest równa, względem kolumny UpVote, co odpowiada wartości ORDER BY. Następnie sprawdzono co się stanie po delikatnej modyfikacji zapytania 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.

9.2 Przechowywanie wartości NULL czy pustej wartości

Częstym zagadnieniem dotyczącym przechowywania danych w tabelach MySQL jest pytanie, w jaki sposób reprezentować brak wartości. Załóżmy, że mamy tabelę studentów, która posiada wiersz z numerem domowym studenta. Zdecydowana większość studentów nie posiada numeru domowego. W jaki zatem sposób ustawić wartość w bazie danych? Brak numeru zapisać jako wartość NULL czy może pusty ciąg znaków? Na początku rozważono kwestię wykorzystania przestrzeni na dysku. Przygotowano cztery następujące tabele:

```
CREATE TABLE test_null_varchar_values(  
k1 varchar(32) not null, k2 varchar(32) not null,  
k3 varchar(32) not null, k4 varchar(32) not null,  
k5 varchar(32) not null, k6 varchar(32) not null,  
k7 varchar(32) not null, k8 varchar(32) not null);
```

```
CREATE TABLE test_not_null_varchar_values(  
k1 varchar(32) not null, k2 varchar(32) not null,  
k3 varchar(32) not null, k4 varchar(32) not null,  
k5 varchar(32) not null, k6 varchar(32) not null,  
k7 varchar(32) not null, k8 varchar(32) not null);
```

```
CREATE TABLE test_not_null_int_values(  
k1 int not null, k2 int not null,  
k3 int not null, k4 int not null,  
k5 int not null, k6 int not null,
```


Rysunek 33: Wyniki polecenia SHOW TABLE STATUS dla tabeli z ośmioma kolumnami

#	Name	Engine	Version	Row_format	Rows	Avg_row_length	Data_length
1	test_not_null_int_values	InnoDB	10	Dynamic	14891	106	1589248
2	test_not_null_varchar_values	InnoDB	10	Dynamic	15024	105	1589248
3	test_null_int_values	InnoDB	10	Dynamic	15107	29	442368
4	test_null_varchar_values	InnoDB	10	Dynamic	15107	29	442368

Rysunek 34: Wyniki polecenia ANALYZE TABLE dla tabeli z jedną kolumną

#	Name	Engine	Version	Row_format	Rows	Avg_row_length	Data_length
1	test_not_null_int_values	InnoDB	10	Dynamic	15069	32	491520
2	test_not_null_varchar_values	InnoDB	10	Dynamic	15107	29	442368
3	test_null_int_values	InnoDB	10	Dynamic	15107	29	442368
4	test_null_varchar_values	InnoDB	10	Dynamic	15107	29	442368

```
k7 int not null, k8 int not null);
```

```
CREATE TABLE test__null_int_values(
k1 int,k2 int,k3 int, k4 int,
k5 int,k6 int,k7 int,k8 int);
```

Następnie tebele zostały wypełnione piętnastoma tysiącami wierszy. W przypadku tabeli, które dopuszczają wartość NULL wypełniono je takimi właśnie wartościami. Dla tabel z kolumnami oznaczonymi jako NOT NULL wypełniono odpowiednio pustym tekstem (") lub wartością 0. Następnie na każdej z tabel wykonano polecenie ANALYZE TABLE, w celu aktualizacji statystyk i wykonano polecenie SHOW TABLE STATUS, którego wyniki umieszczono poniżej na rysunku 33.

Wykorzystanie NULL zredukować rozmiar tabel o ok. 70 %. W kolejnym kroku przeanalizowano sytuację, kiedy w tabeli przechowywana jest tylko jedna kolumnę z możliwymi wartościami NULL. Zmodyfikowano skrypty tworzące tabele tak, żeby każda tabela posiadała tylko jedną kolumnę. W analogiczny sposób wypełniono bazę piętnastoma tysiącami wierszy i dla każdej z tabel wykonano polecenie ANALYZE TABLE. Na rysunku 34 przedstawiono wyniki polecenia SHOW TABLE STATUS dla tabeli.

Jednakowy rozmiar wierszy zawierających osiem wartości NULL oraz wierszy z jedną wartością NULL wynika ze sposobu w jaki MySQL przechowuje informację o kolumnach z wartościami NULL. Dla każdego wiersza, który zawiera kolumny z wartościami przechowywanymi, jest dodatkowy bajt z informacją o kolumnach z wartością NULL. MySQL na każdym bajcie przechowuje informacje dla maksymalnie 8 kolumn (jeden bit

Rysunek 35: Wyniki polecenia ANALYZE TABLE dla dwóch indeksów

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Users		index_merge	reputation_idx,views_idx	views_idx,reputatio...	4,4		305256	100.00	Using intersect(views_idx,reputation_idx); Using where; Using Index

Rysunek 36: Wyniki polecenia ANALYZE TABLE dla pojedynczego indeksu

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Users		ref	reputation_views_idx	reputation_views_idx	8	const,const	404912	100.00	Using Index

dla każdej z kolumn). Gdyby w wierszu zawierającym osiem kolumn, dodana została jeszcze jedna kolumna z dopuszczalną wartością NULL, wtedy MySQL zarezerwowałby dodatkowy bajt dla tego wiersza.

9.3 Indeks na wielu kolumnach czy wiele indeksów na jednej

Przed wersją 5.0 MySQL pozwalał na użycie tylko jednego indeksu dla tabeli, nawet jeżeli potencjalnie użytecznych było więcej. W takim przypadku, aby wyszukiwanie na większej liczbie kolumn korzystało z indeksów, należało utworzyć indeks składający się z wielu kolumn. W wersji 5.0 wprowadzony został algorytm łączenia indeksów (*index merge*). Idea łączenia indeksów umożliwia łączenie różnych indeksów na kolumnach użytych w klauzuli WHERE. Jako przykład posłużono się tabelą *Users* z bazy *StackOverflow*. Na początku założono dwa osobne indeksy na kolumny *reputation* oraz *views* i wykonano analizę zapytania wyszukującego dane z wykorzystaniem obu kolumn.

```
CREATE INDEX reputation_idx ON Users(reputation);
CREATE INDEX views_idx ON Users(views);
EXPLAIN SELECT count(*) FROM Users where reputation = 1 and views = 1;
```

Następnie zastąpiono pojedyncze indeksy jednym wielokolumnowym i wykonano analogiczną analizę.

```
EXPLAIN SELECT COUNT(*) FROM Users WHERE reputation = 1 AND views = 1;
```

Na rysunkach 35 oraz 36 zaprezentowano wyniki polecenia EXPLAIN dla obu przypadków. W obu odfiltrowane zostało 100 % wierszy. W następnej kolejności zmierzono średni czas wykonania obu zapytań. W przypadku pojedynczego indeksu zapytanie wymagało średnio 0.05 sekundy, natomiast przy łączeniu tabel ok. 0.5 sekundy. Na koniec usunięto indeks dla kolumny *Views*, żeby optymalizator wykorzystał tylko indeks *reputation_idx*.

```
CREATE INDEX reputation_idx ON Users(reputation);
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	Users	0000	ref	reputation_idx	reputation_idx	4	const	1246178	10.00	Using where

Rysunek 37: Wyniki polecenia ANALYZE TABLE dla indeksu *reputation_idx*

```
EXPLAIN SELECT COUNT(*) FROM Users WHERE reputation = 1 AND views = 1;
```

Tym razem tylko 10 % wierszy zostało odfiltrowane, a średni czas wykonania zapytania zbliżył się do 8 sekund. Analizując wyniki powyższego eksperymentu można stwierdzić, że mechanizm łączenia tabel nie będzie równie wydajny co pojedynczy indeks na wielu kolumnach, ale może być dobrą alternatywą do pełnego przeszukania lub wykorzystania jedynie jednego indeksu.

9.4 Sztuczny czy naturalny klucz główny

TODO

9.5 Procedury składowane

9.6 Monitorowanie niewydajnych zapytań

Manualne wyszukiwanie niewydajnych zapytań, które są wykonywane na serwerze MySQL może być uciążliwe. Jednym z rozwiązań dostępnych natywnie w MySQL jest mechanizm *Slow Query Log*, który umożliwia zbieranie informacji o długotrwałych zapytaniach. *Slow query log* jest plikiem zawierającym zapytania, których czas wykonania przekracza wartość zdefiniowaną w polu *long_query_time*, a liczba odczytanych wierszy przekracza wartość *min_examined_row_limit*. Dzięki temu analiza pliku pozwala zidentyfikować zapytania, które mogą być dobrymi kandydatami do dalszej optymalizacji. Wadą takiego rozwiązania jest pewien narzut czasowy wynikający z konieczności logowania informacji o zapytaniach. Z tego powodu bardzo istotne jest odpowiednie dobranie wartości czasu, od którego serwer powinien logować dane, ponieważ wybranie zbyt małej wartości może spowodować problemy wydajnościowe serwera, oraz doprowadzić do problemów z użyciem przestrzeni dyskowej ze względu na ogromne rozmiary plików. Logowanie jest domyślnie wyłączone. Aby je włączyć wystarczy wykonać następujące polecenia.

```
SET GLOBAL slow_query_log = 'ON';
SET GLOBAL long_query_time = 0.5; # 0.5 sekundy
```

Bibliografia

- [1] MySQL, *MySQL 8.0 Reference Manual*, <https://dev.mysql.com/doc/refman/8.0/en/>, dostęp sierpień 2020.
 - [2] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, *Wysoko wydajne MySQL*, Helion 2009.
 - [3] Eric Vanier, Birju Shah, Tejaswi Malepati, *Advanced MySQL 8*, Pact 2019.
- todo -mark-whitehorn-relacyjne-bazy-danych
- TODO -na stronie 46 dopisać podsumowanie!! -przeczytać całość jeszcze raz i usunąć małe litery z poleceń SQL!!