Bazy danych
przed 8 wykładem

Andrzej Giniewicz

26.04.2024

Dziś kontynuujemy materiał rozpoczęty na ostatnim wykładzie, dotyczący kluczy. Dotychczas powiązaliśmy pojęcie klucza z intuicją pochodzącą ze skorowidza w książce — pozwala szybciej szukać terminów, ale zajmuje miejsce. Pojęcie klucza jest związane ze skorowidzem na tyle mocno, że oba te terminy często też nazywane są indeksami.

1 Rodzaje kluczy

W SQL wyróżniamy kilka rodzajów kluczy. Klucz może być unikatowy lub nie. Klucze unikatowe mają gwarancję, że nie pojawi się w nich duplikat wartości. Implementacja kluczy unikatowych, może być wydajniejsza ze względu na prędkość wyszukiwania i budowania klucza, ponieważ pojedyncze odnalezienie wiersza kończy procedurę wyszukiwania, podczas gdy w przypadku kluczy nieunikatowych, możemy musieć ponowić sprawdzanie, czy jest to ostatnia obserwacja o danej wartości klucza.

Klucz może składać się z jednej lub więcej kolumn. Jeśli klucz składa się z kilku kolumn, na przykład $(c_1, \ldots, c_k)$, to ciągi kolumn $(c_1, \ldots, c_i)$ dla $1 \leqslant i < k$ nazywamy prefiksami tego klucza. Zapytania, które wykorzystują kolumny z prefiksu któregoś klucza, zwykle działają dużo wydajniej, niż te, które nie wykorzystują kluczy, ponieważ czas wyszukiwania wartości z wykorzystaniem prefiksu klucza jest logarytmiczny, podczas gdy czas wyszukiwania wartości bez klucza jest liniowy.

Spośród wszystkich kluczy unikatowych, jeden możemy wskazać jako klucz główny, o ile w żadnej z jego kolumn nie może się pojawić wartość `NULL`. Ważne, aby klucz główny był stosunkowo mały, ponieważ w InnoDB, najpopularniejszym typie tabel w MySQL i MariaDB, jest doczepiany na końcu każdego klucza jako jego sufiks, im większy klucz główny, tym wobec tego większy każdy inny klucz. Z tego powodu często klucz główny ogranicza się jedynie do jednej niewielkiej kolumny, na przykład `id`, a inne klucze używane w celu przyśpieszenia wyszukiwania, dodajemy do tabeli osobno. Tak się dzieje, jeśli tabela potrzebuje więcej niż jednego klucza. Jeśli tabela potrzebuje tylko jednego klucza i jesteśmy pewni, że tak zostanie, można utworzyć (jedyny) klucz główny składający się z większej liczby kolumn.

Z punktu widzenia implementacji są dwa rodzaje indeksów: oparte o B-drzewa[1] oraz o tablice hashujące[2]. MySQL i MariaDB wspiera indeksy hashujące tylko dla tabel znajdujących się w pamięci (silnik Memory), natomiast pozostałe rodzaje tabel mają indeksy w postaci B-drzewa (między innymi silniki InnoDB i MyISAM). Indeks oparty na tabeli hashującej jest zwykle szybszy, ale wspiera jedynie porównania = i <>, podczas gdy indeks oparty na B-drzewie wspiera wszystkie operatory relacji, warunek z `BETWEEN` oraz sortowanie. Istnieje możliwość zasymulowania indeksu haszującego dla tabel, które go nie wspierają, za pomocą tak zwanych pseudohashy[3]. Jeśli chodzi o tabele znajdujące się na dysku, czyli na przykład tabele InnoDB, rodzaj kluczy nie gra roli, ponieważ i tak wspierany jest tylko jeden ich rodzaj. Rodzaj klucza możemy wybrać tylko dla tabel działających w pamięci, które często służą za nośnik danych tymczasowych. W takiej sytuacji warto pamiętać, że jeśli operacje wykonywane na kolumnach ograniczają się do wyszukiwania za pomocą równości, indeks hashujący będzie lepszym wyborem.

Klucze dla zmiennych tekstowych oparte na B-drzewach pozwalają również na przyśpieszenie zapytań z `LIKE`, ale tylko wtedy, jeśli pierwsza litera jest określona, czyli wzorzec nie zaczyna się od wyrażenia regularnego. Dotąd omówione klucze nie przyśpieszają zapytań typu `LIKE '%a %'`. Aby przyśpieszyć taki rodzaj zapytań, należy zbudować indeks tekstowy `FULLTEXT`. Jest on dostępny tylko w tabelach MyISAM, czyli nie skorzystamy z niego ani w tabeli InnoDB, ani w tabeli Memory. Jeśli wykonujemy dużo zapytań tekstowych, warto rozważyć silnik MyISAM, musimy jednak pamiętać, że jest silnikiem nietransakcyjnym, czyli w sytuacji błędu w zapytaniu, nie ma możliwości wycofania części zmiany. Może to prowadzić do utraty spójności danych. MyISAM wspiera również indeksy przestrzenne `SPATIAL`, które pomagają szybko i wydajnie szukać pól za pomocą pary współrzędnych geograficznych. Indeksy `FULLTEXT` i `SPATIAL` są rzadko stosowane, z tego powodu tabele MyISAM nie są domyślnym sposobem zapisu danych.

Niezależnie od tego, jakie rodzaje kluczy wybierzemy, pamiętajmy, że klucze:

1. zwykle przyśpieszają wyszukiwanie danych w tabeli,
2. mogą spowolnić wyszukiwanie danych dla tabel z bardzo małą liczbą wierszy,
3. spowalniają dodawanie i modyfikowanie danych w tabeli,
4. zwiększają miejsce potrzebne na przechowanie tabeli.

[1] Z pojęciem B-drzewa można zapoznać się z wielu źródeł, w tym z oryginalnego artykułu wprowadzającego ten koncept `https://infolab.usc.edu/csci585/Spring2010/den_ar/indexing.pdf`.

[2] Dla osób chcących odświeżyć sobie informacje o tablicach hashujących, które mogliście spotkać przy okazji omawiania słowników w Pythonie, polecam wykład na ten temat na MIT — `https://www.youtube.com/watch?v=OM_kIqhwbFo`.

[3] Patrz: „Building your own hash indexes" w rozdziale 5 książki „High Performance MySQL".

2 Tworzenie tabel

Omówimy teraz tworzenie tabel, ze szczególnym uwzględnieniem tworzenia tabel tymczasowych i dodawania do nich kluczy, aby stworzyć miejsce do zapisywania cząstkowych wyników trudniejszych zapytań. Tabele tymczasowe działają tak samo jak zwykłe tabele, z wyjątkiem tego, że istnieją tylko w sesji użytkownika. Oznacza to, że tylko osoba, która je utworzyła, ma do nich dostęp. Inne osoby, logujące się nawet na tego samego użytkownika, nie widzą tabel tymczasowych innych użytkowników i mogą utworzyć swoje tabele o tej samej nazwie[4]. Co więcej, jeśli się wylogujemy, tabela tymczasowa zostanie skasowana.

Do tworzenia tabel służy dyrektywa `CREATE`. Możemy zacząć zapytanie od

```
CREATE TABLE nazwa
```

lub

```
CREATE TEMPORARY TABLE nazwa
```

jednak takie zapytanie wykonane dwa razy spowoduje błąd. Jeśli chcemy tworzyć tabelę tylko, kiedy jeszcze nie istnieje, napiszemy

```
CREATE TABLE IF NOT EXISTS nazwa
```

lub

```
CREATE TEMPORARY TABLE IF NOT EXISTS nazwa
```

Jeśli jednak chcemy, aby przy każdym wywołaniu skryptu, tabela się tworzyła od nowa, co ma sens dla zapytań tymczasowych, możemy ją skasować tuż przed jej utworzeniem (uwaga — operacja kasowania jest nieodwracana, nie kasujemy tabel z danymi do zajęć, tylko tabele tymczasowe lub w piaskownicy).

```
DROP TABLE IF EXISTS nazwa;
CREATE TABLE IF NOT EXISTS nazwa
```

lub

```
DROP TEMPORARY TABLE IF EXISTS nazwa;
CREATE TEMPORARY TABLE IF NOT EXISTS nazwa
```

Ponieważ kasowanie i tworzenie tabeli było często używane, w MariaDB pojawił się skrót do powyższego zapisu

[4]Jest tak, ponieważ tabele są związane z sesją użytkownika, a nie z użytkownikiem.

```
CREATE OR REPLACE TABLE nazwa
```

lub

```
CREATE OR REPLACE TEMPORARY TABLE nazwa
```

Niezależnie od rodzaju początku dyrektywy, oznacza ona, że zaczynamy tworzyć tabelę. W dalszych przykładach będziemy pisać jedynie `CREATE ...`, jako jeden z powyższych wariantów sposobów rozpoczęcia tworzenia tabeli. Zaraz po tej linii powinna pojawić się specyfikacja kolumn w nawiasie.

```
CREATE ... (
  kolumna1,
  kolumna2,
  kolumna3,
  ...
)
```

typowy opis kolumny składa się z jej nazwy, następnie typu danych i szeregu możliwych opcji, których wystarczająco przybliżony na nasze potrzeby opis znajduje się poniżej:

- `NULL` (domyślnie) lub `NOT NULL`, informuje SQL, czy kolumna może przechowywać braki danych,
- `DEFAULT wartość`, informuje SQL, jaką wartością uzupełnić braki danych. Gdy kolumna nie ma ustawionego `NOT NULL`, domyślną wartością jest `DEFAULT NULL`;
- `AUTO_INCREMENT`, informuje SQL, że kolumna ta będzie automatycznie numerowana. Można wskazać tylko jedną kolumnę jako `AUTO_INCREMENT`, przy czym musi być jednego z typów całkowitych.

Dodatkowo, jeśli definiujemy klucze składające się z jednej kolumny, możemy wypisać to przy definicji kolumny za pomocą opcji

- `PRIMARY KEY`, definiuje kolumnę jako klucz główny, automatycznie będzie miała ustawione `NOT NULL`,
- `UNIQUE KEY`, definiuje kolumnę jako klucz unikatowy,
- `KEY`, definiuje kolumnę jako klucz niekoniecznie unikatowy.

Na koniec możemy dodać przydatny dla czytelnika kodu komentarz `COMMENT "tekst"`. Częstą praktyką jest ustawianie pierwszej kolumny jako `id`, które jest polem `PRIMARY KEY` i `AUTO_INCREMENT` — dzięki temu mamy niewielki klucz główny, którym nie musimy się przejmować, ponieważ kolejne wiersze tabeli będą automatycznie numerowane.

```
CREATE ... (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  user VARCHAR(40) COMMENT ”Imię i nazwisko”,
  płeć ENUM(’kobieta’, ’mężczyzna’) NOT NULL,
  ‘liczba zamówień‘ INT UNSIGNED NOT NULL DEFAULT 0
)
```

W powyższym przykładzie mamy cztery kolumny: `id`, `user`, `płeć` oraz `liczba zamówień`. Pierwsza kolumna jest automatycznie numerowanym kluczem głównym, czyli kolejne wiersze będą otrzymywać numery od 1 w dal. Druga kolumna może mieć braki danych, jest napisem o długości maksimum 40 znaków i zawiera imię i nazwisko, co jest sprecyzowane komentarzem. Kolejne pole jest wyliczeniem, ma dwie możliwe wartości i nie może mieć braków danych. Ostatnie pole również nie może mieć braków danych, ale ma wartość domyślną zero, więc jeśli użytkownik nie poda wartości kolumny, zapisane w niej zostanie właśnie zero. W ten sposób możemy stworzyć dowolnie dużą tabelę, aż do maksymalnej liczby 4096 kolumn.

Jeśli chcemy stworzyć klucze składające się z więcej niż jednej kolumny, możemy wypisać je poniżej wszystkich kolumn, wciąż w nawiasie.

```
CREATE ... (
  ...,
  KEY nazwa (kolumna1, kolumna2, ...),
  UNIQUE KEY nazwa (kolumna1, kolumna2, ...),
  PRIMARY KEY nazwa (kolumna1, kolumna2, ...)
)
```

Po każdym z tych kluczy może wystąpić jego typ

```
CREATE ... (
  ...,
  KEY nazwa (kolumna1, kolumna2, ...),
  UNIQUE KEY nazwa (kolumna1, kolumna2, ...) USING BTREE,
  PRIMARY KEY nazwa (kolumna1, kolumna2, ...) USING HASH
)
```

Domyślnym typem kluczy są B-drzewa. Po wypisaniu wszystkich kolumn i kluczy i zamknięciu nawiasu, możemy wypisać opcje, oddzielone przecinkami w postaci na przykład

```
CREATE ... (
  ...
) ENGINE=InnoDB, CHARACTER SET=utf8mb4, COLLATE=utf8mb4_polish_ci;
```

W przypadku powyżej mamy określony rodzaj silnika (tutaj InnoDB, inne wartości to na przykład MyISAM lub MEMORY), domyślny zestaw znaków i domyślne kodowanie dla wszystkich pól tekstowych w tabeli. To oczywiście tylko niektóre opcje[5]. Na przykład, aby stworzyć tabelę tymczasową znajdującą się w pamięci i działającą z indeksem hashującycm, napiszemy

```
CREATE TEMPORARY TABLE IF NOT EXISTS tymczasowa (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  imię VARCHAR(40) NOT NULL,
  nazwisko VARCHAR(40) NOT NULL,
  UNIQUE KEY user (imię, nazwisko) USING HASH
) ENGINE=Memory;
```

Po stworzeniu takiej tabeli możemy wypełnić ją danymi z innego zapytania za pomocą

```
INSERT INTO tymczasowa(imię, nazwisko)
SELECT imię, nazwisko FROM ...;
```

i potem traktować tabelę tymczasową jak każdą inną, wykonując na niej kolejne zapytania. Oczywiście, tabele tymczasowe nie muszą być w silniku Memory ani korzystać z indeksów hashujących — tutaj zostało to tylko zademonstrowane jako przykład.

3 Zależności pomiędzy tabelami

Bardzo często pomiędzy tabelami są pewne zależności. Na przykład w tabeli zamówień możemy mieć pole wskazujące na numer klienta.

```
CREATE TABLE użytkownicy (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nazwa VARCHAR(40) NOT NULL,
  hasło VARCHAR(40) NOT NULL COMMENT ”zaszyfrowane hasło”,
  adres VARCHAR(120)
);
CREATE TABLE zamówienia (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  data TIMESTAMP,
  id_użytkownika INT UNSIGNED
);
```

[5]Więcej opcji znajdziemy w dokumentacji `https://mariadb.com/kb/en/create-table/#table-options`.

Zobaczmy, że w tym kodzie zmienna `id_użytkownika` niejako wskazuje na kolumnę `id` w tabeli użytkownicy. Nasz system powinien zadbać o to, abyśmy nie zapomnieli o tej zależności. Po pierwsze, jeśli nie ma użytkownika o danym numerze, nie powinniśmy być w stanie dodać nowego zamówienia dla tego użytkownika, ponieważ nie będziemy wiedzieć, gdzie je wysłać. Takie ograniczenie systemowe zaimplementowane na poziomie bazy danych, nazywamy kluczem obcym. Możemy dodać do tabeli dowolną liczbę kluczy obcych.

```
);
CREATE TABLE zamówienia (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  data TIMESTAMP,
  id_użytkownika INT UNSIGNED,
  FOREIGN KEY użytkownik_istnieje (id_użytkownika) REFERENCES użytkownicy(id)
);
```

Po dodaniu klucza obcego, SQL zadba o to, że nikt nie doda zamówienia nieistniejącego użytkownika. Jest to jeden z tak zwanych więzów integralności. Wcześniej już poznaliśmy inne więzy integralności — `NOT NULL` oraz unikatowe klucze są również więzami integralności, ponieważ system zabrania nam wykonania niektórych zapytań, jeśli doprowadzą do utraty spójności danych.

Co jednak, jeśli nasze dane były spójne, ale po dodaniu zamówienia kasujemy użytkownika? Istnieje kilka możliwości, ale domyślnie system wymusi na nas najpierw ręczne usunięcie wszystkich zamówień użytkownika i dopiero gdy żadne zamówienie nie wskazuje na usuwaną osobę, pozwoli nam ją usunąć. Jest to domyślne zachowanie, które może być zapisane jawnie w definicji klucza

```
FOREIGN KEY ... REFERENCES ... ON UPDATE RESTRICT ON DELETE RESTRICT
```

Oznacza to, że zarazem aktualizacja, jak i kasowanie wierszy będą zakazane, jeśli naruszy to integralność bazy. Inne dwie przydatne opcje to `CASCADE`, które automatyzuje usuwanie wierszy zależnych (zamówień) przy kasowaniu wiersza z tabeli referencyjnej (użytkowników) oraz `SET NULL`, które w tej sytuacji powoduje wpisanie braków danych. W naszym przypadku użytkowników i zamówień sens może mieć

```
FOREIGN KEY ... REFERENCES ... ON UPDATE CASCADE ON DELETE SET NULL
```

ponieważ, gdy z jakiegoś powodu numer użytkownika się zmieni, możemy bezpiecznie zaktualizować go również w zamówieniu, natomiast gdy użytkownika skasujemy, możemy wciąż chcieć zachować informacje historyczne o jego zamówieniu, w celach statystycznych dotyczących sprzedaży. Dane te bez połączenia z tabelą użytkownika (brak danych w polu `id_użytkownika`) będą zanonimizowane, więc mogą zostać zachowane nawet po skasowaniu danych konta. Warto napomknąć, że w Unii Europejskiej obowiązuje prawo, które pozwala użytkownikowi dowolnego systemu teleinformatycznego, zażądać skasowania jego danych

osobowych ze wszystkich baz. Jest to o tyle trudne, że powinno tyczyć się również kopii zapasowych. Wdrożenie dyrektywy unijnej pozwalającej na takie rozwiązanie spędzało i dalej spędza sen z powiek wielu administratorom i analitykom danych, ponieważ wszyscy muszą tak przemyśleć strukturę tabel, żeby ewentualne usunięcie danych osobowych było proste, ale jednocześnie tak, aby usuwać jak najmniej i pozostawiać po takiej operacji anonimowe dane, które wciąż mogą się do czegoś przydać podczas analizy.