Bazy danych
przed 3 wykładem

Andrzej Giniewicz

15.03.2024

Dziś omówimy liczbowe i napisowe typy danych dostępne na serwerze oraz jakie mają implikacje dla zajmowanego miejsca i wydajności. Omówione poniżej typy danych dotyczą tego, co zaimplementowane jest w serwerze baz danych MariaDB. Inne serwery mogą mieć inne nazewnictwo i detale implementacji.

1 Liczby całkowite

Typ liczby całkowitej zapisujemy jako `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT` lub `BIGINT`. Typy te zajmują odpowiednio 1, 2, 3, 4 i 8 bajtów lub innymi słowy 8, 16, 24, 32 i 64 bity. Standardowo każdy z tych typów może reprezentować wartość od -2^{k-1} do $2^{k-1} - 1$, gdzie k to liczba bitów poświęconych na zapis liczby.

typ	bajtów	wartość najmniejsza	wartość największa	miejsca
`TINYINT`	1	−128	127	3
`SMALLINT`	2	−32768	32767	5
`MEDIUMINT`	3	−8388608	8388607	7
`INT`	4	−2147483648	2147483647	10
`BIGINT`	8	−9223372036854775808	9223372036854775807	19

W tabeli powyżej znajduje się również liczba cyfr wartości największej, abyśmy mogli lepiej sobie wyobrazić jej rozmiar bez liczenia cyfr.

Do każdego z wymienionych typów można dodać słowo `UNSIGNED`, które powoduje, że najmniejsza wartość będzie równa 0, natomiast największa $2^k - 1$.

typ	bajtów	wartość największa	miejsca
`TINYINT UNSIGNED`	1	255	3
`SMALLINT UNSIGNED`	2	65535	5
`MEDIUMINT UNSIGNED`	3	16777215	8
`INT UNSIGNED`	4	4294967295	10
`BIGINT UNSIGNED`	8	18446744073709551615	20

Domyślnie, gdy MariaDB przesyła tabelę, informuje program wyświetlający o szerokości kolumny. Szerokość kolumny, jeśli nie podana, będzie równa szerokości najdłuższej wyświetlonej liczby. Istnieje możliwość podania szerokości kolumny w nawiasach, zaraz po `...INT` (przed `UNSIGNED`, jeśli występuje). I tak na przykład `TINYINT(2) UNSIGNED` będzie rezerwowało 2 znaki na kolumnę, pomimo że potrafi przechować więcej informacji. Jeśli jakaś wartość zajmie więcej niż zarezerwowano, kolumna będzie szersza. Jeśli liczba znaków w liczbie będzie mniejsza niż określona, edytory zwykle poprzedzą liczbę odpowiednią liczbą spacji, efektywnie wyrównując liczby do prawej strony. Zmiana sposobu wyświetlania nie wpływa na obliczenia oraz liczbę przechowywaną w pamięci. Jest to kwestia czysto estetyczna i wiele edytorów ignoruje szerokość typów całkowitoliczbowych.

2 Liczby wymierne

Omówimy trzy typy pozwalające wyświetlać podzbiory liczb wymiernych. SQL pozwala zarazem na pracę z liczbami zmiennoprzecinkowymi, jak i stałoprzecinkowymi.

Do liczb zmiennoprzecinkowych używamy `FLOAT` oraz `DOUBLE`, które są odpowiednio 4 i 8 bajtowymi, czyli 32 i 64 bitowymi liczbami zmiennoprzecinkowymi zapisanymi w standardzie IEEE 754. Bity liczby zmiennoprzecinkowej są rozdzielone pomiędzy jeden bit znaku oraz ustaloną liczbę bitów wykładnika i mantysy.

typ	bitów wykładnika (e)	bitów mantysy (m)
`FLOAT`	8	23
`DOUBLE`	11	52

Typ zmiennoprzecinkowy o e bitach licznika i m bitach mantysy potrafi przedstawić zarazem bardzo małe liczby (najmniejsza liczba dodatnia to $2^{2-m-2^{e-1}}$), bardzo duże liczby (największa liczba dodatnia to $(1-2^{-m-1})2^{2^{e-1}}$) oraz wszystkie liczby całkowite co do modułu mniejsze lub równe 2^{m+1} (poza tym zakresem liczby całkowite mogą być przybliżone). Dla typów zdefiniowanych w MariaDB przekłada się to na następujące wartości.

typ	najmniejsza dodatnia	największa skończona	zakres całkowitych
`FLOAT`	$2^{-149} \approx 1.4 \times 10^{-45}$	$(1-2^{-24})2^{128} \approx 3.4 \times 10^{38}$	$2^{24} = 16777216$
`DOUBLE`	$2^{-1074} \approx 4.94 \times 10^{-324}$	$(1-2^{-53})2^{1024} \approx 1.8 \times 10^{308}$	$2^{53} = 9007199254740992$

Oznacza to, że w typie `FLOAT` nie przedstawimy liczby całkowitej 16777217, choć możemy przedstawić 16777216 i 16777218. Niekiedy możemy spotkać typy zmiennoprzecinkowe z liczbami w nawiasie, co wymusza zaokrąglenie do konkretnej liczby miejsc po przecinku — nie jest to jednak zalecane i oficjalna dokumentacja podaje, że należy tej składni unikać. Jeśli potrzebujemy konkretnej liczby miejsc po przecinku, lepiej użyć typu stałoprzecinkowego.

Typ stałoprzecinkowy w MariaDB to `DECIMAL(n, m)`. Typ `DECIMAL(n)` jest równoważny typowi `DECIMAL(n, 0)` natomiast `DECIMAL` typowi `DECIMAL(10, 0)`. Wartość n oznacza liczbę cyfr znaczących liczby (przed i po przecinku), natomiast wartość m liczbę cyfr po przecinku. Dla liczb naturalnych n i m muszą być spełnione nierówności

$$0 < n \leqslant 65, \quad 0 \leqslant m \leqslant \min\{38, n\}.$$

Na przykład `DECIMAL(3,2)` przechowuje trzy cyfry, z czego jedną przed przecinkiem i dwie po przecinku. Liczba $\frac{22}{7} = 3.(142857)$ będzie wobec tego przybliżona do 3.14, stosując zaokrąglenie do najbliższej reprezentowalnej liczby. Wszystkie obliczenia arytmetyczne na typie `DECIMAL` wykonywane są z precyzją 65 miejsc znaczących.

Liczby typu `DECIMAL` są reprezentowane w pamięci za pomocą skompresowanego formatu binarnego. Każdy blok 9 cyfr zajmuje 4 bajty, natomiast pozostała część b cyfr, $0 \leqslant b \leqslant 8$, zajmuje $\left\lfloor \frac{b+1}{2} \right\rfloor$ bajtów. Cyfry przed i po przecinku pakowane są osobno. Oznacza to, że `DECIMAL(n, m)` zajmuje

$$s(n-m) + s(m), \qquad \text{gdzie} \qquad s(x) = 4 \cdot \left\lfloor \frac{x}{9} \right\rfloor + \left\lfloor \frac{(x \bmod 9) + 1}{2} \right\rfloor.$$

Przykładowo, gdybyśmy chcieli przechować 15 miejsc znaczących przed przecinkiem i 10 po przecinku, czyli typ `DECIMAL(25, 10)`, będziemy potrzebować

$$s(15) = 4 \cdot \left\lfloor \frac{15}{9} \right\rfloor + \left\lfloor \frac{(15 \bmod 9) + 1}{2} \right\rfloor = 4 \cdot 1 + \left\lfloor \frac{6+1}{2} \right\rfloor = 4 + 3 = 7$$

bajtów na miejsca przed przecinkiem oraz

$$s(10) = 4 \cdot \left\lfloor \frac{10}{9} \right\rfloor + \left\lfloor \frac{(10 \bmod 9) + 1}{2} \right\rfloor = 4 \cdot 1 + \left\lfloor \frac{1+1}{2} \right\rfloor = 4 + 1 = 5$$

bajtów na miejsca po przecinku, czyli $7 + 5 = 12$ bajtów łącznie. Tyle samo pamięci zajmie typ `DECIMAL(26, 11)`, który może zapisać tyle samo miejsc przed przecinkiem, ale jedno miejsce po przecinku więcej.

Wiedząc, jak obliczyć zajętość pamięci przez liczby typu `DECIMAL` oraz jakie są nasze wymagania dotyczące precyzji, jesteśmy w stanie dobrać najpierw najmniejszy typ reprezentujący liczby z zadaną dokładnością, a następnie niekiedy nieco zwiększyć jego precyzję, bez zwiększania ilości zajętej pamięci.

Typ `DECIMAL` może być bardzo przydatny do zapisu księgowego — standardowym zaleceniem w systemach księgowych jest stosowanie czterech miejsc po przecinku — to pozwala bez zaokrągleń wyliczyć podatek (będący najczęściej całkowitą liczbą punktów procentowych) z grosza. Musimy jednak wziąć pod uwagę to, że obliczenia na liczbach zmiennoprzecinkowych wykonują się wprost na procesorze komputera, zatem są o wiele szybsze, niż na liczbach stałoprzecinkowych. Musimy wybrać pasującą nam do sytuacji równowagę pomiędzy wydajnością a precyzją.

3 Typ logiczny

Typ logiczny nazywa się `BOOLEAN` i zajmuje jeden bajt. Implementacja tego typu jest wykonana za pomocą synonimu do `TINYINT(1)`, czyli w rzeczywistości jest on liczbą całkowitą. Wartość `TRUE` jest synonimem do 1, natomiast `FALSE` jest synonimem do 0. Niemniej jednak, w SQL wartości te nam nie wystarczają — ponieważ oprócz wartości prawda lub fałsz, każda zmienna (nie tylko logiczna, ale również inne typy) może mieć brak danych — `NULL`. Jeśli w jakimś wyrażeniu, którego wartością jest wartość logiczna pojawi się `NULL`, wynikiem tego działania nie może być ani prawda ani fałsz, tylko wartość niewiadomo `UNKNOWN`. Każdy z następujących warunków ma nieznaną wartość:

- `NULL = 0`,
- `NULL <> 0`,
- `NULL = NULL`.

Pojawienie się braku danych reprezentowanego przez `NULL` wymusiło na twórcach baz danych wykorzystanie logiki trójwartościowej. I tak poniższe wyrażenia są prawdziwe:

- `UNKNOWN OR TRUE`,
- `TRUE OR UNKNOWN`,

poniższe są fałszywe:

- `UNKNOWN AND FALSE`,
- `FALSE AND UNKNOWN`,

natomiast wyrażenia poniżej nie mają ustalonej wartości logicznej, czyli mają wartość `UNKNOWN`:

- `UNKNOWN AND TRUE`,
- `TRUE AND UNKNOWN`,
- `UNKNOWN OR FALSE`,
- `FALSE OR UNKNOWN`.

Co ważne, również `NOT UNKNOWN` ma wartość `UNKNOWN`, ponieważ negujemy wartość nieznaną, czyli wynik operacji jest nieznany — przypadek ten jest problematyczny, ponieważ moglibyśmy się spodziewać, stosując podwójne zaprzeczenie, że `NOT UNKNOWN` to `KNOWN`, ale takiej wartości nie ma w SQL.

Do sprawdzania wartości logicznych nie należy używać operatora `=`, służy do tego operator `IS` albo `IS NOT`. Jeśli `b` to jakieś wyrażenie typu `BOOLEAN`, na przykład efekt relacji i operatorów logicznych, to możemy napisać dowolne z wyrażeń:

- `b IS TRUE`,
- `b IS FALSE`,
- `b IS UNKNOWN`,
- `b IS NOT TRUE`,
- `b IS NOT FALSE`,
- `b IS NOT UNKNOWN`.

Zwróćmy uwagę, że w ostatnim przypadku po prawej stronie nie mamy operacji `NOT UNKNOWN` po prawej stronie operacji `IS`, tylko wartość `UNKNOWN` zastosowaną jako prawa strona operatora `IS NOT`, mającego dwa słowa. Prawa strona operatora `IS` nie powinna być wyliczana za pomocą formuł logicznych, tylko powinna być zadana jako stała.

Operator `IS` oraz `IS NOT` stosuje się też do kolumn innych typów, aby sprawdzić, czy wartość w nich jest brakiem danych. Wtedy nie używamy jednak wartości `UNKNOWN` stosowanej do logiki trójwartościowej, tylko wartości `NULL`. Jeśli `x` jest pewną wartością typu innego niż `BOOLEAN`, operator `IS` używany jest następująco:

- `x IS NULL`,
- `x IS NOT NULL`.

4 Napisy

Aby opisać napisy, musimy najpierw przypomnieć sobie podstawowe informacje o kodowaniu znaków. Istnieje wiele kodowań znaków, czyli wiele sposobów tłumaczenia liter na ciąg bajtów. Najpopularniejszy w dzisiejszych czasach jest UTF-8, który każdy znak koduje za pomocą od jednego do czterech bajtów. Kodowanie to pozwala zapisać wszystkie znaki znanych ludziom alfabetów, nie ma natomiast obciążenia wynikającego z dużej liczby możliwych znaków, ponieważ tekst korzystający tylko z zakresu ASCII będzie miał tylko po jednym bajcie na znak. W kodowaniu UTF-8 płacimy pamięcią tylko za te znaki, które rzeczywiście są bardziej egzotyczne. Kodowanie UTF-8 w MariaDB ma skrót `utf8mb4`, co oznacza `utf8` i „multiple bytes: 4". Dlaczego nie po prostu `utf8`? Gdy do MySQL wprowadzano kodowanie UTF-8, zdecydowano wprowadzić jego okrojoną wersję, mającą maksymalnie 3 bajty na znak, a to dlatego, że ówczesne mechanizmy zapisu danych, rezerwowały tyle pamięci, żeby zmieścił się najdłuższy możliwy znak. W efekcie stosowanie kompletnego UTF-8 było mało opłacalne, ponieważ niewiele języków korzystało z czterobajtowych znaków. Sytuacja zmieniła się z czasem i wprowadzono pełne kodowanie, jednakże nazwa `utf8` już była zajęta przez jego trzybajtowy okrojony wariant. Z tego powodu właściwa wersja `utf8` musiała uzyskać jakąś inną nazwę, aby zachować kompatybilność kodu wstecz. MariaDB, która jest odgałęzieniem serwera MySQL, przejęła tę konwencję ze względu na kompatybilność z MySQL. W efekcie zawsze, gdy myślimy UTF-8, powinniśmy w serwerze wybrać `utf8mb4`, a nie `utf8`.

Sama informacja o zestawie znaków nie wystarczy — te same litery w różnych krajach mogą być ułożone w różnych miejscach alfabetu. W związku z tym, oprócz kodowania, określa się porządek znaków. Do każdego typu kolumny można dopisać wyrażenie

```
CHARACTER SET ... COLLATE ...
```

podające zestaw znaków oraz wykorzystane porządkowanie (`COLLATE` pochodzi od słowa `COLLATION` oznaczającego „porównanie" — nie mylić z „kolacją"). Porządkowanie to wyrażenie zbudowane w następujący sposób `kodowanie_język_opcje`. Nas najczęściej będzie interesować porównywanie `utf8mb4_polish_ci`. Opcja `ci` oznacza „case insensitive", czyli

porównywanie bez uwzględnienia wielkości liter. MariaDB i inne bazy SQL zwykle nie mają możliwości ustawienia sortowania rozróżniającego kolejność liter. Oznacza to, że w SQL „Maria" i „MARIA" to takie same napisy i zostaną potraktowane podczas sortowania w ten sposób, że znajdą się obok siebie. Jeśli będziemy chcieli określić w pełni kodowanie oraz kolejność w zmiennej tak, aby była rozpoznawana jako kolumna w języku polskim, na końcu typu dopiszemy

```
CHARACTER SET utf8mb4 COLLATE utf8mb4_polish_ci
```

Pewne fragmenty możemy pominąć, ponieważ w bazie danych obowiązuje kaskadowe ustawianie wartości domyślnych. Podczas instalacji serwera administrator określa domyślne kodowanie znaków i porządkowanie dla wszystkich baz danych. Podczas tworzenia bazy danych, będzie ono wykorzystane, gdy ktoś nie poda innego kodowania i sortowania. Kodowanie bazy danych jest domyślnym kodowaniem dla wszystkich tabel a kodowanie tabeli domyślnym kodowaniem dla każdego wiersza tej tabeli. To samo tyczy się reguł sortowania.

Idąc dalej, wprowadza się trzy główne typy napisów: `CHAR(n)`, `VARCHAR(n)` oraz rodzinę napisów `TEXT`. Omówimy najpierw dwa pierwsze, po czym przejdziemy do rodziny napisów `TEXT`.

Typy `CHAR(n)` oraz `VARCHAR(n)` są podobne. W obu przypadkach informują nas, że stosujemy napis maksymalnej długości znaków n. Szczególny przypadek `CHAR` bez podania wymiaru, oznacza `CHAR(1)`, czyli jeden znak. Różnica pomiędzy napisami jest taka, że podczas zapisu `CHAR(n)` zawsze rezerwuje miejsce na n znaków. Jeśli podamy krótszy napis, uzupełni go spacjami do pełnej długości n znaków. `VARCHAR(n)` natomiast nie uzupełnia napisu, tylko pamięta tyle znaków, ile rzeczywiście jest zapisanych w napisie. Wiąże się to z dodatkową komplikacją w implementacji typu, ponieważ `VARCHAR` oprócz samego napisu, musi pamiętać jego długość. Na zapisanie długości poświęca jeden bajt, jeśli najdłuższy napis, który możemy w nim zapisać, ma mniej niż 256 bajtów oraz dwa bajty w przeciwnym wypadku. Na przykład, jeśli używamy kodowania `utf8mb4`, każdy znak może mieć 4 bajty. Oznacza to, że napisy krótsze niż $n = \frac{256}{4} = 64$ znaki będą miały jeden dodatkowy bajt, natomiast mające 64 lub więcej znaków, dwa dodatkowe bajty. Takiego narzutu nie ma typ `CHAR`, który nie musi pamiętać długości napisu, bo wszystkie napisy mają w nim tę samą długość (są uzupełnione spacjami z prawej strony). Załóżmy kodowanie `utf8mb4` i spójrzmy na wzory opisujące zajętość pamięci.

`CHAR(n)`	$4n$
`VARCHAR(n)`, $n < 64$	$1 + x$, gdzie $0 \leqslant x \leqslant 4n$
`VARCHAR(n)`, $n \geqslant 64$	$2 + x$, gdzie $0 \leqslant x \leqslant 4n$

Wartość x w powyższych wzorach oznacza „dokładnie tyle, ile zajmuje napis". Jeśli napis składa się tylko z liter z zestawu ASCII, x jest długością napisu. Każdy polski znak kodowany jest w UTF-8 za pomocą dwóch bajtów, więc możemy myśleć o x jak o długości napisu plus liczbie polskich znaków. Pamiętajmy jednak, że niektóre znaki, na przykład emoji, zajmują aż 4 bajty.

Maksymalna liczba znaków *n* zależy od kodowania. W MariaDB najdłuższy napis może mieć 65532 bajtów, czyli w kodowaniu `utf8mb4` 16383 znaków. Jest to oczywiście limit górny i zwykle tworzymy mniejsze napisy. Wartość *n* może być równa zero. Wtedy w kolumnie mogą być tylko dwie wartości — pusty napis lub wartość `NULL`.

Jeśli kodujemy znaki za pomocą UTF-8, praktycznie nigdy nie opłaca się stosować `CHAR`, ponieważ musi zarezerwować 4 bajty na najdłuższy możliwy znak, podczas gdy `VARCHAR` rezerwuje tylko to, co potrzeba. Nawet przy napisie długości 1, jeśli rozważamy UTF-8, `CHAR(1)` zajmie 4 bajty, natomiast `VARCHAR(1)` zajmie 1 plus tyle, ile trzeba. O ile nie zamierzamy przechowywać w napisie emoji lub chińskich znaków, to `VARCHAR(1)` zajmie mniej pamięci pomimo narzutu. Inaczej sprawa się ma, gdy wiemy, że napis ma tylko znaki ASCII. Typ zmiennej

```
CHAR(1) CHARACTER SET ascii
```

jest o wiele mniejszy od

```
VARCHAR(1) CHARACTER SET ascii
```

W pierwszym przypadku napis zawsze ma 1 bajt, w drugim 1 bajt dla napisu pustego i dwa bajty dla napisu z jedną literą. Jeśli mamy kolumnę, w której przechowujemy płeć zakodowaną za pomocą jednej litery, np.: `"M"` oraz `"F"`, to `CHAR(1) CHARACTER SET ascii` może być bardzo dobrym typem dla tej zmiennej (choć na kolejnych zajęciach poznamy jeszcze lepszy typ na tę okoliczność).

W niektórych sytuacjach, na przykład przy wczytywaniu danych do pamięci z dysku, MariaDB stosuje tak zwany ustalony rozmiar wiersza. W takich sytuacjach typ `VARCHAR` traktowany jest jako `CHAR`. Jest to optymalizacja wyszukiwania, która, aby pobrać 10 wiersz, może przesunąć się w pamięci w prawo o 10 długości wiersza. Gdyby długość wiersza była różna, nie można by było wyliczyć pozycji wiersza i należałoby przeszukać całą strukturę element po elemencie. Więcej o silnikach przechowywania dowiemy się na kolejnych zajęciach.

Przejdźmy teraz do typu `TEXT`. Został on określony rodziną typów, ponieważ posiada warianty

type	maksymalny rozmiar w bajtach
`TINYTEXT`	$2^8 - 1 = 255 = 0.25KiB$
`TEXT`	$2^{16} - 1 = 65535 = 64KiB$
`MEDIUMTEXT`	$2^{24} - 1 = 16777215 = 16MiB$
`LONGTEXT`	$2^{32} - 1 = 4294967295 = 4GiB$

Każdy kolejny rodzaj pola tekstowego może przechować więcej tekstu, ostatni aż cztery gibibajty. Pamiętajmy, że zależnie od kodowania, liczba znaków może być odpowiednio mniejsza. Sposób, w jaki silnik SQL przechowuje dane tego typu, to pamięta długość (odpowiednio 1, 2, 3 oraz 4 bajty dla typów `TINYTEXT`, `TEXT`, `MEDIUMTEXT` i `LONGTEXT`) oraz wskaźnik (8 bajtów na komputerach 64-bitowych, które stanowią dziś większość dostępnych). Wskaźnik kieruje w inne miejsce, poza tabelą, zależne od implementacji silnika. Podjęto taką decyzję, aby nie tworzyć ogromnych wierszy — dla przypomnienia, aby szybko znajdować odpowiednie wiersze, MariaDB wczytuje tabelę do pamięci, ale dzięki temu zabiegowi, nie będzie wczytywać 4GiB tekstu, tylko po to, by wykonać sumę na innej kolumnie.

Z punktu widzenia przechowywania danych, warto zastanowić się, czy potrzebujemy tak dużych wartości. Jeśli często będziemy wyszukiwać i przetwarzać tekst, ale zmieści się w `VARCHAR` lub `CHAR`, to lepiej użyć ich zamiast `TEXT`. Jeśli natomiast `CHAR` i `VARCHAR` są za małe, `MEDIUMTEXT` i `LONGTEXT` są rozwiązaniem. Po co nam wobec tego `TINYTEXT`? Jeśli wiemy, że w kolumnie jest tekst, którego nie będziemy używać zbyt często, nie będziemy za jego pomocą wyszukiwać ani łączyć wartości, to wykorzystanie pól rodziny `TEXT` może zmniejszyć zajętość pamięci tabel tymczasowych, które są wykorzystywane jako tymczasowe miejsce do obliczeń dla wielu operacji, ponieważ wczytywać będziemy jedynie wskaźnik i długość napisu, a nie sam napis — typy te mogą więc przyśpieszyć obliczenia, jeśli jest w nich tekst, z którego rzadko lub prawie wcale nie korzystamy.