Podstawowa składnia języka MySQL/MariaDB

Chociaż MySQL/MariaDB bazują na specyfikacji SQL to posiadają pewne modyfikacje językowe i administracyjne, które są dla nich charakterystyczne. Nie odróżniają się przy tym zbytnio od pozostałych implementacji – jak to zostało wcześniej wspomniane każde rozwiązanie bazodanowe w jakimś stopniu odbiega od ogólnego standardu. W tym materiale szczegółowo zostaną omówione rozwiązania MySQL. Mogą one w znacznym stopniu być zbieżne z pozostałymi rozwiązaniami, aczkolwiek nie ma gwarancji, że będą działać.

INFORMACJA: Baza MariaDB to projekt otwartoźródłowy założony przez byłego programistę bazy MySQL. Po tym jak firma Sun Systems została przejęta przez Oracle obawiał się on, że MySQL stanie się płatnym bądź zamkniętym projektem. Dlatego rozwidlił (angielskie określenie to 'fork') projekt, a jego nową odnogę nazwał MariaDB. W jego założeniu projekt ten ma być niemal klonem bazy MySQL. W większości projekty te są ze sobą zbieżne. Posiadają jednak sporo różnic, przy czym ilość różnic przybywa wraz z kolejnym numerem wersji bazy. Na stronie projektu MariaDB wymieniane różnice wychodzą przeważnie na korzyść nowego projektu. Nie zmienia to faktu, że serwery MySQL i MariaDB są w stanie ze sobą rozmawiać i wymieniać dane. Używają ponadto tych samych klientów do łączenia się z bazą. Dla użytkownika końcowego może więc wydawać się, iż jest to ten sam projekt (lecz będzie to pojęcie błędne).

Słowo na temat licencji – obie bazy udostępnione są jako projekty otwartoźródłowe. Każdy może pobrać, przeglądać i modyfikować kod projektu. Ponadto może go przekompilować i używać na własny użytek. Ponadto obie bazy dostępne są na licencji GPL co oznacza, że każda modyfikacja kodu musi zostać udostępniona publicznie. Wcześniej MySQL dostępny był jako bezpłatna baza jedynie do zastosowań niekomercyjnych. Komercyjne wykorzystanie równało się z koniecznością uiszczenia opłaty, zależnej od przychodu uzyskanego dzięki bazie danych. W tej chwili nie ma różnicy czy wykorzystujemy bazę komercyjnie lub nie – opłat licencyjnych nie ma.

W dalszej części materiału przyjęto podawać nazwę MySQL. Opis poleceń dotyczy jednak w takim samym stopniu MariaDB.

1. Podstawy składni.

a) używanie odwróconych apostrofów w nazwach elementów bazy

Składnia MySQL nie różni się niczym od standardowej składni SQL. Różnicą jest możliwość (niektórzy,w tym autor niniejszego opracowania, możliwość tę traktują jako konieczność) stosowania odwróconych apostrofów (`) w celu objęcia nazw baz danych, tabel czy kolumn w tabelach. Zabieg ten stosuje się po to by interpreter wiedział iż ma do czynienia właśnie z nazwami własnymi w bazie oraz by wyeliminować pomyłki w przypadku, gdy autor bazy używał spacji jako oddzielacza kolejnych słów w nazwie baz, tabel lub pól.

Przykład polecenia SQL:

```
SELECT ulica, numer, kod_pocztowy FROM adresy WHERE miasto='Olkusz';
```

Przykład polecenia MySQL:

```
SELECT `ulica`,`numer`,`kod_pocztowy` FROM `adresy` WHERE `miato`='Olkusz';
```

Proszę zwrócić uwagę, że dla WARTOŚCI są stosowane zwykłe apostrofy. W innej konfiguracji spowodujemy błąd bazy danych!

b) obiektowy styl odwoływania się do elementów-liści

Elementy na serwerze baz danych tworzą pewną strukturę. Przykładowo kolumna ulica istnieje w tabeli adresy, a ta z kolei należy do bazy danych forum_elektroniczne. Odwołanie do takiego pola dla SQL wygląda tak:

SELECT `forum_elektroniczne`.`adresy`.`ulica` FROM `forum_elektroniczne`.`adresy`;

Dlaczego w ten sposób? Wyobraźmy sobie sytuacje, gdy na serwerze mamy kilka tabel o nazwie adresy, w których posiadamy kolumnę ulica. Skąd serwer ma wiedzieć, o którą tablę nam chodzi? W związku z tym należy go naprowadzić na pożądane wartości. Przeważnie jednak pomija się nazwę bazy danych przy tworzeniu zapytania. Takie rozwiązanie możliwe jest dzięki użyciu polecenia USE, np.

USE `forum_elektroniczne`;

Każde kolejne zapytanie realizowane po tej linii będzie domyślnie podejmowane w tabelach wskazanej bazy danych. Przy prostych zapytaniach, takie jak to powyżej, można także zrezygnować z podawania po słowie SELECT nazwy tabeli – jest ona określona w sekcji FROM. Ostatecznie więc można powyższe zapytanie wykonać w następujący sposób:

USE `forum_elektroniczne`;
SELECT `ulica` FROM `adresy`;

c) model transakcyjny

Ostatnie, o czym należy pamiętać rozpoczynając przygodę z MySQL (i w zasadzie z każdą bazą opartą o SQL) to możliwość ręcznego akceptowania zmian zawartości bazy danych. Domyślnie każda zmiana danych (dodanie, aktualizacja, kasowanie) jest automatycznie akceptowana. Nie jest to złe rozwiązanie, jednak niekiedy może ono wpłynąć negatywnie na szybkość dostępu do bazy danych.

Inną przesłanką do użytkowania ręcznego mechanizmu transakcji jest potencjalna możliwość popełnienia błędu. W tym wypadku, jeżeli do takiego błędu doszło, możemy usunąć nasze modyfikacje bez uszczerbku dla aktualnie składowanych danych. Ponadto istnieje możliwość utworzenia zapisu aktualnie aktywnej transakcji – wtedy cofnięcie jej będzie odbywało się jedynie do tego punktu (wszystkie wcześniejsze operacje zostaną zachowane).

Przykład użycia mechanizmu transakcji:
use `forum`;
START TRANSACTION; */*tutaj rozpoczynamy naszą transakcję*/*
INSERT INTO `cities` (`name`, `description`, `postal`) VALUE ('Częstochowa', 'Pierwsze dodane miasto', '42-200');
INSERT INTO `cities` (`name`, `description`, `postal`) VALUE ('Lublin', 'Drugie dodane miasto', '40-210');
SELECT * FROM `cities`; */*to co wyświetli to zapytanie będzie wyświetleniem danych z MIGAWKI (a nie z bazy danych, która nie będzie zawierać tych rekordów)*/*
SAVEPOINT nowy; */*tworzymy punkt przywracania; to co znajdzuje się przez tym punktem będzie w transakcji bezpieczne*/*
INSERT INTO `cities` (`name`, `postal`) VALUE ('Kraków', '50-120');
INSERT INTO `cities` (`name`, `postal`) VALUE ('Opole', '37-200');
SELECT * FROM `cities`; */*znowu wyświuetlamy dane*/*
ROLLBACK TO nowy; */*cofamy zmiany do nowy - dwa ostatnie rekordy nie zostana zapisane w*

migawce/*
SELECT * FROM `cities`;
COMMIT; */*akceptujemy nasza transakcje zostana dodane dwa pierwsze rekordy*/*
*/*ROLLBACK; - tego polecenia użylibyśmy by wyrzucić wszystkie zmiany z transakcji - również te z zapisanego punktu!*/*

Należy pamiętać, że możemy tworzyć kilka punktów przywracania w ramach jednej transakcji.
Poza tym od napisania START TRANSACTION nic nie zostanie przez nas dodanych w sesji chyba, że wpiszemy słowo COMMIT/ROLLBACK. To one ponownie uruchamiają automatyczne akceptowanie treści.
Istnieje jeszcze jedno polecenie dotyczące zapisanych punktów

RELEASE SAVEPOINT nazw_punktu

Powoduje ono zwolnienie nazwy wskazanego punktu (usunięcie go). NIE USUWA ONO jednak ani transakcji, ani zapytań, które tenże punkt zabezpieczał. Po prostu sam punkt przestaje być dostępny.

Innym sposobem na zablokowanie automatycznych akceptacji operacji jest wydanie polecenia:

SET autocommit=0;

Od tego momentu ŻADNA operacja nie zostanie zapisana w bazie, a jedynie pliku transakcji.
Każde polecenie będzie musiało się kończyć ręcznie wdanym poleceniem COMMIT (dotyczy to całych grup poleceń).

d) atrybuty typów pól MySQL

MySQL posiada następujące dodatkowe atrybuty mogące zostać ustawionymi dla każdego z pól tabeli:
(podane na przykładzie w nawiasach []):
AUTO_INCREMENT – wartość zapisana pod daną kolumną będzie automatycznie zwiększana przy każdorazowym dodawaniu kolejnego wiersza danych; działa na typy liczbowe (tylko jedna kolumna w tabeli może posiadać tę wartość)
UNSIGNED – wartości zapisywane pod kolumną nie mogą być ujemne (typy liczbowe)
ZEROFILL – tylko dla typów liczbowych; jeżeli określimy konkretną ilość cyfr przypadającą na zapisywaną wartość liczbową, a liczba ta będzie zawierać mniejszą liczbę cyfr to baza danych automatycznie uzupełni te braki wypełniając je zerami
NOT NULL – domyślnie dana kolumna może pozostać pusta (NULL) przy uzupełnianiu nowego wiersza danymi; dodanie wspomnianej cechy uniemożliwi dodanie wiersza jeżeli pod daną kolumną nastąpi próba zapisu wartości NULL
UNIQUE – wartości zapisywane pod tą kolumną muszą być unikatowe; jeżeli nastąpi próba zapisu wartości już występującej w którymś z wierszy tabeli (dotyczy tylko rozpatrywanej kolumny) zostanie zwrócony błąd
DEFAULT <wartosc> - jeżeli wartość danej kolumny przy dodawaniu nowego wiersza będzie NULL (pusta, np. kolumna ta zostanie pominięta w zapytaniu INSERT) to we wskazanej komórce pojawi się wprowadzona przez projektanta wartość
PRIMARY KEY – wskazuje na pole, które będzie kluczem głównym w tabeli (podstawowym);
KEY – wskazuje, że tworzone pole będzie polem indeksowanym (może być wiele pól indeksowanych)
COMMENT <opis> - dodaje komentarz (opis) pola w bazie; pozwala to na lepszą orientację po co tworzone było dane pole (informacje głównie dla projektanta).

CHARACTER SET <format> - pozwala na ustawienie kodowania dla tworzonego pola w tabeli
COLLATE <format> - wskazuje metodę porównywania napisów w przypadku przeszukiwania wskazanego pola; w przypadku pominięcia opcji CHARACTER SET pole to oddziałuje na format składowania danych tekstowych.

INFORMACJA: W przypadku pominięcia CHARACTER SET oraz COLLATE informacje o kodowaniu znaków przyjmowane są domyślnie z kodowania znaków w bazie danych!

e) metody porównywania napisów (kodowanie znaków)

w MySQL pod pojęciem porównywania (ang. *collate*) kryje się zachowanie kodowania znaków w tabelach bądź poszczególnych polach. W chwili obecnej najpopularniejszym kodowaniem znaków jest UTF8; pozwala ono bowiem zapisać niemal wszystkie znaki jakie występują w alfabetach ziemskich. MySQL wspiera wiele systemów kodowań – zarówno popularne, jak i mniej popularne oraz wręcz egzotyczne (wykorzystywane sporadycznie). Domyślnym kodowaniem serwera nie jest jednak UTF a latin1_swedish_ci; wynika to z „pochodzenia” bazy danych (tworzona w Szwecji). W związku z tym należy pamiętać, by dla każdej nowo tworzonej bazy zmieniać kodowanie na nam odpowiadające (szczelnie gdy chcemy przechowywać polskie znaki). Jeżeli nie zmienimy kodowania dla bazy to możemy zmieniać kodowanie dla poszczególnych pól w tabelach (na nam odpowiadające).

INFORMACJA: Chociaż UTF8 jest ciekawym rozwiązaniem (uniwersalnym) to należy mieć świadomość, że ani w rzeczywistości, ani w bazie danych nie zajmuje on jednego bajta; ilość bajtów zajmowanych znaków w tym kodowaniu zależna jest bowiem od samego znaku – jeżeli mieści się w standardzie ASCII to rzeczywiście zajmuje jeden bajt;w przeciwnym wypadku zajmuje 2 bajty (polskie znaki) bądź nawet 3 (znaki azjatyckie). W bazie domyślnie rezerwowane są 2 bajty na każdy znak (baza potrafi jednak dynamicznie zmieniać zapotrzebowanie rozmiaru każdego z pól). W bazie dodano także kodowanie o nazwie UTF8mb4. Jak można dowiedzieć się z dokumentacji, kodowanie to stanowi nadpisanie zbioru UTF8 gdyż pozwala zapisywać znaki na 4 bajtach. Tym samym pozwala ono na zapis w bazie wszystkich znaków ze wszystkich języków. Trzeba jednak pamiętać, że używając np. typu CHAR zmuszamy bazę do rezerwacji 4 bajtów na każdy znak (tutaj sugestią programistów jest używanie typu VARCHAR/TEXT).

UWAGA: Kodowanie znaków może okazać się krytycznym czynnikiem dużego wzrostu objętości bazy danych. Dlatego kodowanie pól należy dobierać odpowiednio do zastosowania, tj. w przypadku korzystania jedynie z podstawowego alfabetu w danym polu lepiej jest wybrać kodowanie najprostsze, podczas gdy przy korzystaniu z bazy wielojęzycznej lepiej jest wybierać UTF. Należy też pamiętać, że sam UTF ma kilka odmian – UTF16 (zapis stały na 2 bajtach) oraz UTF32 (zapis stały na 4 bajtach). Chociaż znaki w tych kodowaniach zajmują znacznie więcej miejsca to przeszukiwanie ciągów znakowych dzięki nim jest znacznie szybsze (porównywanie bajtowe).

f) silniki bazy danych

MySQL pozwala na korzystanie z kilku silników baz danych, które można zastosować dla tworzonych tabel. Domyślnym silnikiem składowania danych jest InnoDB, który został opracowany w celu zastąpienia wysłużonego MyISAM. Pomimo tego projektanci tabel mogą zmienić InnoDB na dowolny inny silnik. Dostępne silniki w MySQL:
1) InnoDB – silnik baz danych nowej generacji. Pozwala na pełną obsługę transakcji, kompresję danych przechowywanych w bazie (przy jednoczesnym zmniejszeniu ruchu wejścia/wyjścia), wydajniejsza obsługa indeksów. Ponadto system ten, w przypadku awarii nośnika danych, zapisuje wszystkie informacje o niezatwierdzonych transakcjach (zapamiętuje ostatnie sesje). Najczęściej

używane informacje (zmieniane/odczytywane) przenoszone są, w miarę możliwości, do pamięci operacyjnej maszyny. Dzięki temu dostęp do danych jest znacznie efektywniejszy niż w poprzednich edycjach. Poprawiono także mechanizmy działania przy zapytaniach dodawania/aktualizacji oraz usuwania danych – wykonywane są one znacznie szybciej dzięki mechanizmowi dedykowanej pamięci podręcznej. Od wersji serwera 5.5 jest to DOMYŚLNY SYSTEM SKŁADOWANIA
2) MyISAM – poprzedni domyślny system składowania danych. Silnik ten, podobnie jak InnoDB posiada najważniejsze cechy SQL, jak obsługa kluczy, autonumeracji, wartości unikatowych i indeksów. Nie posiada natomiast tak rozbudowanego systemu pamięci podręcznej, kolejkowania operacji dodawania/aktualizacji danych czy też kompresji danych w locie. Dodatkowo plik bazy danych ograniczony jest do 256TB danych. Silnik ten jest zalecany dla baz, z których odczytujemy dane, a rzadziej zapisujemy.
3) MEMORY (lub HEAP) – silnik zapisujący wszystkie dane jedynie w pamięci operacyjnej. Dlatego też nie jest on dobrym rozwiązaniem na wykorzystywanie go do zapisywania stałych danych. Ze względu na swoje właściwości (pamięć operacyjna jest znacznie szybsza od pamięci masowej) polecana jest do tworzenia baz tymczasowych. W chwili obecnej silnik ten traci jednak na znaczeniu poprzez właściwości InnoDB, który łączy cechy bazy pamięci masowej i operacyjnej.
4) CSV – system składowania oparty o zwykłe pliki tekstowe, w których dane są oddzielone średnikami. Wydajność nie jest najefektywniejsza jednak baza taka może być odczytana niemal przez dowolny program bazodanowy (w tym np. poprzez Libreoffice czy Excel).
5) ARCHIVE – silnik nadający się głównie dla dużych porcji przechowywanych danych. Przede wszystkim system nie obsługuje indeksowania danych. Domyślnie kompresuje wszystkie dane, jakie zostają zapisane pod kolejnymi rekordami. Nie nadaje się do standardowego składowania małych danych/krótkich informacji.
6) EXAMPLE – tego typu tabele nie przechowują żadnych danych. Służą do pokazania/zaprojektowania przyszłych baz danych. Charakterystyczną cechą tego rozwiązania jest tworzenie jedynie pliku bazy danych – bez pliku operacji czy też przechowującego informacji o zdarzeniach. Pozwala na tworzenie struktury tabel jednak nie pozwala na uzupełnianie ich danymi.
7) BLACKHOLE – silnik ten nie przyjmuje żadnych danych. Potrafi tworzyć tabele (strukturę), polecenia INSERT działają jak należy, jednak wszelkie dane są wyrzucane (są pochłaniane w nicość – stąd nazwa). Bazy tego typu pozwalają jednak na tworzenie replikacji danych. Przykładowo zapisujemy dane w tabeli na jednym serwerze z tym systemem składowania. Dane rzecz jasna ulegną na nim zniszczeniu, jednak w przypadku połączenia tegoż serwera z serwerem replikacji zostaną one przesłane na ten drugi serwer i tam zapisane. Rozwiązanie to może być więc tzw. bramą-pośrednikiem (proxy). Nawet w przypadku ataku na niego atakujący nie otrzyma zapisywanych w bazie danych (będą bezpieczne na drugim serwerze).
8) FEDERATED – system składowania zdalnego. Każde zapytanie wykonywane lokalnie działa na serwery stowarzyszone (konfederacja), które przechowują dane. Serwer lokalny przechowuje jedynie dane sesyjne w pliku bazy takim jak na serwerach zdalnych. Tego typu rozwiązanie przyjęło nazywać się rozmytym/rozproszonym. Rozwiązanie to powinno być szczególnie przydatne w takich instytucjach jak sieci sklepów, banki czy placówki pocztowe, gdzie dane zapisywane lokalnie powinny być dostępne poprzez jeden, centralny serwer.
9) MERGE – ten typ silnika przeznaczony jest do użytku z tabelami MyISAM. Pozwala on na łączenie wielu (kolekcja) tabel, w tym na różnych serwerach, dzięki czemu pracują one jako jedna. Tabele te muszą jednak być identyczne pod względem struktury, opcji i parametrów. Charakterystyczną cechą tego rozwiązania jest tworzenie przez serwer 2 plików bazy – jeden lokalny, drugi przechowujący kopię baz zdalnych.
10) NDBCLUSTER – silnik oparty głównie na sieci niezależnych serwerów (NDB – Network DataBase). System ten zakłada chmurę połączonych ze sobą poprzez sieć maszyn, w których dane przechowywane są w pamięci operacyjnej systemów (tak jak w silniku MEMEORY). W tym jednak przypadku nie mamy bezpośredniej kontroli nad każdym z serwerów (ani dostępu do nich). Zapytania oraz konfigurację wykonuje się poprzez serwer pośredniczący (zarządzający).

Dodatkowo, na potrzeby aplikacji korzystających z bazy danych w tym systemie przewidziana jest możliwość włączenia do niego serwerów – węzłów, które pozwalają na odpytywanie bezpośrednio utworzonej chmury danych (oraz zapisu do niej). Węzły te są również bezpośrednio kontrolowane przez serwer-zarządcę. Węzły mogą przechowywać kopie lustrzane baz danych oraz przechowywać kolejkę pytań, które z jakichkolwiek powodów nie mogły zostać wykonane na klastrze danych. Dzięki temu zwiększa się bezawaryjność serwera oraz integralność bazy danych.

2. Typy zmiennych w MySQL.

Standard SQL definiuje wiele typów zmiennych. Niektóre z nich mogą przy pierwszym kontakcie wydawać się zdublowane – jednak jest to mylne wrażenie. Niektóre z typów są szybkie w chwili przeszukiwania jednak mało pojemne; inne potrafią zgromadzić sporą ilość danych (nawet 4 GiB w jednym polu), jednak przeszukiwanie po nich byłoby bardzo wolne. Dzięki mnogości typów wprawny projektant bazy może odpowiednio wyważyć rozmiar baz oraz efektywność jej działania.

W MySQL występują następujące typy danych:

a) podstawowe

INT – najbardziej popularny typ zmiennych liczbowych; pozwala na przechowywanie zmiennych 4- bajtowych (2^{32}), co pozwala na zachowanie liczby ponad 4 miliardowej (ewentualnie połowę jeżeli chcemy zachowywać także wartości ujemne). Możemy decydować ile maksymalnie cyfr będzie mogła posiadać nasza zmienna (np. INT(8) oznacza, że można będzie zapisać liczbę 99999999)

VARCHAR – jeden z popularniejszych typów znakowych. Pozwala na gromadzenie zmiennych, które mają jedynie sprecyzowaną maksymalną ilość znaków (przedział od 0 do 65535 znaków). Pole z tym typem zajmie w bazie tylko tyle bajtów (znaków) ile zostanie do niego wprowadzonych. Reszta przestrzeni zostanie przesunięta do wykorzystania na inne dane. Zmienna jest efektywna w składowaniu danych jednak mniej efektywna podczas przeszukiwania – zmienne posiadające mniej znaków muszą przez bazę zostać powiększone do tej samej ilości bajtów. Przykład:

VARCHAR(20) – zmienna będzie mogła zawierać od 0 do 20 znaków

VARCHAR(500) – zmienna będzie mogła zawierać od 0 do 500 znaków

TEXT – typ do przechowywania zmiennych znakowych posiadający odgórny limit w postaci 65535 znaków. Domyślnie rezerwuje na każdy znak po 2 bajty. Służy głównie do przechowywania krótkich wiadomości, artykułów czy wpisów na stronie bądź programie. NIE PODAJE SIĘ JEGO DŁUGOŚCI (wystarczy napisać sam TEXT).

DATE – typ przechowuje datę w formacie YYYY-MM-DD (rok-miesiąc-dzień). Zmienna potrzebuje na to 3 bajty.

b) numeryczne

TINYINT – jednobajtowa liczba całkowita (2^{8}); zakres wartości -128 – +127 lub 0 – 255 w wersji bezznakowej

SMALLINT – 2 bajtowa liczba całkowita (2^{16})

MEDIUMINT – 3 bajtowa liczba całkowita (2^{24})

BIGINT – 8 bajtowa liczba całkowita (2^{64})

DECIMAL – liczba stałoprzecinkowa – możemy określać zarówno liczbę podstawy jak i części ułamkowej (tutaj odpowiednio podstawione M oraz D). M maksymalnie może wynieść 65 (domyślnie 10), natomiast D 30 (domyślnie 0). Przykład implementacji:

DECIMAL(5,2) – pozwala zapisywać liczby takie jak 3450.34, 123.01, 80000.56

DECIMAL(3,4) – zapisuje np. 345.0001, 120.3456

FLOAT - liczby zmiennoprzecinkowe pojedynczej precyzji (2^{32}).

DOUBLE – liczby zmiennoprzecinkowe podwójnej precyzji (2^{64}).

REAL – synonim double (dodany ze względu na często pojawianie się tego typu w różnych językach programowania). Oznacza zbiór liczb rzeczywistych.

BIT – zmienna typu pole bitowe; pozwala na przechowywanie do 64. Zmienne te są szczególnie

przydatne w przypadku przechowywania różnego rodzaju stanów (np. funkcji użytkowników na stronie WWW lub przechowywanie stanów wejść bądź wyjść urządzeń). Przykładem może być zapis w takiej zmiennej liczby 8 co oznacza, że 4 bit posiada wartość 1 (a bit 4 może być połączony np. z funkcją administratorską).
BOOLEAN – zmienna przechowująca dwie wartości – prawdę lub fałsz. Tak naprawę jest synonimem wartości TINYINT(1) i działa następująco: wartość 0 to fałsz, każda wartość powyżej 0 (1, 2.. 9) to prawda
SERIAL – zmienna jest tzw. aliasem unikatowym. Tego typu zmienna może być nadana tylko jedna w ramach tabeli. Pole z tym typem będzie posiadało następujące cechy: typ BIGINT(20), bez znaku (tylko wartości dodatnie) oraz zmienna ta będzie automatycznie powiększana o 1 za każdym dodaniem nowego rekordu (AUTO_INCREMENT); tym samym zmienna posiada swój indeks (wartość w tym polu musi być unikatowa, tj. nie powtarzać się w żadnym z rekordów). WAŻNE! Po utworzeniu pola o tej wartości nie będzie ono posiadało WARTOŚCI SERIAL! W widoku struktury będzie ono widziane jako BIGINT(20). W rzeczywistości bowiem SERIAL nie istnieje!

c) data i czas
DATETIME – przechowuje datę oraz czas (w formacie YYYY-MM-DD HH:MM:SS.[+milisekundy]). Wielkość pojedynczego pola to 8 bajtów + miejsce na milisekundy (opcjonalne).
TIMESTAPM – czas przechowywany w znaczniku czasu systemu UNIX. Pozwala na zapis daty od 1970-01-01 00:00:01.000000 do 2038-01-19 03:14:07.999999. Wielkość pojedynczego pola to 4 bajty + miejsce na milisekundy (opcjonalne).
TIME – przechowuje sam czas (00:00:00.000000). Zajmuje 3 bajty + miejsce na milisekundy (opcjonalne).
YEAR – przechowuje informacje o roku (YYYY). Zajmuje 1 bajt.

d) zmienne znakowe
CHAR – zmienna przechowująca znaki (0-65535 znaków). W przeciwieństwie do VARCHAR wskazana ilość przechowywanych znaków (np. 10) jest STAŁA – w przypadku krótszego ciągu znakowego (np. 5 znaków) pozostałe 5 znaków zostanie uzupełnionych zerowymi bajtami (NULL). Zmienna ta jest szybka przy przeszukiwaniu jednak znacznie mniej efektywna w przechowywaniu.
TINYTEXT – przechowuje od zera do 255 znaków. Zmienna dobiera swoją wartość automatycznie. Tego typu zmienna, podobnie jak TEXT, nadaje się do przechowywania danych, które są pobierane z bazy jednak nie jest na nich często wykonywane przeszukiwanie.
MEDIUMTEXT – pozwala na przechowywanie do 16,7 mln znaków (2^{24})
LONGTEXT – pozwala na przechowanie do 4,3 mld znaków (2^{32}).

e) zmienne binarne – WSZYSTKIE PONIŻSZE WARTOŚCI stworzone zostały do przechowywania wartości binarnych, tj. plików graficznych, wykonywalnych i innych, które posiadają bajty nie drukowane. Przechowywanie tych wartości pozwala na oszczędzenie miejsca na serwerze (baza kompresuje tego typu dane), ponadto wprowadza porządek w systemie plików (pliki wysyłane na serwer lądują w uporządkowanej tabeli).
BINARY – zmienna jest odpowiednikiem CHAR, jednak przechowuje zmienne binarne (niekoniecznie czytelne dla nas).
VARBINARY – odpowiednik VARCHAR; przechowuje wartości binarne
TINYBLOB – odpowiednik TINYTEXT (do 255 bajtów)
BLOB – odpowiednik TEXT (do 65535 znaków, czyli ok 64 kB danych)
MEDIUMBLOB – odpowiednik MEDIUTEXT (do 16 MB danych)
LONGBLOB – odpowiednik LONGTEXT (do 4 GB danych)

f) zmienne pseudotablicowe (zbiory)
ENUM – pozwala na utworzenie zbioru wartości tekstowych. Zbiór tego typu jest niezwykle efektowny gdyż nie porównuje się w nim wartości tekstowych lecz poprzez wartość binarną. Tego

typu zmienną dobrze zastosować w przypadku ograniczonej ilości (skończonej) wartości mogących być przypisanymi do wskazanego pola.
Przykład implementacji w bazie:
nazwa_pola ENUM('wartosc1', 'wartos2', 'wartosc3')

SET – pozwala na utworzenie zbioru dozwolonych wartości wskazanego pola. Dzięki tej zmiennej możemy ograniczyć wartości zmiennych tekstowych np. do stosowania określonych znaków, wyrazów bądź wyrażeń. Przykładowo implementacja:
zmienna SET ('a','b','c','d')
spowoduje, że w pole zmienna będzie można zapisać następujące wartości:
'',
'a',
'b',
'a,b',
'c',
'd',
'c,d';
'a,c',
'a,d',
'a,b,c'
….
czyli wszystkie możliwe kombinacje 4 podanych liter. Podobnie byłoby z wyrazami. Zestaw ograniczony jest do 64 pozycji (pole bitowe). Jedna wartość = jeden bit; łączenie bitów zezwala na dodanie kolejnych wartości do ciągu (po przecinku).

g) zmienne przestrzenne – tego typu zmienne są przeważnie klasą (obiektem); stosuje się je do opisu zarówno brył matematycznych jak i współrzędnych na mapach czy planach. Do bazy zostały dodane celem lepszego przeszukiwania tego typu wartości (przeszukiwanie bezpośrednie po logitude/latitude jest znacznie efektywniejsze niż np. wyciąganie tego typu danych z ciągu tekstowego/binarnego).
GEOMETRY – tego typu pole może przechowywać dowolną zmienną przestrzenną dostępną w bazie danych. Automatycznie dostosowuje się do wprowadzonych danych. Klasa podstawowa (określana przez dokumentację jako korzeń pozostałych)
POINT – zmienna przechowująca współrzędne pojedynczego punktu (zwyczajowo zwanego X oraz Y).
LINESTRING – zmienna przechowuje informacje na temat przebiegu odcinka wedle ustalonych punktów. Oznacza to więc, że zmienna ta (klasa) przechowuje jeden lub więcej danych punktowych (POINT).
POLYGON – kolejna zmienna budowana z klasy POINT. Przechowuje informacje na temat budowy figury geometrycznej (kolejne punkty wyznaczają jej kształt).
MULTIPOINT – pozwala na przechowywanie więcej niż jednego punktu
MULTILINESTRING – pozwala na przechowywanie informacji na temat wielu odcinków w ramach pojedynczej zmiennej
MULTIPOLYGON – pozwala na przechowanie informacji o wielu figurach utworzonych ze zbioru punktów
GEMETRYCOLLECTION – pozwala na zebranie dowolnych typów zmiennych przestrzennych w ramach jednego pola; wartości mogą być mieszane.

3. Podstawowe polecenia DDL

a) CREATE DATABASE – polecenie to niczym nie różni się od standardu SQL (trzeba pamiętać o odwróconych apostrofach dla nazwy bazy); tworzy bazę danych.

Ogólna składnia polecenia

CREATE DATABASE <nazwa_bazy> [CHARACTER SET = <format>] COLLATE = <format>]

jak widać opcjonalnie możemy ustawić domyślne kodowanie dla bazy danych oraz metodę porównywania napisów (praktycznie wystarczy ustawić metodę porównywania napisów, a kodowanie zmieni się na wybrany format porównania).

Przykład:
CREATE DATABASE `nowa_baza`;

b) DROP DATABASE – usuwa wskazaną bazę danych
Przykład:
DROP DATABASE `nowa_baza`;
c) CREATE TABLE – tworzy nową tabelę w aktualnie wybranej bazie danych. Polecenie posiada wiele dodatkowych parametrów oraz przełączników. Ogólny wzorzec tego zapytania można zapisać w następujący sposób:

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <nazwa_tabeli> (<definicja_pola1>, <definicja_pola2>... <definicja_polan>) <dodatkowe_opcje_tabeli> <dodatkowe_opcje_składowania> [<zapytanie_select_dodające_tresc_dotabeli>]

Poszczególne elementy tego polecenia zostaną omówione na przykładach (celem lepszego zrozumienia).

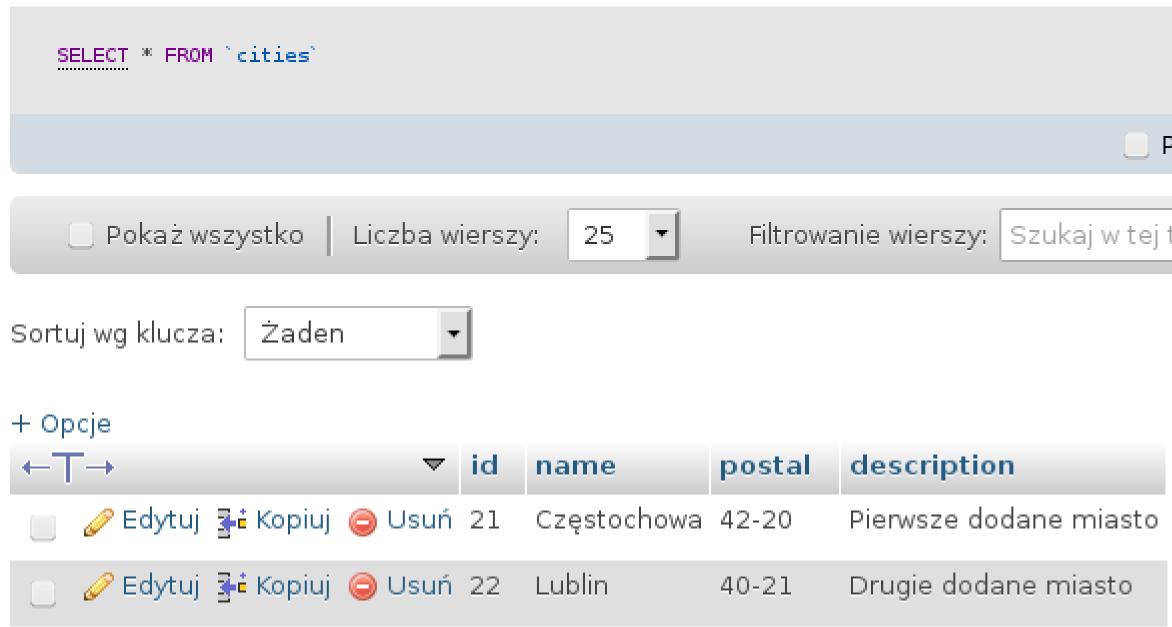
Jak można zauważyć, <nazwa_tabeli> to nazwa, pod jaką ma zostać zapisana nasza tabela w bazie danych. Z kolei <definicja_pola> to kolejne pola (kolumny) dodawane do naszej tabeli.
Przykład:
CREATE TABLE `artykul` (`id` SERIAL, `skrot_inf` VARCHAR(100), `tresc` TEXT, `widoczny` TINYINT(1) UNSIGNED DEFAULT 0);

Polecenie to tworzy tabelę artykuł posiadającą 4 kolumny – **id**, które będzie polem identyfikującym dany wiersz, **skrot_inf**, w którym będą przechowywane streszczenia artykułów, **tresc** przechowująca artykuł oraz **widoczny**, które to pole pełni rolę semafora (przełącznika) decydującego czy artykuł ma być widoczny czy nie (jest to zmienna stworzona pod aplikację korzystającą z baz danych; polecenie SELECT wyświetli wszystkie nie skaskowane rekordy z bazy – bez względu na zawartość tego pola). Proszę zauważyć, że pole widoczny domyślnie dodaje wartość 0 do każdego rekordu, w którym wartość dla tej kolumny nie byłaby ustawiana przez zapytanie.

Tworzona tabela może zostać wypełniona danymi. Wystarczy po definicji nowej tabeli dodać zapytanie SELECT, które pozwoli na wybranie i zapełnienie wskazanych pól w nowej tabeli.

Przykład:
CREATE TABLE `miasta` (`id` SERIAL, `nazwa` VARCHAR(200)) AS SELECT `name` AS 'nazwa' FROM `cities`;

zawartość tabeli cities



Zawartość utworzonej tabeli miasta:

wcześniej były dodane do innej tabeli. Wartości można pobierać także z innych baz danych (i należących do nich tabel).

Jeżeli chcemy jakąś tabelę skopiować (jej strukturę, bez danych) to można posłużyć się operatorem LIKE. Przykład:
CREATE TABLE `uzytkownicy` LIKE `wordpress`.`wp_users`;

Efektem powyższego polecenia będzie utworzenie tabeli użytkownicy w aktualnej bazie danych (tabela będzie nazywać się uzytkownicy). Tabela ta wszystkie kolumny, typy oraz parametry i przełączniki odziedziczy po wskazanej przez nas tabeli (w tym wypadku z innej bazy danych).

Gdybyśmy próbowali ponownie dodać tę samą tablicę spowoduje to błąd SQL i zwrócenie błędu. Dzieje się tak ze względów bezpieczeństwa – potencjalnie nadpisywana tabela mogłaby przecież zawierać dodatkowe dane. W przypadku skryptu błąd ten zatrzymałby wykonywanie go. Dlatego przy tworzeniu tabel można dodać klazulę IF NOT EXISTS. Powoduje ona, że w chwili, kiedy interpreter odnajdzie już bazę o tej nazwie pominie jej tworzenie. Przykład:

```
CREATE TABLE IF NOT EXISTS `uzytkownicy` LIKE `wordpress`.`wp_users`;
```

Teraz MySQL pozwoli na wykonanie skryptu i nie zwróci błędu pomimo że nie utworzył nowej tabeli.

Przy tworzeniu tabeli możemy również nadać tabeli dodatkowe opcje. Najczęściej nie zmienia się żadnych opcji tabeli (pozostawia się wartości domyślne) jednak czasami może zajść potrzeba zmiany np. silnika bazy bądź metody porównywania napisów. Przykład polecenia zmiany opcji tabeli:

```
CREATE TABLE IF NOT EXISTS `uzytkownicy2` (`id` INT(10) COMMENT 'identyfikator bazy')
ENGINE=MyISAM COMMENT='Prosta baza danych' COLLATE='utf8_general_ci';
```

Polecenie to utworzy tabele z silnikiem MyISAM, komentarzem oraz porównywaniem napisów w UTF-8 metodą ogólną. Innymi ciekawymi opcjami tabeli są:
AUTO_INCREMENT=<wartosc> - możemy wybrać startową wartość autonumeracji
CHARACTER SET <format> - domyślne kodowanie znaków tabeli
MAX_ROWS=<wartosc> - maksymalna ilość wierszy, jaką planujemy przechowywać w tabeli
MIN_ROWS=<wartosc> - minimalna ilość wierzy, jaką planujemy przechowywać na serwerze; zmienna ta posiada charakter informacyjny dla bazy (rezerwacja zasobów)

Pozostałe opcje są rzadziej zmieniane/przypisane są ściśle do silnika baz danych. Ich lista znajduje się w dokumentacji - https://dev.mysql.com/doc/refman/5.5/en/create-table.html

Tabele można dzielić na partycje (tak jak dzieli się na partycje dysk HDD/SSD). Partycje pozwalają na rozlokowanie danych np. po wartości określonych kolumn (niekoniecznie kluczy), takich jak daty, początkowe litery nazw (mogą być całe wyrazy) czy liczbach. Dzięki temu, w przypadku naprawdę rozbudowanej bazy danych można szybciej przeszukiwać dane – mechanizm przeszukuje tylko fragment bazy (partycję) zamiast wszystkie wartości od 1 do ostatniego rekordu. Każda partycja może mieć swoją nazwę, dzięki której użytkownik sam może wskazać rejon przeszukiwania (nazwy partycji zapisywane są pod konkretnymi nazwami PARTITION_NAME) dla bazy.

Przykład:

```
CREATE TABLE `uzytkownicy` (`id` INT(10), `imie` VARCHAR(30), `nazwisko`
VARCHAR(30), `wiek` INT(3)) PARTITION BY RANGE (`wiek`) (
	PARTITION grupa1 VALUES LESS THAN (6),
	PARTITION grupa2 VALUES LESS THAN (15),
	PARTITION grupa3 VALUES LESS THAN (21),
	PARTITION grupa4 VALUES LESS THAN (30)
);
```

Powyższe zapytanie utworzy nam nową tabele uzytkownicy, w której wartości będą rozłożone w partycjach w zależności od podanego wieku. Załóżmy teraz dodanie takich oto wartości

```
INSERT INTO `uzytkownicy` VALUES (1, 'Adam', 'Nowak', 45),
(2, 'Janina', 'Połać', 20),
(3, 'Lucjan', 'Dobrowolski', 30),
```

(4, 'Marianna', 'Kowalska', 29),
(5, 'Dobromir', 'Śpiewak', 15);

Okaże się, że serwer zwróci błąd – nie ma partycji, która umożliwiałaby dodanie wartości 45 (podobnie stanie się z wartością 30). Jak więc poradzić sobie z tym problemem? Wystarczy zmodyfikować naszą tabelę (w poniższym wypadku wyrzucimy poprzednią, po czym dodamy nową, prawidłową):

```
DROP TABLE `uzytkownicy`;
CREATE TABLE `uzytkownicy` (`id` INT(10), `imie` VARCHAR(30), `nazwisko` 
VARCHAR(30), `wiek` INT(3)) PARTITION BY RANGE (`wiek`) (
	PARTITION grupa1 VALUES LESS THAN (6),
	PARTITION grupa2 VALUES LESS THAN (15),
	PARTITION grupa3 VALUES LESS THAN (21),
	PARTITION grupa4 VALUES LESS THAN (30),
	PARTITION grupa5 VALUES LESS THAN (MAXVALUE)
);
```

Teraz dodanie wcześniejszych wartości jest możliwe. Odczytanie ilości wartości zgromadzonych w poszczególnych partycjach możliwe jest poprzez takie zapytanie:

```
SELECT `PARTITION_NAME`, `TABLE_ROWS` FROM 
`INFORMATION_SCHEMA`.`PARTITIONS` WHERE `TABLE_NAME`='uzytkownicy';
```

```
SELECT `PARTITION_NAME` , `TABLE_ROWS` FROM `INFORMATION_SCHEMA` . `PARTITIONS` WHERE `TABLE_NAME` = 'uzytkownicy'
```

Profilowanie [Edi

Pokaż wszystko | Liczba wierszy: 25 Filtrowanie wierszy: Szukaj w tej tabeli

+ Opcje

PARTITION_NAME	TABLE_ROWS
grupa1	0
grupa2	0
grupa3	2
grupa4	1
grupa5	2

Jak widać w grupie 3 mamy zapisane dwa rekordy (wiek 15 i 20) jeden w grupie 4 (29), reszta natomiast trafiła do grupy 5 (30 i więcej).

Innym sposobem partycjonowania jest wrzucanie wartości z listy – jeżeli wartość dodawana odpowiada pozycji na danej liście to jest tam dodawana jako rekord.

Przykład:

```
CREATE TABLE `uzytkownicy` (`id` INT(10), `imie` VARCHAR(30), `nazwisko` 
VARCHAR(30), `wiek` INT(3)) PARTITION BY LIST (`wiek`) (
	PARTITION grupa1 VALUES IN (6,10,11,12),
	PARTITION grupa2 VALUES IN (15,29),
	PARTITION grupa3 VALUES IN (20,22),
	PARTITION grupa4 VALUES IN (30)
);
```

```
INSERT INTO `uzytkownicy` VALUES (1, 'Adam', 'Nowak', 22),
(2, 'Janina', 'Połać', 20),
(3, 'Lucjan', 'Dobrowolski', 30),
(4, 'Marianna', 'Kowalska', 29),
(5, 'Dobromir', 'Śpiewak', 15);
```

Proszę zauważyć, że mamy tutaj do czynienia z listą zamkniętą – tylko wartości ściśle odpowiadające podanym w nawiasach mogą znaleźć się we wskazanej partycji. Podobnie jak poprzednio nie ma możliwości dodania wartości nie wchodzących w skład którejkolwiek listy.

Następna możliwość partycjonowania pozwala na łączenie dwóch przedstawionych metod. Różnica polega na tym, że wpływ na przydział danych do partycji może mieć więcej niż jedna kolumna (pole).

Przykład:

```
CREATE TABLE `uzytkownicy` (`id` INT(10), `imie` VARCHAR(30), `nazwisko`
VARCHAR(30), `wiek` INT(3), `nr_ewidencji` CHAR(4)) PARTITION BY LIST COLUMNS
(`nr_ewidencji`) (
	PARTITION grupa1 VALUES IN ('AAA9', 'AAA7', 'BBB1'),
	PARTITION grupa2 VALUES IN ('BBB2', 'ABA1', 'CCC2'),
	PARTITION grupa3 VALUES IN ('DDD2', 'LZA5')
);

INSERT INTO `uzytkownicy` VALUES (1, 'Adam', 'Nowak', 22, 'CCC2'),
(2, 'Janina', 'Połać', 20, 'CCC2'),
(3, 'Lucjan', 'Dobrowolski', 30, 'LZA5'),
(4, 'Marianna', 'Kowalska', 29, 'AAA7'),
(5, 'Dobromir', 'Śpiewak', 15, 'BBB1');
```

Tym razem do testowanej tabeli doszła nowa kolumna – nr_ewidencji. W tym wypadku bez partycjonowania po kolumnie otrzymalibyśmy błąd, że listy wartości muszą zawierać się w typie INT. Po aktualizacji zapytania (dodanie słowa COLUMNS) błąd zniknął i możemy dodawać do listy także wyrazy.

Przykład:

```
CREATE TABLE `uzytkownicy` (`id` INT(10), `imie` VARCHAR(30), `nazwisko`
VARCHAR(30), `wiek` INT(3), `nr_ewidencji` CHAR(4)) PARTITION BY LIST
COLUMNS(`wiek`, `nr_ewidencji`) (
	PARTITION grupa1 VALUES IN ((12, 'AAA4'), (22, 'BBB1')),
	PARTITION grupa2 VALUES IN ((30, 'CCC2'), (40, 'LZA5'))
);

INSERT INTO `uzytkownicy` VALUES (1, 'Adam', 'Nowak', 30, 'CCC2'),
(2, 'Janina', 'Połać', 30, 'CCC2'),
(3, 'Lucjan', 'Dobrowolski', 40, 'LZA5'),
(4, 'Marianna', 'Kowalska', 12, 'AAA4'),
(5, 'Dobromir', 'Śpiewak', 22, 'BBB1');
```

Przykład pokazuje możliwość dodawania więcej niż jednej kolumny w partycjonowaniu po liście. Pola nie muszą mieć tych samych typów.

Przykład:

```
CREATE TABLE `uzytkownicy` (`id` INT(10), `imie` VARCHAR(30), `nazwisko`
VARCHAR(30), `wiek` INT(3), `nr_ewidencji` CHAR(4)) PARTITION BY RANGE
COLUMNS(`wiek`, `nr_ewidencji`) (
	PARTITION grupa1 VALUES LESS THAN (12, 'AAA4'),
	PARTITION grupa2 VALUES LESS THAN (22, 'BBB1'),
	PARTITION grupa3 VALUES LESS THAN (30, 'CCC2'),
	PARTITION grupa4 VALUES LESS THAN (40, 'LZA5')
);

INSERT INTO `uzytkownicy` VALUES (1, 'Adam', 'Nowak', 22, 'CCC2'),
(2, 'Janina', 'Połać', 20, 'CCC2'),
(3, 'Lucjan', 'Dobrowolski', 30, 'LZA5'),
(4, 'Marianna', 'Kowalska', 29, 'AAA7'),
(5, 'Dobromir', 'Śpiewak', 15, 'BBB1');
```

W tym wypadku mamy brane pod uwagę są pola wiek oraz nr_ewidencji. Takie rozwiązanie pozwala na późniejsze szybsze przeszukiwanie kolumn i zwracanie wyników w przypadku odpytywania poprzez pola, które zostały ustawione jako wyznaczniki partycji.

MySQL posiada także możliwość samoczynnego partycjonowania danych (rozkładania ich na partycje). Do tego celu wykorzystywana jest partycja typu HASH (z angielskiego można to tłumaczyć jako siekanie, krzyżowanie/kratowanie). Partycja tego typu pozwala na tworzenie partycji zarówno po wartościach pól jak i po wyrażeniach. Najlepszym przykładem wyrażenia może być pole zawierające datę. Pole pozwala zamieniać datę np. na numer dnia, wyciągać z daty rok czy miesiąc.

Abstrahując od formy hashowania (dla tego określenia nie przyjął się niestety żaden polski odpowiednik) baza na podstawie uzyskanego wyniku rozdziela dane po partycjach automatycznie – projektant ma wpływ jedynie na ilość tworzonych partycji.

Przykład:

```
CREATE TABLE `uzytkownicy` (`id` INT(10), `imie` VARCHAR(30), `nazwisko`
VARCHAR(30), `wiek` INT(3)) PARTITION BY HASH(`wiek`) PARTITIONS 2;

INSERT INTO `uzytkownicy` VALUES (1, 'Adam', 'Nowak', 22),
(2, 'Janina', 'Połać', 20),
(3, 'Lucjan', 'Dobrowolski', 30),
(4, 'Marianna', 'Kowalska', 29),
(5, 'Dobromir', 'Śpiewak', 15),
(6, 'Sławomir', 'Śpiewak', 67),
(7, 'Zuzanna', 'Listkiewicz', 30),
(8, 'Izydor', 'Leśniewski', 15),
(9, 'Iwona', 'Adamczyk', 45),
(10, 'Kamila', 'Deresz', 13);
```

W tym momencie dane będą rozkładane w dwóch partycjach (w zależności od wartości pola wiek). Powyższe zapytanie spowoduje dodanie 4 pozycji do pierwszej partycji oraz 6 do drugiej

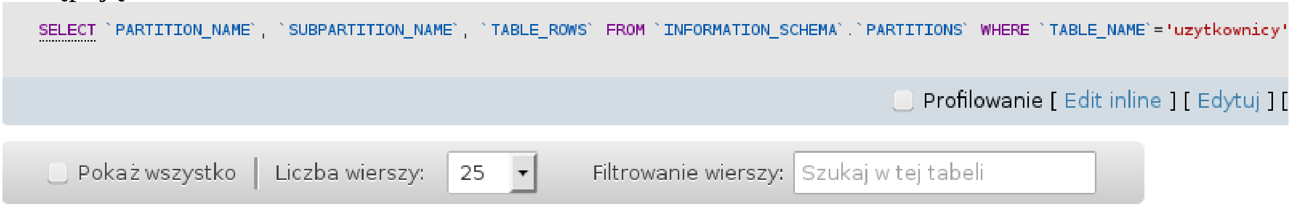
Odmianą partycji haszującej jest partycja liniowa haszująca (LINEAR HASH). Działanie obydwu różni zastosowany algorytm. Liniowa wersja jest efektywniejsza dla wyrażeń złożonych. Budowa samego zapytania jest identyczna (słowo HASH poprzedzone zostaje słowem LINEAR)

Ostatnim sposobem partycjonowania jest metoda po kluczu (KEY). Działa na podobne zasadzie do HASH (nawet występuje w dwóch odmianach – klucz oraz klucz liniowy) jednak w przeciwieństwie do hashowania działa na kluczach głównych bądź na kluczach indeksujących, których wartości określone są jako unikatowe (UNIQUE). Dla każdego klucza biorącego udział w partycjonowaniu danych baza sama dobiera wyrażenie haszujące dane, które później rozmieszczane są po odpowiednich partycjach.

Przykład:
 DROP TABLE `uzytkownicy`;
CREATE TABLE `uzytkownicy` (`id` INT(10) PRIMARY KEY, `imie` VARCHAR(30), `nazwisko` VARCHAR(30), `wiek` INT(3)) PARTITION BY KEY() PARTITIONS 5;

Dane umieszczane w bazie te same co w poprzednim przykładzie. Dane zostały rozmieszczone w 2 i piątej partycji (po 5). Pozostałe partycje pozostały puste.
Ostatnią metodą jest system podpartycji. Mechanizm działania przypomina tworzenie partycji rozszerzonej na dysku twardym. Partycja ta może posiadać jedną bądź więcej partycji logicznych co pozwala na lepsze planowanie rozmieszczenia danych na dysku twardym. Podobnie jest z system podpartycjonowania w MySQL.
Przykład:
CREATE TABLE `uzytkownicy` (`id` INT(10), `imie` VARCHAR(30), `nazwisko` VARCHAR(30), `wiek` INT(3))
PARTITION BY RANGE (`wiek`)
SUBPARTITION BY HASH(`wiek`)
SUBPARTITIONS 2 (
PARTITION grupa1 VALUES LESS THAN (15),
PARTITION grupa2 VALUES LESS THAN (25),
PARTITION grupa3 VALUES LESS THAN (MAXVALUE)
);

W powyższym przykładzie utworzone zostanie w sumie 6 partycji (3 główne zawierające 2 podpartycje). Po umieszczeniu 10 rekordów z poprzednich przykładów rozłożenie danych będzie następujące:



+ Opcje

PARTITION_NAME	SUBPARTITION_NAME	TABLE_ROWS
grupa1	grupa1sp0	0
grupa1	grupa1sp1	1
grupa2	grupa2sp0	2
grupa2	grupa2sp1	2
grupa3	grupa3sp0	2
grupa3	grupa3sp1	3

Na koniec pozostało wyjaśnienie klauzuli TEMPORARY. Jej zastosowanie ma miejsce w chwili, kiedy potrzebujemy utworzyć tabelę do tymczasowego przechowania określonej ilości wartości z innych tabel (np. połączenie rekordów, potrzeba skorzystania w procedurze/funkcji ze zmiennej

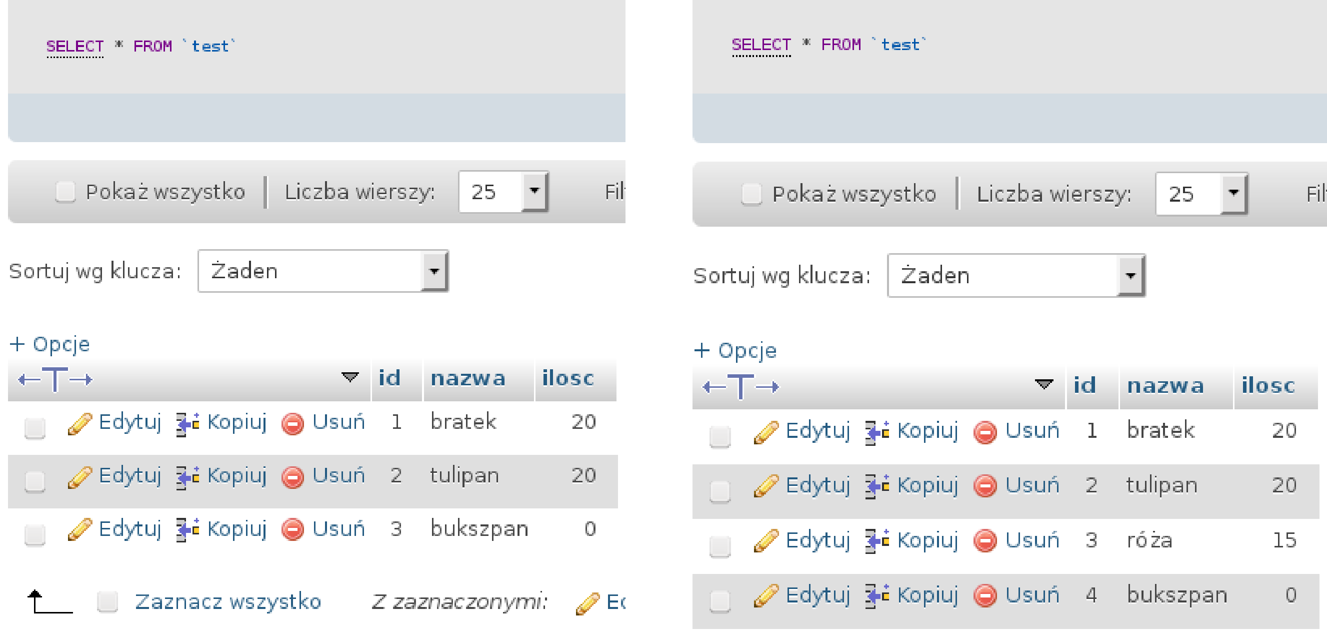
tablicowej, utworzenie nowej tablicy z wyników kilku zapytań itp.). Tablice tymczasowe widoczne są tylko dla sesji użytkownika, który je utworzył. Wraz z zakończeniem sesji tablice te, o ile nie zostały ręcznie skasowane, usuwane są bezpowrotnie z bazy danych.
Przykład:
CREATE TEMPORARY TABLE `uzytkownicy` (`id` INT(10), `name` VARCHAR(30));
INSERT INTO `uzytkownicy` VALUES (1, 'Mirek');
SELECT * FROM `uzytkownicy`;

Proszę zauważyć, że po ponownej próbie odpytania tabeli tymczasowej w phpMyAdmin otrzymamy komunikat iż takowa tabela nie istnieje w bazie! Oznacza to tyle, że sesja dla tabeli tymczasowych kończy się wraz z wykonaniem ostatniego polecenia w ciągu zapytań.

d) INSERT – polecenie pozwala na dodanie nowych wartości (jednej bądź kilku jednocześnie) do wskazanej tabeli. Składnia ogólna polecenia:

INSERT [LOW_PRIORITY/DELAYED/HIGH_PRIORITY] [IGNORE]
[INTO] <nazwa_tabeli> [(<nazwaPola1>, <nazwaPola2>, …, <nazwaPolaN>)]
{VALUE/VALUES} (<wartosc1>,<wartosc2>, …, <wartoscN>)[, (<wartosc1>,<wartosc2>, …, <wartoscN>) (…)] [ON DUPLICATE KEY UPDATE <nazwa_kolumny>=<wartosc> [,<nazwa_kolumny>=<wartosc>, (…)]]

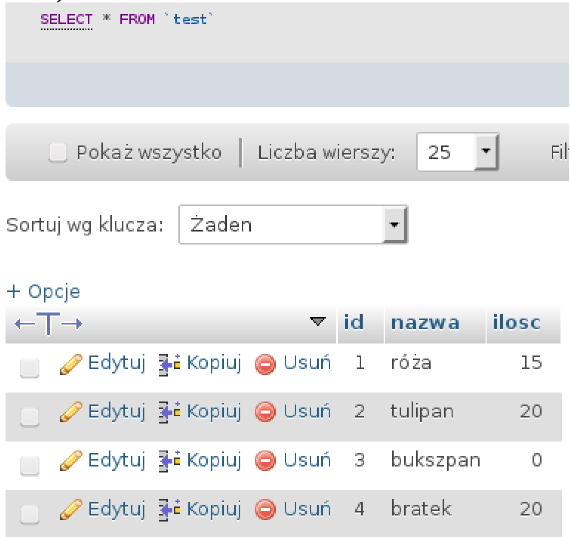
Polecenie jest mniej rozbudowane od polecenia tworzenia tabeli. Uwagę przykuwa ilość opcjonalnych klauzul (kwadratowe nawiasy).
Najprostszą postać polecenia mieliśmy zaprezentowaną w poprzednim podpunkcie (przy dodawaniu wartości do naszych tabel). Przyjrzyjmy się teraz opcjonalnym opcjom po klauzuli INSERT.
W pierwszym nawiasie mamy trzy opcje:
LOW_PRIORITY – zapytanie będzie czekało w kolejce na wykonanie; przed nim wykonają się inne zapytania INSERT oraz tabela nie może być odczytywana przez jakiegokolwiek klienta. W przypadku obciążonych systemów zapytanie może nigdy się nie wykonać!
DELAYED – zapytanie umieszczane jest w buforze zapytań. Dodawanie zawartości rozpoczyna się w chwili, gdy nikt inny nie korzysta z bazy. Jeżeli pojawia się nowe zapytanie do bazy, polecenie to jest zawieszane i kontynuowane po wykonaniu się tegoż nowego zapytania.
HIGH_PRIORITY – zapytanie wykonuje się natychmiast, nadpisując działania nawet innych aktualnie działających poleceń.
Kolejnym opcjonalnym słowem jest IGNORE. Pozwala ono, w przypadku wykonywania wielu poleceń, na zignorowanie błędu dodawania nowego rekordu. Przykładowo dodajemy nowy rekord posiadający duplikat wartości w kolumnie z atrybutem UNIQUE. INSERT w tym momencie wykonałby błąd. Z opisywaną klauzulą zignoruje błąd i wykona kolejne polecenia/dodania wierszy (o ile użytkownik takowe dołączył). Oczywiście wadliwy wiersz NIE DODA się do tabeli!
Słowo INTO należy do standardu SQL. W przypadku MySQL jest ono opcjonalne!
MySQL dopuszcza na zamienne wykorzystywanie kluazuli VALUE/VALUES (ten sam efekt!).
Wyrażenie ON DULICATE KEY UPDATE jest przydatne w chwili, gdy dodajemy nowe wiersze danych, w których mogą zostać powielone wartości pól unikatowych. W tym przypadku pola te zostaną nadpisane wskazaną wartością (np. powiększoną o 1).
Przykład:
INSERT INTO `test` VALUE (3, 'róża', 15) ON DUPLICATE KEY UPDATE `id`= `id` + 1;
Polecenie spróbuje dodać do bazy naszą różę. Ponieważ klucz id o wartości 3 już istnieje, polecenie nadpisze go nową wartością (4), a naszą różę doda z wartością 3 klucza id.



Zrzuty pokazują stan tabeli przed wykonaniem polecenia oraz tuż po wykonaniu.

Powyższy przypadek będzie działał jedynie wtedy, gdy będziemy mieli w bazie wartości 1,2,3 i zechcemy jednocześnie dodać wiersz z wartością id 3. Jeżeli próbowalibyśmy dodać wiersz z id 1 to otrzymalibyśmy błąd zmiany wartości – baza próbowałaby aktualizować wartość 1 na 2, lecz wartość 2 także istnieje w bazie. Oznacza to oczywiście błąd i zakończenie działania.
W związku z tym można posłużyć się takim oto (jak na razie skomplikowanym) poleceniem, które przegląda wartości id w bazie i do maksymalnej zapisanej w bazie wartości dodaje 1:
DO (SELECT DISTINCT @zm:=(SELECT MAX(`id`) FROM `test`) FROM `test`);
INSERT INTO `test` (`id`, `nazwa`, `ilosc`) VALUE (1, 'róża', 15) ON DUPLICATE KEY UPDATE `id`=@zm + 1

Polecenie najpierw wykonuje polecenie podwójne polecenie SELECT; pierwszy SELECT przypisuje zmiennej zm wartość drugiego polecenie SELECT (wybierającego wartość maksymalną z bazy). Ponieważ wartość maksymalna pojawiłaby się tyle samo razy ile wierszy jest w bazie użyto słowa DISTINCT (wybiera tylko wartości unikatowe). W celu nie wyświetlania wyniku polecenia wybierania danych zamknięto je w polecenie DO. Na koniec zmienna wykorzystywana jest do powiększenia indeksu w przypadku wykrycia duplikatu. Efekt działania powyższego polecenia (baza wyjściowa jak poprzednio):



4. Podstawowe polecenie MDL

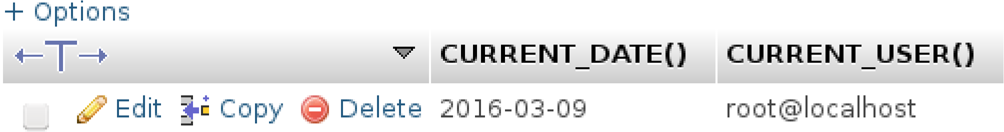
a) SELECT – jedno z najbardziej podstawowych poleceń pozwalających na odczyt wartości z bazy danych. Ogólna postać polecenia:

```
SELECT [ALL/DISTINCT/DISTINCTROW] [HIGH_PRIORITY] [STRAIGHT_JOIN]
[SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
[SQL_CACHE/SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]  <kolumna_badz_dzialanie1>
[AS <nazwa_kolumny>], <kolumna_badz_dzialanie2> [AS <nazwa_kolumny>],...,
<nazwa_badz_dzialanieN> [AS <nazwa_kolumny>]
[FROM <nazwa_tabeli_badz_lista_tabel>
        [WHERE <warunki>]
        [GROUP BY {nazwa_kolumny1/wyrazenie1/pozycja1} [ASC/DESC],
{nazwa_kolumny2/wyrazenie2/pozycja2} [ASC/DESC], …,
{nazwa_kolumnyN/wyrazenieN/pozycjaN} [ASC/DESC] [WITH ROLLUP]]
        [HAVING <wyrazenie>]
        [ORDER BY {nazwa_kolumny1/wyrazenie1/pozycja1} [ASC/DESC],
{nazwa_kolumny2/wyrazenie2/pozycja2} [ASC/DESC], …,
{nazwa_kolumnyN/wyrazenieN/pozycjaN} [ASC/DESC]]
        [LIMIT {[przesuniecie,] ilosc_wierszy/ilosc_wierszy OFFSET przesuniecie}]
        [PROCEDURE nazwa_procedury(argument1,argument2,...,argumentN)]
        [{INTO OUTFILE 'nazwa_pliku' [CHARACTER SET <format>] <opcje_eksportu>/
          INTO DUMPFILE 'nazwa_pliku' /
          INTO zmienna1 [, zmienna2,...,zmiennaN]}]
        [{FOR UPDATE/LOCK IN SHARE MODE}]
]
```

Jak widać zapytanie jest rozbudowane. Większość jego części stanowi opcjonalne polecenia (kwadratowe nawiasy), natomiast wymagane, aczkolwiek wybieralne opcje oznaczone zostały objęte nawiasami klamrowymi. Dla lepszej czytelności najbardziej rozbudowana klauzula dodatkowa została oznaczona szarym tłem, natomiast jej najbardziej rozbudowane klauzule dodatkowe oznaczono niebieskim kolorem czcionki. KOLOR NIE MA WPŁYWU na ważność klauzul!

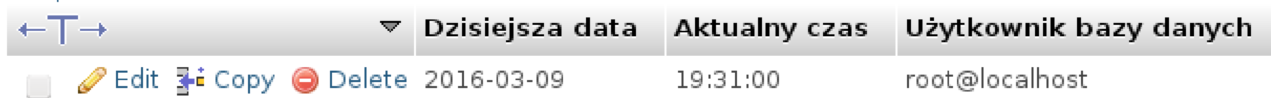
Podobnie jak w innych systemach bazodanowych (PostgreSQL, MS SQL) w MySQL nie jest wymagana klauzula FROM (i jej dodatkowe parametry oraz opcje). Rozwiązanie to jest przydatne w chwili, kiedy zadajemy pytanie do bazy np. o aktualną godzinę bądź chcemy uzyskać informację o wyniku jakiegoś działania matematycznego.
Przykład:
SELECT CURRENT_DATE(), CURRENT_USER();

Powyższe polecenie wyświetli aktualną datę oraz nazwę aktualnego użytkownika.



SELECT CURRENT_DATE() AS 'Dzisiejsza data', CURRENT_TIME() AS 'Aktualny czas', CURRENT_USER() AS 'Użytkownik bazy danych';

Wynik działania:


+ Options

	Dzisiejsza data	Aktualny czas	Użytkownik bazy danych
Edit Copy Delete	2016-03-09	19:31:00	root@localhost

INFORMACJA: W standardzie SQL (oraz niektórych rozwiązaniach – np. serwer Oracle) nie można opuścić klauzuli FROM; stad do podobnych zapytań o czas czy wynik działań matematycznych można „zapytać" tabelę DUAL, która tworzy tzw. puste odwołanie (dummy). MySQL również posiada taką bazę (gdyby ktoś wolał używać klauzuli FROM) aczkolwiek jej działanie w niczym nie różni od tego, w którym FROM zostaje pominięte.

Najprostszym, często używanym w małych tabelach zapytaniem jest:

SELECT * FROM `uzytkownicy`;

Zapytanie to wybiera wszystkie kolumny z tabeli (znak gwiazdki to gwarantuje), której nazwa znajduje się w klauzuli FROM (tutaj jest to tabela użytkownicy). Ponieważ klauzula WHERE nie została użyta ani też inna klauzula ograniczająca wybierania rekordów zapytanie to zwróci i wyświetli WSZYSTKIE rekordy z bazy:

+ Opcje

	id	imie	nazwisko	wiek
Edytuj Kopiuj Usuń	1	Adam	Nowak	22
Edytuj Kopiuj Usuń	2	Janina	Połać	20
Edytuj Kopiuj Usuń	3	Lucjan	Dobrowolski	30
Edytuj Kopiuj Usuń	4	Marianna	Kowalska	29
Edytuj Kopiuj Usuń	5	Dobromir	Śpiewak	15
Edytuj Kopiuj Usuń	6	Sławomir	Śpiewak	67
Edytuj Kopiuj Usuń	7	Zuzanna	Listkiewicz	30
Edytuj Kopiuj Usuń	8	Izydor	Leśniewski	15
Edytuj Kopiuj Usuń	9	Iwona	Adamczyk	45
Edytuj Kopiuj Usuń	10	Kamila	Deresz	13

Celem dalszego testowania polecenia SELECT wykonamy na naszej bazie testowej nowe operacje dodające nową tabelę miasta, dodające nową kolumnę w tabeli uzytkownicy oraz ustawiające zależności pomiędzy polem id tabeli miasta a nowo dodaną kolumną (dodany zostaje klucz główny). Polecenia użyte poniżej zostaną omówione w późniejszym terminie.

CREATE TABLE `miasta` (id INT(10) PRIMARY KEY, `nazwa` VARCHAR(200), `kod_pocztowy` CHAR(6) UNIQUE);
ALTER TABLE `uzytkownicy` ADD COLUMN `miasto` INT(10) INDEX;
ALTER TABLE `uzytkownicy` ADD CONSTRAINT `id_uzytkownicy_fk` FOREIGN KEY

(`miasto`) REFERENCES `miasta`(`id`) ON DELETE RESTRICT ON UPDATE RESTRICT;
INSERT INTO `miasta` (`nazwa`, `kod_pocztowy`) VALUES ('Częstochowa', '42-200'),
('Lublin', '33-140'),
('Warszawa', '01-145'),
('Warszawa', '01-150'),
('Warszawa', '00-145'),
('Opole', '40-678'),
('Częstochowa', '42-218'),
('Szczecin', '20-001'),
('Jarocin', '56-145'),
('Suwałki', '90-100'),
('Radom', '14-600');

Pozostaje jeszcze nadać komórkom (krotkom) nowej kolumny odpowiednie wartości. Polecenie aktualizacji również będzie szerzej omawiane później.

UPDATE `uzytkownicy` SET `miasto` = 3 WHERE `id` IN (2,5,7);
UPDATE `uzytkownicy` SET `miasto` = 8 WHERE `id` IN (1,3);
UPDATE `uzytkownicy` SET `miasto` = 5 WHERE `id` IN (4,6);
UPDATE `uzytkownicy` SET `miasto` = 4 WHERE `id` IN (8,9,10);

Gdybyśmy podali wartości spoza id w tabeli miasta spowodowałoby to oczywiście błąd bazy (opcja ścisła).

Teraz spróbujmy wybrać wartości zarówno z jednej jak i drugiej tabeli:
SELECT * FROM `uzytkownicy`, `miasta`;

Zapytanie rzeczywiście wyświetli wszystko z jednej jak i drugiej tabeli, przy czym druga z wymienionych tabel będzie tworzyła indeks dla wyświetlenia wartości z drugiej tabeli. Sumarycznie (10 rekordów w pierwszej i 11 rekordów w drugiej tabeli) da nam odpowiedź zawierającą 110 rekordów, gdzie każdy rekord z tabeli miast będzie powielony 10 razy (dla każdego rekordu tabeli użytkownicy).

Ponieważ tego typu wynik raczej nie będzie nas interesował można wyświetlić tylko interesujące nas wyniki (przykładowo będzie nas interesowała nazwa miasta i kod pocztowy tylko dla określonych wartości id miast)

SELECT `imie`, `nazwisko`, `nazwa`, `kod_pocztowy` FROM `uzytkownicy`, `miasta` WHERE `uzytkownicy`.`miasto` = 3;

W tym wypadku również otrzymaliśmy dane nie takie jak chcemy. Problem polega na tym, że miasto z id 3 (Warszawa) zostanie wybrana 33 razy (3 osoby mają przypisane do siebie to miasto; dodatkowo każdy z rekordów tabeli użytkownicy, jak poprzednio, jest trzykrotnie wywołany).

W celu naprawienia zapytania należy dodać jeszcze jeden warunek

SELECT `imie`, `nazwisko`, `nazwa`, `kod_pocztowy` FROM `uzytkownicy`, `miasta` WHERE `uzytkownicy`.`miasto` = 3 AND `uzytkownicy`.`miasto` = `miasta`.`id`

Teraz zapytanie zwróci upragniony wynik – osoby, które są związane z Warszawą (id 3)

INFORMACJA: Zapytanie, chociaż działa, nie jest zgodne z konwencją SQL. Poprawnie w

przypadku tego typu zapytań jest podawanie nazwy tabeli oraz pola w sekcji SELECT. Zatem w pełni poprawnie wyglądające zapytanie przybrałoby postać:
SELECT `uzytkownicy`.`imie`, `uzytkownicy`.`nazwisko`, `miasta`.`nazwa`, `miasta`.`kod_pocztowy` FROM `uzytkownicy`, `miasta` WHERE `uzytkownicy`.`miasto` = 3 AND `uzytkownicy`.`miasto` = `miasta`.`id`
Zapytanie to staje się jednak mało czytelne, zaś pisanie pełnej nazwy tabeli może być niekomfortowym rozwiązaniem. Dlatego przy nazwach baz również można zastosować zamienniki (aliasy):
SELECT `a`.`imie`, `a`.`nazwisko`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = 3 AND `a`.`miasto` = `b`.`id`;
należy przy tym pamiętać, że zamiast apostrofów przy tego typu aliasach stosuje się ich odwróconą wersję.
Stosowanie odwołań przez nazwy tabel (aliasów) zapobiega problemom w chwili gdy dwie tabele będą miały identycznie brzmiące nazwy pól.

Teraz wybierzmy wszystkich (i wszystko) z tabeli uzytkownicy i przypiszmy do tych rekordów nazwy miast oraz kody pocztowe
SELECT `a`.`id`, `a`.imie, `a`.`nazwisko`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = `b`.`id`;

Zapytania w bazie danych można grupować po określonych wartościach:
SELECT `a`.`id`, `a`.imie, `a`.`nazwisko`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = `b`.`id` GROUP BY `b`.`kod_pocztowy`;
Wynikiem działania będzie:

id	imie	nazwisko	nazwa	kod_pocztowy
4	Marianna	Kowalska	Warszawa	00-145
2	Janina	Połać	Warszawa	01-145
8	Izydor	Leśniewski	Warszawa	01-150
1	Adam	Nowak	Szczecin	20-001

Grupowania można dokonywać po większej ilości kolumn:
SELECT `a`.`id`, `a`.imie, `a`.`nazwisko`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = `b`.`id` GROUP BY `b`.`kod_pocztowy`, `a`.`nazwisko`;

Spróbujmy teraz posortować wyniki po kodach pocztowych:
SELECT `a`.`id`, `a`.imie, `a`.`nazwisko`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = `b`.`id` ORDER BY `b`.`kod_pocztowy`;

Zmiany najlepiej widać poprzez niepoukładane identyfikatory wierszy.
Grupowania można dokonywać po wielu kolumnach:
SELECT `a`.`id`, `a`.imie, `a`.`nazwisko`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = `b`.`id` ORDER BY `b`.`kod_pocztowy`,`a`.`nazwisko` ;

W tym wypadku sortowanie nastąpi po kodzie pocztowym oraz nazwisku.
Innym sposobem na sortowanie jest podanie numeru kolumny (w klauzuli SELECT), po którym będzie następowało grupowanie:
SELECT `a`.`id`, `a`.imie, `a`.`nazwisko`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = `b`.`id` ORDER BY 4,3;

W tym wypadku najpierw nastąpi sortowanie po nazwie miasta, po czym wyniki posortowane będą wedle nazwisk (**numerowanie kolumn rozpoczyna się od 1**).

Pogrupowane wyniki można sortować od najmniejszego do największego (rosnąco – ASCENDING) lub od największego do najmniejszego (malejąco – DESCENDING). Domyślnie sortowanie odbywa się rosnąco (ASC). Można je jednak zmienić dodając po nazwie kolumny w ORDER BY słowo DESC:

SELECT `a`.`id`, `a`.imie, `a`.`nazwisko`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = `b`.`id` ORDER BY 4,3 DESC;

W tym momencie nazwiska będą ułożone od Z do A (wcześniej od A do Z).

Istnieje możliwość łączenia ze sobą klauzul GROUP BY oraz ORDER BY:
SELECT `a`.`id`, `a`.imie, `a`.`nazwisko`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = `b`.`id` GROUP BY `b`.`kod_pocztowy`, `a`.`nazwisko` ORDER BY `a`.`nazwisko` ;

id	imie	nazwisko ▲ 1	nazwa	kod_pocztowy
9	Iwona	Adamczyk	Warszawa	01-150
10	Kamila	Deresz	Warszawa	01-150
3	Lucjan	Dobrowolski	Szczecin	20-001
4	Marianna	Kowalska	Warszawa	00-145
8	Izydor	Leśniewski	Warszawa	01-150
7	Zuzanna	Listkiewicz	Warszawa	01-145
1	Adam	Nowak	Szczecin	20-001
2	Janina	Połać	Warszawa	01-145
5	Dobromir	Śpiewak	Warszawa	01-145
6	Sławomir	Śpiewak	Warszawa	00-145

Można też odwrócić wyświetlanie wyników klucza głównego:

SELECT * FROM `uzytkownicy` ORDER BY `id` DESC;

Dla następnego przykładu nasza tabela uzytkownicy wymaga edycji:
ALTER TABLE `uzytkownicy` ADD COLUMN `komentarze` INT(4);
UPDATE `uzytkownicy` SET `komentarze`=FLOOR((RAND()*1000));

Powyższe zapytania kolejno tworzą nową kolumnę w tabeli uzytkownicy, po czym uzupełniają dane dla każdego rekordu wartością losową. Ponieważ funkcja RAND() w MySQL loduje liczby z przedziału 0..1, wynik ten został pomnożony o 1000 oraz zaokrąglony w dół do całości (funkcja FLOOR).

Teraz zobaczmy wynik naszego poprzedniego zapytania wzbogacony o wylosowane wartości komentarzy:
SELECT `a`.`id`, `a`.imie, `a`.`nazwisko`, `a`.`komentarze`, `b`.`nazwa`, `b`.`kod_pocztowy` FROM `uzytkownicy` AS `a`, `miasta` AS `b` WHERE `a`.`miasto` = `b`.`id`;

+ Opcje

id	imie	nazwisko	komentarze	nazwa	kod_pocztowy
1	Adam	Nowak	798	Szczecin	20-001
2	Janina	Połać	85	Warszawa	01-145
3	Lucjan	Dobrowolski	31	Szczecin	20-001
4	Marianna	Kowalska	901	Warszawa	00-145
5	Dobromir	Śpiewak	413	Warszawa	01-145
6	Sławomir	Śpiewak	362	Warszawa	00-145
7	Zuzanna	Listkiewicz	570	Warszawa	01-145
8	Izydor	Leśniewski	764	Warszawa	01-150
9	Iwona	Adamczyk	114	Warszawa	01-150
10	Kamila	Deresz	275	Warszawa	01-150

Standardowo można by odpytać bazę takim zapytaniem :
SELECT `uzytkownicy`.`id`, `imie`, `nazwisko`, MAX(`komentarze`), `nazwa`, `kod_pocztowy` FROM `uzytkownicy`, `miasta` WHERE `miasta`.`id` = `uzytkownicy`.`miasto` AND MAX(`komentarze`) > 300 GROUP BY `nazwisko`;
JEDNAK ZWRÓCI ONO BŁĄD! Dzieje się tak dlatego, że klauzula WHERE nie może posiadać warunków budowanych w oparciu o funkcję.

Do tego typu celów można wykorzystać dodatkową klauzulę HAVING. Pozwala ona właśnie między innymi na ustawianie warunków dla funkcji/agregatorów (sumatorów):

SELECT `uzytkownicy`.`id`, `imie`, `nazwisko`, MAX(`komentarze`), `nazwa`, `kod_pocztowy` FROM `uzytkownicy`, `miasta` WHERE `miasta`.`id` = `uzytkownicy`.`miasto` HAVING MAX(`komentarze`) > 300 ;

Jednak odpowiedź na zapytanie raczej nie będzie dla nas interesująca:

+ Opcje

id	imie	nazwisko	MAX(`komentarze`)	nazwa	kod_pocztowy
1	Adam	Nowak	901	Szczecin	20-001

Stało się tak dlatego, że funkcja MAX wybiera tylko najwyższy wynik (maksymalna wartość w kolumnie). Jeżeli dodatkowo ustawiliśmy warunek, iż wybieramy wartości większe od 300 z tej kolumny to otrzymaliśmy najwyższy wynik (i nic więcej). Dla kontrastu jeżeli zmienilibyśmy znak na mniejszość (<) to nie otrzymalibyśmy żadnych wyników – wybrany jest tylko jeden rekord (z największą wartością). W przypadku pogrupowania zapytania (po dowolnej kolumnie) otrzymamy wyniki z wartościami w kolumnie MAX bez wybrania największej wartości (funkcja wydaje się nie działać). Jednak jeżeli dodamy klauzulę HAVING:

SELECT `uzytkownicy`.`id`, `imie`, `nazwisko`, MAX(`komentarze`), `nazwa`, `kod_pocztowy` FROM `uzytkownicy`, `miasta` WHERE `miasta`.`id` = `uzytkownicy`.`miasto` GROUP BY 1 HAVING MAX(`komentarze`) > 300;

Wyniki będą wyświetlone tak jak byśmy sobie tego życzyli:

id	imie	nazwisko	MAX(`komentarze`)	nazwa	kod_pocztowy
1	Adam	Nowak	798	Szczecin	20-001
4	Marianna	Kowalska	901	Warszawa	00-145
5	Dobromir	Śpiewak	413	Warszawa	01-145
6	Sławomir	Śpiewak	362	Warszawa	00-145
7	Zuzanna	Listkiewicz	570	Warszawa	01-145
8	Izydor	Leśniewski	764	Warszawa	01-150

Należy pamiętać by nie używać HAVING w operacjach, do których nadaje się WHERE (standardowe porównania)!
Trzeba też pamiętać, że HAVING działa bez optymalizacji, po dokonaniu WSZYSTKICH POPRZEDNICH WARUNKÓW I OPERACJI!
Standard SQL zakłada, że klauzula ta powinna zawierać jedynie odwołania do pól wymienionych w GROUP BY lub do użytych funkcji/agregatów w SELECT. MySQL pozwala jednak na „łamanie" tej zasady.
Ostatnim zastosowaniem HAVING jest możliwość ujednoznacznienia działania klauzuli zapytania. Funkcjonalność ta jest o tyle pożądana, że MySQL dopuszcza duplikowanie nazw wyświetlanych kolumn w zapytaniu (powodowane np. poprzez aliasy). Jeżeli alias został nałożony na funkcję, a następnie występuje odwołanie do niego w celu np. pogrupowania wyniku bądź wyświetlenia tylko pożądanych wartości moglibyśmy napotkać błąd (pod uwagę brany będzie alias zamiast pola w bazie). HAVING najpierw przegląda pola SELECT lecz następnie sprawdza pola w bazie (nawet te, które nie są brane pod uwagę w poleceniu SELECT). Dzięki temu ujednolica odwołanie i nie powoduje błędów w warunkach (wszystko powinno działać jak powinno)

Klauzula LIMIT określa ile ma zostać wyświetlonych rekordów pobranych ze wskazanej tabeli.
Istnieją trzy sposoby użycia tej klauzuli
1) SELECT * FROM <tabela>
LIMIT 5 #pobierze pierwszych 5 wierszy
2) SELECT * FROM <tabela>
LIMIT 5,10 #pobierze wiersze od 6 do 15
3) SELECT * FROM <tabela>
LIMIT 5 OFFSET 10 #pobierze rekord 10-15
Rozwiązanie numer 3 jest zgodne ze standardem SQL (Oracle, PostgreSQL, SQLite, MsSQL); dwa pierwsze to rozwiązania dostępne jedynie w MySQL.
Przykład użycia:
SELECT * FROM `uzytkownicy` LIMIT 5 OFFSET 4;

←T→	id	imie	nazwisko	wiek	miasto	komentarze
Edytuj Kopiuj Usuń	5	Dobromir	Śpiewak	15	3	413
Edytuj Kopiuj Usuń	6	Sławomir	Śpiewak	67	5	362
Edytuj Kopiuj Usuń	7	Zuzanna	Listkiewicz	30	3	570
Edytuj Kopiuj Usuń	8	Izydor	Leśniewski	15	4	764
Edytuj Kopiuj Usuń	9	Iwona	Adamczyk	45	4	114

	id	imie	nazwisko	wiek	miasto	komentarze
Edytuj Kopiuj Usuń	5	Dobromir	Śpiewak	15	3	413
Edytuj Kopiuj Usuń	6	Sławomir	Śpiewak	67	5	362
Edytuj Kopiuj Usuń	7	Zuzanna	Listkiewicz	30	3	570
Edytuj Kopiuj Usuń	8	Izydor	Leśniewski	15	4	764
Edytuj Kopiuj Usuń	9	Iwona	Adamczyk	45	4	114

	id	imie	nazwisko	wiek	miasto	komentarze
Edytuj Kopiuj Usuń	1	Adam	Nowak	22	8	798
Edytuj Kopiuj Usuń	2	Janina	Połać	20	3	85
Edytuj Kopiuj Usuń	3	Lucjan	Dobrowolski	30	8	31
Edytuj Kopiuj Usuń	4	Marianna	Kowalska	29	5	901
Edytuj Kopiuj Usuń	5	Dobromir	Śpiewak	15	3	413

W tym miejscu warto zająć się dodatkowymi przełącznikami i opcjami polecenia SELECT (mając już dwie w miarę wypełnione tabele).
- ALL – domyślnie działanie; wybiera wszystkie pasujące rekordy z bazy danych
- DISTINCT/DISTINCTROW – obie opcje mają identyczne działanie. Jeżeli się pojawiają to zapytanie zwraca tylko te rekordy, których wartości nie duplikują się. Przykład:

SELECT DISTINCT `nazwa` FROM `miasta`;

polecenie wybierze tylko nie powtarzające się nazwy (powiela się nazwa Warszawa oraz Częstochowa)

- HIGH_PRIORITY – nadaje poleceniu SELECT wyskoki priorytet; skutkuje to zablokowaniem bazy na czas wykonania odczytu. Wymusza pierwszeństwo nawet przed zapytaniami dodającymi/aktualizującymi dane z wysokim priorytetem. Należy mieć na uwadze by stosować ten przełącznik dla zapytań naprawdę ważnych
- STRAIGHT_JOIN – wymusza by łączenie wielu tabel zapytania (powyższe przykłady przeważnie pracowały na dwóch tabelach) odbywało się zgodnie z listą klauzuli FROM.
-SQL_SMALL_RESULT – szczególnie przydane z przełącznikiem DISCTINCT; podpowiada interpreterowi zapytań jak dużo rekordów może zawierać odpowiedź (optymalizacja). W tym wypadku silnik używa szybkiej tabeli tymczasowej tworzonej w pamięci operacyjnej
- SQL_BIG_RESULT – opcja zbieżna do poprzedniej. W tym jednak wypadku interpreter przenosi tymczasową tablicę wyników na pamięć masową, użytkując (z pierwszeństwem) jako klucza kolumn wymienionych w GROUP BY.
- SQL_BUFFER_RESULT – przemieszcza wyniki zapytania do tabeli tymczasowej. Dzięki temu tabela może szybciej zostać zwolniona do zapisu/aktualizacji/innych operacji bazodanowych.
- SQL_CALC_FOUND_ROWS – silnik MySQL przelicza ilość zwróconych rekordów przez zapytanie. Szczególnie przydatne w przypadku, gdy wykorzystujemy klauzulę LIMIT (zwrot tylko części wyników). Wartość zwróconą można uzyskać poprzez zapytanie:

SELECT FOUND_ROWS()
- SQL_CACHE – silnik ma przenosić wyniki zapytania do pamięci podręcznej zapytań.
- SQL_NO_CACHE – wyniki zapytanie nie są przenoszone do pamięci podręcznej.

Klauzula PROCEDURE pozwala na wywołanie procedury wraz z zapytaniem. Najlepszym przykładem użycia tej klauzuli jest wbudowana procedura ANALYSE(), pozwalająca na przegląd informacji co pomocnych przy optymalizacji tabel i zapytań.
Przykład:
SELECT `uzytkownicy`.`id`, `imie`, `nazwisko`, MAX(`komentarze`), `nazwa`, `kod_pocztowy` FROM `uzytkownicy`, `miasta` WHERE `miasta`.`id` = `uzytkownicy`.`miasto` GROUP BY 1 HAVING MAX(`komentarze`) > 300 PROCEDURE ANALYSE();

Field_name	Min_value	Max_value	Min_length	Max_length	Empties_or_zeros	Nulls	Avg_value_or_avg_length	Std	Opt
teb_test.uzytkownicy.id	2	9	1	1	0	0	6.1667	2.2669	ENU
teb_test.uzytkownicy.imie	Dobromir	Zuzanna	5	9	0	0	6.8333	*NULL*	ENU NOT
teb_test.uzytkownicy.nazwisko	Adamczyk	Śpiewak	7	11	0	0	8.8333	*NULL*	ENU NOT
MAX(`komentarze`)	362	901	3	3	0	0	634.6667	200.8330	ENU
teb_test.miasta.nazwa	Warszawa	Warszawa	8	8	0	0	8.0000	*NULL*	ENU
teb_test.miasta.kod_pocztowy	00-145	01-150	6	6	0	0	6.0000	*NULL*	ENU

Klauzula INTO FILE będzie omawiana szerzej w innym terminie. Ogólnie pozwala ona zapisać informacje wynikowe w pliku we wskazanej lokalizacji. Przydatne jeżeli potrzebne jest przenieść bazy/uzupełnić danymi bazy tzw. płaskie (Excel).

Dwie końcowe klauzule, FOR UPDATE oraz LOCK IN SHARE MODE (wykluczają się wzajemnie) pozwalają na:
- FOR UPDATE zamyka aktualnie przepatrywany/wybierany rekord zarówno przez zapisem/aktualizacją jak i pobieraniem przez inne zapytanie.
- LOCK IN SHARE MODE zamyka przepatrywany rekord dla zapisu/aktualizacji, jednak inne zapytania czytające mają do niego taki sam dostęp jak aktualne zapytanie.

ZADANIA:
1. Dlaczego w przedstawianych przykładach partycjonowania tablice nie posiadają żadnych kluczy (za wyjątkiem partycjonowania po kluczu)?
2. W jaki sposób uzyskać wartości kolumn zapisanych w konkretnej partycji?
3. Który z przedstawionych sposobów dodawania kluczy (głównego, indeksu czy obcego) jest zgodny ze standardem SQL?
4. Jak zapisać kilka odpowiedzi SELECT do jednej tablicy tymczasowej? Jak z tablicy tymczasowej stworzyć normalną tablicę?
5. Czy tabela tymczasowa powinna zniknąć wraz z wykonaniem na niej ostatniego polecenia?
6. Przy których parametrach poleceń tworzących można użyć klauzuli DEFAULT? Co ona daje?
7. Napisać zapytanie dodające do tabeli nowy rekord, którego wartości zostaną zaczerpnięte z innej tabeli. Wykonać pobranie rekordów z duplikatami kluczy głównych (jednak mają się one wykonać, nie zaś zgłosić błąd).
8. Czy istnieje sposób na dodanie serii rekordów do partycjonowanej tabeli w przypadku gdy warunki tychże partycji nie są w stanie zakwalifikować danych (np. partycji zakresu – RANGE, w której nie przewidziano dodania jakiejś wartości). W jaki sposób należy zbudować zapytanie INSERT?
9. Czy można dodawać wiele wierszy w ramach jednego polecenia INSERT?
10. Czy jest możliwe by klauzula ON DUPLICATE KEY UPDATE oddziaływała na więcej niż

jedno pole?
11. Czy istnieją jakieś obostrzenia odnośnie GROUP BY (ilości pól, występowania ich w klauzuli itp.)?
12. Która z klauzul działa po wykonaniu się klauzuli HAVING? Z czego wyniki ścisła kolejka?

ZADANIE DODATKOWE (CELUJĄCA):
Proszę zastanowić się w jaki sposób można zastąpić operację partycjonowania tablicy. Szczególnie chodzi o przypadek przeszukiwania tablicy posiadającej rozbudowane słownie artykuły, w których stosowanie partycjonowanie może być niewystarczające (sprawdza się ono najlepiej w przypadku pojedynczych wartości).

ŹRÓDŁA:
http://dev.mysql.com/doc/refman/5.5/en/
http://dev.mysql.com/doc/refman/5.6/en/partitioning-selection.html