# Skalowanie baz danych na przykładzie MySQL

Rafał Łasocha

Wrocław, dnia 11 lutego 2017 r.

## 1 Skalowanie systemów informatycznych

Dla wielu aplikacji wystarczający jest tylko jeden serwer bazodanowy. Niektóre z nich jednak, zbierają tyle informacji, że prędzej czy później dochodzą do momentu, w którym dane nie mieszczą się na jednym serwerze i jest konieczność zmian w infrastrukturze, żeby te dane pomieścić.

## 2 Co to znaczy, że system jest skalowalny?

Twórcy usług bazodanowych (lub innych usług, z których korzystają programiści), często posługują się terminem „skalowalność" w odniesieniu do ich produktów. Poza tym, że używają tego słowa sprzedawcy, żeby reklamować swój produkt, można ten termin w miarę konkretnie zdefiniować. Najpierw jednak zdefiniujmy pojęcie „wydolności" systemu. Oczywiście, dokładna definicja jak bardzo wydolny jest system baz danych, w dużej mierze zależy od jego wykorzystania. Różne aplikacje mają różny stosunek zapisów do odczytów, różne wielkości tabel (w niektórych systemach wszystkie tabele są mniej więcej jednakowo duże, w innych jest tylko kilka tabel które składają się na większość systemu i to wydajność odczytów/zapisów tych kilku tabel próbujemy zoptymalizować) etc. Intuicyjnie możemy myśleć o wydolności w ten sposób: jak dużą możemy osiągnąć przepustowość (w liczbie zapytaniach na sekundę na przykład) utrzymując rozsądny czas odpowiedzi. Dla przykładu, typowe testy wydajnościowy w stylu „wyślemy 1000000 zapytań SQL i policzymy czas, w którym serwer się wyrobi z obsłużeniem wszystkich" to nie jest do końca to, co nas interesuje, bo w takim przypadku serwer będzie przetwarzał zapytania pod maksymalnym obciążeniem. Nas interesuje test „będziemy wysyłać 1000 zapytań na minutę, i dostaniemy odpowiedź ¡ 3ms w 90% przypadków". Co trzeba podkreślić, wydolność bardzo zależy od naszego systemu, bo te „1000 zapytań na minutę", to nie są dowolne, abstrakcyjne zapytania, tylko takie, które występują w naszej aplikacji. Stąd, na wydolność naszej aplikacji ma wpływ liczba użytkowników, wielkość przetrzymywanych danych na serwerze, wielkość przetrzymywanych danych w przeliczeniu na jednego użytkownika („prawo

papieża", zawsze może być w naszej aplikacji kilka użytkowników, na których przypada nieproporcjonalnie dużo danych – i to też musimy wziąć pod uwagę, bo Papież również chciałby korzystać swobodnie z naszej aplikacji). Jeśli już rozumiemy pojęcie wydolności, które możemy jakoś określić dla konkretnej skonfigurowanej maszyny („wydolność naszego systemu to 1000 zapytań na minutę"), to oczywistym dla nas jest, że wiele aplikacji w swoim cyklu rozwoju ma taki moment, że potrzebuje zwiększyć tę wydolność. I tutaj możemy przejść do definicji skalowalności. Skalowalność mówi nam, do jakiego stopnia zwiększa nam się wydolność systemu, gdy dodajemy kolejne serwery. Pierwszą naszą myślą może być, że gdy mamy 1 serwer i dodamy drugi, to wydolność zwiększy się o 100% (skalowalność liniowa – rys 1.). Oczywiście zdrowy rozsądek podpowiada nam, że każdy taki serwer ma jakiś stały narzut na swoje działanie, więc prawdopodobnie skalowalność jest krzywą trochę poniżej liniowej (rys 2.). Skalowalność wielu systemów wygląda właśnie w ten sposób i wzięcie pod uwagę stałego „kosztu" na każdy serwer nazywamy prawem skalowalności Amdahla. Jeszcze zdrowszy rozsądek podpowie nam, że taki system na pewno ma jakieś granice, do którego możemy dokładać dodatkowe serwery i ciągle mieć z tego zysk. Dlatego też jest uniwersalne prawo skalowalności, które bierze również pod uwagę fakt, że serwery w danym systemie mogą potrzebować wymieniać ze sobą informacje (na zasadzie „każdy z każdym"). To powoduje, że mamy narzut $N^2$ do wydolności na każdym serwerze, więc jest taki moment, gdzie dołożenie kolejnego serwera pogorszy wydolność całego systemu! Większość systemów stosuje się właśnie do uniwersalnego prawa skalowalności. Oczywiście jak każdy model, jest to tylko przybliżenie – niektóre systemy mogą nie potrzebować akurat $N^2$ narzutu na komunikację, tylko dowolną inną funkcję, a może się zdarzyć też, że skalowalność będzie większa niż liniowa (rozpatrzmy przypadek gdzie mamy jeden serwer, który ma obciążony zarówno pamięć jak i dysk twardy i system operacyjny ciągle „swapuje", po czym dokładamy drugi serwer, dzięki któremu wszystkie dane mieszczą się w pamięci RAM).

# 3 Skalowanie MySQL

Rozważymy dwa sposoby na skalowanie systemu bazodanowego – skalowanie pionowe i skalowanie poziome. Skalowanie pionowe polega na tym, że ciągle mamy tylko jeden serwer, ale zwiększamy jego parametry (pamięć RAM, liczba CPU), żeby był bardziej wydolny. W skalowaniu poziomym, zwiększamy liczbę maszyn.

# 4 Skalowanie pionowe

Skalowanie pionowe jest świetnym rozwiązaniem aby kupić sobie trochę czasu. Jeśli jeszcze nie wiemy czy nasza aplikacja odniesie sukces, skalowanie pionowe jest lepszym wyborem, bo kosztuje znacznie mniej – zmiana maszyny na lepszą jest znacznie prostszą operacją niż skonfigurowanie całej infrastruktury, przygotowanie aplikacji do korzystania z wielu serwerów, przygotowanie procedur tworzenia/przywracania kopii zapasowych etc. A jako że

z reguły najdroższą rzeczą w firmie są wynagrodzenia pracowników IT, to skalowanie pionowe jest bardziej opłacalne. Główną wadą skalowania pionowego jest fakt, że nie można go robić w nieskończoność. Parametry serwerów mają swoje rozsądne granice, i im lepsze serwery kupujemy, tym droższe robią się zasoby i w pewnym momencie zaczyna się opłacać przejście na skalowanie poziome. Poza tym, musimy pilnować aby aktualizować wersję MySQLa – nie możemy oczekiwać, żeby kod napisany i testowany np. 10 lat temu działał wydajnie na systemie z 256GB RAMu – nikt tego wtedy nie miał możliwości przetestować na takim sprzęcie.

# 5 Skalowanie poziome

Skalowanie poziome możemy podzielić na trzy rodzaje: replikację, dzielenie funkcjonalne i sharding. Replikacja jest najprostszą formą i pozwala przenieść część odczytów na inne serwery, ale nie pomoże nam, jeśli naszym wąskim gardłem są zapisy. Zarówno dzielenie funkcjonalne jak i sharding rozwiązuje problem wydolności zapisów.

# 6 Dzielenie funkcjonalne

Dzielenie funkcjonalne jest prostym pomysłem, niewiele mającego wspólnego z konkretną bazą danych i jej możliwościami (w przeciwieństwie do np. replikacji, która musi być wspierana przez bazę danych, żeby ją efektywnie zrobić). Jeśli nasza aplikacja jest niewydolna żeby jeden serwer ją utrzymał, to jest spora szansa, że jest bardzo duża - duża, w sensie liczby różnych funkcjonalności, a nie liczbie użytkowników. Możemy ją więc podzielić, ze względu na funkcjonalności, jeśli za te funkcjonalności korzystają z raczej rozdzielnych zestawów tabel. Mówię „raczej", bo prawdopodobnie każda korzysta z tabeli użytkowników. Wyobraźmy sobie jakiś serwis informacyjny. Możemy bazę danych pociąć na kilka oddzielnych dużych funkcjonalności: witryna z bieżącymi wiadomościami, forum dla czytelników, helpdesk i właśnie baza użytkowników. Udało nam się więc podzielić, niewielkim kosztem serwis na kilka baz danych i prawdopodobnie osiągnąć skalowalność bliską liniowej (bazy danych są całkowicie rozłączne i nie komunikują się ze sobą). Jak widać dzielenie funkcjonalne jest względnie proste (w porównaniu do shardingu, który opisany będzie w dalszej części artykułu) i nie niesie ze sobą zbyt wielu zagrożeń (w momentach gdzie rzeczywiście musimy porozmawiać z więcej niż jedną bazą danych, być może musimy się pogodzić z brakiem transakcyjności – ale jeśli dzielenie funkcjonalne jest zrobione poprawnie, to są to rzadkie przypadki). Koniec końców niestety, w naszej aplikacji najprawdopodobniej jest takich rozdzielnych modułów kilka, albo kilkanaście, więc nie możemy dzielić w ten sposób naszej aplikacji w nieskończoność. W szczególności, może okazać się, że tylko jeden z tych już rozdzielonych baz danych potrzebuje bardziej skomplikowanego rozwiązania (shardingu), więc z reguły dzielenie funkcjonalne idzie w parze z shardingiem i sharding tego podziału nie likwiduje.

# 7 Sharding

Jest to aktualnie najpopularniejsza metoda skalowania bazy danych MySQL. Wystarczy wspomnieć, że korzystają z tego rozwiązania takie firmy jak Facebook, Instagram czy Uber. Jak wspomniane zostało wyżej, sharding idzie w parze z dzieleniem funkcjonalnym. Z reguły jest jakaś baza danych, która trzyma dane globalne, które są potrzebne wszystkim (np. właśnie listę użytkowników), często broniona przez klaster szybkiego cache, np. memcached. Aplikacje które korzystają z bazy danych z shardingiem, raczej o tym wiedzą. Potrzebna jest jakaś programistyczna abstrakcja, która decyduje do jakiego serwera bazodanowego wysłać żądania SQL i można to zrobić tak, żeby aplikacja nie wiedziała z kim rozmawia, ale jest to niewskazane. Równomierna dystrybucja danych po wszystkich serwerach jest nieefektywna (co pokażemy później), więc chcemy robić to trochę mądrzej.

# 8 Wybór klucza shardującego

Jeśli uznamy że potrzebujemy shardingu, jedną z pierwszych decyzji do podjęcia jest wybór klucza shardującego. Klucz ten decyduje, na jakim shardzie wyląduje rekord. W MySQL NDB Cluster, który jest gotowym rozwiązaniem do wdrażania shardowanej bazy danych, rekordy są równomiernie rozprowadzane po serwerach wg haszowanego klucza głównego w danej tabeli. To proste rozwiązanie, ale z nienajlepszą wydajnością. Wyobraźmy sobie platformę blogową, gdzie użytkownicy mają posty i piszą komentarze pod postami innych autorów. Żeby wyświetlić stronę główną takiego bloga, musimy pobrać wszystkie posty danego użytkownika. Skoro posty są rozłożone równomiernie na wszystkich serwerach, to musimy zrobić (najlepiej równolegle) zapytania do wszystkich serwerów i potem połączyć te dane (dodatkowe utrudnienia: co jeśli zapytanie miało klauzulę ORDER BY, LIMIT, HAVING - to wszystko musi być wtedy obsłużone w abstrakcji programistycznej po stronie klienta). Jest to zrobienia, ale widać, że może da się tutaj zrobić coś lepiej. Dlatego najczęściej dobrym kluczem jest identyfikator użytkownika (jeśli pomyślimy np. o witrynie społecznościowej) lub identyfikator klienta (częste w aplikacjach modelu SaaS, gdzie jest zakładane konto dla całej firmy i wiele kont użytkownika podlega pod tą firmę).

Mamy tutaj jeszcze inny problem. Załóżmy że mamy dwa widoki: 1) wyświetlenie posta ze wszystkimi komentarzami pod spodem 2) wyświetlenie profilu użytkownika, ze wszystkimi jego komentarzami Jak w takim wypadku rozprowadzać komentarze po serwerach? Możemy to zrobić wg identyfikatora użytkownika lub identyfikatora posta, ale wtedy zawsze jeden z powyższych widoków będzie wymagał zapytań do wszystkich serwerów. Częstym rozwiązaniem tutaj jest denormalizacja i duplikacja danych. Oczywiście, trzymanie zduplikowanych danych to jest pewien koszt, ale może w naszym przypadku wydajność jest ważniejsza. Można też się zastanowić czy na pewno wszystkie dane o komentarzach są potrzebne w obu zastosowaniach - może w przypadku profilu użytkownika chcemy wyświetlić jedynie nagłówki komentarzy?

# 9 Zapytania do wielu shardów

Niezależnie od zastosowanych technik, nie uda się nam wybrać klucza który by idealnie podzielił dane tak, że wszystkie zapytania odwoływałyby się tylko do jednego sharda. Na naszej platformie blogowej, możemy chcieć mieć np. listę najpopularniejszych postów, więc siłą rzeczy musielibyśmy wysłać zapytania do wszystkich shardów. Jest kilka technik do radzenia sobie z takimi zapytaniami: * warstwa cache dla ciężkich zapytań * tabele, które przetrzymują wyniki trudniejszych zapytań - dane o najpopularniejszych postach możemy na bieżąco przetwarzać w tle i wrzucać je do tabeli z wynikami. Taką tabelę z wynikami możemy wtedy albo zduplikować na wszystkich shardach, albo wrzucić w oddzielną bazę danych.

Nie tylko zapytania są trudne do zrealizowania w shardowanych bazach danych - jeśli mamy shardy, to tracimy też klucze obce (pomiędzy shardami) i transakcyjność (zależnie od rozwiązania, są sposoby aby transakcyjność odzyskać).

Poza odczytem danych, trudniejsze też może być ich usunięcie - jeśli chcemy usunąć użytkownika, to zamiast robić żądania do wszystkich shardów wykonujące usuwanie danych, możemy tylko dodać użytkownika do usunięcia, a samym usuwaniem danych zajmie się jakiś proces działający w tle.

# 10 Przydzielanie shardów do serwerów

Niekoniecznie musi być tak, że na jednym serwerze jest tylko jeden shard z danymi - wprost przeciwnie, zaraz wytłumaczę dlaczego lepszym pomysłem jest posiadanie wielu małych shardów na kazdym serwerze i jakie są zalety i wady obu rozwiązań.

Wyobraźmy sobie że mamy na serwerze 100GB danych - możemy je podzielić na 1 shard 100 GBajtowy lub 100 shardów wielkości 1GB. Pierwszą sprawą którą można wziąć pod uwagę, to zmiany struktury tabel. W MySQL, wiele z operacji zmian struktury blokuje całą tabelę (takie jak np. zmiana długości pola VARCHAR, dodawanie niektórych indexów, dodawanie kolumn w wersji ¡= 5.5 itd.). Lepiej jest więc aby proces zmiany struktury był robiony po kolei i blokował każdy mały shard na kilka minut, niż żeby jeden duży shard został zablokowany na godzinę. Mniejsze shardy są też łatwe do przenoszenia, bo operacja przeniesienia jednego shardu na drugi jest znacznie prostsza niż operacja rozdzielenia sharda na kilka części.

Są tylko dwie wady małych shardów: po pierwsze, prawdopodobnie rośnie liczba zapytań pomiędzy różnymi shardami (z którymi nauczyliśmy się sobie radzić w poprzedniej sekcji). Drugą wadą jest to, że jeśli mamy aktualnie działającą aplikację i przynoszący pieniądze biznes, operacja rozdzielenia bazy na dwie części (i dodanie infrastruktury do tego) może być znacznie prostsze niż rozdzielenie bazy na 1000 części. W pierwszym przypadku możemy sobie pewnie jeszcze pozwolić na robienie czegoś ręcznie, ale dwa razy, w drugim już musimy zautomatyzować wszystkie operacje działające na bazie danych.

# 11 Implementacja shardów na serwerach

Przechodząc bardziej do technicznych szczegółów - jak reprezentować shard danych na serwerze? Jest kilka najpopularniejszych sposobów: * każdy shard jest osobną bazą danych o takiej samej nazwie jak oryginalna * każdy shard jest osobną bazą danych o nazwie z sufiksem numerycznym oznaczającym numer sharda (np. bookclub_23, tabele to bookclub_23.comments itd.) * na jednym serwerze jest tylko jedna baza danych, a tabele mają sufiks numeryczny oznaczający numer sharda (np. bookclub.comments_23) * sufiks z nr sharda jest zarówno w nazwie bazy danych, jak i nazwie tabeli (bookclub_23.comments_23) * połączenie dowolnego z powyższych, można też wziąć pod uwagę uruchomienie kilku różnych instancji MySQL na danym serwerze obsługującą pewną liczbę shardów, itd.

Wszystkie podane sposoby są łatwe do wprowadzenia w nowej aplikacji, więc tam doradzane jest użycie sufiksu zarówno w tabeli, jak i bazie danych, żeby uniknąć ewentualnych pomyłek i błędów. Jednak jest to też sposób najbardziej inwazyjny dla kodu produkcyjnego. Dodanie numeru sharda tylko do bazy danych jest średnio inwazyjne w kodzie aplikacji - pewnie musimy ten numer gdzieś przekazać przy rozpoczęciu żądania do serwera, ale kod wykonujący zapytania SQL może zostać taki, jaki był przed dodaniem shardingu do aplikacji. Jeśli sufiks jest też w nazwie tabel, to zmian może być znacznie więcej, prawdopodobnie każde zapytanie SQL w systemie będzie musiało być lekko zmodyfikowane.

Podczas rozrzucania shardów po serwerach, warto też wziąć pod uwagę lokalizację użytkownika - jeśli nasza firma jest globalna, użytkowników z Europy można wrzucić do innego DC (zlokalizowanego w Europie) niż użytkowników z Azji.