Propagowanie zmian w skalowalnej multibazie danych
(An update propagator for joint scalable storage)

AUTOREFERAT ROZPRAWY DOKTORSKIEJ
10 czerwca 2012

Paweł Leszczyński

Wydział Matematyki i Informatyki
Uniwersytet Mikołaja Kopernika, ul. Chopina 12/18, 87-100 Toruń
pawel.leszczynski@mat.umk.pl

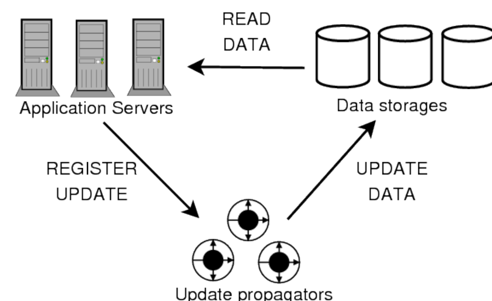
1 Wstęp

W erze Web 2.0 podstawową rolę nowoczesnych aplikacji internetowych odgrywa treść generowana przez użytkowników. Aplikacje stają się coraz bardziej spersonalizowane i przyjazne użytkownikowi, co powoduje wzrost obciążenia serwerów. Co gorsza, wzrost ten jest nieprzewidywalny. W wielu przypadkach błyskawiczny sukces, a więc także wzrost ruchu, prowadzi do niestabilnego działania aplikacji, co może skończyć się porażką biznesową przedsięwzięcia. Aby tego uniknąć, wszystkie elementy architektury systemu powinny być skalowalne. Problem skalowania bazy danych jest niebanalny, a tradycyjna architektura z pojedynczą relacyjną bazą danych jest niewystarczająca. W tej sytuacji powszechną praktyką jest aplikowanie dodatkowych (No)SQLowych baz danych w systemie, co rozwiązuje problem skalowalności krytycznych danych. Przy takim rozwiązaniu mimo, że dane są w kilku różnych bazach, stanowią jednak składowe tego samego systemu. Zbiory przechowywanych danych nie są więc rozłączne. Niezbędne są zatem mechanizmy do ich synchronizacji. Synchronizacja zaprogramowana *ad hoc* może prowadzić do błędów systemu, które są trudne do wykrycia i naprawienia, a więc także kosztowne. W pracy doktorskiej opisujemy metodę łączenia wielu składów o różnych charakterystykach w pojedynczy, spójny i skalowalny skład.

Kompromis spójności i skalowalności jest aktualnym problemem badawczym. Autorzy [1] opisują zasady i problemy przy tworzeniu skalowalnej bazy danych i wskazują na silną potrzebę wypełnienia luki pomiędzy bazami relacyjnymi, a bazami typu *klucz-wartość*. Autorzy [16] prezentują swoje przewidywania odnośnie multibaz danych (*multidatabase system* – MDBMS). Zauważają, że heterogeniczność składowych systemu stanowi

rzeczywisty problem i, aby go rozwiązać, niezbędne są pewne ograniczenia i uproszczenia składni zapytań.

Przykładem systemu podobnego do naszego jest Cloudy [7]. Składa się on z wielu baz danych połączonych w jeden system. W odróżnieniu od prezentowanego systemu, umożliwia on jednocześnie zapisy i odczyty. W konsekwencji tworzy on warstwę abstrakcji i ukrywa bazy danych, będące składowymi systemu. Założeniem naszego systemu jest stworzenie propagatora zmian i umożliwienie aplikacji bezpośrednich odczytów ze składowych baz systemu, patrz Rysunek 1. Naszym zdaniem stworzenie jednolitego API dla odczytu z dowolnych baz jest bardzo trudne, a utrzymanie go aktualnego dla najnowszych wersji wszystkich baz, po prostu niemożliwe.



Rysunek 1. Architektura systemu propagatora

Zbadaliśmy też istniejące mechanizmy do utrzymywania i aktualizacji widoków zmaterializowanych [17], w tym także *FlexViews* [5], implementację widoków zmaterializowanych dla MySQLa stworzoną w oparciu o [12,14]. Zastosowania takich rozwiązań w naszym przypadku są niestety ograniczone. Po pierwsze, widoki zmaterializowane przechowywane są w tej samej bazie co dane źródłowe. Po drugie, w naszym przypadku dane źródłowe nie istnieją. Wreszcie po trzecie, większość prac poświęconych widokom zmaterializowanym tworzy algorytmy oparte o procedury SQLowe. W naszym przypadku, dane przechowywane są w bazach, które nie muszą być relacyjne i tym samym nie dają dostępu do danych przy użyciu SQLa.

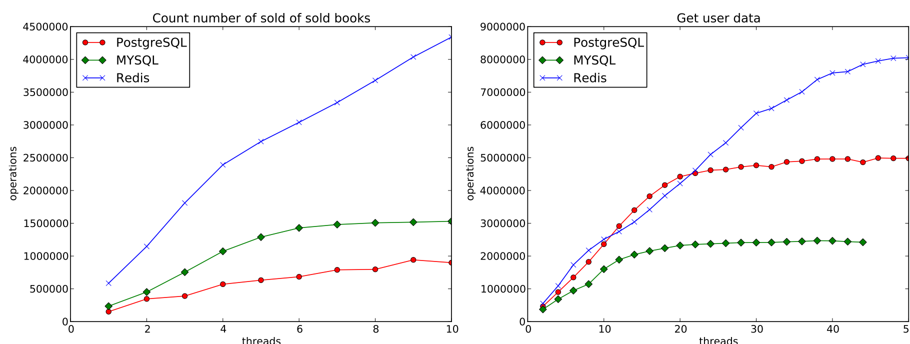
2 Przykład motywujący badania

Rozważmy jako przykład księgarnię internetową, która umożliwia swoim użytkownikom przeglądanie i wyszukiwanie pozycji, dokonywane zakupu

oraz pisanie opinii na temat produktu. Uproszczona baza danych takiego systemu składa się z czterech tabel zawierających: dane o oferowanych książkach, kupionych produktach, użytkownikach oraz opinie o książkach.

Kiedy liczba użytkowników zaczyna gwałtownie wzrastać, pojawia się pytanie: jaka baza danych będzie najlepszym rozwiązaniem dla systemu. W erze *NoSQL* naturalną odpowiedzią jest: żadna. Nie ma pojedynczego rozwiązania, które będzie najlepiej pasowało we wszystkich elementach aplikacji.

Mechanizm wyszukiwania jest krytyczny dla powodzenia biznesu. Rozwiązania takie jak Sphinx [2] lub Solr [15] (serwer oparty na silniku Lucene [6]) są często stosowane w praktyce, gdyż są zdecydowanie bardziej wydajne od RDBMS (relacyjne bazy danych) i zwracają trafniejsze wyniki. Umożliwiają one konfigurację funkcji wagowej decydującej o kolejności wyników oraz obsługują odmianę języka polskiego. Liczba opinii o książkach może szybko wzrastać, warto więc przechowywać je w skalowalnych bazach np. Cassandrze [8]. Informacje o książkach wymagają wysokiej dostępności i szybkich odczytów, ponieważ są wyświetlane na każdej stronie produktu. Pojedyncze zapytanie pobiera dane o pojedynczym produkcie, można więc je wydajnie przechowywać w bazie typu *klucz-wartość* [4,9]. Z kolei wymaganiem biznesowym może być, aby dane finansowe były przechowywane w relacyjnej bazie danych, która gwarantuje transakcyjność i bezpieczeństwo danych.



Rysunek 2. Na wykresach przedstawiono liczbę wykonanych operacji w zależności od liczby klientów je wykonujących. Wykres po lewej stronie przedstawia wyniki dla zapytania zwracającego dane o pojedynczej książce, w tym liczbę sprzedanych egzemplarzy. Z kolei wykres po prawej stronie dotyczy zapytania o dane użytkownika. W czasie testów, oddzielny wątek co 5ms modyfikował testowane dane.

Nasze przypuszczenia zostały zweryfikowane eksperymentalnie. Przetestowaliśmy bazy MySQL, PostgreSQL, Solr i Redis. Zbadaliśmy wyszukiwanie pełnotekstowe książek i opinii. Silnik InnoDB MySQLa nie wspiera indeksów pełnotekstowych. Wprawdzie PostgreSQL je wspiera, ale i tak jest zdecydowanie wolniejszy od Solra. W ciągu jednominutowego testu PostgreSQL wykonał 5 zapytań, podczas gdy Solr wykonał na tych samych danych 232 zapytania. Na rysunku 2 zaprezentowano wyniki dwóch kolejnych testów. W pierwszym z nich badamy zapytanie zwracające dane o książce i liczbie sprzedanych egzemplarzy. Porównaliśmy dwie bazy relacyjne z Redisem, który w wartości dla pojedynczego klucza zawiera dane o książce i licznik. Drugi test porównuje zapytanie pobierające dane pojedynczego użytkownika.

Te badania pokazują dlaczego warto tworzyć systemy oparte o architekturę heterogeniczną. Oczywiście przykładowy system można zaimplementować samodzielnie. Wyobraźmy sobie jednak dwóch programistów A i B pracujących nad systemem. A tworzy moduł z listą użytkowników, który kupili dany produkt. Aby działał on w sposób wydajny, decyduje się przechowywać nazwy użytkowników w bazie zakupionych produktów. Kilka miesięcy później B implementuje funkcjonalność zmiany nazwy użytkownika zupełnie nieświadomy wcześniejszych działań A. Nazwa użytkownika zmieniana jest w jednej bazie i po wdrożeniu nowej funkcjonalności dane tracą spójność. Kiedy błąd zostaje wykryty, jest przydzielany do A. Programista A wie, że funkcjonalność działała poprawnie i nic ostatnio nie było zmieniane. Nie wie jednak nic o zmianie B i jej konsekwencjach. Tego typu błędy są trudne do wykrycia przed wdrożeniem. Co gorsza, ich naprawa jest czasochłonna, a więc także kosztowna. Celem prezentowanej rozprawy jest ich uniknięcie i stworzenie spójnego składu, który gwarantuje spójność przechowywanych danych w różnych składach.

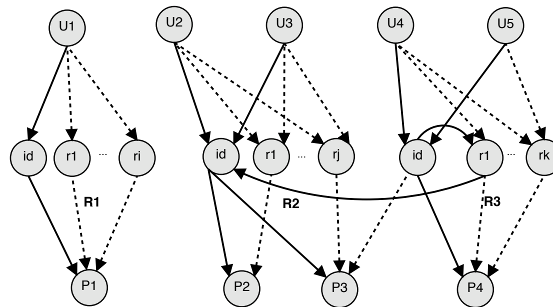
3 Problem spójności danych

W naszych badaniach zakładamy, że dane aplikacji są relacyjne, z pewnymi ograniczeniami. Przyjmujemy, że każda relacja posiada jednoelementowy klucz główny i dopuszczamy powiązania *jeden-do-wielu* pomiędzy relacjami. Założenia te są powszechnie przestrzegane w przemyśle. Pomimo, że schemat bazy większości systemów e-commerce przestrzega tych zasad, ich dane są przechowywane w różnych składach, w tym także nierelacyjnych. Składy te zawierają projekcje bazowego relacyjnego schematu. Każdy atrybut bazowego schematu jest zawarty w pewnej projekcji, a niekiedy jest on przechowywany redundantnie w wielu z nich.

Nawet jeśli dane są przechowywane w nierelacyjnej bazie danych, to ich struktura pozostaje relacyjna. Przykładowo, bazy *klucz-wartość* mogą przechowywać pojedyncze krotki identyfikowane przez klucz składający się z nazwy tabeli i wartości klucza głównego. W skład naszego systemu mogą wchodzić dowolne składy, które przechowują wynik projekcji dozwolonej w naszym modelu danych oraz implementujące właściwy *sterownik*. Sterownik powinien zawierać metody do modyfikacji pojedynczej krotki w projekcji: *add*, *modify*, *delete* oraz metodę, która zwraca krotkę z projekcji na podstawie zadanego klucza głównego. Dużą zaletą konstruowanego systemu jest fakt, że zajmuje się on jedynie właściwą propagacją zapisów, nie dotykając sposobu odczytu. Świat baz NoSQL jest bardzo dynamiczny i dzięki takiej konstrukcji stworzenie sterownika dla nowej bazy sprowadza się do utworzenia czterech metod.

Operacja zapisu jest zgłaszana bezpośrednio do systemu propagatora. Zakładamy też, że pojedynczy zapis modyfikuje jedną krotkę w relacji. Utworzony algorytm propaguje zgłoszoną zmianę w projekcjach. Jego konstrukcja jest oparta o graf propagacji, składający się z następujących wierzchołków:

- Każdy atrybut relacji schematu danych tworzy wierzchołek.
- Każda przechowywana projekcja tworzy wierzchołek.
- Każdy wzorzec operacji zapisu tworzy wierzchołek. Wzorzec składa się z modyfikowanej relacji, typu zapisu oraz listy modyfikowanych wartości.



Rysunek 3. Przykładowy graf propagacji. $U_1, \ldots, U_5$ oznaczają wierzchołki modyfikacji danych, $R_1, \ldots, R_3$ to relacje ze schematu, a $P_1, \ldots, P_4$ to projekcje przechowujące dane. Zakładamy, że R_2 jest w relacji *jeden-do-wielu* z R_3.

Na rysunku 3 przedstawiono przykładowy graf propagacji. Krawędzie w grafie są skierowane. Rozróżniamy *mocne* i *słabe* krawędzie.

- Każdy wierzchołek zapisu danych jest połączony *mocną* krawędzią z wierzchołkiem klucza głównego modyfikowanej relacji oraz *słabymi* krawędziami ze wszystkimi modyfikowanymi jej atrybutami.
- Klucz główny relacji jest połączony *mocną* krawędzią ze wszystkimi kluczami obcymi wewnątrz relacji.
- Klucz obcy jest połączony *mocną* krawędzią z kluczem głównym obcej relacji.
- Każda projekcja jest połączona *mocną* krawędzią z kluczem głównym pewnej relacji.
- Każda projekcja jest połączona *słabymi* krawędziami ze wszystkimi zawartymi w niej atrybutami relacji.

Kiedy operacja zapisu zostaje zarejestrowana, system na podstawie grafu określa projekcje, które wymagają modyfikacji i modyfikuje krotki. Konstrukcja grafu jest autorska i w oparciu o nią definiujemy problem spójności danych, *Data Consistency Problem - DCP*, postawiony w rozprawie doktorskiej.

Definicja 1 (Problem spójności danych–DCP). *Załóżmy, że system ma projekcje $P_1, P_2, \ldots, P_j$ zawierające dane w stanach T_1, T_2, ..., T_j. Operacja zapisu U modyfikuje krotkę pewnej relacji, co zmienia stan projekcji na T'_1, T'_2, ..., T'_j, gdzie $T_i = T'_i$ jeśli projekcja P_i nie uległa zmianie. Spójny propagatar zmian to obliczalna funcja F taka, że:*

$$F(U, T_1, \ldots, T_j) = (T'_1, \ldots, T'_j). \tag{1}$$

Przyjmijmy, że n oznacza rozmiar wszystkich projekcji przechowywanych w systemie:

$$n = |P_1| + |P_2| + \ldots + |P_j|, \tag{2}$$

oraz niech l oznacza liczbę wszystkich atrybutów projekcji zsumowaną z liczbą wszystkich powiązan *jeden-do-wielu* wykorzystanych w projekcjach $P_1, P_2, \ldots, P_j$. Ponadto niech m oznacza

$$m = |V| + |E_{weak}| + |E_{strong}| + l, \tag{3}$$

gdzie V jest zbiorem wierzchołków grafu, a E_{strong}, E_{weak} zbiorami *mocnych* i *słabych* krawędzi grafu. Wówczas m zależy od rozmiaru grafu, a tym samym od schematu relacji. Złożoność obliczeniowa algorytmów DCP jest funkcją n oraz m. Rozprawa doktorska zawiera algorytm o złożoności niezależnej od n. Dzięki temu konstruowany system propagacji jest niezależny od rozmiaru danych, a więc skalowalny.

4 Rezultaty

W rozprawie zawarty został algorytm rozwiązujący problem DCP. Udowodniliśmy też, że jest on poprawny. Oszacowana została jego złożoność obliczeniowa i jest ona funkcją niezależną od rozmiaru danych. W oparciu o algorytm utworzony został *PropScale*, system zapewniający spójność między elementami połączonego składu.

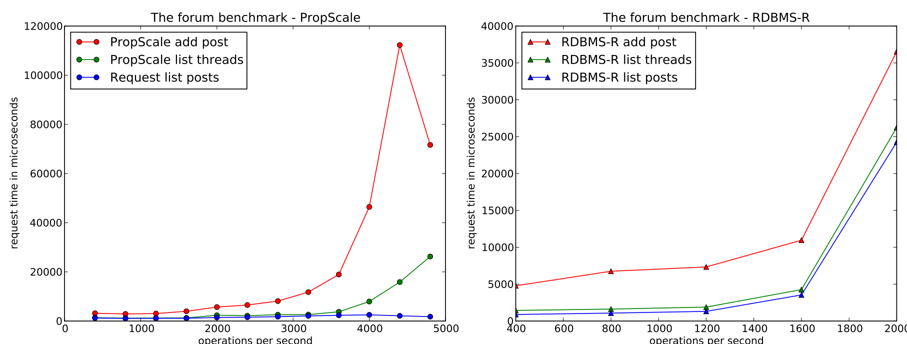
W 2012 precyzyjny opis algorytmu został przyjęty do druku w czasopiśmie *Fundamenta Informaticae*. Wyniki eksperymentalne dla implementacji zostały przyjęte na konferencję ADBIS 2012. Rozprawa doktorska zawiera dodatkowo rozdział poświęcony spójnemu cachowaniu, który był przedmiotem pracy badawczej na początku prowadzonych badań. Przedstawiony algorytm propagacji stanowi uogólnienie poprzednich rezultatów, które zostały opublikowane w [10,11].

Coraz większą popularność zyskują rozproszone bazy danych w chmurach. Firmy decydują się korzystać z zewnętrznych składów jak np. Amazon SimpleDB [13] ze względu na ich skalowalność, a także koszt. Wiele firm nie chce jednak umieszczać wszystkich danych na zewnątrz, czego przykładem mogą być dane finansowe, których bezpieczeństwo jest krytyczne dla organizacji. W takim przypadku pojawia się problem integracji wielu składów w spójny system, co jest możliwe dzięki prezentowanemu systemowi.

Kolejną korzyścią wynikającą z *PropScala* jest możliwość definiowania redundantnych danych statystycznych w projekcjach dla często odczytywanych statystyk. Denormalizacja jest powszechnie wykorzystywana nawet w relacyjnej bazie danych. *PropScale* zapewnia automatyczną aktualizację danych i umożliwia przechowywanie ich w oddzielnych systemach np. wysoko dostępnej bazie typu *klucz-wartość*.

Przydatność takiego mechanizmu została zbadana na przykładzie forum społecznościowego. Przyjmijmy, że forum składa się z trzech tabel: *forum*, *thread* i *post*. W eksperymencie zakładamy istnienie 100 forów, 10 tysięcy wątków po 100 wpisów każdy. Badamy wzorce dostępu do danych, które naszym zdaniem są najczęstsze dla takiego systemu: dodawanie nowego wpisu, pobieranie listy forów, pobieranie listy wątków dla danego forum oraz listy wpisów w wątku. Zwracana lista forów i wątków powinna zawierać informacje o ostatnich wpisach, ich autorach oraz dacie dodania. Dodatkowo powinna ona zawierać liczniki wpisów/wątków w nich zawartych. W eksperymencie badamy zapytania dodające wpis oraz zwracające pierwsze 20 elementów list wpisów/wątków, wykonywane z prawdopodobieństwem 15%, 45% i 45% odpowiednio. Porównujemy trzy warianty bazy

danych: MySQL ze schematem w znormalizowanej postaci, MySQL pod denormalizacji z redundantnymi kolumnami oraz *PropScale* korzystający ze znormalizowanej bazy MySQL oraz Redisa (bazy typu *klucz-wartość*). W pierwszym wariancie osiągnięto przepustowość 5ops (operacji na sekundę) dla ruchu generowanego w pojedynczym wątku. Podczas testu średni czas wykonania zapytania zwracającego wątek wyniósł 376 milisekund, co dyskwalifikuje ten wariant z praktycznych zastosowań. Na rysunku 4 zaprezentowano wyniki dla pozostałych dwóch wariantów: *PropScale* działa w sposób wydajny dla ponad 3000ops, osiągając maksymalną przepustowość 4800ops podczas gdy MySQL ze zdenormalizowanym schematem radzi sobie zaledwie przy 2000ops.



Rysunek 4. RDBMS-R i PropScale odpowiadają wariantom: MySQL z redundantnymi kolumnami oraz Propscalowi integrującemu MySQL z Redisem. Koniec wykresu oznacza ostatni pozytywny test – maksymalne obciążenie dla danej konfiguracji.

Przeprowadzone badania dotykają kompromisu pomiędzy spójnością a dostępnością rozproszonego systemu [3]. Bazy relacyjne zapewniają transakcyjność, ale nie skalują się dobrze. Z kolei rozwiązania NoSQL oferują wysoką dostępność w zamian za ryzyko niespójności. Naszym zdaniem zaprezentowany algorytm i system pozwalają na znalezienie optymalnego kompromisu CAP (Consistency - Availability - Partition Tolerance) dostosowanego do konkretnych zastosowań. Umożliwia on podzielenie bazy danych na części i wybranie dla każdej z nich optymalnego rozwiązania przy zachowaniu spójności między nimi.

Literatura

1. D. Agrawal, A. E. Abbadi, S. Antony, and S. Das. Data management challenges in cloud computing infrastructures. In S. Kikuchi, S. Sachdeva, and S. Bhalla,

editors, *DNIS*, volume 5999 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2010.
2. A. Aksyonoff. Introduction to search with Sphinx: From installation to relevance tuning, 2011.
3. E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, page 7, New York, NY, USA, 2000. ACM.
4. S. Chu. MemcacheDB: A complete guide, Mar. 2008.
5. Flexviews. Incrementally refreshable materialized views for MySQL, Jan. 2012.
6. E. Hatcher and O. Gospodnetic. *Lucene in Action (In Action series).* Manning Publications Co., Greenwich, CT, USA, 2004.
7. D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser. Cloudy: A modular cloud storage system. *PVLDB*, 3(2):1533–1536, 2010.
8. A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.
9. R. M. Lerner. At the forge: Redis. *Linux J.*, 2010(197), Sept. 2010.
10. P. Leszczynski and K. Stencel. Consistent cache maintenance for database driven websites. In F. Daniel and F. M. Facca, editors, *ICWE Workshops*, volume 6385 of *Lecture Notes in Computer Science*, pages 576–580. Springer, 2010.
11. P. Leszczynski and K. Stencel. Consistent caching of data objects in database driven websites. In B. Catania, M. Ivanovic, and B. Thalheim, editors, *ADBIS*, volume 6295 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2010.
12. I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In J. Peckham, editor, *SIGMOD Conference*, pages 100–111. ACM Press, 1997.
13. J. Murty. *Programming Amazon web services - S3, EC2, SQS, FPS, and SimpleDB: outsource your infrastructure.* O'Reilly, 2008.
14. K. Salem, K. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: asynchronous incremental view maintenance. *SIGMOD Rec.*, 29(2):129–140, May 2000.
15. D. Smiley and E. Pugh. *Apache Solr 3 Enterprise Search Server.* Packt, 2011.
16. P. Valduriez. Principles of distributed data management in 2020? In A. Hameurlain, S. W. Liddle, K.-D. Schewe, and X. Zhou, editors, *DEXA (1)*, volume 6860 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2011.
17. C. Zaniolo, editor. *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986.* ACM Press, 1986.