STUDIA INFORMATICA 2009
Volume 30 Number 2B (84)

Małgorzata BACH, Marcin STAŃCZAK, Aleksandra WERNER
Politechnika Śląska, Instytut Informatyki

WPŁYW PRZYJĘTEGO MODELU WERSJONOWANIA DANYCH NA EFEKTYWNOŚĆ RELACYJNEJ BAZY DANYCH

Streszczenie. W pracy skoncentrowano się na zagadnieniach związanych z wersjonowaniem relacyjnej bazy danych oraz przedstawiono propozycje dwóch rozwiązań, pozwalających na zapamiętanie i odtwarzanie stanu bazy z dowolnego momentu czasu. Dodatkowo przeprowadzono testy, które wskazały potencjalne obszary zastosowań zaproponowanych mechanizmów wersjonowania.

Słowa kluczowe: dekompozycja pozioma, dekompozycja pionowa, wersjonowanie danych, odtwarzanie stanu bazy

THE INFLUENCE OF DATA VERSIONING ON RELATIONAL DATABASE EFFICIENCY

Summary. The fundamental concepts of relational data versioning possibilities and the proposals of full and incremental versioning methods are contained in this paper. Additionally, applied method influence to the database size and the time of database state restoring are presented in this article.

Keywords: horizontal-temporal decomposition, vertical-temporal decomposition, data versioning, database state-restoring

1. Wprowadzenie

W związku z tym, że w bardzo wielu dziedzinach życia codziennego konieczne jest pamiętanie danych zmieniających się w czasie oraz z uwagi na fakt, iż uwzględnienie osi czasu pozwala m. in. na zachowywanie historii zmian danych, w ostatniej dekadzie znacząco wzrosło zainteresowanie możliwością reprezentacji czasowego charakteru danych w systemach relacyjnych baz danych.

O ile konwencjonalne systemy zarządzania relacyjnymi bazami danych fizycznie aktualizują dane (usuwają z dysku wartości stare i zapisują nowe), to w bazach historycznych dane aktualizowane są „logicznie" – tzn. nieaktualne dane pozostają na dysku, a jedynie ich stare i nowe wartości („wersje" danych) są odpowiednio znacznikowane czasem. Pozwala to na odtwarzanie stanu bazy danych z cofnięciem do dowolnie wybranego momentu w czasie, a nie punktu odtwarzania uzależnionego np. od czasu utworzenia ostatniej archiwalnej kopii danych.

Rozwiązania ułatwiające śledzenie i odtwarzanie stanu historycznego pojawiają się już od pewnego czasu w niektórych systemach relacyjnych baz danych (mechanizm Flashback Queries w systemie Oracle począwszy od wersji 9i czy zapytania historyczne w systemie PostgreSQL[1]), ale nadal nie jest to powszechna praktyka (brak jest takich możliwości np. w bardzo popularnych serwerach MS SQL i DB2). Dodatkowo proponowane funkcjonalności często wydają się mieć ograniczone zastosowanie. Przykładowo, Oracle 10g przechowuje wartość danych archiwalnych tak długo, jak tylko ich kopia zapisana jest w segmentach wycofania, ale jest to pojedynczy zestaw danych, z którego nie można odtworzyć dłuższej historii zmian. Ponadto, aby móc wykorzystać oferowane udogodnienie, administrator bazy musi wstępnie oszacować, a potem zadeklarować (jednostką są sekundy!) przewidywany czas przechowywania danych w segmentach. To wszystko powoduje, że wciąż aktualne jest poszukiwanie rozwiązania problemu wersjonowania danych zapisanych w bazie relacyjnej.

Pomimo, iż relacyjne bazy danych posiadają pewne ograniczenia wynikające m. in. z konieczności odwzorowania przestrzeni obiektów dziedziny przedmiotowej na zbiór „płaskich" tabel (relacji), których wartości atrybutów należą do dziedzin zawierających jedynie wartości atomowe, stanowią one dobrą podstawę do budowy systemu wersjonowania danych, choć ilość i rozmiar przechowywanych w tych bazach danych mają bezpośredni wpływ na wydajność przetwarzanych zapytań. Dzieje się tak dlatego, że każda modyfikacja tabeli bazy danych (czy to przez usunięcie wskazanych wierszy, czy przez aktualizację wartości żądanych kolumn) prowadzi docelowo do zmiany wartości (przynajmniej jednego) identyfikatora czasowego, powodując również w niektórych przypadkach znaczne zwiększenie liczby jej wierszy.

Wymienione czynniki powodują, że przy projektowaniu systemu wersjonującego dane należy rozważyć zastosowanie jednego z rozwiązań, znanych pod nazwami: dekompozycja pozioma – dekompozycja pionowa [1].

Dekompozycja pozioma polega na podziale wierszy danych pamiętanych w pojedynczej tabeli pomiędzy dwie tabele. Jedna z nich (tzw. historyczna) zawiera informacje historyczne, a druga (tzw. bieżąca) – informacje dotyczące stanu bieżącego.

[1] W wybranych wersjach tego systemu.

Przy zastosowaniu takiego rozwiązania włączenie mechanizmu rozdzielania danych bieżących i historii spowodowałoby, że:

- każdy wiersz o znanym punkcie początkowym i końcowym przedziału czasu transakcyjnego[2] zostanie przepisany do tabeli historycznej i usunięty lub zmodyfikowany w tabeli bieżącej,
- tabela bieżąca będzie przechowywać jedynie dane bieżące i nie jest wymagane, aby dodawać do niej dodatkowe kolumny związane z procesem wersjonowania (tzn. przechowujące znaczniki czasowe).

Z kolei, dekompozycja pionowa polega na rozdzieleniu wszystkich zmieniających się w czasie atrybutów pomiędzy osobne tabele relacyjne [3].

Z uwagi na fakt, że z zagadnieniem dekompozycji pionowej wiąże się ściśle problem identyfikacji zależności wielowartościowych o charakterze temporalnym oraz z tym, że zakres przebudowy aplikacji kierującej zapytania do historycznej bazy danych jest bardzo duży[3] [2], w dalszej części pracy skupiono się tylko na rozwiązaniu pierwszym.

W kontekście omawianej powyżej celowości rozdziału danych bieżących od historycznych można przeanalizować fragment bazy danych `SZPITAL`, rejestrującej podstawowe dane o pacjentach i odnotowującej każdy pobyt pacjenta w szpitalu oraz jego każdorazową wizytę w przychodni przyszpitalnej.

Ponieważ wspomniana baza była utworzona w systemie MS SQL Server, dlatego też punktem wyjścia dla dalszych analiz był dla autorów właśnie wspomniany system zarządzania relacyjną bazą danych.

W związku z tym, że zapytania dotyczące pacjentów przychodni stanowią jedynie niewielki procent ogółu zadawanych zapytań (pacjentom takim nie są np. wydawane żadne leki z apteki oddziałowej, chociaż ich wkład w fizyczny rozrost tabeli jest identyczny jak w przypadku pacjentów hospitalizowanych), można np. stwierdzić, że celowe jest odseparowanie danych o jednodniowych pobytach pacjentów na terenie przychodni przyszpitalnej od pozostałych danych, informujących o dłuższym pobycie pacjentów w szpitalu. Można również zastanawiać się nad korzyściami płynącymi z rozdzielenia tych kolumn tabeli `PACJENT`, których wartości zmieniają się w czasie częściej niż inne (`waga`, `wzrost`).

[2] Przez czas transakcyjny rozumiany jest czas, w którym dane (fakty, obiekty) są obecne w bazie (zapamiętane w bazie), niezależnie od tego, kiedy fakt miał miejsce w rzeczywistości [5].

[3] Przebudowa aplikacji wiąże się z korektą wszystkich zapytań kierowanych do tabel historycznych, która w praktyce sprowadza się do napisania zbioru wszystkich zapytań od nowa.

2. Przykłady implementacji mechanizmu automatycznego wersjonowania tabel relacyjnej bazy danych

W trakcie tworzenia systemu wersjonowania przyjęto, iż proces zapisywania danych historycznych powinien być niewidoczny dla użytkownika bazy i odbywać się bez jego interwencji. Nie powinien też mieć wpływu na korzystanie z aktualnych danych. Transparentność systemu polegać miała na braku konieczności modyfikacji zarówno zapytań kierowanych do bazy danych, jak i sposobu dodawania nowych wierszy czy modyfikacji i usuwania istniejących. Ponadto, nie powinien on generować zauważalnych narzutów czasowych dla istniejących aplikacji. W kolejnym podrozdziale zaprezentowane zostaną dwa mechanizmy wersjonowania zastosowane w omawianym systemie.

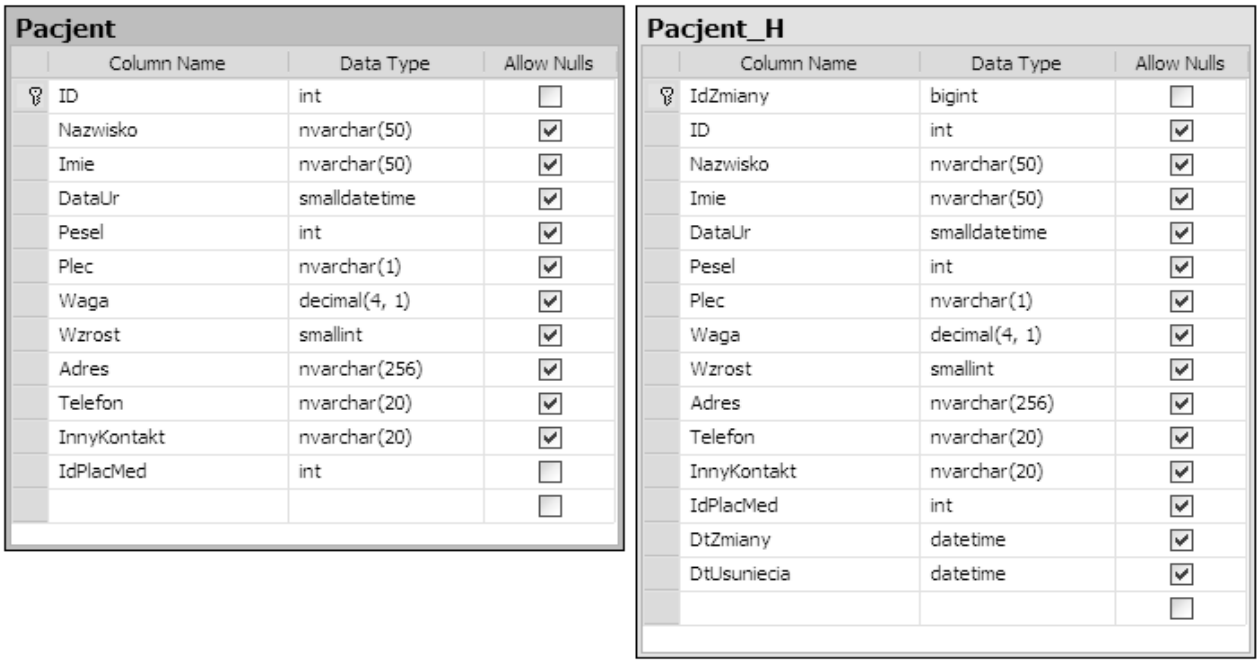
2.1. Mechanizmy wersjonowania całościowego i przyrostowego

W punkcie 1. przedstawiono pokrótce zagadnienia związane z pionową i poziomą dekompozycją relacji. W ramach prezentowanego systemu zaimplementowano dwa sposoby wersjonowania, które są pewnymi wariantami dekompozycji poziomej. Zostały one nazwane odpowiednio wersjonowaniem całościowym i przyrostowym.

Ich idea jest następująca.

Dla każdej z tabel, które powinny być wersjonowane, tworzone są w bazie nowe tabele – ich historyczne odpowiedniki – poszerzone o trzy kolumny: identyfikator zmiany (`IdZmiany`), datę modyfikacji (`DtZmiany`) oraz datę usunięcia (`DtUsuniecia`). Własność `Allow Nulls` każdej z kolumn w tabelach historycznych (o ogólnej nazwie `xxx_H`), poza identyfikatorem zmiany, powinna być ustawiona na true, co umożliwi wstawienie wartości `NULL`. Na rys. 1 przedstawiono schemat przykładowej tabeli przechowującej bieżące dane personalne pacjentów pewnej placówki medycznej (tabela `PACJENT`) oraz jej zmodyfikowaną dla potrzeb wersjonowania danych strukturę (`PACJENT_H`).

W momencie uruchomienia systemu umożliwiającego wersjonowanie tworzone są tabele `xxx_H`, a następnie kopiowane są do nich wszystkie dane z tabel oryginalnych, przy czym dodatkowo wartości w kolumnach: `DtZmiany` oraz `DtUsuniecia` ustawiane są odpowiednio na: aktualną datę i wartość `NULL`. Jest to część wspólna dla obu mechanizmów: całościowego i przyrostowego. Różnią się one natomiast sposobem odzwierciedlenia w tabeli historycznej późniejszych operacji modyfikacji (`UPDATE`) oraz procedurami przywracania danych.



Pacjent

	Column Name	Data Type	Allow Nulls
🔑	ID	int	☐
	Nazwisko	nvarchar(50)	☑
	Imie	nvarchar(50)	☑
	DataUr	smalldatetime	☑
	Pesel	int	☑
	Plec	nvarchar(1)	☑
	Waga	decimal(4, 1)	☑
	Wzrost	smallint	☑
	Adres	nvarchar(256)	☑
	Telefon	nvarchar(20)	☑
	InnyKontakt	nvarchar(20)	☑
	IdPlacMed	int	☐
			☐

Pacjent_H

	Column Name	Data Type	Allow Nulls
🔑	IdZmiany	bigint	☐
	ID	int	☑
	Nazwisko	nvarchar(50)	☑
	Imie	nvarchar(50)	☑
	DataUr	smalldatetime	☑
	Pesel	int	☑
	Plec	nvarchar(1)	☑
	Waga	decimal(4, 1)	☑
	Wzrost	smallint	☑
	Adres	nvarchar(256)	☑
	Telefon	nvarchar(20)	☑
	InnyKontakt	nvarchar(20)	☑
	IdPlacMed	int	☑
	DtZmiany	datetime	☑
	DtUsuniecia	datetime	☑
			☐

Rys. 1. Struktura tabel historycznych
Fig. 1. Structure of history tables

W mechanizmie całościowym, podczas operacji aktualizacji, do tabel historycznych kopiowane są całe wiersze danych, natomiast przywracanie stanu bazy oparte jest na prostych operacjach `SELECT`.

Z kolei mechanizm wersjonowania przyrostowego – w odróżnieniu od wersjonowania całościowego – wstawia do tabeli historycznej jedynie te dane, które uległy zmianie. Pozostałe kolumny uzupełniane są wartościami `NULL`. Procedury przywracające dane oparte są na dwóch zagnieżdżonych kursorach, co powoduje znaczny przyrost czasu niezbędnego do ich wykonania. Mechanizm ten ma jedną zasadniczą zaletę: dla aplikacji, gdzie występuje dużo operacji `UPDATE`, powoduje o wiele mniejszy przyrost wielkości pliku bazy, co pokazano w punkcie 3.

2.2. Zastosowane mechanizmy zapewniające wersjonowanie bazy danych

Mechanizm wersjonujący oparty został na wyzwalaczach (ang. *trigger*) wykonywanych po operacjach DML (`UPDATE`, `INSERT`, `DELETE`) na wierszach danych [4]. Przykładowo, dla rozwiązania całościowego:

- `AfterInsertTrigger_xxx` kopiuje cały wstawiony wiersz do tabeli historycznej, automatycznie wpisując w kolumnie DtZmiany aktualną datę, a w kolumnie `DtUsuniecia` – `NULL`.
- `AfterUpdateTrigger_xxx` znajduje w tabeli historycznej ostatni wpis dla modyfikowanego wiersza, ustawia mu datę usunięcia na aktualną, a następnie kopiuje

cały wstawiony wiersz do tabeli historycznej, automatycznie wpisując w kolumnach `DtZmiany` aktualną datę, a w `DtUsuniecia` – `NULL`.

- `AfterDeleteTrigger_xxx` znajduje w tabeli historycznej ostatni wpis dla modyfikowanego wiersza i ustawia `DtUsuniecia` na datę aktualną.

Podglądanie i przywracanie stanu dla zadanej daty zapewniają odpowiednie procedury składowane *xxx*`_restore`. Ich działanie polega na wyszukiwaniu wierszy o maksymalnej dacie zmiany mniejszej od zadanej i dacie usunięcia równej `NULL`, a następnie przekopiowaniu danych do tabel tymczasowych – w przypadku konieczności odczytu (”podglądnięcia”) stanu historycznego – lub oryginalnych – w przypadku jego fizycznego przywracania.

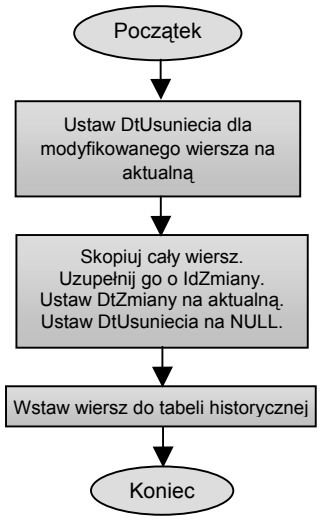
Podobnie jak w przypadku wersjonowania całościowego mechanizm przyrostowy oparty jest na wyzwalaczach wykonywanych po operacjach na wierszach danych, przy czy dla operacji `INSERT` wyzwalacze w obu przypadkach działają identycznie. Różnice występują dla operacji modyfikacji i usuwania danych (`UPDATE` i `DELETE`):

- `AfterUpdateTrigger_`*xxx* wstawia nowy wiersz do tabeli historycznej. W odróżnieniu jednak od wersjonowania całościowego wstawia jedynie dane, które uległy zmianie, a pozostałe kolumny uzupełnia wartościami `NULL`. Ponadto, automatycznie wpisuje w kolumnę `DtZmiany` aktualną datę, a w kolumnę `DtUsuniecia` – `NULL`.
- `AfterDeleteTrigger_`*xxx* wyszukuje w tabeli historycznej wszystkie wpisy o `ID` równym `ID` modyfikowanego wiersza i ustawia im `DtUsuniecia` na aktualną datę.

Przywracanie stanu tabeli z zadanego momentu, podobnie jak w poprzednim mechanizmie, realizowane jest za pomocą procedury składowanej, jednakże w tym przypadku jej działanie oparto na dwóch zagnieżdżonych kursorach, wyłuskujących z wierszy wartości modyfikowanych kolumna i uzupełniających dane w oryginalnej tabeli.

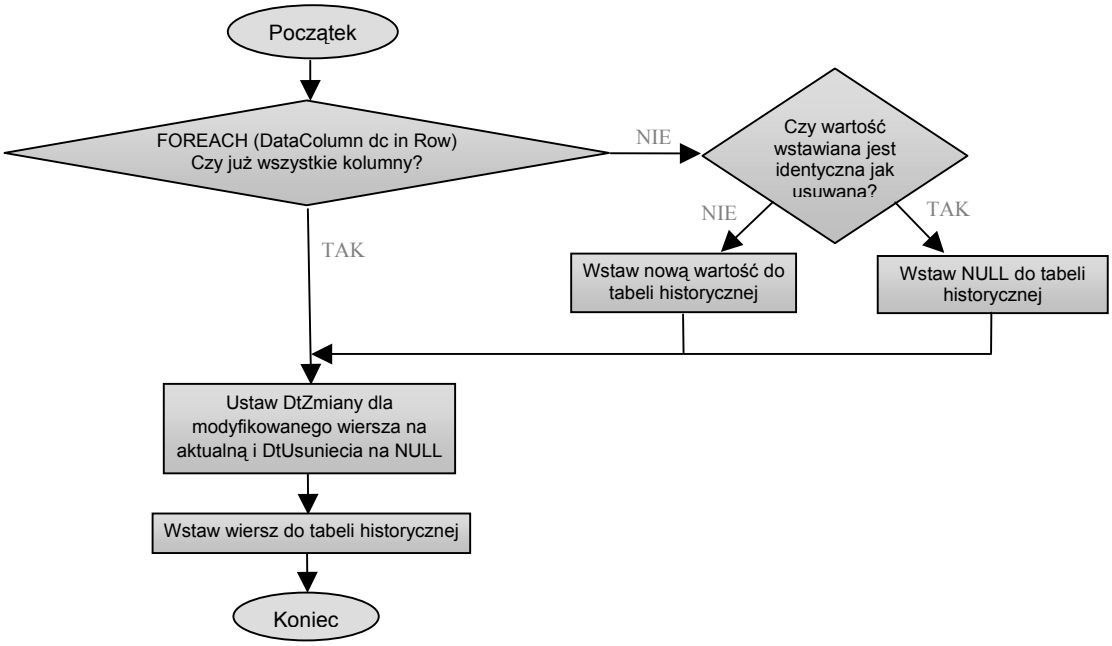
Dla lepszego zobrazowania różnic w działaniu wyzwalaczy `AfterUpdate` zaimplementowanych w obu przypadkach przedstawiono schematy blokowe obrazujące ich działanie (odpowiednio: na rys. 2 dla mechanizmu całościowego i na rys. 3 dla mechanizmu przyrostowego).

Algorytm wyzwalacza `AfterUpdate` w przypadku mechanizmu przyrostowego jest nieco bardziej skomplikowany. Wartość każdej kolumny modyfikowanego wiersza (poza kluczem głównym) porównywana jest z poprzednią wartością (na rys. 3 pętla przechodząca przez wszystkie kolumny wiersza zaprezentowana została przy użyciu składni języka C#: `FOREACH (DataColumn dc in Row)`).



Rys. 2. Wyzwalacz AfterUpdate mechanizmu całościowego
Fig. 2. Trigger AfterUpdate of full method

Jeśli wartości te są identyczne, do tabeli historycznej trafia wartość NULL, w przeciwnym przypadku – nowa wartość. Uzyskano w ten sposób sporą oszczędność pamięci kosztem szybkości działania mechanizmu. Uzupełniana jest ponadto o `DtZmiany` nowo wstawianego wiersza danych. Warto zauważyć, że w odróżnieniu od wyzwalacza dla mechanizmu całościowego, ten opisywany nie ustawia `DtUsunięcia` na datę aktualną przy modyfikacji wiersza. Spowodowane jest to wymogami algorytmu przywracania danych, korzystającego z zagnieżdżonych kursorów, w których sprawdzana jest tylko `DtZmiany`.



Rys. 3. Wyzwalacz AfterUpdate mechanizmu przyrostowego
Fig. 3. Trigger AfterUpdate of incremental method

3. Przykładowe testy wydajnościowe zastosowanych rozwiązań

Jednym z głównych wymagań stawianych przed tworzonym systemem było, aby zastosowane mechanizmy wersjonowania nie powodowały zauważalnego spowolnienia przebiegu korzystających z nich aplikacji. Przeprowadzone w systemach MS SQL Server oraz Oracle 10g testy wydajnościowe pokazały, iż wymaganie to zostało spełnione. Wykonanie wyzwalaczy związanych z operacjami `INSERT`, `UPDATE` i `DELETE`, uruchamianymi na 100 wierszach danych, wprowadzało narzuty czasowe poniżej jednej sekundy. Można zatem uznać, że narzut ten nie jest krytyczny w zastosowaniach, gdzie nie występuje masowe wstawianie czy modyfikacja wierszy.

Aby oszacować skalowalność rozwiązania, wykonane zostały testy na bazie zawierającej informacje o pacjentach. Wersjonowaniu poddano tabelę `PACJENT`, której struktura przedstawiona została na rys 1. Testy przebiegały według następującego algorytmu:

- Zapisano informację o wielkości pliku (dla SQL Servera był to plik o rozszerzeniu .mdf) dla testowanej bazy.
- Uruchomiono mechanizm konwertujący bazę do stanu umożliwiającego wersjonowanie (utworzono tabelę `PACJENT_H`). Zapisano czas wykonania i wielkość pliku po wykonaniu tej operacji.
- Dwukrotnie uruchomiono skrypt modyfikujący 25 000 wierszy i zapisano czas wykonania i wielkość pliku bazy danych po modyfikacji.
- Wykonano operację przywracania bazy do stanu po pierwszej modyfikacji 25 000 wierszy danych. Zapisano czas wykonania i wielkość pliku bazy (a w przypadku mechanizmu przyrostowego również pliku logu).

Ponieważ proces zakładania mechanizmów wersjonujących, polegający na utworzeniu tabel historycznych i przekopiowaniu do nich danych, w obu wersjach (całościowej i przyrostowej) przebiega identycznie, stąd też nie zauważono różnic w przebiegu dwóch pierwszych kroków przedstawionego powyżej algorytmu. Różnice występowały w przyroście rozmiaru pliku bazy po modyfikacji kolejnych porcji danych oraz w czasie potrzebnym do przywrócenia stanu bazy.

Jak widać w tabeli 1, dla mechanizmu całościowego aktualizacja zestawu 25 000 wierszy powodowała – przy mniejszej liczbie zapamiętanych w tabeli danych – przyrost pliku bazy rzędu 13 do 14 MB, co dla początkowej liczby 25 000 wierszy stanowiło wzrost rozmiaru bazy o ponad 126%. Procentowo, przyrost ten sukcesywnie zmniejszał się wraz ze zwiększaniem zbioru danych, na których następnie dokonywano operacji modyfikacji określonej liczby wierszy (przykładowo dla wyjściowej liczby 6 400 000 wierszy wyniósł już tylko 0,5% w stosunku do rozmiaru początkowego), co oczywiście wynikało z tego, iż niewielka

część wyjściowych danych ulegała jakimkolwiek zmianom. Nie zmieniało to jednak faktu, że każdorazowo po modyfikacji zestawu 25 000 danych plik bazy zwiększał swój rozmiar praktycznie o stałą wielkość. Spostrzeżenia te potwierdziły się również po wykonaniu testów w systemie Oracle.

Tabela 1

Porównanie rozmiarów plików bazy dla wersjonowania całościowego

Liczba wierszy w tabeli	Wielkość pliku .mdf po założeniu mechanizmu wersjonowania (baza zawiera tabele oryginalne i historyczne) (MB)	Wielkość pliku .mdf po wykonaniu operacji modyfikacji 25000 wierszy danych dla wersjonowania całościowego (MB)
25000	11,1	25,1
50000	20,1	33,1
100000	37,1	51,1
200000	71,1	84,1
800000	274	287
1600000	546	559
3200000	1085	1095
6400000	2171	2181

Jeżeli chodzi o mechanizm przyrostowy, to obserwowano wzrost pliku bazy w granicach od 2 do 4 MB, co przy początkowym rozmiarze testowanej bazy 11,1 MB oznaczało przyrost wielkości pliku rzędu 18%. Należy zauważyć, że przytoczone proporcje mogą ulec zmianie, bowiem wielkość pliku w przypadku mechanizmu przyrostowego jest ściśle związana z typami atrybutów wersjonowanych tabel relacyjnych. Im więcej atrybutów o typach zmiennej długości (varchar, nvarchar, varbinary), tym większy zysk z zastosowania przyrostowego mechanizmu wersjonowania, gdyż w przypadku kiedy dane ww. typów nie są przedmiotem operacji `UPDATE`, są one wypełniane wartościami `NULL`, co skutkuje dla nich zerową zajętością pamięci.

Tabela 2

Porównanie czasów niezbędnych do odtworzenia stanu bazy

Liczba wierszy w tabeli	Czas odtwarzania bazy do stanu po pierwszej serii modyfikacji 25000 wierszy dla mechanizmu całościowego (s)	Czas odtwarzania bazy do stanu po pierwszej serii modyfikacji 25000 wierszy danych dla mechanizmu przyrostowego (s)
25000	1	288
50000	2	788
100000	4	2468
200000	8	8435

W tabeli 2 porównano czasy potrzebne dla przywrócenia stanu bazy. Ze względu na dużą czasochłonność mechanizmu przyrostowego pomiary dla tej metody wykonane zostały dla tabeli o maksymalnej liczbie 200 000 wierszy. Dla mechanizmu całościowego pomiary zakończono dla 6 400 000 wierszy. Czas w tym przypadku wyniósł 432 s. Tak więc widać, że czas potrzebny do przywrócenia stanu bazy w obu przypadkach jest skrajnie różny. Mechanizm przyrostowy wymaga długotrwałego obciążenia maszyny. Ponadto, przywrócenie stanu

bazy powodowało znaczące powiększenie się dziennika transakcji (plik .ldf), co nie występowało w przypadku mechanizmu całościowego.

4. Podsumowanie

W ramach prezentowanej pracy przeanalizowano różne sposoby wersjonowania bazy danych. Zaimplementowano dwa mechanizmy nazwane całościowym i przyrostowym. Przeprowadzone testy pozwoliły porównać oba rozwiązania i wskazać obszary potencjalnych zastosowań.

I tak, mechanizm przyrostowy może być wykorzystywany w odniesieniu do systemów, w których operacje modyfikacji danych wykonywane są bardzo często – zwłaszcza na tabelach relacyjnych z dużą liczbą atrybutów o typach zmiennej długości. Powinien on znaleźć zastosowanie w aplikacjach, gdzie nie jest wymagany częsty podgląd stanów z przeszłości czy przywracania bazy, natomiast wymagana jest oszczędność pamięci. Przywracanie stanu bazy natomiast należy przy tym rozwiązaniu traktować tylko jako mechanizm awaryjny i wykorzystywany sporadycznie.

W sytuacjach gdy konieczność odtwarzania stanu bazy (rozumianego jako fizyczne przywracanie lub tylko odczytanie pewnego stanu historycznego) występuje dość często, należy wziąć pod uwagę możliwość wykorzystania mechanizmu wersjonowania całościowego, gdyż jego niewątpliwą zaletą jest zdecydowanie krótszy, w stosunku do rozwiązania poprzedniego, czas przywracania stanu bazy. Wydaje się, że przeznaczeniem tego mechanizmu mogą być przede wszystkim systemy, gdzie dane modyfikowane są rzadko. Jego zasadniczą wadą jest bowiem stosunkowo szybki przyrost wielkości pliku bazy, ze względu na kopiowanie całego modyfikowanego wiersza do tabeli historycznej. Tak więc trzeba w tym przypadku liczyć się z koniecznością zakupu pojemniejszych pamięci, co w chwili obecnej nie powinno jednak stanowić zasadniczego problemu.

BIBLIOGRAFIA

1. Date C. J.: An Introduction to Database Systems, Addison Wesley, 2000.
2. Werner A.: Dobór modelu temporalnych baz danych dla celów optymalizacji zapytań, rozprawa doktorska, Gliwice, 2005.
3. Werner A.: Strojenie projektu temporalnej bazy danych, Konferencja BDAS, Ustroń 2005.
4. Stańczak M.: Opracowanie i implementacja mechanizmu automatycznego wersjonowania tabel w Microsoft SQL Server, praca dyplomowa, Gliwice 2008.

5. Jensen Ch., Clifford J., Gadia S.K.: A consensus Glossary of Temporal Database Concepts, International ARPA/NSF Workshop, Arlington, Texas, 1993; strona internetowa (stan na rok 2009) ftp://cs.arizona.edu/tsql/doc.
6. Dokumentacja techniczna Oracle 10g i Oracle 11g ; strona internetowa (stan na rok 2009) http://www.oracle.com.
7. Dokumentacja techniczna MS SQL Server; strona internetowa (stan na rok 2009) http://www.microsoft.com.

Recenzent: Dr inż. Tomasz Traczyk

Wpłynęło do Redakcji 3 lutego 2009 r.

Abstract

The topic of this article is the problem of data versioning influence to the database efficiency. The article is organized as follows. The first part (section 1 and 2) introduces the idea of relational table horizontal and vertical decompositions, and adduces the proposals of data versioning methods – full and incremental. Besides, it presents the database mechanisms that render these versioning. The second part (section 3) includes the results of exemplary database efficiency tests, when the data file accrued size and restoring processing time were observed. The final conclusions about the usage of both presented methods, are offered in section four.

Adresy

Małgorzata BACH: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, malgorzata.ach@polsl.pl.
Aleksandra WERNER: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, aleksandra.werner@polsl.pl.
Marcin STAŃCZAK: absolwent kierunku Informatyka Politechniki Śląskiej.