ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1997
Seria: INFORMATYKA z. 33 Nr kol. 1377

Małgorzata BACH, Aleksandra WERNER
Politechnika Śląska, Instytut Informatyki

ROZSZERZENIA MODELU RELACYJNEGO W EKSPERYMENTALNYM SYSTEMIE POSTGRES95

Streszczenie. System Postgres95 odgrywa znaczącą rolę w rozszerzaniu relacyjnych baz danych o abstrakcyjne typy danych, wspomaganie czasowych własności danych oraz aktywne reguły. W niniejszym artykule zaprezentowano powyższe, nowatorskie cechy systemu wraz z przykładami ich użycia. Praca zawiera ponadto omówienie różnic między relacyjnym, pasywnym modelem danych a modelem aktywnym, pozwalającym modelować skomplikowane fragmenty rzeczywistości.

EXTENSIONS OF RELATIONAL MODEL IN EXPERIMENTAL SYSTEM POSTGRES95

Summary. System Postgres95 is of great importance in extending relational database systems with abstract data types, time travel and active rules. This article presents its innovatory features with application examples. In this paper we also describe differences between relational, passive data model and active one, which support modeling of complicated fragments of reality.

1. Wstęp

Na rynku komercyjnym wciąż zdecydowanie dominują systemy zarządzania bazami danych oparte na relacyjnym modelu danych. Niewątpliwą zaletą takiego rozwiązania jest jego prostota, stosunkowo wysoka efektywność i łatwość zastosowania w systemach administracyjnych, bankowych, gospodarki materiałowej czy magazynowej. Jednakże wymogi współczesnych aplikacji, modelujących skomplikowane fragmenty rzeczywistości, często przerastają możliwości tego modelu. Przykładowo, próby zastosowania relacyjnych baz

danych do szeroko rozumianego wspomagania podejmowania decyzji wykazały potrzebę rozszerzenia modelu relacyjnego o nowe elementy. Próbę sprostania tym wymogom podjął zespół prof. M. Stonebrakera na Uniwersytecie w Kalifornii, opracowując system Postgres95. System ten jest następcą systemu Ingres, który przez szereg lat był rozszerzany o nowe cechy, w tym abstrakcyjne typy danych i wyzwalacze. System Postgres95 został zaprojektowany z myślą o włączeniu tych nowych cech, razem z obiektami złożonymi, zapytaniami rekurencyjnymi oraz zarządzaniem danymi historycznymi i wersjami.

System ten, zwany systemem postrelacyjnym, nie jest produktem komercyjnym, został jednak udostępniony przez autorów w sieci Internet pod adresem:

http://www.postgreSQL.org

W niniejszym opracowaniu zostaną zaprezentowane rozszerzenia systemu Postgres95 w stosunku do systemów relacyjnych, ze szczególnym uwzględnieniem możliwości tworzenia złożonych typów danych i zadawania pytań historycznych.

2. Krótka charakterystyka relacyjnego modelu danych

Jak już wspomniano na wstępie, najbardziej rozpowszechnionym modelem danych jest model relacyjny, stanowiący podstawę większości współczesnych systemów zarządzania bazami danych (ang. DBMS - Database Management System).

Relacyjny model danych bazuje na pochodzącym z teorii mnogości pojęciu relacji, czyli podzbioru iloczynu kartezjańskiego listy dziedzin, przy czym dziedzina jest rozumiana jako zbiór wartości atrybutu. Element relacji to krotka. Reprezentacją relacji jest dwuwymiarowa tablica, w której wiersze odpowiadają krotkom, a kolumny atrybutom. Tablica taka jest z kolei naturalną ilustracją pliku rekordów, w którym nazwy pól odpowiadają nazwom kolumn, zaś rekordy poszczególnym wierszom. Zbiór nazw atrybutów nazywa się schematem relacji.

Niewątpliwą zaletą modelu relacyjnego jest jego prostota, ogólność zastosowań i przejrzystość struktur, ma on jednak pewne znaczące ograniczenie, polegające na wymaganiu, by schemat każdej relacji był co najmniej w pierwszej postaci normalnej (1NF), co oznacza, iż dziedziny dla wszystkich atrybutów mogą zawierać jedynie wartości atomowe.

Konieczność odwzorowania przestrzeni obiektów dziedziny przedmiotowej na zbiór płaskich relacji uniemożliwia reprezentowanie w sposób naturalny obiektów świata rzeczywistego. Próba wyeliminowania tego ograniczenia stanowiła jeden z powodów rozpoczęcia badań nad tzw. modelami postrelacyjnymi, czyli modelami o bogatszej

semantyce niż model relacyjny. W systemach postrelacyjnych elementy krotek mogą stanowić tablice, krotki, zbiory, czy nawet relacje (tzw. zagnieżdżone relacje).

3. Model danych systemu Postgres95

Model danych systemu Postgres95, w porównaniu z tradycyjnym modelem relacyjnym, oferuje kilka dodatkowych konstrukcji:

- strukturalne typy atrybutów,
- klasy (ang. classes), będące zgrupowaniem wszystkich obiektów, które mają ten sam zbiór atrybutów i metod,
- dziedziczenie wielokrotne (ang.inheritance), wynikające z faktu, że klasy tworzą hierarchiczne zależności,
- rozszerzalne funkcje.

System ten został również wyposażony w mechanizmy: tworzenia i przetwarzania reguł oraz wspomagania czasowych własności danych, może więc stanowić podstawę do budowy aktywnych i dedukcyjnych baz danych.

4. Typy danych

W systemie Postgres95 istnieją dwa typy atrybutów: **atomowe** (ang. atomic) i **strukturalne** (ang. structured) definiowane przez użytkowników. Typy atomowe stanowią przede wszystkim typy predefiniowane w systemie, takie jak: *int2, int4, float4, varchar* itd. choć mogą być również definiowane przez użytkowników. W związku z tym, w systemie Postgres95 typy danych określa się mianem abstrakcyjnych typów danych (ang. ADT).

Stworzenie typu abstrakcyjnego wymaga zdefiniowania struktury danych i operatorów stosowanych dla tej struktury.

W systemie Postgres95 definicja struktury danych składa się z:

① nazwy typu danych,
② rozmiaru wewnętrznej reprezentacji danych w bajtach,
③ procedur konwersji wartości z reprezentacji zewnętrznej na wewnętrzną i odwrotnie,
④ wartości domyślnych (opcjonalnie).

Natomiast w skład definicji skorelowanego ze zdefiniowanym typem operatora wchodzi:

① symbol operacji,
② określenie ilości operandów,
③ procedura stanowiąca implementację operatora

oraz

④ inne wartości opcjonalne.

W tym rozdziale zostaną omówione korzyści wypływające z możliwości wykorzystania złożonych typów danych.

4.1. Przykłady zastosowań typów strukturalnych

Przypuśćmy, że chcemy zapamiętać informacje o kwartalnych premiach pracowników pewnego przedsiębiorstwa.

W tradycyjnym systemie relacyjnym musielibyśmy stworzyć tablicę o następującym schemacie:

PREMIE (*nazwisko, numer_kwartału, premia_kwartalna*)

gdzie: atrybut *nazwisko* jest typu znakowego (np. *varchar*),
atrybut *numer_kwartału* jest typu całkowitego (np. *int4*),
atrybut *premia_kwartalna* jest typu zmiennoprzecinkowego (np. *float*).

W języku SQL realizujemy to zleceniem:

```
create table PREMIE (Nazwisko varchar, Numer_kwartału int4, Premia_kwartalna float)
```

Możliwość wykorzystania w Postgresie95 tablic jedno- lub wielowymiarowych o stałym lub zmiennym rozmiarze pozwala na zapamiętanie informacji o premiach w następujący sposób: atrybut *nazwisko* definiujemy jako tablicę znakową o zmiennej długości, natomiast *premię_kwartalną* jako tablicę czteroelementową, przy czym kolejne elementy tej tablicy są 8-znakowymi liczbami całkowitymi; wskaźnikiem jest numer kwartału:

```
create table PREMIE1 (Nazwisko text, Premia_kwartalna int4[])
```

W pierwszym ujęciu każdemu pracownikowi odpowiadają cztery rekordy, w których zapamiętana jest premia, jaką otrzymał w poszczególnych kwartałach (tabela 1).

Tabela 1

Przykładowa zawartość relacji *PREMIE* dla dwóch pracowników

Nazwisko	Numer_kwartału	Premia_kwartalna
Nowak	1	200
Nowak	2	400
Nowak	3	300
Nowak	4	600
Pawłowski	1	300
Pawłowski	2	400
Pawłowski	3	300
Pawłowski	4	800

Wykorzystując możliwości systemu Postgres95, można te same dane przedstawić również tak:

Tabela 2

Przykładowa zawartość relacji *PREMIE1* dla dwóch pracowników

Nazwisko	Premia_kwartalna
Nowak	{200,400,300,600}
Pawłowski	{300,400,300,800}

W przedstawionej powyżej relacji *PREMIE1* do zapamiętania tej samej informacji wystarczają dwa rekordy – a więc wielkość pliku, w którym zapisywane są dane, jest znacznie mniejsza. Można również przypuszczać, iż czasy realizacji dowolnego zadania wyszukiwania będą w obu przypadkach różne.

Zmierzono czasy potrzebne na odnalezienie informacji o wielkości premii, jaką otrzymał pracownik *Pawłowski* w pierwszym kwartale (testy przeprowadzono dla relacji zawierających informacje o 1000 pracownikach).

W pierwszym przypadku zadano pytanie:

```
select Premia_kwartalna from PREMIE where nazwisko='Pawłowski' and Numer_kwartału=1
```

w drugim:

```
select PREMIE1.Premia_kwartalna[1] from PREMIE1 where nazwisko='Pawłowski'
```

Zgodnie z oczekiwaniami, czas potrzebny na odszukanie danych w pierwszym przypadku był około 3-krotnie dłuższy.

W kolejnym przykładzie dotyczącym informacji o studentach, jeden z atrybutów jest zdefiniowany jako tablica dwuwymiarowa. Informacje zorganizowano w dwojaki sposób. Raz, jako:

```
create table STUDENT (wydział varchar, rok_st int4, lb_gr_dziek int4, lb_stud int4)
```

gdzie: atrybut *wydział* określa nazwę wydziału,
atrybut *rok_st* oznacza rok studiów,
atrybut *lb_gr_dziek* oznacza liczbę grup dziekańskich,
atrybut *lb_stud* określa liczbę studentów studiujących na danym roku.

Zakładając, że studia na danym wydziale trwają pięć lat, do zapamiętania informacji o każdym z wydziałów potrzebne jest pięć rekordów (tabela 3).

Tabela 3

Przykładowe wypełnienie relacji *STUDENT* dla jednego wydziału

Wydział	rok_st	lb_gr_dziek	lb_stud
Górniczy	1	6	120
Górniczy	2	4	80
Górniczy	3	5	110
Górniczy	4	5	105
Górniczy	4	4	98

Wykorzystując możliwości systemu Postgres95, można także stworzyć inną strukturę:

```
create table STUDENT1 (wydział varchar, lb_gr_lb_st int4[][]),
```

gdzie: atrybut *wydział* oznacza nazwę wydziału,
atrybut *lb_gr_lb_st* jest tablicą dwuwymiarową, przy czym pierwszy wymiar jest identyfikowany z liczbą grup, a drugi z liczbą studentów. Wskaźnikiem tej tablicy jest rok studiów.

Jeden rekord relacji *STUDENT1* zawiera tą samą informację, do której zapamiętania poprzednio wykorzystano pięć rekordów (tabela 4).

Tabela 4

Przykładowe wypełnienie relacji *STUDENT1* dla jednego wydziału

Wydział	lb_gr_lb_st
Górniczy	{{6,4,5,5,4},{120,80,110,105,98}}

Aby odnaleźć liczbę studentów piątego roku Wydziału Górniczego, w pierwszym przypadku należy wydać polecenie:

```
select il_stud from STUDENT where rok_st=5 and wydział='Górniczy'
```

w drugim:

```
select STUDENT.il_gr_il_st[2][5] where wydział='Górniczy'.
```

W przypadku stosowania typów tablicowych, nie bez znaczenia wydaje się również fakt zwiększenia zwięzłości zapisu poleceń odczytu/modyfikacji danych w bazie.

4.2. Definiowanie typów strukturalnych

Do zdefiniowania własnego typu konieczne jest wcześniejsze stworzenie w języku C funkcji do wprowadzania i wyprowadzania danych.

Na przykład stwórzmy typ *MTIME*, pozwalający na wpisywanie do tabeli *CZAS* godzin przyjazdu/odjazdu pociągu, w formacie [hh:mm].

```
create table CZAS                /* definicja tablicy CZAS*/
 (
 id_pociagu text,
 przyj MTIME,
 odj MTIME
 );


create type MTIME                /* definicja typu */
 (
 internallength = 8,
 input = MTIME_IN,               /* funkcja do wprowadzania danych */
 output = MTIME_OUT              /* funkcja do wyprowadzania danych */
 );
```

Definicje funkcji: *MTIME_IN* oraz *MTIME_OUT* muszą być dołączone do systemu przez wydanie polecenia *create function*:

```
create function MTIME_IN(opaque) returns MTIME
  as '/home/pg/mtime/MTIME.SO' language 'C';

create function MTIME_OUT(opaque) returns opaque
  as '/home/pg/mtime/MTIME.SO' language 'C';
```

Plik *MTIME.SO* powstaje w wyniku kompilacji funkcji napisanej w języku C. Przykładowa postać funkcji:

```
typedef struct mTime {              /* typ danej */
    int    godzina;
    int    minuta;
} mTime;

mTime *MTIME_IN(char *str)     /*  funkcja wprowadzająca daną z łańcucha tekstowego na wejściu */
{
int g, m;
mTime *result;
  if (sscanf(str, " %d : %d )", &g, &m) != 2)
  {
  elog(WARN, "mtime_in: blad w \"%s\", str);
  return NULL;
  }
result = (mTime *)palloc(sizeof(mTime));
result->godzina = g;
result->minuta = m;
return(result);
}

char *MTIME_OUT(mTime *mtime)    /*  funkcja wyprowadzająca daną w postaci łańcucha tekstowego */
{
char *result;
if (mtime == NULL) return(NULL);
result = (char *) palloc(10);
sprintf(result, "%d:%d", mtime->godzina, mtime->minuta);
return(result);
}
```

Po zrealizowaniu wyszczególnionych poprzednio etapów, możliwe jest wprowadzenie do relacji *CZAS* godzin przyjazdu i odjazdu pociągu w ustalonym poprzednio formacie:

```
insert into CZAS values ('WWA12O','10:12','12:14');
insert into CZAS values ('GLC11O','7:30','14:20');
```

Następną fazą jest określenie operacji, które można przeprowadzić na danych typu *MTIME*. Przykładem niech będzie operacja sumowania dwóch zmiennych typu *MTIME*.

```
create operator +                    /* definiowanie operatora dla abstrakcyjnego typu danych */
 (
 leftarg = MTIME,
 rightarg = MTIME,
 procedure = MTIME_ADD,        /* procedura stanowiąca implementację operatora */
 commutator = +
 );
```

Tak jak poprzednio, funkcja *MTIME_ADD* musi zostać napisana w języku C, a następnie dołączona do systemu Postgres95 (polecenie *create function*).

```
mTime *MTIME_ADD(mTime *a, mTime *b)  /* przykładowa postać funkcji MTIME_ADD w C */
{
int reszta=0;
mTime *result;
result = (mTime *)palloc(sizeof(mTime));
result->minuta = a->minuta + b->minuta;
if (result->minuta > 59)                    /* jeśli przepełnienie minut */
 {
 result->minuta -= 60;
 reszta=1;
 }
result->godzina = a->godzina + b->godzina + reszta;
return (result);
}

create function MTIME_ADD(mtime,mtime) returns MTIME
as '/home/pg/mtime/mtime.so' language 'C';
```

Ilustracja działania operatora sumującego:

```
select * from CZAS;

id_pociagu  przyj   odj
--------    ----    ------
WWA12O      10:12   12:14
GLC11O      7:30    14:20
```

```
select (przyj+'0:10') as SPOZNIENIA, id_pociagu from CZAS;
SPOZNIENIA    id_pociagu
------------  ----------
10:22         WWA12O
7:40          GLC11O
```

5. Zapytania historyczne

Jedną z cenniejszych cech systemu Postgres95 jest możliwość realizacji danych historycznych. Postgres95 jest jednym z kilku systemów, razem z Vision, który uwzględnia wspomaganie czasowych właściwości danych.

W bardzo wielu dziedzinach życia codziennego (inżynieria, nauka, medycyna) konieczne jest pamiętanie danych zmieniających się w czasie (ang. time-varying data). W klasycznych (relacyjnych) bazach danych, czas traktuje się w sposób uproszczony, sprowadzając go wyłącznie do atrybutu reprezentowanego obiektu. Aby uniknąć ograniczenia, programiści stosują nienaturalne rozwiązania, tworząc np. kopie archiwalne relacji (*sprzedaż w miesiącu X*).

Obecnie brak jest na rynku jakiegokolwiek komercyjnego systemu zarządzania temporalną bazą danych (ang. Temporal Database Management System - TDBMS). Wiąże się to przede wszystkim z różnicami pomiędzy poszczególnymi danymi zmieniającymi się w czasie oraz z faktem stosowania w systemach informatycznych różnych wariantów reprezentacji czasu, takich jak:

wariant 1. Czas zdefiniowany przez użytkownika (ang. user-defined time).Czas ten jest jednym z atrybutów bazy danych; pozostałe atrybuty nie są od niego zależne. Przykładem czasu definiowanego przez użytkownika może być: data urodzenia, data produkcji wyrobu, data przyjścia faktury.

wariant 2. Czas, w którym dane są prawdziwe w modelowanej rzeczywistości (ang. valid time). Dziedziną tego czasu jest okres od początku do końca okresu trwania modelowanej rzeczywistości. Atrybuty są funkcjonalnie czasowo od siebie zależne. Przykładem ilustrującym ten typ czasu może być fakt: "Anna mieszkała w Gliwicach od 1 kwietnia 1995 do 13 marca 1996".

wariant 3. Czas, w którym dane są zapamiętywane w bazie - tzw. czas transakcyjny (ang. transaction time). Czas transakcyjny rozpoczyna się w momencie wprowadzania danych do bazy, a kończy gdy dane zostają skorygowane; w tym okresie czasu atrybuty są od siebie zależne.

W chwili obecnej dla użytkowników dostępne są jedynie niekomercyjne, eksperymentalne systemy TDBMS. Do takich systemów można zaliczyć Postgres95.

Jakie są korzyści wynikające ze stosowania temporalnych baz danych?

W literaturze dotyczącej tematyki TDBMS wymienia się kilka zagadnień (dziedzin), wymagających uwzględnienia osi czasu:

- automatyczne śledzenie danych zapamiętanych w bazie (np. transakcje finansowe),
- wzrost integralności danych,
- zachowywanie historii zmian atrybutów,
- lepsze uszczegółowienie danych potrzebnych dla hurtowni danych (ang. data mining),
- możliwość tworzenia bilansów materiałowych i tendencji.

Nie bez znaczenia jest również fakt, że temporalny model danych jest niezależny od granularności czasu (gdzie *granularność* jest najmniejszą cząstką czasu, która może być przyjęta przez system).

5.1. Modele danych temporalnych

W literaturze przedmiotu spotyka się trzy zasadnicze modele, różniące się między sobą sposobem ujęcia czasu.

- *Snapshot model* – model "migawka"

W tym modelu reprezentowane są dane z pewnej ściśle określonej chwili. Model relacyjny (w ujęciu klasycznym) używa migawki do reprezentacji bieżącego stanu danych. Możliwe jest więc zadanie pytania migawkowego (ang. snapshot query).

Przykładem tabeli migawkowej (ang. snapshot table) może być konwencjonalna relacja, zawierająca atrybuty czasowe (tzn. definiowane przez użytkownika), reprezentująca wycinek świata rzeczywistego w danej chwili. W takiej tabeli historia nie jest zapamiętywana, a zmiany i uaktualnienia są destrukcyjne.

Relacje o tej strukturze pozwalają uzyskać odpowiedź na jeden typ pytania: *Jaka informacja o danym atrybucie została aktualnie wpisana do bazy?*

Tabela 5

Przykładowe wypełnienie tabeli typu *migawka*

Nazwisko	Wydział	Pensja
Kowalski	EL	850
Nowak	RAU2	800

Zakładając, że korzystamy z relacji o schemacie:

PŁACA (*nazwisko, wydział, pensja*),

posiadającej dwa rekordy dotyczące płac pracowników: Kowalskiego i Nowaka (tabela 5), wykonanie jakichkolwiek zmian w tej relacji (np. poleceniami *INSERT* oraz *DELETE*) spowoduje zmianę zawartości relacji *PŁACA* (tabela 6).

```
delete from PLACA where nazwisko = 'Kowalski';
insert into placa values ('Michno','EL',900);
```

Tabela 6

Zawartość relacji *PŁACA* po dokonanych zmianach

Nazwisko	Wydział	Pensja
Nowak	RAU2	800
Michno	EL	900

- *Event model* - model z buforowaniem (zastosowany w systemie Postgres95).

W modelu tym zachodzi rejestracja zdarzeń w czasie, przy czym zdarzenie jest wystąpieniem danych temporalnych w pewnej chwili czasu. Tabele zdarzeń (ang. event tables) są z natury "append-only" (co oznacza, że błędne zapisy są traktowane jako element historii opisu).

Przykładem danych, które mogą być zamodelowane za pomocą zdarzeń, są np. transakcje finansowe, rezerwacje, sprzedaż.

- *State model* - model stanu.

W modelu statycznym zachodzi rejestracja stanu bazy w czasie (z definicji: *stan* jest to coś, co istnieje przez pewien czas). Każdy zapisany stan jest znacznikowany czasem, w którym stan istnieje; np. raty kredytowe, polisy ubezpieczeniowe, bilanse materiałowe.

5.2. Przykład

Podstawowym typem zapytań, w przypadku historycznych baz danych jest pytanie o historię wartości atrybutów obiektów opisanych w bazie.

Załóżmy istnienie relacji *PRACOWNICY* o schemacie:

PRACOWNICY (*nr_pracownika, nazwisko, imię, nr_zespołu*)

Dzięki mechanizmowi zapytań historycznych w systemie Postgres95 możliwe jest uzyskanie odpowiedzi na następujące zapytania:

① wypisz wszystkich pracowników zatrudnionych obecnie w zespole nr 5,

```
select * from PRACOWNICY where nr_zespołu = 5;
```

lub:

```
select * from PRACOWNICY ['now'] where nr_zespołu = 5;
```

② wypisz stan osobowy zespołu nr 5, w dniu 5 grudnia 1993, o godzinie 13^{40},

```
select * from PRACOWNICY ['December 5,13:40:00,1993'] where nr_zespołu = 5;
```

③ wypisz wszystkich pracowników zatrudnionych w zespole nr 5 w okresie: od 3 lipca 1991 roku do 13 października 1994 roku,

```
select * from PRACOWNICY ['July 3,1991','October 13,1994'] where nr_zespołu = 5;
```

④ wypisz wszystkich pracowników, którzy kiedykolwiek pracowali w zespole nr 5 (aż do chwili obecnej).

```
select * from PRACOWNICY [,'now'] where nr_zespołu = 5;
```

W rzeczywistym systemie możemy się spodziewać znacznie bardziej skomplikowanych struktur niż pokazane poprzednio. Oto przykład bazy danych dla magazynu artykułów spożywczych (uproszczony do celów dydaktycznych; nie bierze pod uwagę np. zwrotów, strat i innych dokumentów, które mogą wystąpić w rzeczywistości).

Baza danych składa się z tabel:

STANY_MAGAZYNOWE (*identyfikator towaru, ilość, cena, data ważności*)

DOKUMENTY_NAGŁÓWKI (*łącznik_do_pozycji, nr_dokumentu, przyjęcie/wydanie_towaru, data_dokumentu, kontrahent*)

DOKUMENTY_POZYCJE (*łącznik, identyfikator towaru, data ważności, cena, ilość*)

TOWARY (*identyfikator towaru, nazwa towaru, jednostka miary*)

Jeśli do magazynu przywożony jest towar, wówczas do systemu wprowadzany jest dokument (uzupełniane są relacje: *DOKUMENTY_NAGŁÓWKI* i *DOKUMENTY_POZYCJE*), który jest następnie księgowany, co oznacza, że kolejne pozycje z dokumentu dodawane są do stanu magazynowego (wypełniana jest relacja *STANY_MAGAZYNOWE*). W chwili sprzedaży do odpowiedniej tabeli wprowadzany jest dokument, ewidencjonujący wydanie towaru konkretnemu klientowi. Dokument jest księgowany, co oznacza odjęcie towaru ze stanu magazynu.

Wykorzystując bazy temporalne można zadać następujące zapytania:

Jaki był zapas magazynowy na dany dzień dla danego asortymentu?

Jaka była wartość magazynu na konkretny dzień (ilość * cena)?

Uzyskanie odpowiedzi na powyższe pytania w relacyjnej bazie danych byłoby bardzo czasochłonne.

6. Reguły

Konwencjonalne systemy baz danych są pasywne: wykonują jedynie odpowiedzi na zapytania (czy uogólniając: transakcje), dokładnie sformułowane przez użytkownika (lub aplikację). Jednakże dla wielu zastosowań ważne jest monitorowanie sytuacji i natychmiastowa odpowiedź na występujące wydarzenie. W aktywnych bazach danych zachowanie to zostało urzeczywistnione przez zastosowanie mechanizmu definiowania i przetwarzania reguł produkcyjnych (ang. production rules) o typowej formie:

ON *zdarzenie*
IF *warunek*
THEN *akcja*

Są to tzw. reguły ECA (ang. Event Condition Action).

Reguły będące wystąpieniami tak zwanych predykatów wirtualnych umożliwiają wywodzenie nowych faktów na podstawie faktów przechowywanych w bazie danych oraz innych faktów wirtualnych, otrzymanych w wyniku wcześniejszego zastosowania reguł innych predykatów. Reguły umożliwiają ponadto specyfikę więzów integralności, ograniczających zbiór możliwych stanów bazy danych oraz modelowanie i kontrolę poprawnej kolejności procesów inicjowanych w systemie.

6.1. Zdarzenia

Zdarzeniami uaktywniającymi (wyzwalającymi) regułę są zwykle modyfikacje danych zapisanych w bazie, jednakże szczegóły i stopień złożoności zdarzeń, warunków i akcji różnią się znacznie w poszczególnych systemach. Niektóre języki manipulowania danymi (np. HiPAC) oferują możliwość definiowania dodatkowych typów zdarzeń:

- zdarzeń czasowych (ang. temporal events), w tym:
  - zdarzeń absolutnych (np. 08:00:00, 1 January 1996)
  - zdarzeń relacyjnych (np. 5 secs after ...)
  - zdarzeń okresowych (np. 08:00:00 every Friday)

oraz:

- zdarzeń złożonych (ang. composite events), będących dowolną kombinacją zdarzeń czasowych i dostępu do danych, określoną za pomocą odpowiednich operatorów (ang. event composition operators) np. *before commit*, *after insert*.

W eksperymentalnym systemie zarządzania bazą danych - Postgres95 - zdarzeniami uaktywniającymi reguły mogą być operacje wprowadzania, usuwania, aktualizacji oraz wyszukiwania danych (*INSERT*, *DELETE*, *UPDATE*, *SELECT*).

6.2. Warunek

Warunek zdefiniowany w treści reguły jest sprawdzany na podstawie danych zawartych w bazie. Jeśli warunek jest spełniony, podejmowana jest określona akcja. W wielu językach regułowych warunki mogą odnosić się do modyfikowanych danych lub stanu bazy danych poprzedzającego modyfikację. W systemie Postgres95 istnieje możliwość definiowania takich warunków przechodnich (tzn. warunków definiowanych na podstawie zmian stanu bazy danych), za pomocą specjalnych zmiennych krotkowych: *new* i *current*. *New* określa uaktualnioną w wyniku operacji modyfikacji wartość atrybutu, *current* wartość bieżącą.

W zależności od typu akcji w regule, występują pewne korelacje:

① reguła wyzwolona poleceniem *INSERT* ma dostęp tylko do nowej wartości atrybutu. Ponieważ wiersz został stworzony przez *INSERT*, stara wartość = *NULL*,

② reguła wyzwolona poleceniem *UPDATE* ma dostęp zarówno do starej, jak i nowej wartości atrybutu,

③ reguła wyzwolona przez polecenie *DELETE* ma dostęp jedynie do starej wartości kolumny. Ponieważ wiersz już nie istnieje, nowa wartość = *NULL*.

6.3. Akcja

Część akcyjna w definicji reguły jest wykonywana po wystąpieniu zdarzenia, w chwili spełnienia odpowiedniego warunku. Jest to przetwarzanie zorientowane na krotkę (ang. tuple oriented rule processing).

Podstawowy algorytm przetwarzania reguł w Postgres95:

1) następuje modyfikacja krotki,

2) operacja modyfikacji spełnia warunki reguł $R_1..R_n$ i uaktywnia je,

3) każda reguła R_i wykonuje swoją akcję, mogącą wyzwalać te same lub dodatkowe reguły.

Przetwarzanie reguł jest rekursywne.

Akcja może spowodować wykonanie takiego samego polecenia, które uaktywniło regułę. W związku z tym akcja reguły może być wyróżniona znacznikiem *instead*. Bez tego znacznika akcja będzie wykonywana wraz z poleceniem wyzwalającym regułę. Użycie *instead nothing* powoduje, że nie będą podejmowane żadne akcje.

6.4. Przykłady

Przypuśćmy, że istnieją klasy: *PRACOWNIK* oraz *DOCHODY* o schematach:

PRACOWNIK (*nr_pracownika*, *nazwisko*, *imię*, *nr_zespołu*)

DOCHODY (*nr_pracownika*, *kwota*, *nr_tematu*)

Zdefiniowano następującą regułę:

```
create rule REGUŁA1
as on insert to PRACOWNIK
do instead
    delete from PRACOWNIK where imię = 'Jan' and nazwisko = 'Kowalski'
```

Uaktywnienie reguły *REGUŁA1* następuje przez wykonanie polecenia *INSERT*, o przykładowej postaci:

```
insert into PRACOWNIK values (10, 'Nowak', 'Marek', 1)
```

Wynikiem działania polecenia *create rule* będzie skasowanie z relacji *PRACOWNIK* rekordu dotyczącego *Jana Kowalskiego*. Dane o *Marku Nowaku* nie zostaną wprowadzone do bazy, ponieważ zastosowano frazę *instead*.

Innym przykładem może być definicja reguły obniżającej pensję wszystkim pracownikom, w przypadku zatrudnienia nowego:

```
create rule REGUŁA2
as on insert to PRACOWNIK
do
    update DOCHODY set kwota = kwota - 50
```

Zdarzeniem wyzwalającym regułę *REGUŁA2*, jest modyfikacja relacji *PRACOWNIK* zleceniem *INSERT:*

```
insert into PRACOWNIK values (100, 'Jeziorański', 'Stanisław', 1)
```

Wynikiem działania reguły jest jednoczesne zmniejszenie kwot wypłaconych pracownikom w ramach poszczególnych tematów oraz dopisanie do relacji *PRACOWNIK* informacji o zatrudnieniu nowego pracownika – *Stanisława Jeziorańskiego*.

Ze względu na to, że język zapytań w systemie Postgres95 jest okrojonym językiem SQL3, w systemie tym reguły służą przede wszystkim do definicji więzów integralności.

Na przykład – dla przedstawionej powyżej klasy *PRACOWNIK* – konieczne jest zagwarantowanie, żeby atrybut *nr_pracownika* przyjmował wartości większe od zera. Aby to osiągnąć, wystarczy zdefiniować regułę o proponowanej postaci:

```
create rule REGUŁA3
as on insert to PRACOWNIK where nr_pracownika <=0
do instead nothing
```

Oczywiście, nie wszystko można zamodelować w systemie, bazując jedynie na regułach. Przykładowo, gdyby konieczne było wypełnienie kolumny *nr_pracownika* następującymi po sobie liczbami całkowitymi, prostszym (i bardziej zrozumiałym niż reguły) rozwiązaniem, okazuje się być zdefiniowanie odpowiedniej funkcji.

Oto proponowane rozwiązanie problemu:

```
create table MY_OIDS (f1 int4);
insert into MY_OIDS values (1);
create function NEW_OIDS() returns int4 as
    'update MY_OIDS set f1 = f1 + 1;
    select f1 from MY_OIDS;'
    language 'sql';
insert into PRACOWNIK values (NEW_OIDS(), 'Kowalczyk', 'Piotr', 3);
```

7. Podsumowanie

Wzrastające wymagania współczesnych aplikacji modelujących skomplikowane fragmenty rzeczywistości zmuszają do rozszerzenia relacyjnego modelu danych. W tym zakresie znaczącą rolę odgrywa system Postgres95, pozwalający na lepszą reprezentację bardziej złożonych struktur danych. W niniejszym artykule przedstawiono sposób definiowania i korzystania z abstrakcyjnych typów danych i procedur oraz zaprezentowano te cechy systemu, niespotykane w systemach konwencjonalnych, które pozwalają traktować go jako podstawę do tworzenia aktywnych i dedukcyjnych baz danych. Podjęto próbę porównania pewnych mechanizmów (np. dotyczących zapytań historycznych) zastosowanych w systemie Postgres95 z innymi znanymi rozwiązaniami.

W związku z tym, że system Postgres95 jest w dalszym ciągu rozwijany (co miesiąc dostępna jest w Internecie nowa, uaktualniona wersja systemu), obecna wersja (v6.2b10) systemu Postgres95 różni się - w pewnym stopniu - od wersji badanej(v6.0). System został obecnie rozszerzony - między innymi - o możliwość definiowania triggerów, dołączono jako standardowe typy: TIME i DATE wraz z procedurami wprowadzania i wyprowadzania danych w żądanym formacie oraz funkcjami agregującymi. Wszystko to wyraźnie wskazuje na fakt, że twórcy systemu chcą maksymalnie zbliżyć relacyjne bazy danych do obiektowych. Umożliwi to dużej grupie użytkowników relacyjnych baz danych, korzystających z ogólnie zaakceptowanego standardu języka baz danych, na bezkonfliktowe przejście do baz danych następnej generacji.

LITERATURA

[1] Yu Andrew, Chen Jolly: The POSTGRES95 User Manual. University of California. Berkeley 1995.

[2] Stonebraker M., Jhingran A., Goh J., Potamianos S.: On rules, procedures, caching and views in Data Base Systems. University of California. Berkeley 1995.

[3] Rowe L., Stonebraker M.: The POSTGRES Data Model. University of California. Berkeley 1988.

[4] Kim Won: Wprowadzenie do obiektowych baz danych. WNT. Warszawa 1996.

[5] Januchta M., Królikowski Z.: POSTGRES jako narzędzie budowy aktywnych i dedukcyjnych systemów baz danych. Informatyka 1994, nr 12, s. 4-11.

[6] Wieczerzycki W.: Dedukcyjne bazy danych. Informatyka 1994, nr 2, s. 2–7.

[7] Widom J., Hanson E., Dayal U.: Active Database Systems. Massachusetts 1994.

[8] Snodgrass R., Pissinou N., Elmasri R.: Towards an Infrastructure for Temporal Databases. Report of ARPA/NSF Workshop. University of Arizona 1994.

Recenzent: Dr hab. inż. Stanisław Wołek

Wpłynęło do Redakcji 13 października 1997 r.

Abstract

To meet the needs of many real-world application, certain concepts from Temporal and Active Databases must be integrated. A new generation of database systems, support active rules for the detection of events, occuring in the database environment and provide mechanisms to store time-varying data in database. Although there is no widely used commercial active-and-temporal database system, Postgres95 is seemed to be very good candidate for being next-generation database system.

This report presents some peculiarities of Postgres95, not existing in traditional DBMS. In section 2 we review concepts of relational database model, in sections: 4, 5, 6 we discuss choosen mechanisms, structures and concepts of Postgres95 such as: composite and user-defined types (Section 4), time travel (Section 5), rule system (Section 6). This paper also demonstrates some examples, applying: non-atomic values of attributes (Section 4.1, Table 2), composite types (Section 4.2), Event-Condition-Action rules (Section 6.4) and

historical queries (Section 5.1). The article concludes with a general recapitulation of the sytem Postgres95 which suggests, that it will be of great importance in creating new-generation database systems.