STUDIA INFORMATICA 2013
Volume 34 Number 2B (112)

Marcin KALETA, Janusz CHWASTOWSKI, Krzysztof CZAJKOWSKI
Instytut Teleinformatyki, Politechnika Krakowska

NIEZALEŻNE PARTYCJONOWANIE DANYCH W BAZACH ORACLE

Streszczenie. Partycjonowanie danych ma duże znaczenie w projektowaniu i implementacji zaawansowanych rozwiązań baz danych. Ze względu na wydajność, bezpieczeństwo i elastyczność w zarządzaniu, rozwiązanie to znalazło szerokie zastosowanie w bazach danych i hurtowniach danych. Opcja ta występuje w wielu serwerach, jednak w przypadku serwerów komercyjnych dostępna jest tylko w wersjach najdroższych. W artykule przedstawiono autorską implementację partycjonowania tabel i indeksów w środowisku Oracle, która może być wykorzystywana w wersjach oprogramowania, które tego mechanizmu nie mają.

Słowa kluczowe: baza danych, partycjonowanie danych, optymalizacja

INDEPENDENT DATA PARTITIONING IN ORACLE DATABASES

Summary. Data partitioning plays an important role in the designing and implementation of advanced database solutions. Due to the performance, protection and flexibility in administration, such a solution has very wide applications in databases and data warehouses. This option exists in many servers, but in the case of commercial systems it is available only in the most expensive versions. In this paper, the authors' implementation of partitioning is presented. This implementation can be used in the version of database servers that do not offer a native partitioning option.

Keywords: database, data partitioning, tuning

1. Wstęp

Dane o dużych rozmiarach przechowywane w systemach baz danych generują wiele trudności, z jakimi muszą się na co dzień mierzyć projektanci, programiści i administratorzy tego typu rozwiązań. Cześć z tych problemów dotyczy zapewnienia dostępności na wypadek awarii, inne skupiają się wokół kwestii wydajności operowania na dużych woluminach, jesz-

cze inne związane są z bieżącymi czynnościami administracyjnymi. Niemal w każdym z problemów użyteczną opcją jest możliwość podziału dużych struktur danych na mniejsze fragmenty (zwykle automatycznie zarządzane), co określane jest mianem partycjonowania – dotyczy to zarówno samych danych, jak i indeksów budowanych na tych danych. Podejście takie daje możliwość rozpraszania danych na wielu niezależnych urządzeniach fizycznych (dyskach), co zwiększa zarówno odporność na awarię (uszkodzenie pliku powoduje czasowy brak dostępu tylko do fragmentu tabeli, podczas gdy pozostałe jej części mogą być w ciągłym użyciu), jak i wydajność (duże operacje mogą być wykonywane równolegle na wielu fragmentach tabeli). Sam fakt podziału tabeli na partycje, nawet umieszczone na tym samym dysku, daje spore korzyści, pozwalając na wykorzystanie mechanizmu polegającego na realizacji operacji tylko na tych partycjach, których zawartości dotyczą kryteria lub kryterium umieszczone w klauzulach filtrujących w poleceniach języka SQL (w środowisku Oracle zwanego partition pruning). Jest to korzystne również przy łączeniu tabel, gdyż łączone mogą być dane z poszczególnych partycji. Podejście takie znacząco zwiększa szybkość realizacji zoptymalizowanego w ten sposób planu wykonania polecenia, poprzez bardzo znaczące zawężenie obszaru przetwarzanych danych. Również bieżące czynności administracyjne mogą być wykonywane szybciej, dzięki elastyczności wynikającej z możliwości niezależnego zarządzania częściami tabeli – ich przenoszenia, kompresowania, obcinania, usuwania, odtwarzania itp. Bardzo istotną korzyścią wynikającą z idei partycjonowania jest możliwość podziału indeksów. Dzięki temu łatwiej jest zarządzać indeksami, a w szczególności dokonywać ich przebudowy, która w takiej sytuacji może być wykonywana tylko na części indeksu. Partycjonowanie może być realizowane dla indeksów zarówno zwykłych, jak i bitmapowych. Dzięki tak licznym zaletom, partycjonowanie znajduje szerokie zastosowanie zarówno w bazach danych, jak i hurtowniach danych.

Partycjonowanie jest obecnie implementowane w wielu popularnych systemach baz danych: Oracle Server [1], IBM DB2 [2], Microsoft SQL Server [3], MySQL Server [4], PostgreSQL [5]. W systemach komercyjnych, chociaż mają one wersje darmowe (np. Oracle Express Edition, IBM DB2 Express-C, MS SQL Server Express), mechanizm partycjonowania jest zazwyczaj płatną opcją, dostępną tylko w wybranych wersjach wyżej wymienionych środowisk. W przypadków systemów firmy Oracle, partycjonowanie dostępne jest tylko w najdroższej i najbogatszej w opcje wersji Enterprise Edition. Niniejszy artykuł prezentuje autorską implementację mechanizmu partycjonowania, wykonaną w środowisku Oracle. Zaprezentowano zarówno metodę implementacji, jak i testy porównawcze z opcją standardowo dostępną w Oracle Enterprise Edition.

2. Partycjonowanie w systemie Oracle

Partycjonowanie standardowo implementowane jest jako podział jednej struktury przechowującej dane – tabeli (mającej jeden tzw. segment danych), na mniejsze porcje – partycje (z których każda ma własny segment), z wykorzystaniem wskazanej listy kolumn – tzw. klucza partycji (liczba kolumn ograniczona jest od 1 do 16). Każdy wiersz tabeli przyporządkowany jest do dokładnie jednej partycji (i przechowywany jest w jej segmencie). Serwer automatycznie kieruje operacje języka SQL: Insert, Update, Delete, do właściwej partycji. Możliwe jest utworzenie do miliona partycji dla jednej tabeli [1].

System Oracle udostępnia wiele rodzajów partycjonowania danych:

- zakresowe (Range) – oparte na zakresach wartości (czasu, liczb lub tekstów),
- listowe (List) – ustalana jest lista wartości dyskretnych (np. teksów) i dla każdej wartości (lub ich grupy) tworzona jest konkretna partycja,
- oparte na funkcji mieszającej (zwane również haszowym) (Hash) – o przynależności do danej partycji decyduje wynik zwracany przez funkcję haszującą, a system zapewnia równomierny rozkład rekordów w partycjach,
- systemowe (System) – o tym, które dane znajdą się w której partycji, decyduje programista,
- referencyjne (Reference) – partycjonowanie jednej tabeli odbywa się na podstawie partycjonowania innej tabeli, przy wykorzystaniu klucza obcego,
- interwałowe (Interval) – jest to rozszerzenie partycjonowania zakresowego, w którym kolejne partycje tworzone są automatycznie, według z góry określonego schematu.

Dostępna jest również możliwość tworzenia tzw. popdpartycji (supartitions), czyli podziału partycji na mniejsze części, stosując inny klucz podziału (inne kolumny). Mówimy wówczas o partycjonowaniu złożonym (dwupoziomowo), które ma następujące warianty:

- zakresowo-zakresowe (Range-Range),
- zakresowo-haszowe (Range-Hash),
- zakresowo-listowe (Range-List),
- listowo-zakresowe (List-Range),
- listowo-haszowe (List-Hash),
- listowo-listowe (List-List).

Możliwe jest również partycjonowanie indeksów, które cechuje się tymi samymi korzyściami, jak ma to miejsce w przypadku tabel, pozwalając jednocześnie na selektywną przebudowę wybranych części (partycji) indeksu, co znacznie skraca czas takiej operacji. Partycjonowanie indeksów może odbywać się:

- lokalnie – co oznacza podział analogiczny do tabeli, na kolumnach której nałożony jest dany indeks,

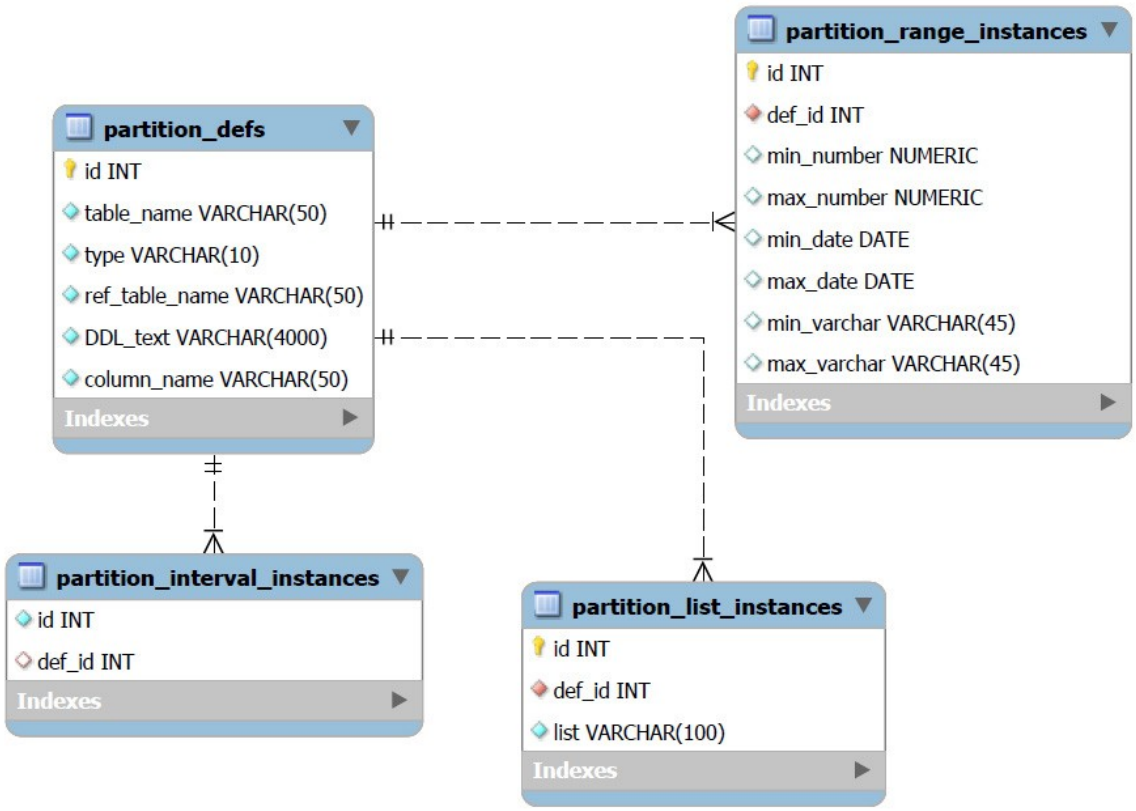
- globalnie – podział indeksu na partycje wykonany jest inaczej (według innego modelu lub innych kolumn) niż podział tabeli. Możliwe są partycjonowania: zakresowe i haszowe.

Dla partycjonowanej tabeli możliwe jest także tworzenie indeksów, które nie są partycjonowane.

Partycjonowanie zarówno tabel, jak i indeksów realizowane jest za pomocą języka SQL. Wykorzystywane są do tego celu odpowiednie klauzule poleceń CREATE TABLE i CREATE INDEX. Również zarządzanie partycjami (ich tworzenie, usuwanie, przenoszenie, łączenie i dzielenie) odbywa się za pomocą dodatkowych klauzul wyżej wymienionych poleceń. Optymalizator w trakcie tworzenia planu wykonania polecenia, mając informację na temat partycjonowanych tabel i indeksów oraz ich partycji, jest w stanie generować korzystniejsze plany wykonania niż w przypadku organizacji danych bez wykorzystania partycjonowania.

3. Proponowana implementacja partycjonowania tabel

3.1. Przygotowanie partycjonowania



Rys. 1. Diagram encji tabel bazy danych z informacjami o partycjonowaniu
Fig. 1. Entity diagram of database with information about partitioning

Opisana w niniejszym artykule implementacja dostarcza funkcjonalności partycjonowania danych oraz struktur indeksujących w bazie danych. Opracowane zostały implementacje następujących metod partycjonowania:

- partycjonowanie zakresowe,
- partycjonowanie listowe,
- partycjonowanie interwałowe,
- partycjonowanie referencyjne.

Na rysunku 1 przedstawiono schemat tabel bazy danych, przechowującej informacje na temat utworzonych partycjonowań.

Implementacja została zrealizowana w formie pakietu języka PL/SQL. Pierwszym etapem jest zdefiniowanie konkretnego partycjonowania dla danej tabeli – procedury *create_range_partitioning, create_list_partitioning create_interval_partitioning, create_reference_partitioning*. Procedury te dodają stosowne wpisy na temat nałożonych partycjonowań do tabeli *partition_defs*, która jest wykorzystywana na dalszych etapach.

Kolejnym etapem jest utworzenie poszczególnych partycji. W przypadku partycjonowań zakresowego i listowego, tworzenie nowych partycji odbywa się za pomocą wywołania procedur z pakietu (odpowiednio *create_range_partitioning, create_list_partitioning*), natomiast dla pozostałych metod tworzenie partycji odbywa się automatycznie. Podczas tworzenia partycji istnieje możliwość wskazania przestrzeni tabel, gdzie umieszczona zostanie partycja (parametr *p_tbs_name* we wszystkich procedurach *create_*_partition*).

Ostatnim etapem jest stworzenie wszystkich niezbędnych wyzwalaczy oraz widoków. Tworzone są one poprzez wywołanie procedury *build_range_partitioning, build_list_partitioning, build_reference_partitioning* albo *build_interval_partitioning*, odpowiednio do typu partycjonowania. Procedury te korzystają z dynamicznego natywnego SQL [6]. Wszystkie procedury umieszczone są w jednym pakiecie, a procedura dotycząca tworzenia partycji zakresowych jest przeciążona i posiada trzy wersje:

```
create_range_partitioning(p_table_name IN VARCHAR2, p_column_name IN VARCHAR2,
p_type IN VARCHAR2, p_DDL_text IN VARCHAR2);
create_range_partition(p_table_name IN VARCHAR2, p_tbs_name IN VARCHAR2,
p_maxvalue IN NUMBER);
create_range_partition(p_table_name IN VARCHAR2, p_tbs_name IN VARCHAR2,
p_maxvalue IN DATE);
create_range_partition(p_table_name IN VARCHAR2, p_tbs_name IN VARCHAR2,
p_maxvalue IN VARCHAR2);
build_range_partitioning(p_table_name IN VARCHAR2);

create_list_partitioning(p_table_name IN VARCHAR2, p_column_name IN VARCHAR2,
p_DDL_text IN VARCHAR2);
create_list_partition(p_table_name IN VARCHAR2, p_tbs_name IN VARCHAR2,
p_list_value IN VARCHAR2);
build_list_partitioning(p_table_name IN VARCHAR2);
```

```
create_interval_partitioning(p_table_name IN VARCHAR2,p_column_name IN VARCHAR2,
p_DDL_text IN VARCHAR2);
build_interval_partitioning(p_table_name IN VARCHAR2, p_interval IN NUMBER);

create_reference_partitioning(p_table_name IN VARCHAR2, p_ref_table_name IN
VARCHAR2, p_column_name IN VARCHAR2, p_DDL_text IN VARCHAR2);
```

Realizacja podpartycjonowania zakłada umieszczenie jako nazwy tabeli nazwy widoku (realizującego pierwszy poziom partycjonowania). W ten sposób można uzyskać liczbę poziomów partycjonowania większą niż 2, co jest maksymalną liczbą poziomów partycjonowania Oracle. Jednak nadmierna ich liczba z pewnością komplikuje strukturę bazy danych oraz utrudnia pracę optymalizatorowi.

Fragment kodu umożliwiający automatyczne przebudowywanie wyzwalaczy typu INSTEAD OF dla wstawiania, kasowania i modyfikowania oraz automatycznej aktualizacji perspektywy dla wyrażeń SELECT, w przypadku partycjonowania zakresowego, został zamieszczony poniżej.

Na wejściu procedura otrzymuje nazwę tabeli, dla której ma nastąpić przebudowa widoku i wyzwalaczy. Do zmiennej typu rekordowego def_record (zdefiniowanego w tym samym pakiecie, ale jako zmienna globalna, poza procedurą) wczytywane są informacje o partycjonowaniu, które następnie wykorzystywane są w dalszej części procedury. Jako pierwszy przebudowywany jest widok (a dokładnie przygotowywana jest instrukcja, mająca tę czynność zrealizować), którego treść konstruowana jest na podstawie pętli LOOP, z wykorzystaniem operacji UNION ALL (aby dołączać wyniki z kolejnych tabel pełniących rolę partycji). Następnie przygotowywane są instrukcje przebudowujące trzy wyzwalacze typu INSTEAD OF, które będę realizować operacje INSERT, UPDATE i DELETE na fizycznych tabelach (które służą jako partycje). Całość działania procedury kończy się wykonaniem przygotowanych wcześniej instrukcji, tworzących widok i wyzwalacze (cztery operacje natywnego dynamicznego SQL – EXECUTE IMMEDIATE).

```
build_range_partitioning(p_table_name IN VARCHAR2)
BEGIN
    SELECT * INTO def_record FROM partition_defs WHERE UPPER(table_name) =
      UPPER(p_table_name);
       SELECT COUNT(1) INTO v_count FROM partition_range_instances WHERE def_id
    =
      def_record.id;
       v_view := 'CREATE OR REPLACE VIEW '||p_table_name||' AS ';
       FOR rec IN (SELECT * FROM partition_range_instances WHERE def_id =
    ef_record.id) LOOP
           v_i := v_i + 1;
       v_view := v_view || ' SELECT * FROM ' || def_record.table_name || rec.id
       || ' WHERE ' || def_record.column_name || ' BETWEEN ' || CASE WHEN
       TO_CHAR(rec.min_number) IS NOT NULL THEN TO_CHAR(rec.min_number) WHEN
       rec.min_date IS NOT NULL THEN 'TO_DATE(''' ||
       TO_CHAR(rec.min_date,'yyyymmddhh24:mi:ss')||''','yyyymmddhh24:mi:ss'')'
       WHEN rec.min_varchar IS NOT NULL THEN '''' || rec.min_varchar || '''' END
       || ' AND ' || CASE WHEN TO_CHAR(rec.max_number) IS NOT NULL THEN
       TO_CHAR(rec.max_number) WHEN rec.max_date IS NOT NULL THEN 'TO_DATE('''
       ||
       TO_CHAR(rec.max_date,'yyyymmddhh24:mi:ss')||''','yyyymmddhh24:mi:ss'')'
```

```
    WHEN rec.max_varchar IS NOT NULL THEN '''' || rec.max_varchar || ''''
    END;
        IF v_i != v_count THEN
            v_view := v_view || ' UNION ALL ';
        ELSE
            NULL;
        END IF;
    END LOOP;
    v_i := 0;
    v_insert := 'CREATE OR REPLACE TRIGGER i' || def_record.table_name ||
    ' INSTEAD OF INSERT ON ' || def_record.table_name || ' FOR EACH ROW
    DECLARE BEGIN CASE ';
    FOR rec IN (SELECT * FROM partition_range_instances WHERE def_id =
ef_record.id) LOOP
        v_i := v_i + 1;
    v_insert := v_insert || ' WHEN :new.' || def_record.column_name || '
    BETWEEN ' || CASE WHEN TO_CHAR(rec.min_number) IS NOT NULL THEN
    TO_CHAR(rec.min_number) WHEN rec.min_date IS NOT NULL THEN 'TO_DATE('''
    ||
    TO_CHAR(rec.min_date,'yyyymmddhh24:mi:ss')||''','''yyyymmddhh24:mi:ss'')'
    WHEN rec.min_varchar IS NOT NULL THEN '''' || rec.min_varchar || '''' END
    || ' AND ' || CASE WHEN TO_CHAR(rec.max_number) IS NOT NULL THEN
    TO_CHAR(rec.max_number) WHEN rec.max_date IS NOT NULL THEN 'TO_DATE('''
    ||
    TO_CHAR(rec.max_date,'yyyymmddhh24:mi:ss')||''','''yyyymmddhh24:mi:ss'')'
    WHEN rec.max_varchar IS NOT NULL THEN '''' || rec.max_varchar || '''' END
    ||
           ' THEN INSERT INTO '|| def_record.table_name || rec.id || ' VALUES
    ' || get_insert_columns(def_record.table_name||1);
        IF v_i != v_count THEN
            NULL;
        ELSE
            v_insert := v_insert || ' END CASE; END;';
        END IF;
    END LOOP;
    v_i := 0;
    v_delete := 'CREATE OR REPLACE TRIGGER d' || def_record.table_name ||
    ' INSTEAD OF DELETE ON ' || def_record.table_name || ' FOR EACH ROW
    DECLARE BEGIN CASE ';
    FOR rec IN (SELECT * FROM partition_range_instances WHERE def_id =
ef_record.id) LOOP
        v_i := v_i + 1;
    v_delete := v_delete || ' WHEN :old.' || def_record.column_name || '
    BETWEEN ' || CASE WHEN TO_CHAR(rec.min_number) IS NOT NULL THEN
    TO_CHAR(rec.min_number) WHEN rec.min_date IS NOT NULL THEN 'TO_DATE('''
    ||
    TO_CHAR(rec.min_date,'yyyymmddhh24:mi:ss')||''','''yyyymmddhh24:mi:ss'')'
    WHEN rec.min_varchar IS NOT NULL THEN '''' || rec.min_varchar || '''' END
    || ' AND ' || CASE WHEN TO_CHAR(rec.max_number) IS NOT NULL THEN
    TO_CHAR(rec.max_number) WHEN rec.max_date IS NOT NULL THEN 'TO_DATE('''
    ||
    TO_CHAR(rec.max_date,'yyyymmddhh24:mi:ss')||''','''yyyymmddhh24:mi:ss'')'
    WHEN rec.max_varchar IS NOT NULL THEN '''' || rec.max_varchar || '''' END
    || ' THEN DELETE FROM '|| def_record.table_name || rec.id || ' WHERE ' ||
    def_record.column_name || ' = :old.' || def_record.column_name || ' ; ';
        IF v_i != v_count THEN
            NULL;
        ELSE
            v_delete := v_delete || ' END CASE; END;';
        END IF;
    END LOOP;
    v_i := 0;
    v_update := 'CREATE OR REPLACE TRIGGER u' || def_record.table_name ||
    ' INSTEAD OF UPDATE ON ' || def_record.table_name || ' FOR EACH ROW
    DECLARE BEGIN CASE ';
```

```
        FOR rec IN (SELECT * FROM partition_range_instances WHERE def_id =
    ef_record.id) LOOP
            v_i := v_i + 1;
        v_update := v_update || ' WHEN :old.' || def_record.column_name || '
        BETWEEN ' || CASE WHEN TO_CHAR(rec.min_number) IS NOT NULL THEN
        TO_CHAR(rec.min_number) WHEN rec.min_date IS NOT NULL THEN 'TO_DATE('''
        ||
        TO_CHAR(rec.min_date,'yyyymmddhh24:mi:ss')||''','yyyymmddhh24:mi:ss'')'
        WHEN rec.min_varchar IS NOT NULL THEN '''' || rec.min_varchar || '''' END
        || ' AND ' || CASE WHEN TO_CHAR(rec.max_number) IS NOT NULL THEN
        TO_CHAR(rec.max_number) WHEN rec.max_date IS NOT NULL THEN 'TO_DATE('''
        ||
        TO_CHAR(rec.max_date,'yyyymmddhh24:mi:ss')||''','yyyymmddhh24:mi:ss'')'
        WHEN rec.max_varchar IS NOT NULL THEN '''' || rec.max_varchar || '''' END
        ||
            ' THEN UPDATE '|| def_record.table_name || rec.id || ' SET ' ||
        get_update_columns(def_record.table_name||1) || ' WHERE ' ||
        def_record.column_name || ' = :old.' || def_record.column_name || ' ; ';
            IF v_i != v_count THEN
                NULL;
            ELSE
                v_update := v_update || ' END CASE; END;';
            END IF;
        END LOOP;
        EXECUTE IMMEDIATE v_view;
        EXECUTE IMMEDIATE v_insert;
        EXECUTE IMMEDIATE v_delete;
        EXECUTE IMMEDIATE v_update;
END;
```

Efektem działania powyższej procedury są między innymi widok (dla operacji SELECT) oraz wyzwalacz (dla INSERT) o strukturach analogicznych do poniższego kodu:

```
CREATE VIEW nazwa_widoku AS
    SELECT kolumna1, kolumna2, kolumna3 FROM tabela WHERE kol_partycjonowana
        BETWEEN wartość1 AND wartość2
    UNION ALL
    SELECT kolumna1, kolumna2, kolumna3 FROM tabela2 WHERE kol_partycjonowana
        BETWEEN wartość2 AND wartość3
    ... ;
CREATE OR REPLACE TRIGGER inazwa_tabeli
    INSTEAD OF INSERT ON nazwa_tabeli FOR EACH ROW
    DECLARE
    BEGIN
        CASE WHEN :new.nazwa_kolumny BETWEEN wartość1 AND wartość2
        THEN INSERT INTO nazwa_tabeli_partycji VALUES ...
    ... ;
```

3.2. Metodyka pomiaru

Pomiar wydajności stworzonych algorytmów został przeprowadzony w następujący sposób. Czas wykonania każdej z operacji: SELECT, INSERT, DELETE oraz UPDATE, był mierzony w funkcji liczby jej wykonań. Pomiar ten został przeprowadzony w następujących przypadkach: bez zastosowania partycjonowania, przy zastosowaniu partycjonowania dostarczonego przez Oracle oraz w przypadku zastosowania implementacji autorskiej partycjonowania. Za miary efektywności przyjęto współczynniki zdefiniowane poniżej:

$$r(n) = \frac{czas\ wykonania\ n\ operacji\ z\ wykorzystaniem\ partycjonowania\ autorskiego}{czas\ wykonania\ n\ operacji\ z\ wykorzystaniem\ partycjonowania\ Oracle},$$

$$d(n) = \frac{czas\ wykonania\ n\ operacji\ z\ wykorzystaniem\ partycjonowania\ autorskiego}{czas\ wykonania\ n\ operacji\ bez\ zastosowania\ partycjonowania}.$$

Wszystkie wyniki opierały się na zbiorze 2 000 000 000 rekordów w bazie danych, zajmujących około 120 GB. Rekordy zostały wygenerowane poprzez wykorzystanie generatora liczb losowych w Oracle, zawartego w pakiecie DBMS_RANDOM [8]. Rekordy obejmowały kolumny: liczbową (identyfikator wiersza), tekstową oraz typu data.

Dla operacji SELECT przeprowadzono testy na zapytaniach prostych (na jednej tabeli, z wyszukiwaniem pojedynczych wartości oraz wyszukiwaniem zakresowym), a także połączeniach (INNER JOIN i OUTER JOIN).

Należy w tym miejscu zauważyć, iż w celu potwierdzenia poprawności uzyskanych wyników testy wydajnościowe zostały przeprowadzone również dla mniejszego (12 GB) zbioru danych. W trakcie prowadzenia pomiarów okazało się, że błędy stosunków r i d są zaniedbywalne.

3.3. Partycjonowanie zakresowe

Przeprowadzono analizę różnych możliwych rozwiązań implementacji partycjonowania zakresowego. Rozważano zastosowanie natywnego SQL, realizowanego poleceniem EXECUTE IMMEDIATE. Analizie poddana została również operacja SELECT ... FROM TABLE().Testy wydajnościowe wykazały jednak, iż obie metody są wolniejsze od wykonywania czystego PL/SQL wewnątrz procedur oraz funkcji. Konsekwencją przeprowadzonych testów było wybranie czystego PL/SQL z logiką CASE-WHEN.

Partycjonowanie zostało oparte na widoku będącym złączeniem wszystkich tabel. W celu obsługi dodawania rekordów do partycjonowanej tabeli, utworzono wyzwalacz INSTEAD OF INSERT FOR EACH ROW. Zadaniem tego wyzwalacza jest porównywanie kolumny klucza ze wzorcem, który został stworzony podczas definiowania partycjonowania tabeli. Wyzwalacz INSTEAD OF UPDATE FOR EACH ROW umożliwia natomiast modyfikację rekordów w partycjonowanej tabeli. W przypadku zmiany kolumny klucza na klucz znajdujący się w innym zakresie, wykonywane są dwie operacje: INSERT oraz DELETE. Wyzwalacz ten zachowuje się więc w analogiczny sposób do tradycyjnego rozwiązania, istniejącego w bazie danych ORACLE, z aktywną opcją Enable Row Movement. Na potrzeby usuwania rekordów z odpowiednich tabel według kolumny klucza, opracowany został wyzwalacz INSTEAD OF DELETE FOR EACH ROW.

Wyniki przedstawione w tabeli 1 wskazują, że prezentowany w niniejszej pracy mechanizm partycjonowania zakresowego dla operacji INSERT okazał się znacznie wolniejszy

zarówno od mechanizmu Oracle, jak i w przypadku braku partycjonowania. Współczynniki *r* i *d* przyjęły bowiem wartości znacząco przewyższające 1. Niska wydajność proponowanego rozwiązania wynika w głównej mierze z wysokiej złożoności czasowej logiki CASE-WHEN.

Tabela 1

Względne czasy wykonania operacji dla partycjonowania zakresowego

Liczba operacji	**Operacja INSERT**		**Operacja DELETE**		**Operacja UPDATE**		**Operacja SELECT**	
	r	**d**	**r**	**d**	**r**	**d**	**r**	**d**
10	N/A	N/A	1,08	0,29	1,08	0,27	1,02	0,23
100	N/A	N/A	1,07	0,27	1,07	0,28	1,01	0,23
500	4,98	6,66	1,07	0,27	1,06	0,29	1,01	0,24
1000	5,01	6,68	1,08	0,28	1,07	0,29	1,01	0,23
2000	4,99	6,67	N/A	N/A	N/A	N/A	N/A	N/A
5000	5,00	6,67	N/A	N/A	N/A	N/A	N/A	N/A
10000	5,00	6,67	N/A	N/A	N/A	N/A	N/A	N/A

W przypadku operacji DELETE i UPDATE, za pomocą stworzonego mechanizmu jest możliwe uzyskanie wyników zbliżonych do tych, które osiągane są przy wykorzystaniu partycjonowania dostarczonego przez Oracle. Biorąc pod uwagę, iż w operacjach typu DELETE i UPDATE najwięcej czasu zajmuje wyszukanie rekordu, a nie samo jego fizyczne usunięcie lub aktualizacja, partycjonowanie sporządzone w ramach tej pracy osiąga stosunkowo dobry wynik.

Omawiana implementacja okazała się z kolei bardzo szybka dla operacji typu SELECT. Współczynnik r przyjął dla tej operacji wartość bliską 1. Wyniki uzyskiwane za pomocą proponowanego mechanizmu są niemal identyczne z tymi, które osiągane są przez wykorzystanie mechanizmu Oracle. Współczynnik d dla operacji SELECT ukształtował się natomiast znacząco poniżej 1. Stworzony mechanizm cechuje się zatem wysoką wydajnością w porównaniu do sytuacji, gdy mechanizm partycjonowania nie jest stosowany.

Należy zauważyć, iż dla operacji SELECT przetestowane zostały zapytania złożone (OUTER JOIN, INNER JOIN) oraz zapytania proste. Wyłącznie w przypadku zapytań prostych, w których w warunku WHERE nie było kolumny klucza, wszystkie trzy sposoby przechowywania danych dla operacji SELECT okazywały się być jednakowe.

3.4. Partycjonowanie listowe

Implementacja partycjonowania listowego zrealizowana została w sposób analogiczny do implementacji partycjonowania zakresowego. Wyniki testów przedstawiono w tabeli 2.

Opisany w niniejszej pracy mechanizm partycjonowania listowego dla operacji typu INSERT okazał się znacznie mniej wydajny od mechanizmu dostarczonego przez Oracle oraz od sytuacji, gdy mechanizm partycjonowania nie jest stosowany. Dla operacji typu DELETE i UPDATE proponowane rozwiązanie cechuje się wysoką wydajnością. Uzyskane

wyniki są podobne do tych, które osiągane są za pomocą partycjonowania dostarczanego przez Oracle. Uzyskany czas wykonywania operacji SELECT dla proponowanego mechanizmu partycjonowania listowego jest zatem niemal identyczny ze średnim czasem wykonania operacji SELECT dla mechanizmu partycjonowania listowego Oracle.

Tabela 2

Względne czasy wykonania operacji dla partycjonowania listowego

Liczba operacji	Operacja INSERT		Operacja DELETE		Operacja UPDATE		Operacja SELECT	
	r	d	r	d	r	d	r	d
10	N/A	N/A	1,06	0,23	1,06	0,22	1,03	0,22
100	N/A	N/A	1,07	0,24	1,03	0,23	1,02	0,23
500	3,39	5,62	1,06	0,23	1,02	0,24	1,01	0,24
1000	3,38	5,63	1,06	0,23	1,02	0,23	1,01	0,23
2000	3,37	5,63	N/A	N/A	N/A	N/A	N/A	N/A
5000	3,38	5,62	N/A	N/A	N/A	N/A	N/A	N/A
10000	3,38	5,62	N/A	N/A	N/A	N/A	N/A	N/A

3.5. Partycjonowanie interwałowe

Implementacja partycjonowania interwałowego znacząco różni się od implementacji obu wcześniej omówionych typów partycjonowania. Wynika to z konieczności zapewnienia automatycznego tworzenia wymaganych partycji. Implementacja oraz testy wydajnościowe przeprowadzone zostały dla dwóch rozwiązań. Pierwsze zakłada wykorzystanie pakietu DBMS_JOB i tworzenie partycji w wyznaczonych godzinach (w czasie mniejszego obciążenia serwera). Drugie rozwiązanie opiera się na wykorzystaniu wyzwalacza systemowego ON SYSTEM ERROR, który umożliwia wychwycenie wyjątku, w tym przypadku związanego z brakiem partycji i jej utworzeniem. Rozwiązanie opierające się na wykorzystaniu pakietu DBMS_JOB, mimo iż jest bardzo wydajne, gdyż nie wymaga dynamicznego przebudowywania widoków oraz wyzwalaczy podczas pracy systemu, cechuje się niewystarczającą elastycznością. W sytuacji gdy znacząca liczba rekordów zostaje wstawiona do tabeli w okresie zwiększonej aktywności serwera bazy danych, mechanizm związany z widokami nie przynosi korzyści w postaci skrócenia czasu operacji. Rozwiązanie wykorzystujące dynamiczny SQL wraz z wyzwalaczem systemowym typu ON SYSTEM ERROR charakteryzuje się natomiast wysoką elastycznością, przy znacząco ograniczonej wydajności.

Z testów, których wyniki przedstawiono w tabeli 3, wynika, że w przypadku operacji INSERT oba rozważane rozwiązania okazały się gorsze od implementacji Oracle oraz od sytuacji braku partycjonowania, przy czym wykorzystanie DBMS_JOB jest bardziej wydajne od zastosowania natywnego dynamicznego SQL.

W przypadku pozostałych operacji obie proponowane implementacje cechują się porównywalną wydajnością. Wydajność tę uznać należy za stosunkowo wysoką zarówno w odnie-

sieniu do mechanizmu partycjonowania Oracle, jak i w odniesieniu do sytuacji, gdy mechanizm partycjonowania nie jest stosowany.

Tabela 3

Względne czasy wykonania operacji dla partycjonowania interwałowego

Liczba operacji	Operacja INSERT z DBMS_JOB		Operacja INSERT z natywnym dynamicznym SQL		Operacja DELETE		Operacja UPDATE		Operacja SELECT	
	r	d	r	d	r	d	r	d	r	d
10	N/A	N/A	N/A	N/A	1,08	0,27	1,07	0,27	1,02	0,23
100	N/A	N/A	N/A	N/A	1,08	0,27	1,07	0,28	1,02	0,24
500	5,31	16,39	8,35	29,51	1,07	0,27	1,06	0,29	1,01	0,24
1000	5,30	16,39	8,36	29,54	1,08	0,28	1,07	0,28	1,01	0,23
2000	5,29	16,38	8,38	29,67	N/A	N/A	N/A	N/A	N/A	N/A
5000	5,30	16,39	8,40	29,98	N/A	N/A	N/A	N/A	N/A	N/A
10000	5,30	16,38	8,56	33,33	N/A	N/A	N/A	N/A	N/A	N/A

3.6. Partycjonowanie referencyjne

W celu zaimplementowania partycjonowania referencyjnego, rozszerzono pakiet PARTITIONS o dodatkowe funkcje. Funkcje te dodają nowe partycje referencyjne (w tabeli podrzędnej) w momencie dodawania nowych partycji zakresowych w tabeli nadrzędnej. Wyzwalacze INSTEAD OF FOR EACH ROW odpowiedzialne są za poszukiwanie w tabeli referencyjnej odpowiedniej wartości. Po jej znalezieniu wykonują one wywoływaną operację na odpowiedniej tabeli partycjonowania referencyjnego.

W tabeli 4 zaprezentowano wyniki testów dla tego przypadku. Opracowany mechanizm partycjonowania referencyjnego dla operacji typu INSERT i SELECT okazał się mniej wydajny zarówno w porównaniu do mechanizmu Oracle, jak i w porównaniu do sytuacji, gdy mechanizm partycjonowania nie jest stosowany. Natomiast dla operacji UPDATE i DELETE opracowane rozwiązanie jest mniej wydajne od implementacji Oracle, ale wydajniejsze od sytuacji, gdy partycjonowanie nie było stosowane.

Tabela 4

Względne czasy wykonania operacji dla partycjonowania referencyjnego

Liczba operacji	Operacja INSERT		Operacja DELETE		Operacja UPDATE		Operacja SELECT	
	r	d	r	d	r	d	r	d
10	N/A	N/A	1,23	0,33	1,21	0,35	1,21	0,31
100	N/A	N/A	1,22	0,34	1,22	0,33	1,21	0,31
500	7,22	26,98	1,22	0,33	1,21	0,33	1,21	0,32
1000	7,22	26,99	1,22	0,33	1,22	0,33	1,21	0,31
2000	7,23	27,00	N/A	N/A	N/A	N/A	N/A	N/A
5000	7,23	27,01	N/A	N/A	N/A	N/A	N/A	N/A
10000	7,23	27,00	N/A	N/A	N/A	N/A	N/A	N/A

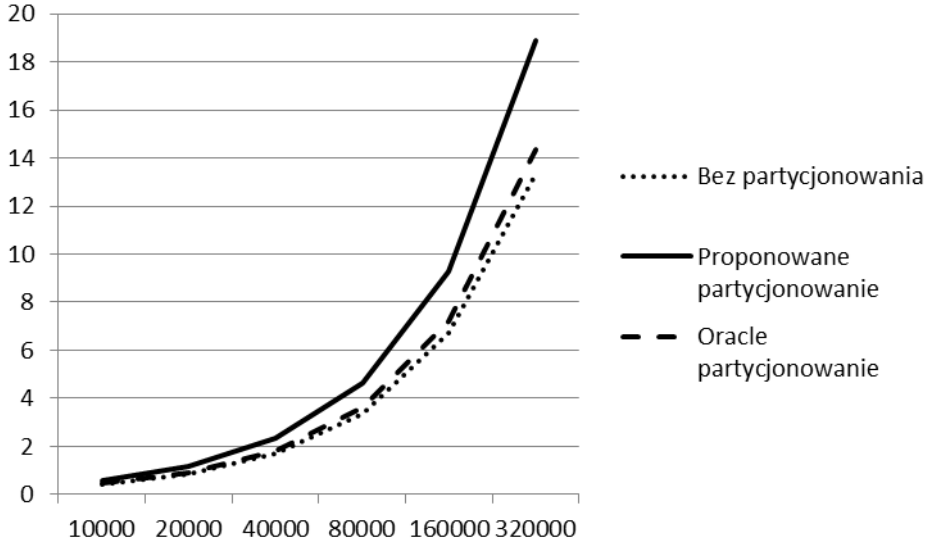
4. Proponowana implementacja partycjonowania indeksów

Podczas partycjonowania tabeli, indeksy, które są dla tej tabeli tworzone, mogą być również partycjonowane. Indeksy dzielone są według poszczególnych partycji lub w sposób od nich niezależny. Dzięki temu możliwe staje się znaczne przyspieszenie operacji wyszukiwania interesującego rekordu dla tabel z bardzo dużą liczbą rekordów [7].

Stworzenie rozwiązania działającego równie szybko co indeksy partycjonowane Oracle okazało się jednak niemożliwe. Wszystkie operacje wykorzystujące indeksy na tabelach partycjonowanych mechanizmem Oracle okazały się znacznie szybsze od operacji na indeksach stworzonych w ramach prezentowanej implementacji, co było spowodowane brakiem możliwości automatycznego odwołania się do wartości ROWID (jednoznacznych identyfikatorów poszczególnych wierszy) oraz bardzo czasochłonnym przebudowywaniem indeksów. Jedyną dostępną formą partycjonowania indeksów okazało się tworzenie indeksu na każdej z pojedynczych tabel będących partycjami.

4.1. Partycjonowanie zakresowe dla indeksów

Implementacja partycjonowania zakresowego indeksów przebiegała analogicznie do implementacji partycjonowania zakresowego dla tabel. Dodatkowo stworzona została procedura CREATE_INDEX, która korzystając z dynamicznego SQL, automatycznie tworzy indeksy na danej kolumnie we wszystkich tabelach.



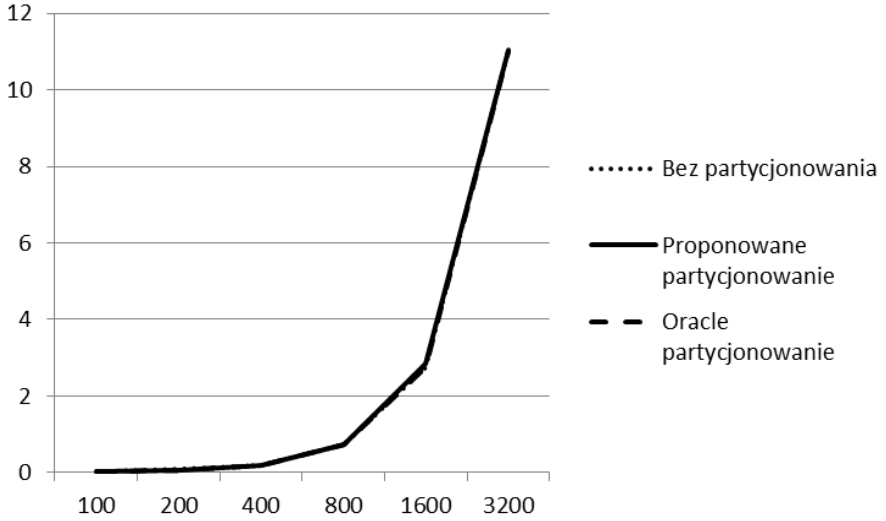
Rys. 2. Zależność czasu wykonania operacji prostych typu SELECT od liczby tych operacji dla kolumny partycjonowanej zakresowo

Fig. 2. Dependence between execution time of simple SELECT operations and numbers of these operations for range partitioning of a column

Na rysunku 2 przedstawiono zależność czasu wykonania operacji od liczby operacji prostych typu SELECT dla kolumny partycjonowanej zakresowo (wyszukiwania z operatorem

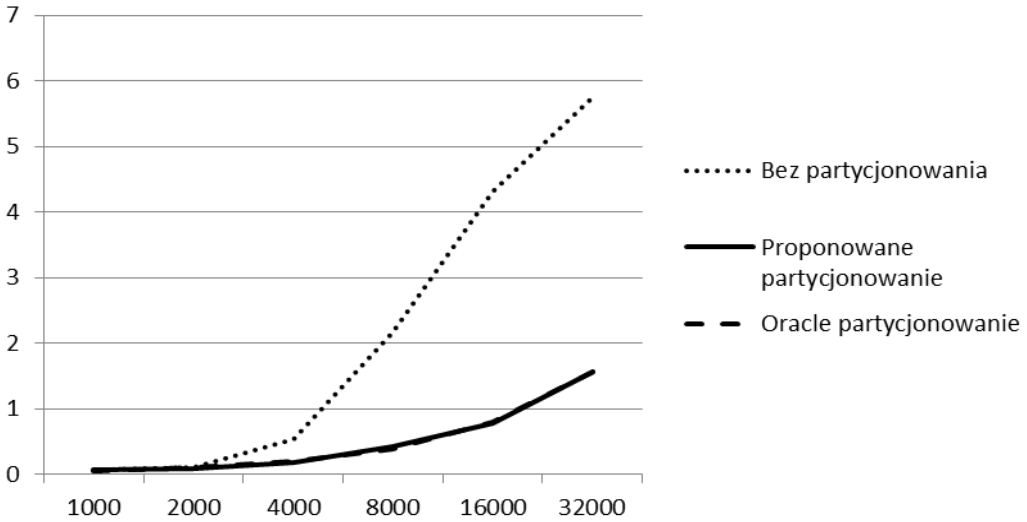
„=” po kolumnie indeksowanej, będącej kluczem partycji). Wyniki osiągnięte dla proponowanego mechanizmu partycjonowania zakresowego indeksów oraz dla mechanizmu Oracle okazały się mniej wydajne od wyników uzyskanych dla operacji prostych typu SELECT, gdy mechanizm partycjonowania nie został wykorzystany.

Na rysunku 3 zilustrowano zależność czasu wykonania od liczby operacji złożonych (złożone warunki, złączenia tabel) typu SELECT dla kolumny partycjonowanej zakresowo. W przypadku operacji złożonych typu SELECT, wydajność wszystkich sposobów przechowywania danych ukształtowała się na zbliżonym poziomie. Na wyniki wydajnościowe żadnego wpływu nie wywarły typy złączeń tabel: OUTER JOIN bądź też INNER JOIN.



Rys. 3. Zależność czasu wykonania operacji złożonych typu SELECT od liczby tych operacji dla kolumny partycjonowanej zakresowo

Fig. 3. Dependence between execution time execution time of complex SELECT operations and numbers of these operations for range partitioning of a column



Rys. 4. Zależność czasu wykonania operacji złożonych typu SELECT od liczby tych operacji dla partycjonowania zakresowego kolumny z dodatkowymi indeksami

Fig. 4. Dependence between execution time of complex SELECT operations and numbers of these operations for range partitioning of a column with additional indexes

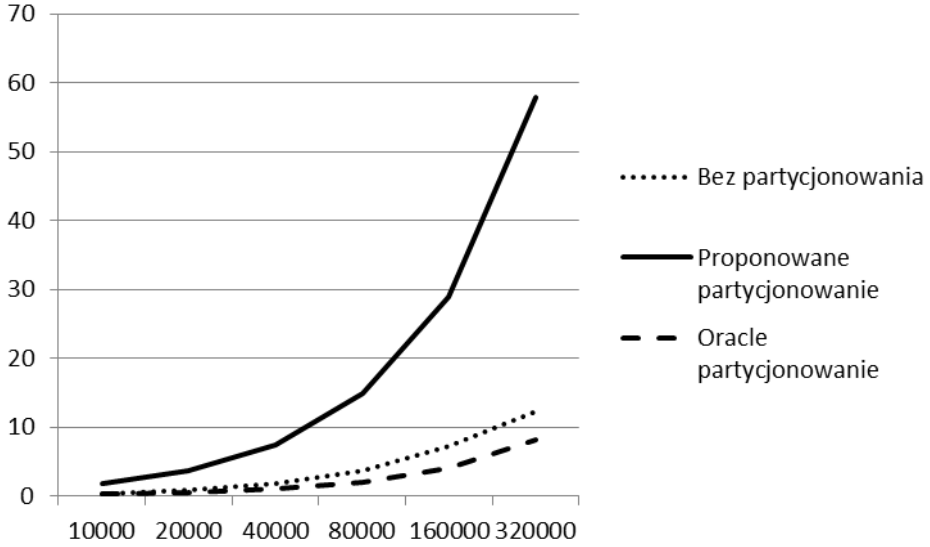
Rysunek 4 prezentuje zależność pomiędzy czasem wykonania a liczbą operacji złożonych typu SELECT dla partycjonowania zakresowego kolumny z dodatkowymi indeksami. Opracowany mechanizm partycjonowania zakresowego kolumny z dodatkowymi indeksami cechuje się zbliżoną wydajnością do mechanizmu partycjonowania Oracle. Wydajność obu mechanizmów kształtuje się na poziomie wyższym niż w przypadku, gdy mechanizm partycjonowania nie jest stosowany.

4.2. Partycjonowanie listowe dla indeksów

Testy wydajnościowe wykazały, iż niską wydajnością dla operacji prostych typu SELECT cechują się zarówno opracowany mechanizm partycjonowania listowego kolumn, jak i mechanizm Oracle. Znacznie lepsze wyniki wydajnościowe osiągnięte zostały, gdy nie został zastosowany żaden z mechanizmów partycjonowania. Jest to sytuacja analogiczna do przypadku partycjonowania zakresowego indeksów.

Również w przypadku operacji złożonych typu SELECT dla kolumny partycjonowanej listowo występuje analogia do przypadku stosowania partycjonowania zakresowego. Wszystkie trzy testowane warianty: partycjonowanie Oracle, opracowane partycjonowanie oraz brak partycjonowania, dają bardzo zbliżone wyniki.

Na rysunku 5 zaprezentowano zależność między czasem a liczbą operacji złożonych typu SELECT dla partycjonowania listowego kolumny z dodatkowymi indeksami. Wyższą wydajność niż w przypadku, gdy mechanizm partycjonowania nie został wykorzystany, osiągnęły, zarówno proponowany mechanizm partycjonowania listowego kolumny z dodatkowymi indeksami, jak i mechanizm partycjonowania Oracle.



Rys. 5. Zależność czasu wykonania operacji od liczby operacji złożonych typu SELECT dla partycjonowania listowego kolumny z dodatkowymi indeksami

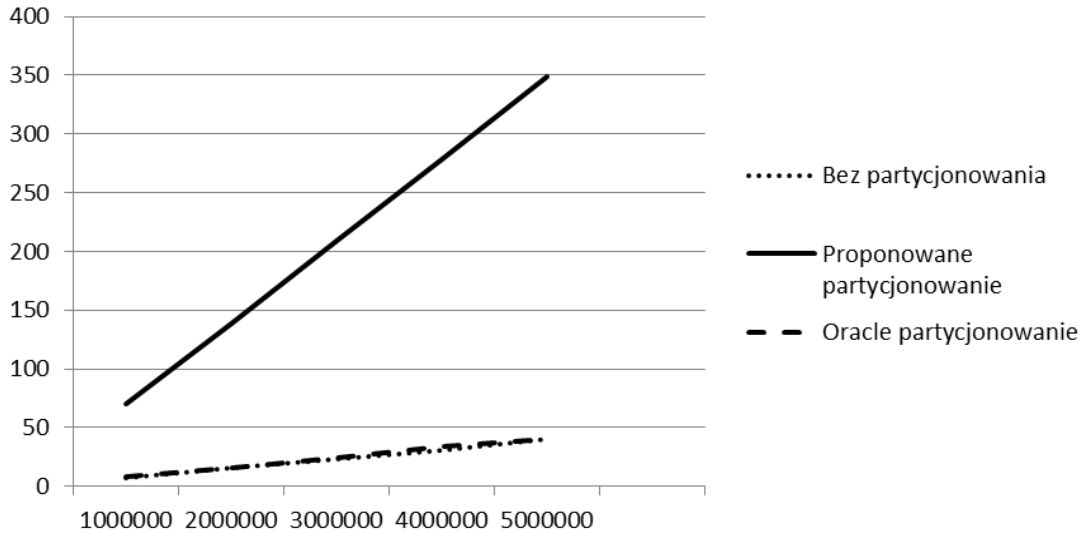
Fig. 5. Dependence between execution time of complex SELECT operations and numbers of these operations for list partitioning of a column with additional indexes

5. Podpartycjonowanie

Podpartycjonowanie polega na podziale partycji tabeli na części zwane podpartycjami (ang. *subpartitions*). Mechanizm partycjonowania Oracle umożliwia dokonanie wyłącznie dwóch poziomów partycjonowania. Przedstawiony w niniejszej pracy mechanizm oferuje bardziej rozbudowane możliwości odnośnie do podpartycjonowania – pozwala on na dowolną liczbę poziomów partycjonowania.

Najlepsze rezultaty w niniejszej pracy osiągnięto dla danych z trzema poziomami podpartycjonowań: zakresowym – listowym – zakresowym. Należy pamiętać, iż wraz ze wzrostem liczby poziomów podpartycjonowania zwiększa się złożoność obliczeniowa całego mechanizmu partycjonowania. Konieczne staje się sprawdzenie coraz większej liczby warunków oraz wykonanie coraz większej liczby operacji, co prowadzi do zmniejszania się wydajności mechanizmu przetwarzania danych.

Rysunek 6 obrazuje zależność czasu od liczby operacji typu INSERT dla podpartycjonowania. Opracowana implementacja podpartycjonowania wykazuje się efektywnością niższą zarówno od podpartycjonowania Oracle, jak i w przypadku jej braku.



Rys. 6. Zależność czasu wykonania operacji typu INSERT od liczby tych operacji dla podpartycjonowania

Fig. 6. Dependence between execution time of INSERT operations and numbers of these operations for subpartitions

W celu zaprezentowania zależności pomiędzy średnim czasem wykonania operacji SELECT dla tworzonego podpartycjonowania a średnim czasem wykonania operacji SELECT dla podpartycjonowania Oracle oraz dla braku partycjonowania, posłużono się współczynnikami r i d. Współczynniki te zdefiniowano analogicznie do definicji w punkcie 3.2. W przypadku operacji złożonych typu SELECT, stworzony mechanizm podpartycjonowania zakres – lista – zakres osiągnął stosunkowo wysokie wyniki wydajności. Czas wyszukiwania w tabeli zawierającej 2 mld rekordów (wygenerowanych losowo za pomocą pakietu

DBMS_RANDOM) okazał się być nieznacznie krótszy od mechanizmu podpartycjonowania Oracle (tabela 5). Wynik wydajności dla współczynnika r ukształtował się zatem na poziomie nieznacznie niższym od 1 – około 5%. Równie obiecujące okazały się wyniki wydajności opisanego mechanizmu podpartycjonowania w odniesieniu do sytuacji, gdy mechanizm partycjonowania nie jest wykorzystywany – skrócenie czasu wykonania o około 80%.

Tabela 5

Współczynniki względne czasów wykonania operacji SELECT dla podpartycjonowania

Liczba operacji	**Operacja SELECT**	
	r	**d**
10	0,95	0,19
100	0,96	0,19
500	0,95	0,18
1000	0,95	0,19

6. Podsumowanie

W artykule przedstawiono przykład autorskiego rozwiązania dla partycjonowania: zakresowego, listowego, interwałowego oraz referencyjnego. Przeprowadzone pomiary pokazują, że w przypadku jego wykorzystania operacja INSERT jest znacznie (kilka do parędziesięciu razy) wolniejsza niż w przypadku wykorzystania mechanizmu partycjonowania dostarczonego przez Oracle lub gdy nie wykorzystywano partycjonowania. Natomiast analogiczne pomiary przeprowadzone dla operacji DELETE, UPDATE i SELECT wskazują, że autorskie rozwiązanie jest do około 10% wolniejsze niż to dostarczone przez Oracle i znacząco szybsze niż w przypadku braku partycjonowania, dla przypadków pełnych przeglądów tabeli (full table scan) lub wyszukiwania i złączeń po nieindeksowanych kolumnach (operacje te mogą zostać ograniczone do wybranych partycji). W przypadku zastosowania podpartycjonowania rozwiązanie prezentowane powyżej prowadzi do około 5% skrócenia czasu operacji w stosunku do tego uzyskiwanego przy zastosowania oprogramowania Oracle oraz około 80% w stosunku do braku podpartycjonowania.

Przedstawiona implementacja zawiera również opcję partycjonowania indeksów. W przypadku złożonych operacji SELECT dla partycjonowania zakresowego kolumn z dodatkowymi indeksami, efektywność proponowanego rozwiązania jest niemal taka sama, jak w przypadku rozwiązania Oracle i znacznie korzystniejsza w stosunku do sytuacji, gdy partycjonowanie nie jest stosowane (rysunek 3). Taka sama sytuacja występuje dla przypadku listowego partycjonowania indeksów.

Wykonane pomiary świadczą o praktycznej niezależności uzyskanych wyników od liczby przeprowadzonych operacji.

Zaprezentowane rozwiązanie w porównaniu do partycjonowania dostarczanego przez firmę Oracle ma zarówno zalety, jak i wady. Niewątpliwym ograniczeniem proponowanego rozwiązania jest brak możliwości obsługi wszystkich rodzajów partycjonowania, jakie oferowane są przez producenta serwera. Wadą jest również konieczność przygotowania odpowiednich struktur. Przewaga rozwiązania Oracle przejawia się także w możliwości łączenia partycji, jeżeli warunki ograniczające łączne tabele pozwalają zawęzić przeglądany zakres danych tylko do wybranych segmentów danych (inaczej mówiąc konkretnych partycji), a optymalizator tworzy plan wykonania, mający łączyć tylko te partycje (funkcjonalność zwana parallel partition-wise join). Takiego rozwiązania nie obejmuje proponowana implementacja. Istotnymi zaletami proponowanego podejścia są natomiast: brak konieczności ponoszenia dodatkowych kosztów związanych z zakupem opcji partycjonowania, możliwość wykorzystania w wersjach serwera innych niż Enterprise Edition oraz możliwość elastycznego dostosowania do specyficznych potrzeb (np. partycjonowanie na więcej niż dwóch poziomach). Zaprezentowana implementacja nie obejmuje, w aktualnej wersji, partycjonowania po wielu kolumnach (partycjonowanie Oracle dopuszcza taką możliwość nawet na 16 kolumnach, dla partycjonowań range i hash). Jednak z uwagi na fakt, iż proponowana implementacja pozwala na partycjonowanie na wielu poziomach, nie wydaje się to istotnym ograniczeniem. Proponowane rozwiązanie jest również na tyle elastyczne, że możliwe jest rozbudowanie mechanizmów o partycjonowanie wielokolumnowe.

BIBLIOGRAFIA

1. Oracle® Database VLDB and Partitioning Guide 11g Release 2 (11.2).
2. Database Partitioning, Table Partitioning, and MDC for DB2 9. International Technical Support Organization, 2007.
3. Talmage R.: Partitioned Table and Index Strategies Using SQL Server 2008. SQL Server Technical Article, 2008.
4. MySQL Reference Manual, Chapter 17. Partitioning: http://dev.mysql.com/doc/refman/5.6/en/partitioning.html.
5. PostgreSQL, Documentation, 5.9. Partitioning: http://www.postgresql.org/docs/current/static/ddl-partitioning.html.
6. Oracle® Database PL/SQL Language Reference 11g Release 2 (11.2).
7. Lightstone S., Teorey T., Nadeau T.: Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more. Morgan Kaufmann Publishers, 2007.
8. Feuerstein S., Beresniewicz J., Dawes C.: Oracle PL/SQL. Pakiety i funkcje. Leksykon kieszonkowy. Helion, Gliwice 2001.

Wpłynęło do Redakcji 30 grudnia 2012 r.

Abstract

Data partitioning plays an important role in the designing and implementation of the advanced database solutions. Due to the performance, protection and flexibility in administration, such a solution has a very wide range of applications in databases and data warehouses. This option exists in many servers, but in the case of commercial systems it is available only in the most expensive versions. In this paper, the authors' devised partitioning implementation is presented. This implementation can be used in the version of database servers that do not include a native partitioning option.

In the presented approach, the table partitioning by range, list, hash and reference is implemented. It has been shown (cf. Tables 1, 2 and 3) that the devised partitioning solution by range, list and interval, respectively, has nearly the same efficiency in the SELECT, DELETE and UPDATE operations as that seen for the Oracle implementation. In the case of the INSERT operation, the Oracle implementation as well as the proposed one are less efficient when compared to the no-partitioning case. In the case of the reference partitioning (see Table 4), the effects of the proposed solution for the SELECT, UPDATE and DELETE operations are not so satisfactory as those for other partitioning methods. The measured efficiency is much higher than in the case of the partitioning not being applied.

The presented implementation includes also the partitioning for indexes. In the case of complex SELECT operations for the range partitioned column with additional indexes, the efficiency of the proposed solution is nearly the same as that of the Oracle implementation and much higher than the one obtained without partitioning (cf. Fig. 3). The same situation is observed in the case of the list partitioning of indexes.

The subpartitioning mechanism implemented in the presented solution delivers results comparable to those seen for the Oracle subpartitioning option (Fig. 5). Contrary to the Oracle mechanism, the proposed solution enables any number of partitioning levels.

Adresy

Marcin KALETA: Politechnika Krakowska, Wydział Fizyki, Matematyki i Informatyki, Instytut Teleinformatyki, ul. Warszawska 24, 31-155 Kraków, Polska, kaleta.marcin88@gmail.com.

Janusz CHWASTOWSKI: Politechnika Krakowska, Wydział Fizyki, Matematyki i Informatyki, Instytut Teleinformatyki, ul. Warszawska 24, 31-155 Kraków, Polska, jchwastowski@pk.edu.pl.

Krzysztof CZAJKOWSKI: Politechnika Krakowska, Wydział Fizyki, Matematyki i Informatyki, Instytut Teleinformatyki, ul. Warszawska 24, 31-155 Kraków, Polska, kc@pk.edu.pl.