Wojciech Sitek
Numer indeksu: 293148

Sprawozdanie BD Optymalizacja

1. Wprowadzenie

Baza danych opiera się na modelowaniu ochrony danych osobowych w firmie. W tym ćwiczeniu wykorzystano relacje EMPLOYEE (zawierająca dane o pracownikach) oraz ACCESS_LOG (zawierająca dane o dostępie do określonych danych osobowych przez określonych pracowników). W relacji EMPLOYEE znajduje się 18309, a w ACCESS_LOG 9947 wygenerowanych przez procedurę PL/SQL rekordów z losowymi danymi.

2. Problem badawczy

Skanowanie

Chciano dowiedzieć się, jaki jest rozkład ilości generowanych logów przez pracowników, to znaczy ilu pracowników wygenerowało jeden *log*, ilu dwa, i tak dalej. W tym celu powstało poniższe zapytanie, które zostało poddane testom optymalizacyjnym:

```
select logow_na_pracownika, count(logow_na_pracownika) ilosc_emp_z_tyloma_log
from (
        select count(emp_id) logow_na_pracownika from Access_log
        group by emp_id
)
group by logow_na_pracownika
order by ilosc_emp_z_tyloma_log desc;
```

Zapytanie daje następujący wynik:

LOGOW_NA_PRACOWNIKA	ILOSC_EMP_Z_TYLOMA_LOG
1	5821
2	1587
3	251
4	42
5	5
6	1

Złączenie

Drugi problem dotyczy łączenia dwóch relacji. Zapytanie ma na celu wypisanie typu dostępu do danych osobowych i departamentu, w którym pracuje pracownik, który jest połączony z tym rekordem. W tym celu utworzono następujące zapytanie:

```
SELECT ACCESS_TYPE, DEPARTMENT
FROM ACCESS_LOG AL
JOIN EMPLOYEE E
ON AL.EMP_ID = E.ID;
```

Dało ono następujące wyniki:

ACCESS_TYPE|DEPARTMENT
N (null)
M WEITI
N WEITI
N WEITI
N WEITI
N MINI
M WEITI
M UW
D WEITI
D kakfylgbufmxrn
A ohjkxt
A drkdclvanei
M ybpmsjhatyshzgf
A piafhnu
A wfifys
I tak dalej…

Indeks

Aby umożliwić różne rodzaje skanowania, stworzono indeks na kluczu obcym pracowników za pomocą następującego zapytania:

```
create index acc_log_emp on access_log(emp_id);
```

3. Optymalizacja skanowania

W celu przeprowadzenia analizy optymalizacji skanowania, wykorzystano różne podpowiedzi dla optymalizatora, i obserwowano „Plan Wykonania” zapytania w Oracle SQL Developer. Oto wyniki przeprowadzonych testów:

Rodzaj skanowania	**Podpowiedź w zapytaniu**	***Cardinality***	**Koszt**
Fast full index scan	/*+INDEX_FFS(ACCESS_LOG)*/	9947	20
Index scan	/*+INDEX(ACCESS_LOG) */	9947	95
Full table scan	/*+FULL(ACCESS_LOG)*/	9947	139

Wnioski

Zgodnie z powyższą tabelą, łatwo zauważyć, że koszt wykonania zapytania przy użyciu optymalizacji związanej z indeksem jest zdecydowanie mniejszy, co poprawia wydajność bazy danych (przy jednoczesnym koszcie pamięciowym przy przechowywaniu i czasowym przy uaktualnianiu indeksów).

Użyty pełny skan tabeli, bez użycia indeksu, spowodował siedmiokrotny wzrost kosztu. Jest to spowodowane koniecznością przeszukiwania w głównej pamięci bazy danych. Warto zwrócić uwagę na fakt, iż przyśpieszenie wykonania najbardziej zagnieżdżonego zapytania SELECT przyśpiesza wykonanie pozostałych podzapytań (koszt pozostałych operacji jest porównywalny do podanego w tabeli).

Dochodzimy więc do wniosku, iż wtedy, kiedy tylko jest to możliwe, należy używać optymalizacji, ponieważ poprawia ona zdecydowanie szybkość bazy danych. Warto zaznaczyć, że w większości przypadków ta optymalizacja jest dokonywana automatycznie przez system bazy danych, gdy tylko jest to możliwe.

W naszych uwarunkowaniach zasobowo-sprzętowych (mała ilość danych, porównywalnie do ilości miejsca na dysku serwerowym) jest możliwe stworzenie indeksu na poszczególnych kolumnach, co pozwala na najbardziej optymalny sposób skanowania tabeli, czyli *Fast full index scan*. W przypadku braku pamięci lub gdyby zależało nam na szybkich operacjach dodawania, modyfikowania i usuwania rekordów z tabeli, wtedy stworzenie indeksu mogłoby być nieopłacalne, a co za tym idzie, trzeba by było zadowolić się skanowaniem pełnym (*full scan)*.

4. Optymalizacja złączeń

Analogicznie do skanowania, przeprowadzono analizę kosztu z różnymi podpowiedziami dla optymalizatora dla złączeń (join). W poniższej tabeli przedstawione są wyniki analizy.

Rodzaj skanowania	**Podpowiedź w zapytaniu**	***Cardinality***	**Koszt**
Nested loop join	/*+USE_NL (AL E)*/	9947	10088
Sort-merge join	/*+USE_MERGE(AL E) */	9947	302
Hash join	/*+USE_HASH(AL E)*/	9947	300

Wnioski

Złączenie metodą zagnieżdżonej pętli jest najbardziej kosztowne, ponieważ wymaga przejścia po iloczynie kartezjańskim dwóch tabel, co można zauważyć na ponad 30 razy większym koszcie operacji złączenia dwóch tabel wypełnionych danymi niż w pozostałych sposobach złączenia. *Sort-merge join* oraz *hash join* zawierają najprawdopodobniej optymalne, bardzo podobne wyniki kosztowe. Wykorzystanie *hash join* jest więc w przypadku tych uwarunkowań zasobowo-sprzętowych (stosunkowo mała ilość danych, w porównaniu do miejsca na dysku, a więc możliwość zachowania danych tymczasowych w pamięci serwera bazy danych) optymalnym sposobem na przeprowadzenie złączenia.