Bazy danych

8. Złączenia — ciąg dalszy. Grupowanie.

P. F. Góra

http://th-www.if.uj.edu.pl/zfs/gora/

semestr letni 2007/08

Filtry

Wyobraźmy sobie zapytanie

```
SELECT ... FROM T1
JOIN T2 ON ...
WHERE 𝒫(T1) AND 𝒬(T2) AND ℛ(T1,T2);
```

Predykaty $\mathcal{P}$ i $\mathcal{Q}$, działające tylko na kolumnach tabel, odpowiednio, $\mathtt{T}_1$ i $\mathtt{T}_2$, są *filtrami*, wybierają bowiem pewne podzbiory wierszy tych tabel. Predykat $\mathcal{R}$, działający na kolumnach obu tabel, można (i należy) traktować jako fragment warunku złączenia (złączenie theta). Im *mniejszy* procent wierszy wybiera z tabeli filtr, tym *większą* ma on selektywność. Dla efektywności złączenia korzystne jest używanie filtru o największej selektywności możliwie najwcześniej.

Jak baza danych realizuje złączenia

Złączenia wewnętrzne zdefiniowane są jako podzbiory iloczynu kartezjańskiego odpowiednich tabel, jednak *na ogół* nie są one realizowane w ten sposób, iż najpierw wykonywany jest iloczyn kartezjański, potem zaś wybierane są odpowiednie wiersze. Sposób realizacji złączenia nie może wpłynąć na ostateczny wynik zapytania, ale może wpłynąć (i wpływa!) na czas realizacji zapytania, zajętość pamięci itp. Jak zatem baza danych realizuje złączenie? Najczęściej używa się następujących trzech algorytmów:

1. Złączenie pętli zagnieżdżonych (*nested loops*)

1. Baza przegląda pierwszą tabelę wejściową. Wiersze nie spełniające filtru nałożonego tylko na tą tabelę odrzuca, wiersze spełniające filtr przekazuje dalej.
2. Do każdego wiersza z pierwszej tabeli dopasowywane są wiersze z drugiej tabeli, spełniające warunek złączenia (złączenie wewnętrzne) lub wartości `NULL`, jeśli wierwszy takowych nie ma (złączenie zewnętrzne). Odrzucane są wiersze nie spełniające warunków dla dotychczas wykorzystanych tabel, czyli filtru dla drugiej tabeli i warunków obejmujących łącznie pierwszą i drugą tabelę.
3. Analogicznie postępujemy dla trzeciej i każdej następnej tabeli.

Takie złączenie ma postać zagnieżdżonych pętli — najbardziej zewnętrzna obiega pierwszą tabelę wejściową, najbardziej wewnętrzna — ostatnią. Z tego względu istotne jest, aby *pierwsza*, najbardziej zewnętrzna pętla, odrzucała możliwie dużo wierszy oraz żeby połączenia następowały po kolumnach indeksowanych, wówczas bowiem łatwo jest znaleźć wiersze pasujące do aktualnego klucza złączenia.

Na każdym etapie wymagana jest jedynie informacja o aktualnie przetwarzanej pozycji oraz zawartość konstruowanego w danej chwili wiersza wynikowego — cały proces nie wymaga dużej pamięci.

Złączenie pętli zagnieżdżonych może mieć warunki złączenia w postaci nierówności. Wiele RDBMS wyraźnie preferuje ten typ złączenia.

2. Złączenie haszujące (mieszające, *hash join*)

Stosuje się tylko do złączeń wewnętrznych, w których warunki złączenia mają postać równości. *Teoretycznie* jest to wówczas najszybszy algorytm złączenia, ale *praktycznie* tak wcale nie musi być.

Złączane tabele przetwarzane są niezależnie. Cały algorytm przebiega w dwu fazach:

- W *fazie budowania* dla mniejszej (po zastosowaniu filtru) tabeli tworzona jest *tablica haszująca* (tablica mieszająca, *hash table*), powstała przez zastosowanie *funkcji haszującej* do kluczy złączenia. Teoretycznie rozmieszcza on "hasze" przyporządkowane różnym kluczom równomiernie w pamięci. Algorytm działa szczególnie szybko, jeśli cała tablica haszująca mieści się w pamięci.

- W *fazie wyszukiwania* sekwencyjnie przeglądana jest większa tabela. Na kluczu złączenia każdego wiersza wykonywana jest ta sama funkcja haszująca; jeżeli odpowiedni element znajduje się w tablicy haszującej dla *pierwszej* tabeli, wiersze są łączone. Jeżeli nie, wiersz drugiej tabeli jest odrzucany. Jeżeli tablica haszująca znajduje się w całości w pamięci, średni czas wyszukiwania elementów jest stały i niezależny od rozmiarów tablicy — to właśnie stanowi o efektywności tego algorytmu.

Problemy ze złączeniem haszującym

Efektywność złączenia haszującego silnie zależy od doboru funkcji haszującej. Idealna funkcja haszująca ma tę własność, że zmiana pojedynczego bitu w kluczu, zmienia połowę bitów hasza, i zmiana ta jest niezależna od zmian spowodowanych przez zmianę dowolnego innego bitu w kluczu. Idealne funkcje haszujące są trudne do zaprojektowania lub kosztowne obliczeniowo, stosuje się więc funkcje "gorsze", co jednak prowadzić może do *kolizji*: Fukncja haszująca dwóm różnym kluczom usiłuje przypisać tę samą wartość hasza. Aby tego uniknąć, stosuje się różne techniki, co jednak powiększa koszt obliczeniowy algorytmu.

Innym problemem jest *grupowanie*: Wbrew założeniom, funkcje haszujące mają tendencję do nierównomiernego rozmieszczania haszy w pamięci, co zmniejsza efektywnośc wykorzystania pamięci i powiększa czas wyszukiwania.

3. Złączenie sortująco-scalające (*sort and merge*)

Tabele odczytuje się niezależnie i stosuje do nich właściwe filtry, po czym wynikowe zbiory wierszy sortuje się względem klucza złączenia. Następnie dwie posortowane listy zostają scalone. Baza danych odczytuje na przemian wiersze z każdej listy. Jeżeli warunek złączenia ma postać równości, baza porównuje górne wiersze i odrzuca te, które znajdują się na posortowanej liście wcześniej niż górny wiersz drugiej tabeli, zwraca zaś te, które wzajemnie sobie odpowiadają. Procedura scalania jest szybka, ale procedura wstępnego sortowania jest wolna, o ile nie ma gwarancji, że *oba* zbiory wierszy mieszczą się w pamięci.

Uwaga!

Jeżeli ze względów estetycznych lub innych *wynikowy* zbiór wierszy pewnego zapytania ma być posortowany, to jeśli ten wynikowy zbiór w całości nie mieści się w pamięci, może to *znacznie* spowolnić czas wykonywania zapytania.

Wymuszanie kolejności wykonywania złączeń

Przypuśćmy, że łaczymy więcej niż dwie tabele i że warunek złączenia ma postać

$$\texttt{... AND } T_1.k_2 = T_2.k_2 \texttt{ AND } T_1.k_3 = T_3.k_3 \texttt{ ...}$$

Chcemy wykonać pętle zagnieżdżone po tabelach T_1 i T_2 *przed* odwołaniem do tabeli T_3. Aby odroczyć wykonanie złączenia, *trzeba uczynić je zależnym* od danych ze złączenia, które powinno zostać wykonane wcześniej.

```
... AND T1.k2=T2.k2 AND T1.k3+0*T2.k2=T3.k3 ...
```

Druga wersja jest logicznie równoważna pierwszej, jednak baza interpretując je, po lewej stronie drugiego złączenia trafia na wyrażenie, które zależy tak od tabeli T_1, jak i T_2 (nie ma znaczenia, że wartość z tabeli T_2 nie może wpłynąć na wynik), nie wykona więc złączenia z T_3 przed złączeniem z T_2.

Uwaga: Oczywiście to samo stosuje się do złączeń sformułowanych w postaci

```
... T1 JOIN T2 ON T1.k2=T2.k2 JOIN T3 ON T1.k3=T3.k3 ...
```

Podzapytania

- Podzapytania pozwalają na tworzenie *strukturalnych* podzapytań, co umożliwia izolowanie poszczególnych części instrukcji. Istnieniu podzapytań SQL zawdzięcza słowo “strukturalny” w swojej nazwie.

- Podzapytania zapewniają alternatywny sposób wykonywania zadań, które w inny sposób można realizować tylko poprzez skomplikowane złączenia. Niektórych zadań *nie da* się wykonać bez podzapytań.

- Podzapytania zwiększają czytelność kodu.

Podzapytania — operatory `ANY`, `IN`, `ALL`

Przypuśćmy, że mamy dwie tabele:

```
mysql> SELECT * FROM Alfa;
+--------+--------+
| Litera | Liczba |
+--------+--------+
| A      |     10 |
| B      |     20 |
| C      |     25 |
+--------+--------+
3 rows in set (0.01 sec)
```

```
mysql> SELECT * FROM Beta;
+--------+------------+
| Liczba | InnaLiczba |
+--------+------------+
|     20 |         30 |
|     20 |         18 |
|     25 |         30 |
|     27 |         38 |
|     20 |         -5 |
|     10 |       NULL |
+--------+------------+
6 rows in set (0.00 sec)
```

coś operator_porównania `ANY` *(podzapytanie)*

oznacza, iż “coś” musi spełniać odpowiednią relację z *jakimiś* wynikami podzapytania

```
mysql> SELECT * FROM Alfa WHERE
    -> Liczba > ANY
    -> (SELECT Liczba FROM Beta);
+--------+--------+
| Litera | Liczba |
+--------+--------+
| B      |     20 |
| C      |     25 |
+--------+--------+
2 rows in set (0.10 sec)
```

```
mysql> SELECT Litera FROM Alfa WHERE
    -> Liczba <= ANY
    -> (SELECT InnaLiczba/3 FROM Beta);
+--------+
| Litera |
+--------+
| A      |
+--------+
1 row in set (0.00 sec)
```

Słowo kluczowe `ALL` oznacza, że warunek musi być spełniony dla wszystkich wierszy zwracanych przez podzapytanie.

```
mysql> SELECT * FROM Alfa
    -> WHERE Liczba > ALL
    -> (SELECT InnaLiczba FROM Beta);
Empty set (0.09 sec)
```

```
mysql> SELECT * FROM Alfa
    -> WHERE Liczba > ALL
    -> (SELECT 0.5*InnaLiczba FROM Beta);
+--------+--------+
| Litera | Liczba |
+--------+--------+
| B      |     20 |
| C      |     25 |
+--------+--------+
2 rows in set (0.01 sec)
```

Słowo kluczowe `IN` jest równoważne z warunkiem `= ANY`.

```
mysql> SELECT * FROM Alfa
    -> WHERE Liczba IN
    -> (SELECT InnaLiczba/3 FROM Beta);
+--------+--------+
| Litera | Liczba |
+--------+--------+
| A      |     10 |
+--------+--------+
1 row in set (0.00 sec)
```

```
mysql> SELECT * FROM Alfa
    -> WHERE Liczba = ANY
    -> (SELECT InnaLiczba/3 FROM Beta);
+--------+--------+
| Litera | Liczba |
+--------+--------+
| A      |     10 |
+--------+--------+
1 row in set (0.00 sec)
```

Podzapytania — operator `EXISTS`

`... EXISTS (`*podzapytanie*`) ...`

warunek jest prawdziwy jeśli podzapytanie zwraca niepusty zbiór wierszy

```
mysql> SELECT * FROM Alfa
    -> WHERE EXISTS
    -> (SELECT * FROM Beta WHERE
    -> InnaLiczba > 100);
Empty set (0.00 sec)
```

```
mysql> SELECT * FROM Alfa
    -> WHERE EXISTS
    -> (SELECT * FROM Beta WHERE
    -> InnaLiczba < 50);
+--------+--------+
| Litera | Liczba |
+--------+--------+
| A      |     10 |
| B      |     20 |
| C      |     25 |
+--------+--------+
3 rows in set (0.00 sec)
```

Podzapytania skorelowane

```
mysql> SELECT * FROM Alfa
    -> WHERE EXISTS
    -> (SELECT * FROM Beta WHERE InnaLiczba/3=Alfa.Liczba);
+--------+--------+
| Litera | Liczba |
+--------+--------+
| A      |     10 |
+--------+--------+
1 row in set (0.01 sec)
```

Występujące tutaj podzapytanie zawiera odwołanie do kolumny występującej w zapytaniu zewnętrznym. Podzapytania tego typu nazywamy *skorelowanymi*.

W powyższym przykładzie podzapytanie wykonywane jest dla każdego wiersza tabeli `Alfa`, a więc przy ustalonej wartości wielkości `Alfa.Liczba`. Dla podanych danych, warunek `EXISTS` jest prawdziwy tylko w jednym przypadku.

Przykład

Zadanie: Wypiszmy te i tylko te wiersze z tabeli `Beta`, w których wartość atrybutu `InnaLiczba` występuje dwukrotnie.

```
mysql> SELECT * FROM Beta AS B1 WHERE
    -> (SELECT COUNT(*) FROM Beta AS B2
    ->   WHERE B2.InnaLiczba=B1.InnaLiczba) = 2;
+--------+------------+
| Liczba | InnaLiczba |
+--------+------------+
|     20 |         30 |
|     25 |         30 |
+--------+------------+
2 rows in set (0.00 sec)
```

Bez podzapytania tego zadania nie dałoby się wykonać — nie da się go zastąpić złączeniem.

Funkcje agregujące

Funkcje agregujące dostarczają podsumowaną (“zagregowaną”) informację z wielu krotek (wierszy).

Najważniejsze funkcje agregujące

`COUNT(·)`	zlicza wiersze
`COUNT(DISTINCT ·)`	zlicza *różne* wystąpienia w wierszach
`SUM(·)`	podaje sumę wartości liczbowych
`AVG(·)`	podaje średnią arytmetyczną z wartości liczbowych
`STD(·), STDDEV(·)`	podaje odchylenie standardowe wartości liczbowych
`VARIANCE(·)`	podaje wariancję wartości liczbowych
`MIN(·), MAX(·)`	podaje najmniejszą i największą wartość

```
mysql> SELECT * FROM Beta;
+--------+------------+
| Liczba | InnaLiczba |
+--------+------------+
|     20 |         30 |
|     20 |         18 |
|     25 |         30 |
|     27 |         38 |
|     20 |         -5 |
|     10 |       NULL |
+--------+------------+
6 rows in set (0.00 sec)

mysql> SELECT COUNT(*), COUNT(InnaLiczba), COUNT(DISTINCT InnaLiczba) FROM Beta;
+----------+-------------------+----------------------------+
| COUNT(*) | COUNT(InnaLiczba) | COUNT(DISTINCT InnaLiczba) |
+----------+-------------------+----------------------------+
|        6 |                 5 |                          4 |
+----------+-------------------+----------------------------+
1 row in set (0.00 sec)
```

Proszę zwrócić uwagę jak obsługiwane są wartości `NULL`.

Grupowanie

Początkowe wiersze pewnej tabeli:

```
mysql> SELECT * FROM Zamowienia Limit 8;
+-------+-----------+---------+--------------+-------------+
| NrZam | NrKlienta | Kwota   | DataZlozenia | DataZaplaty |
+-------+-----------+---------+--------------+-------------+
|     1 |         6 | 4543.78 | 2008-04-15   | 2008-04-22  |
|     2 |        11 | 4702.25 | 2008-01-11   | 2008-01-13  |
|     3 |         8 |  796.73 | 2008-03-08   | 2008-03-23  |
|     4 |        12 | 7298.23 | 2008-02-13   | NULL        |
|     5 |         5 | 5314.03 | 2008-03-27   | NULL        |
|     6 |        11 | 1122.14 | 2008-02-26   | 2008-02-28  |
|     7 |         7 | 2091.78 | 2008-02-02   | NULL        |
|     8 |         2 | 6757.77 | 2008-03-03   | 2008-04-28  |
+-------+-----------+---------+--------------+-------------+
8 rows in set (0.00 sec)
```

Ile zamówień złożył klient nr 2?

```
mysql> SELECT COUNT(*) FROM Zamowienia WHERE NrKlienta = 2;
+----------+
| COUNT(*) |
+----------+
|       13 |
+----------+
1 row in set (0.38 sec)
```

A ile klient nr 11?

```
mysql> SELECT COUNT(*) FROM Zamowienia WHERE NrKlienta = 11;
+----------+
| COUNT(*) |
+----------+
|       17 |
+----------+
1 row in set (0.00 sec)
```

Moglibyśmy tak robić dla każdego klienta, ale to byłoby nudne ☹, przede wszystkim zaś nie wiemy *ilu jest klientów*. A chcielibyśmy to mieć dla wszystkich. . .

```
mysql> SELECT NrKlienta, COUNT(*) FROM Zamowienia
    -> GROUP BY NrKlienta;
+-----------+----------+
| NrKlienta | COUNT(*) |
+-----------+----------+
|         1 |       16 |
|         2 |       13 |
|         3 |       11 |
|         4 |       10 |
|         5 |        9 |
|         6 |       12 |
|         7 |       13 |
|         8 |       11 |
|         9 |       13 |
|        10 |        9 |
|        11 |       17 |
|        12 |       16 |
+-----------+----------+
12 rows in set (0.10 sec)
```

Ile zamówień i o jakiej *łącznej* wartości złożylim klienci w poszczególnych miesiącach?

```
mysql> SELECT NrKlienta, MONTH(DataZlozenia), COUNT(*), SUM(Kwota) FROM Zamowien
ia
    -> GROUP BY Nrklienta, Month(DataZlozenia);
+-----------+---------------------+----------+------------------+
| NrKlienta | MONTH(DataZlozenia) | COUNT(*) | SUM(Kwota)       |
+-----------+---------------------+----------+------------------+
|         1 |                   1 |        3 | 10813.6398925781 |
|         1 |                   2 |        1 | 343.029998779297 |
|         1 |                   3 |        2 | 3961.80007743835 |
|         1 |                   4 |       10 | 51952.9199676514 |
|         2 |                   1 |        3 | 20283.9799804688 |
|         2 |                   2 |        4 | 18671.8201904297 |
|         2 |                   3 |        4 | 22451.0202636719 |
|         2 |                   4 |        2 |  9186.7900390625 |
.....................................................................
|        12 |                   3 |        5 | 34011.7495117188 |
|        12 |                   4 |        3 | 12239.5299072266 |
+-----------+---------------------+----------+------------------+
45 rows in set (0.00 sec)
```