Bazy danych

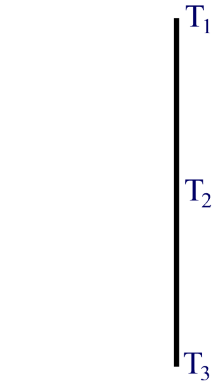
10. Uzupełnienia i przykłady

P. F. Góra

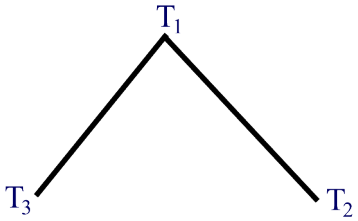
http://th-www.if.uj.edu.pl/zfs/gora/

2011/12

I. Typy złączeń




```
SELECT ... FROM T1
JOIN T2 ON T1.kp=T2.kq
JOIN T3 ON T2.kr=T3.ks
WHERE ...;
```




```
SELECT ... FROM T1
JOIN T2 ON T1.kp=T2.kq
JOIN T3 ON T1.kr=T3.ks
WHERE ...;
```

Każdy wierzchołek grafu reprezentuje tabelę, każda krawędź złączenie

```
JOIN ... ON ...
```

Tego typu złączenia “drzewiaste”, mogące obejmować kilka, kilkanaście, kilkadziesiąt lub nawet więcej ☺ tabel, są charakterystyczne dla złączeń, które obejmują dane z wielu tabel, porozdzielane pomiędzy nimi głównie ze względów normalizacyjnych.

W SQL daje się jednak realizować także złączenia o innej strukturze, w szczególności złączenia, w których pojawiają się cykle.

Zupełnie inny typ złączenia

Przypuśćmy, że mamy trzy tabele z takimi oto przykładowymi danymi:

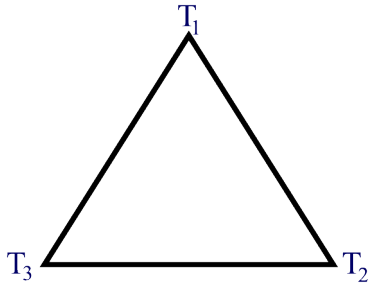
```
mysql> SELECT * FROM T1;
+------+------+
| I    | J    |
+------+------+
| 1    | 1    |
| 2    | 1    |
| 3    | 2    |
+------+------+
3 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM T2;
+------+------+
| K    | L    |
+------+------+
| 1    | A    |
| 2    | B    |
| 3    | C    |
+------+------+
3 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM T3;
+------+------+
| M    | N    |
+------+------+
| A    | 1    |
| A    | 2    |
| B    | 1    |
| C    | 2    |
| C    | 3    |
+------+------+
5 rows in set (0.00 sec)
```

Dla takich tabel można zaprojektować złączenie *“zapętlone”*:

```
mysql> SELECT Jeden.I AS I, Jeden.J AS J, K, L, M, N FROM T1 AS Jeden
    -> JOIN T2 ON Jeden.I=T2.K
    -> JOIN T3 ON T2.L=T3.M
    -> JOIN T1 As Dwa ON T3.N=Dwa.J
    -> WHERE Jeden.I=Dwa.I AND Jeden.J=Dwa.J;
+------+------+------+------+------+------+
| I    | J    | K    | L    | M    | N    |
+------+------+------+------+------+------+
| 1    | 1    | 1    | A    | A    | 1    |
| 2    | 1    | 2    | B    | B    | 1    |
| 3    | 2    | 3    | C    | C    | 2    |
+------+------+------+------+------+------+
3 rows in set (0.00 sec)
```



Ponieważ tabela `T1` występuje w tym złączeniu dwa razy, każde wystąpienie musi mieć unikalny alias. Warunek `WHERE` stanowi, że oba wystąpienia tabeli `T1` odnoszą się do tych samych wierszy.

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, `T1` i `T2`, 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 $\texttt{T}_1$, jak i $\texttt{T}_2$ (nie ma znaczenia, że wartość z tabeli $\texttt{T}_2$ nie może wpłynąć na wynik), nie wykona więc złączenia z $\texttt{T}_3$ przed złączeniem z $\texttt{T}_2$.

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 ...
```

II. Typowe i proste przykłady

Dane są dwie tabele o następujących strukturach:

```
mysql> SELECT * FROM Zamowienia LIMIT 5;
+-------+-----------+---------+--------------+-------------+
| NrZam | NrKlienta | Kwota   | DataZlozenia | DataZaplaty |
+-------+-----------+---------+--------------+-------------+
|     1 |         6 | 4543.78 | 2009-04-15   | 2009-04-22  |
|     2 |        11 | 4702.25 | 2009-01-11   | 2009-01-13  |
|     3 |         8 |  796.73 | 2009-03-08   | 2009-03-23  |
|     4 |        12 | 7298.23 | 2009-02-13   | NULL        |
|     5 |         5 | 5314.03 | 2009-03-27   | NULL        |
+-------+-----------+---------+--------------+-------------+
5 rows in set (0.02 sec)
mysql> SELECT * FROM Klienci LIMIT 5;
+-----------+---------------------+-------------+
| NrKlienta | Nazwa               | Miasto      |
+-----------+---------------------+-------------+
|         1 | Benedetto           | Kraków      |
|         2 | Paolo               | Bolesławiec |
|         3 | Cuda Niewidy        | Kraków      |
|         4 | Najsłynniejsza Firma | Leszno      |
|         5 | Nowoczesny Sklep    | Lublin      |
+-----------+---------------------+-------------+
5 rows in set (0.00 sec)
```

Problem: Znajdź całkowitą wartość zamówień złożonych u klientów w poszczególnych miastach. Ogranicz się do zamówień mających niepusty atrybut `DataZaplaty`.

```
mysql> SELECT Miasto, SUM(Kwota) AS SumaZamowien FROM
    -> Zamowienia NATURAL JOIN Klienci
    -> WHERE NOT ISNULL(DataZaplaty)
    -> GROUP BY Miasto;
+-------------+------------------+
| Miasto      | SumaZamowien     |
+-------------+------------------+
| Bolesławiec | 95918.2000732422 |
| Kraków      | 257165.238189697 |
| Leszno      | 33653.4202880859 |
| Lublin      |         24762.25 |
| Puck        | 34288.3803710938 |
+-------------+------------------+
5 rows in set (0.00 sec)
```

To samo z “ładniejszym” formatowaniem:

```
mysql> SELECT Miasto, CAST(SUM(Kwota) AS DECIMAL(10,2)) AS SumaZamowien FROM
    -> Zamowienia NATURAL JOIN Klienci
    -> WHERE NOT ISNULL(DataZaplaty)
    -> GROUP BY Miasto;
+-------------+--------------+
| Miasto      | SumaZamowien |
+-------------+--------------+
| Bolesławiec |     95918.20 |
| Kraków      |    257165.24 |
| Leszno      |     33653.42 |
| Lublin      |     24762.25 |
| Puck        |     34288.38 |
+-------------+--------------+
5 rows in set (0.00 sec)
```

Problem: Znajdź całkowitą wartość zamówień złożonych u klientów w poszczególnych miastach w poszczególnych miesiącach. Ogranicz się do zamówień mających niepusty atrybut `DataZaplaty`.

```
mysql> SELECT Miasto, MONTHNAME(DataZlozenia) AS Miesiac,
    -> CAST(SUM(Kwota) AS DECIMAL(10,2)) AS SumaZamowien
    -> FROM Zamowienia NATURAL JOIN Klienci
    -> WHERE NOT ISNULL(DataZaplaty)
    -> GROUP BY Miasto, MONTHNAME(DataZlozenia);
+-------------+----------+--------------+
| Miasto      | Miesiac  | SumaZamowien |
+-------------+----------+--------------+
| Bolesławiec | April    |      8971.48 |
| Bolesławiec | February |     32498.41 |
| Bolesławiec | January  |     27384.19 |
| Bolesławiec | March    |     27064.12 |
| Kraków      | April    |     87670.48 |
.........................................
| Puck        | January  |     19508.60 |
| Puck        | March    |     14779.78 |
+-------------+----------+--------------+
16 rows in set (0.00 sec)
```

Proste grupowanie

```
mysql> CREATE TABLE sklepy
    -> (nroperacji SMALLINT UNSIGNED NOT NULL PRIMARY KEY,
    ->  nrsklepu TINYINT UNSIGNED NOT NULL,
    ->  kiedy DATE NOT NULL,
    ->  kwota DECIMAL(6,2) NOT NULL,
    ->  INDEX kiedy (kiedy));
Query OK, 0 rows affected (0.13 sec)

mysql> LOAD DATA LOCAL INFILE "nowesklepy.txt" INTO TABLE sklepy
    -> LINES TERMINATED BY '\r\n';
Query OK, 512 rows affected (0.77 sec)
Records: 512  Deleted: 0  Skipped: 0  Warnings: 0
```

Chcemy się dowiedzieć ile wynosiła średnia kwota operacji w poszczególnych sklepach. Określenie "w poszczególnych sklepach" *natychmiast* sugeruje użycie grupowania.

```
mysql> SELECT nrsklepu, CAST(AVG(kwota) AS DECIMAL(6,2)) AS "średnia operacja"
    -> FROM sklepy
    -> GROUP BY nrsklepu;
+----------+------------------+
| nrsklepu | średnia operacja |
+----------+------------------+
|        1 |            48.93 |
|        2 |            51.27 |
|        3 |            50.83 |
|        4 |            46.50 |
|        5 |            40.49 |
|        6 |            51.01 |
|        7 |            44.68 |
|        8 |            50.82 |
|        9 |            48.95 |
|       10 |            47.11 |
|       11 |            46.65 |
|       12 |            52.12 |
|       13 |            40.67 |
|       14 |            57.64 |
|       15 |            45.72 |
|       16 |            52.08 |
+----------+------------------+
16 rows in set (0.02 sec)
```

Największa wartość

Teraz chcemy dowiedzieć się która z powyższych średnich jest największa. Trzeba w tym celu wybrać największą wartość z drugiej kolumny wyniku poprzedniego grupowania. Chcemy to jednak zrobić bez kosztownego sortowania, czyli wywoływani klauzuli `ORDER BY`:

```
mysql> SELECT CAST(MAX(srednia) AS DECIMAL(6,2)) AS najwieksza
    -> FROM (SELECT nrsklepu, AVG(kwota) AS srednia
    ->       FROM sklepy
    ->       GROUP by nrsklepu) AS z1;
+------------+
| najwieksza |
+------------+
|      57.64 |
+------------+
1 row in set (0.00 sec)
```

A teraz chcemy się dowiedzieć który sklep uzyskał tą największą średnią. Próbujemy

```
mysql> SELECT nrsklepu, CAST(MAX(srednia) AS DECIMAL(6,2)) AS najwieksza
    -> FROM (SELECT nrsklepu, AVG(kwota) AS srednia
    ->       FROM sklepy
    ->       GROUP by nrsklepu) AS z1;
+----------+------------+
| nrsklepu | najwieksza |
+----------+------------+
|        1 |      57.64 |
+----------+------------+
1 row in set (0.00 sec)
```

Otrzymaliśmy **NIEPOPRAWNY** wynik. Stało się tak na skutek umieszczenia na liście `SELECT` funkcji agregującej (`MAX`) i atrybutu, po którym nie grupujemy. W takim wypadku wartość tego atrybutu brana jest z jakiejś przypadkowej krotki.

Prawidłowo należy zrobić to tak:

```
mysql> SELECT nrsklepu, srednia
    -> FROM (SELECT nrsklepu, AVG(kwota) AS srednia FROM sklepy GROUP by nrsklepu
    ->       ) AS z1
    -> WHERE srednia = (SELECT MAX(s)
    ->                  FROM (SELECT AVG(kwota) AS s FROM sklepy GROUP by nrsklepu
    ->                        ) AS z2
    ->                 );
+----------+-----------+
| nrsklepu | srednia   |
+----------+-----------+
|       14 | 57.635172 |
+----------+-----------+
1 row in set (0.03 sec)
```

W celu znalezienia wartości drugiej od góry, trzeba wybrać wartość największą z podzbioru nie zawierającego największej średniej z całości. A więc na przykład zapytanie wybierające dwie największe średnie wygląda tak:

```
mysql> SELECT nrsklepu, CAST(srednia AS DECIMAL(6,2)) AS najwieksza
    -> FROM (SELECT nrsklepu, AVG(kwota) AS srednia FROM sklepy GROUP by nrsklepu) AS z1
    -> WHERE
    ->    srednia = (SELECT MAX(s)
    ->               FROM (SELECT AVG(kwota) AS s
    ->                     FROM sklepy
    ->                     GROUP by nrsklepu
    ->                    ) AS z2
    ->              )
    -> OR
    ->    srednia = (SELECT MAX(s)
    ->               FROM (SELECT AVG(kwota) AS s
    ->                     FROM sklepy
    ->                     GROUP by nrsklepu
    ->                     HAVING s < (SELECT MAX(s)
    ->                                 FROM (SELECT AVG(kwota) AS s
    ->                                       FROM sklepy
    ->                                       GROUP by nrsklepu
    ->                                      ) AS z3
    ->                                )
    ->                    ) AS z4
    ->              )
    -> ORDER BY najwieksza DESC;
```

```
+----------+------------+
| nrsklepu | najwieksza |
+----------+------------+
|       14 |      57.64 |
|       12 |      52.12 |
+----------+------------+
2 rows in set (0.02 sec)
```

Zwracam uwagę na zastosowanie klauzuli `HAVING` w zagnieżdżonym podzapytaniu oraz na użycie aliasów. Podzapytania zwracające tabele muszą mieć aliasy. Podzapytania zwracające jedną liczbę (tu: wynik działania funkcji `MAX()`) *nie mogą* mieć aliasów.

Użycie tabeli tymczasowej

W powyższym przykładzie to samo (a w każdym razie bardzo podobne) podzapytanie wykonywane jest cztery razy. Jest to podzapytanie nieskorelowane, możemy więc zaufać optymalizatorowi, iż nie nakaże on faktycznego wielokrotnego wykonywania tej operacji. Jednak w wypadku wątpliwości, możemy skorzystać z tabeli tymczasowej, a raczej z pewnej cechy MySQL, jaką są tabele przechowywane w pamięci, nie na dysku (giną one po zamknięciu procesu/wątku, który je utworzył).

```
mysql> DROP TABLE IF EXISTS xxx;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> CREATE TABLE xxx
    -> Engine = Memory
    -> SELECT nrsklepu, AVG(kwota) AS srednia
    -> FROM sklepy
    -> GROUP BY nrsklepu;
Query OK, 16 rows affected, 14 warnings (0.72 sec)
Records: 16  Duplicates: 0  Warnings: 14
```

Kolumny (atrybuty) tabeli `xxx` mają nazwy i typy atrybutów zapytania, które je zapełnia.

```
mysql> SELECT nrsklepu, CAST(srednia AS DECIMAL(6,2)) AS najwieksza
    -> FROM xxx
    -> WHERE
    ->    srednia = (SELECT MAX(srednia)
    ->               FROM xxx
    ->              )
    -> OR
    ->    srednia = (SELECT MAX(srednia)
    ->               FROM (SELECT *
    ->                     FROM xxx
    ->                     WHERE srednia < (SELECT MAX(srednia)
    ->                                      FROM xxx
    ->                                     )
    ->                    ) AS z
    ->              )
    -> ORDER BY najwieksza DESC;
+----------+------------+
| nrsklepu | najwieksza |
+----------+------------+
|       14 |      57.64 |
|       12 |      52.12 |
+----------+------------+
2 rows in set (0.00 sec)
```

Użycie zmiennej tymczasowej

Korzystające ze zmiennej tymczasowej, zapis można jeszcze bardziej uprościć, zyskując dodatkowo pewność, że największa średnia obliczana jest tylko raz:

```
mysql> SET @najw = (SELECT MAX(srednia) FROM xxx);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT nrsklepu, CAST(srednia AS DECIMAL(6,2)) AS najwieksza
    -> FROM xxx
    -> WHERE
    ->    srednia = @najw
    -> OR
    ->    srednia = (SELECT MAX(srednia)
    ->                FROM (SELECT *
    ->                      FROM xxx
    ->                      WHERE srednia < @najw
    ->                     ) AS z
    ->               )
    -> ORDER BY najwieksza DESC;
```

Niedopasowane zbiory (Przykład seksistowski)

```
mysql> CREATE TABLE faceci
    -> (Facet VARCHAR(16) NOT NULL
        PRIMARY KEY,
    ->  IQ TINYINT UNSIGNED NOT NULL);
Query OK, 0 rows affected (0.22 sec)

mysql> SELECT * FROM faceci;
+-----------+-----+
| Facet     | IQ  |
+-----------+-----+
| Pafnucy   |  96 |
| Pankracy  | 127 |
| Pantaleon |  89 |
| Polikarp  | 112 |
| Prot      | 103 |
+-----------+-----+
5 rows in set (0.06 sec)
```

```
mysql> CREATE TABLE baby
    -> (Baba VARCHAR(16) NOT NULL
        PRIMARY KEY,
    ->  IQ TINYINT UNSIGNED NOT NULL);
Query OK, 0 rows affected (0.06 sec)

mysql> SELECT * FROM baby;
+----------+-----+
| Baba     | IQ  |
+----------+-----+
| Edwarda  | 100 |
| Edyta    |  92 |
| Eleonora | 129 |
| Eliza    | 130 |
| Elwira   |  91 |
| Emma     | 101 |
| Ewelina  |  99 |
+----------+-----+
7 rows in set (0.00 sec)
```

Mężczyźni, którzy boją się kobiet, szukają partnerek o niższym IQ. Jakie mają możliwości w podanym przykładzie?

```
mysql> SELECT Facet, Baba AS 'Głupsza' FROM
    -> faceci LEFT JOIN baby ON faceci.IQ > baby.IQ
    -> ORDER BY faceci.IQ DESC, baby.IQ DESC;
+-----------+---------+
| Facet     | Głupsza |
+-----------+---------+
| Pankracy  | Emma    |
| Pankracy  | Edwarda |
| Pankracy  | Ewelina |
| Pankracy  | Edyta   |
| Pankracy  | Elwira  |
| Polikarp  | Emma    |
| Polikarp  | Edwarda |
| Polikarp  | Ewelina |
| Polikarp  | Edyta   |
| Polikarp  | Elwira  |
| Prot      | Emma    |
| Prot      | Edwarda |
| Prot      | Ewelina |
| Prot      | Edyta   |
```

```
| Prot      | Elwira  |
| Pafnucy   | Edyta   |
| Pafnucy   | Elwira  |
| Pantaleon | NULL    |
+-----------+---------+
18 rows in set (0.00 sec)
```

Użyliśmy lewego złączenia, aby widać było, że niezbyt inteligentny Pantaleon nie może znaleźć żadnej partnerki.

Jak wskazać facetów, którzy nie mogą nikogo znaleźć?

```
mysql> SELECT Facet AS "Najgłupszy" FROM
    -> faceci LEFT JOIN baby ON faceci.IQ > baby.IQ
    -> WHERE ISNULL(baba);
+------------+
| Najgłupszy |
+------------+
| Pantaleon  |
+------------+
1 row in set (0.05 sec)
```

To samo można zrobić za pomocą zapytania skorelowanego z predykatem `NOT EXISTS`; ten drugi sposób jest lepszy ze względów wydajnościowych:

```
mysql> SELECT Facet AS "Najgłupszy" FROM faceci
    -> WHERE NOT EXISTS
    ->   (SELECT Baba FROM baby
    ->    WHERE faceci.IQ > baby.IQ);
+------------+
| Najgłupszy |
+------------+
| Pantaleon  |
+------------+
1 row in set (0.00 sec)
```

A teraz wskażmy baby bardziej inteligentne od wszystkich facetów:

```
mysql> SELECT Baba AS "Super baba" FROM baby
    -> WHERE NOT EXISTS
    ->   (SELECT Facet FROM faceci
    ->    WHERE faceci.IQ > baby.IQ);
+------------+
| Super baba |
+------------+
| Eleonora   |
| Eliza      |
+------------+
2 rows in set (0.00 sec)
```

Oceny studentów

Prowadzący zajęcia przechowuje w tabeli identyfikatory, imiona i nazwiska studentów. W osobnej tabeli przechowuje identyfikatory studentów oraz oceny, jakie otrzymali oni z dwu kolokwiów. Jeśli ktoś nie pisał kolokwium, zapamiętywana jest wartość `NULL`. Należy mieć pewność, że każdy oceniony student faktycznie jest opisany w tabeli z nazwiskami studentów. W tym celu należy zastosować mechanizm kluczy obcych.

```
mysql> CREATE TABLE studenci
    -> (nrstudenta TINYINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->  Imie VARCHAR(16), Nazwisko VARCHAR(16))
    -> Engine=InnoDB;
Query OK, 0 rows affected (0.13 sec)

mysql> CREATE TABLE kolokwia
    -> (nrstudenta TINYINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->  kol1 DECIMAL(2,1), kol2 DECIMAL(2,1),
    ->  FOREIGN KEY (nrstudenta) REFERENCES studenci (nrstudenta)
    ->  ON UPDATE CASCADE ON DELETE RESTRICT)
    -> Engine=InnoDB;
Query OK, 0 rows affected (0.23 sec)
```

Przykładowe dane wyglądają tak:

```
mysql> select * from studenci;
+------------+---------+--------------+
| nrstudenta | Imie    | Nazwisko     |
+------------+---------+--------------+
|          1 | Alicja  | Zajączkowska |
|          2 | Bogdan  | Yeti         |
|          3 | Czesław | Woźniak      |
|          4 | Damian  | Urbanowicz   |
|          5 | Eliza   | Tokarz       |
|          6 | Fabiola | Stryjkowska  |
+------------+---------+--------------+
6 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM kolokwia;
+------------+------+------+
| nrstudenta | kol1 | kol2 |
+------------+------+------+
|          1 |  3.0 |  5.0 |
|          2 |  5.0 |  4.0 |
|          3 |  3.0 | NULL |
|          4 | NULL |  3.5 |
|          5 | NULL | NULL |
+------------+------+------+
5 rows in set (0.00 sec)
```

Studentom trzeba wystawić oceny — każdy otrzymuje lepszą ocenę z kolokwiów. Jeśli ktoś pisał tylko jedno kolokwium, otrzymuje tę ocenę. Jeśli ktoś nie pisał żadnego, otrzymuje ndst. Należy uwzględnić osoby zdefiniowne w tabeli studenci, które nie zostały jednak wpisane do tabeli z wynikami; traktujemy je tak, jakby nie pisały żadnego kolokwium.

Trzeba odpowiednio obsłużyć wartości `NULL`. Cały problem rozwiązujemy przez zagnieżdżone instrukcje `CASE`.

```
mysql> SELECT Imie, Nazwisko,
    ->   CASE
    ->     WHEN NOT ISNULL(kol1) AND NOT ISNULL(kol2) THEN
    ->          CASE
    ->            WHEN kol1 > kol2 THEN kol1
    ->            ELSE kol2
    ->          END
    ->     ELSE CASE
    ->            WHEN NOT ISNULL(kol1) THEN kol1
    ->            WHEN NOT ISNULL(kol2) THEN kol2
    ->            ELSE 2.0
    ->          END
    ->   END AS ocena
    -> FROM (SELECT * FROM studenci LEFT JOIN kolokwia USING(nrstudenta)) AS z;
```

Zwracam uwagę na obowiązkowe użycie aliasu.

```
+---------+--------------+-------+
| Imie    | Nazwisko     | ocena |
+---------+--------------+-------+
| Alicja  | Zajączkowska |   5.0 |
| Bogdan  | Yeti         |   5.0 |
| Czesław | Woźniak      |   3.0 |
| Damian  | Urbanowicz   |   3.5 |
| Eliza   | Tokarz       |   2.0 |
| Fabiola | Stryjkowska  |   2.0 |
+---------+--------------+-------+
6 rows in set (0.09 sec)
```

Zrobione i niezrobione zadania

Rozważmy podobny problem. Oprócz tabeli `studenci` (użyjemy tej samej, co w poprzednim przykładzie), mamy tabelę `zadania` z numerami i opisami zadań, oraz tabelę pomostową `zrobione`, w której zaznaczamy kto zrobił które zadanie. Rzecz jasna dane w tabeli `zrobione` muszą się odnosić do *istniejących* danych z dwu pozostałych tabel.

```
mysql> CREATE TABLE zadania
    -> (nrzadania  TINYINT UNSIGNED NOT NULL PRIMARY KEY,
    ->  opis VARCHAR(16) NOT NULL)
    -> Engine = InnoDB;
Query OK, 0 rows affected (0.08 sec)

mysql> CREATE TABLE zrobione
    -> (nrstudenta TINYINT UNSIGNED NOT NULL,
    ->  nrzadania  TINYINT UNSIGNED NOT NULL,
    ->  PRIMARY KEY (nrstudenta,nrzadania),
    ->  FOREIGN KEY (nrstudenta) REFERENCES studenci (nrstudenta)
    ->  ON DELETE RESTRICT ON UPDATE CASCADE,
    ->  FOREIGN KEY (nrzadania)  REFERENCES zadania (nrzadania)
    ->  ON DELETE RESTRICT ON UPDATE CASCADE)
    -> Engine = InnoDB;
Query OK, 0 rows affected (0.11 sec)
```

Wydaje się, że bardzo łatwo jest policzyć kto ile zadań zrobił:

```
mysql> SELECT Imie, Nazwisko, COUNT(*) AS "Zrobił"
    -> FROM studenci NATURAL JOIN zrobione
    -> GROUP BY nrstudenta
    -> ORDER BY Nazwisko;
+---------+--------------+--------+
| Imie    | Nazwisko     | Zrobił |
+---------+--------------+--------+
| Eliza   | Tokarz       |     10 |
| Damian  | Urbanowicz   |     10 |
| Czesław | Woźniak      |     12 |
| Bogdan  | Yeti         |      7 |
| Alicja  | Zajączkowska |      7 |
+---------+--------------+--------+
5 rows in set (0.02 sec)
```

Jak się okazuje, to rozwiązanie nie uwzględnia osób, które nie zrobiły żadnego zadania. Wydaje się, że wystarczy użyć złączenia zewnętrznego:

```
mysql> SELECT Imie, Nazwisko, COUNT(*) AS "Zrobił"
    -> FROM studenci LEFT JOIN zrobione USING(nrstudenta)
    -> GROUP BY nrstudenta
    -> ORDER BY Nazwisko;
+---------+--------------+--------+
| Imie    | Nazwisko     | Zrobił |
+---------+--------------+--------+
| Fabiola | Stryjkowska  |      1 |
| Eliza   | Tokarz       |     10 |
| Damian  | Urbanowicz   |     10 |
| Czesław | Woźniak      |     12 |
| Bogdan  | Yeti         |      7 |
| Alicja  | Zajączkowska |      7 |
+---------+--------------+--------+
6 rows in set (0.00 sec)
```

W wynikach pojawiła się co prawda osoba, która nie zrobiła żadnego zadania, ale z fałszywym wynikiem! Bierze się to stąd, że lewe złączenie wprowadziło do tabeli, po której zliczamy, wiersz odpowiadający tej osobie. Zauważmy, że klauzula `WHERE NOT ISNULL(nrzadania)` nie rozwiąże problemu, gdyż sprawi, że osoba, która nie zrobiła żadnego zadania, ponownie nie zostanie policzona.

Trzeba uciec się do pewnego triku:

```
mysql> SELECT Imie, Nazwisko,
    ->   SUM(CASE
    ->         WHEN NOT ISNULL(nrzadania) THEN 1
    ->         ELSE 0
    ->       END
    ->      ) AS "Zrobił"
    -> FROM studenci LEFT JOIN zrobione USING(nrstudenta)
    -> GROUP BY nrstudenta
    -> ORDER BY Nazwisko;
+---------+-------------+--------+
| Imie    | Nazwisko    | Zrobił |
+---------+-------------+--------+
| Fabiola | Stryjkowska |      0 |
| Eliza   | Tokarz      |     10 |
| Damian  | Urbanowicz  |     10 |
| Czesław | Woźniak     |     12 |
| Bogdan  | Yeti        |      7 |
| Alicja  | Zajączkowska|      7 |
+---------+-------------+--------+
6 rows in set (0.08 sec)
```

A jak szybko policzyć ilu zadań student *nie zrobił*? Trzeba odjąć liczbę zrobionych zadań od liczby wszystkich zadań.

```
mysql> SELECT Imie, Nazwisko,
    ->  (SELECT COUNT(*) FROM zadania) - SUM(CASE
    ->                                        WHEN NOT ISNULL(nrzadania) THEN 1
    ->                                        ELSE 0
    ->                                      END
    ->                                     ) AS "Nie zrobił"
    -> FROM studenci LEFT JOIN zrobione USING(nrstudenta)
    -> GROUP BY nrstudenta
    -> ORDER BY Nazwisko;
+---------+--------------+------------+
| Imie    | Nazwisko     | Nie zrobił |
+---------+--------------+------------+
| Fabiola | Stryjkowska  |         12 |
| Eliza   | Tokarz       |          2 |
| Damian  | Urbanowicz   |          2 |
| Czesław | Woźniak      |          0 |
| Bogdan  | Yeti         |          5 |
| Alicja  | Zajączkowska |          5 |
+---------+--------------+------------+
6 rows in set (0.08 sec)
```

Optymalizator zapewni, że podzapytanie zliczające wiersze w tabeli zadania wykona się tylko raz.

Nieco trudniejsze jest nie samo zliczenie, ale *wypisanie numerów zadań*, których student *nie zrobił*. Zastanówmy się: Tabela `zrobione` zawiera pary `(nrstudenta,nrzadania)`, odpowiadające zadaniom zrobionym przez studenta. Natomiast iloczyn kartezjański kolumny `nrstudenta` z tabeli `studenci` oraz kolumny `nrzadania` z tabeli `zadania` zawiera *wszystkie* możliwe pary. Trzeba więc wziąć różnicę mnogościową wspomnianego iloczynu kartezjańskiego i tabeli `zrobione`.

Można to zrobić na dwa sposoby. Po pierwsze, wykorzystując podzapytanie skorelowane i predykat `NOT EXISTS`:

```
mysql> SELECT Imie, Nazwisko, nrzadania
    -> FROM studenci CROSS JOIN zadania
    -> WHERE NOT EXISTS
    ->   (SELECT * FROM zrobione
    ->    WHERE zrobione.nrstudenta = studenci.nrstudenta
    ->    AND   zrobione.nrzadania  = zadania.nrzadania)
    -> ORDER BY Nazwisko, nrzadania;
+---------+--------------+-----------+
| Imie    | Nazwisko     | nrzadania |
+---------+--------------+-----------+
| Fabiola | Stryjkowska  |         1 |
| Fabiola | Stryjkowska  |         2 |
.....................................
| Alicja  | Zajączkowska |         6 |
| Alicja  | Zajączkowska |         7 |
+---------+--------------+-----------+
26 rows in set (0.00 sec)
```

Po drugie, pamiętając, że tabele są zbiorami, wykorzystamy podzapytanie nieskorelowane i predykat `NOT IN`:

```
mysql> SELECT Imie, Nazwisko, nrzadania
    -> FROM studenci CROSS JOIN zadania
    -> WHERE (nrstudenta,nrzadania) NOT IN
    ->   (SELECT * FROM zrobione)
    -> ORDER BY Nazwisko, nrzadania;
+---------+--------------+-----------+
| Imie    | Nazwisko     | nrzadania |
+---------+--------------+-----------+
| Fabiola | Stryjkowska  |         1 |
| Fabiola | Stryjkowska  |         2 |
.....................................
| Alicja  | Zajączkowska |         6 |
| Alicja  | Zajączkowska |         7 |
+---------+--------------+-----------+
26 rows in set (0.00 sec)
```

Ten drugi sposób dla dużych tabel będzie zdecydowanie szybszy.

Brakujące wartości w sekwencji

Klasycznym zagadnieniem w SQL jest znajdywanie brakujących wartości w sekwencjach liczb całkowitych*. Na przykład w sekwencji $\{1, 2, 3, 4, 6, 7, 10, 11, 13, 17, 18\}$ brakującymi wartościami są $\{5, 8, 9, 12, 14, 15, 16\}$.

*Problem ten można łatwo uogólnić na sekwencje innych obiektów, dla których zdefiniowane jest pojęcie “następnego”.

Niech tabela `liczby` zawiera powyższą sekwencję. Zacznijmy od znalezienia następnika każdej z liczb w tabeli. Robimy to przez samozłączenie i grupowanie — grupowanie odpowiada za realizację operacji “dla każdej liczby”.

```
mysql> SELECT l1.i AS m, MIN(l2.i) AS "następnik"
    -> FROM liczby AS l1, LICZBY AS l2
    -> WHERE l2.i > l1.i
    -> GROUP BY l1.i;
+----+-----------+
| m  | następnik |
+----+-----------+
|  1 |         2 |
|  2 |         3 |
|  3 |         4 |
|  4 |         6 |
|  6 |         7 |
|  7 |        10 |
| 10 |        11 |
| 11 |        13 |
| 13 |        17 |
| 17 |        18 |
+----+-----------+
10 rows in set (0.00 sec)
```

Powyższe samozłączenie można też zapisać nieco inaczej.

```
mysql> SELECT l1.i AS m, MIN(l2.i) AS "następnik"
    -> FROM liczby AS l1 JOIN liczby AS l2 ON l2.i > l1.i
    -> GROUP BY l1.i;
+----+-----------+
| m  | następnik |
+----+-----------+
|  1 |         2 |
|  2 |         3 |
|  3 |         4 |
|  4 |         6 |
|  6 |         7 |
|  7 |        10 |
| 10 |        11 |
| 11 |        13 |
| 13 |        17 |
| 17 |        18 |
+----+-----------+
10 rows in set (0.00 sec)
```

Gdy mamy znalezione następniki, łatwo jest znaleźć początki i końce przerw w sekwencji danych. Przerwa sacharakteryzowana jest tym, że następnik liczby w tabeli nie jest liczbą następną w zbiorze liczb naturalnych.

```
mysql> SELECT m+1 AS "początek przerwy", nastepnik-1 AS "koniec przerwy"
    -> FROM (SELECT l1.i AS m, MIN(l2.i) AS nastepnik
    ->       FROM liczby AS l1 JOIN liczby AS l2 ON l2.i > l1.i
    ->       GROUP BY l1.i
    ->      ) AS x
    -> WHERE nastepnik > m+1;
+------------------+----------------+
| początek przerwy | koniec przerwy |
+------------------+----------------+
|                5 |              5 |
|                8 |              9 |
|               12 |             12 |
|               14 |             16 |
+------------------+----------------+
4 rows in set (0.00 sec)
```

Kłopot natomiast sprawi wypisanie faktycznych brakujących liczb (w naszym przykładzie problem będzie z liczbą 15) — nie umiem tego zrobić za pomocą zapytania, a jedynie za pomocą procedury składowanej.