МГТУ ИМ. Н.Э. БАУМАНА

Базы Данных

Семинар 3

Москва, 2018

DDL (Продолжение)

Убедительно прошу начать семинар с повторения материала предыдущегог занятия. Возьмите предметную область и напишите 3-4 сложных линейных запроса (со всеми инструкциями). Без подзапросов и прочих наворотов. Дайте студентам время усвоить материал, пусть сами продитктуют запрос. Убедитесь, что простые запросы - не проблема. На лабах буду с пристрастием спрашивать чем отличается where от having.

После того, как понятно, как писать простые запросы, необходимо объяснить что такое подзапрос.

Подзапросы, внутренние или вложенные запросы – есть не что иное, как запрос внутри запроса. Обычно, подзапрос используется в конструкции WHERE или FROM. И, в большинстве случаев, подзапрос используется, когда вы можете получить значение с помощью запроса, но не знаете конкретного результата.

Подзапросы являются альтернативным путем получения данных из множества таблиц.

Вопрос в зал: Почему SQL допускает подзапросы? (Так как в основе лежит реляционная модель и реляционная алгебра, то результатом выполнения любой операции реляционной алгебры над отношениями также является отношением. Эта особенность называется свойством реляционной замкнутости.)

Подзапрос в WHERE обычно используют такие операторы, как IN, NOT IN

```
-- Когда список имен заранее пределен
SELECT first_name, last_name, subject
FROM student_details
WHERE games NOT IN ('Cricket', 'Football');

-- Если список имен неизвестен
SELECT id, first_name
FROM student_details
WHERE first_name IN (SELECT first_name
                     FROM student_details
                     WHERE subject =  'Science');
```

Подзапрос в FROM. В таких запросах обязательно наличие псевдонима (Alias) после закрытой скобки подапроса.

```
SELECT  COUNT(made_only_recharge) AS made_only_recharge
FROM
   (
      SELECT DISTINCT (identifiant) AS made_only_recharge
      FROM cdr_data
      WHERE CALLEDNUMBER = '0130'
      EXCEPT
      SELECT DISTINCT (identifiant) AS made_only_recharge
      FROM cdr_data
      WHERE CALLEDNUMBER != '0130'
   ) AS derivedTable
```

Наряду с операторами сравнения такими, как =, <, >, >=, <= и др., можно использовать подзапросы с конструкциями: SELECT, INSERT, UPDATE, DELETE.

JOIN

JOIN — оператор языка SQL, который является реализацией операции соединения реляционной алгебры. Входит в предложение FROM операторов SELECT, UPDATE и DELETE.

Операция соединения, как и другие бинарные операции, предназначена для обеспечения выборки данных из двух таблиц и включения этих данных в один результирующий набор. Отличительными особенностями операции соединения являются следующие:

- в схему таблицы-результата входят столбцы обеих исходных таблиц (таблиц-операндов), то есть схема результата является «сцеплением» схем операндов;
- каждая строка таблицы-результата является «сцеплением» строки из одной таблицы-операнда со строкой второй таблицы-операнда.

Определение того, какие именно исходные строки войдут в результат и в каких сочетаниях, зависит от типа операции соединения и от явно заданного *условия соединения*. Условие соединения, то есть условие сопоставления

строк исходных таблиц друг с другом, представляет собой логическое выражение (предикат).

При необходимости соединения не двух, а нескольких таблиц, операция соединения применяется несколько раз (последовательно).

По механизму работы соединения различают три типа JOIN-ов. Далее рассматриваются внутренние соединения, на основе данных алгоритмов несложно написать внешние. Код выписывать не обязательно, достаточно прогнать пару примеров, чтобы былыи понятны основные различия:

Nested Loops Join
Самый базовый алгоритм объединения двух списков сейчас проходят в школах. Суть очень простая — для каждого элемента первого списка пройдемся по всем элементам второго списка; если ключи элементов вдруг оказались равны — запишем совпадение в результирующую таблицу. Для реализации этого алгоритма достаточно двух вложенных циклов, именно поэтому он так и называется.

```
public static  <K, V1, V2> List<Triple<K, V1, V2>> nestedLoopsJoin(List<Pair<K, V1>> left, List<Pair<K, V2>> right) {
   List<Triple<K, V1, V2>> result = new ArrayList<>();
   for (Pair<K, V1> leftPair: left) {
      for (Pair<K, V2> rightPair: right) {
         if (Objects.equals(leftPair.k, rightPair.k)) {
            result.add(new Triple<>(leftPair.k, leftPair.v, rightPair.v));
         }
      }
   }
   return result;
}
```

Основной плюс этого метода — полное безразличие к входным данным. Алгоритм работает для любых двух таблиц, не требует никаких индексов и перекладываний данных в памяти, а также прост в реализации. На практике это означает, что достаточно просто бежать по диску двумя курсорами и периодически выплёвывать в сокет совпадения.

Однако ж, фатальный минус алгоритма — высокая временная сложность O(N*M) (квадратичная асимптотика). Например, для джойна пары небольших таблиц в 100к и 500к записей потребуется сделать аж 100.000 * 500.000 = 50.000.000.000 (50 млрд) операций сравнения. Запросы с таким джойном будут выполняться невежливо долго, часто именно они — причина беспощадных тормозов кривеньких самописных CMS'ок.

Современные РСУБД используют nested loops join в самых безнадежных случаях, когда не удается применить никакую оптимизацию или затраты на оптимизация несравнимы с размерами таблиц.

Hash Join

Если размер одной из таблиц позволяет засунуть ее целиком в память, значит, на ее основе можно сделать хеш-таблицу и быстренько искать в ней нужные ключи. Проговорим чуть подробнее.

Проверим размер обоих списков. Возьмем меньший из списков, прочтем его полностью и загрузим в память, построив HashMap. Теперь вернемся к большему списку и пойдем по нему курсором с начала. Для каждого ключа проверим, нет ли такого же в хеш-таблице. Если есть — запишем совпадение в результирующую таблицу.

Временная сложность этого алгоритма падает до линейной O(N+M), но требуется дополнительная память.

```
public static  <K, V1, V2> List<Triple<K, V1, V2>> hashJoin(List<Pair<K, V1>> left,
List<Pair<K, V2>> right) {
    Map<K, V2> hash = new HashMap<>(right.size());
    for (Pair<K, V2> rightPair: right) {
        hash.put(rightPair.k, rightPair.v);
    }

    List<Triple<K, V1, V2>> result = new ArrayList<>();
    for (Pair<K, V1> leftPair: left) {
        if (hash.containsKey(leftPair.k)) {
            result.add(new Triple<>(leftPair.k, leftPair.v, hash.get(leftPair.k)));
        }
    }
```

```
    return result;
}
```

Что важно, во времена динозавров считалось, что в память нужно загрузить правую таблицу, а итерироваться по левой. У нормальных РСУБД сейчас есть статистика cardinality, и они сами определяют порядок джойна, но если по какой-то причине статистика недоступна, то в память грузится именно правая таблица. Это важно помнить при работе с молодыми корявыми инструментами типа Cloudera Impala.

Merge Join

А теперь представим, что данные в обоих списках заранее отсортированы, например, по возрастанию. Так бывает, если у нас были индексы по этим таблицам, или же если мы отсортировали данные на предыдущих стадиях запроса. Как вы наверняка помните, два отсортированных списка можно склеить в один отсортированный за линейное время — именно на этом основан алгоритм сортировки слиянием. Здесь нам предстоит похожая задача, но вместо того, чтобы склеивать списки, мы будем искать в них общие элементы.

Итак, ставим по курсору в начало обоих списков. Если ключи под курсорами равны, записываем совпадение в результирующую таблицу. Если же нет, смотрим, под каким из курсоров ключ меньше. Двигаем курсор над меньшим ключом на один вперед, тем самым “догоняя” другой курсор.

```
public static <K extends Comparable<K>, V1, V2> List<Triple<K, V1, V2>>
mergeJoin(
        List<Pair<K, V1>> left,
        List<Pair<K, V2>> right
) {
    List<Triple<K, V1, V2>> result = new ArrayList<>();
    Iterator<Pair<K, V1>> leftIter = left.listIterator();
    Iterator<Pair<K, V2>> rightIter = right.listIterator();
    Pair<K, V1> leftPair = leftIter.next();
    Pair<K, V2> rightPair = rightIter.next();
```

```
    while (true)  {
        int compare = leftPair.k.compareTo(rightPair.k);
        if (compare < 0) {
            if (leftIter.hasNext()) {
                leftPair = leftIter.next();
            } else {
                break;
            }
        } else if (compare > 0) {
            if (rightIter.hasNext()) {
                rightPair = rightIter.next();
            } else {
                break;
            }
        } else {
            result.add(new Triple<>(leftPair.k, leftPair.v, rightPair.v));
            if (leftIter.hasNext() && rightIter.hasNext()) {
                leftPair = leftIter.next();
                rightPair = rightIter.next();
            } else {
                break;
            }
        }
    }
    return result;
}
```

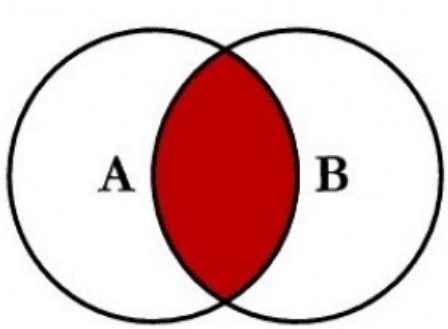
Если данные отсортированы, то временная сложность алгоритма линейная O(M+N) и не требуется никакой дополнительной памяти. Если же данные не отсортированы, то нужно сначала их отсортировать. Из-за этого временная сложность возрастает до O(M log M + N log N), плюс появляются дополнительные требования к памяти.

Виды JOIN-ов (по типу соединений)

Вместе составьте две таблицы, которые потом будете соединять. Для каждого вида нарисовать диаграмму Эйлера + написать выходную таблицу. Да много, но это правда НАДО.

Внутренне соединение:

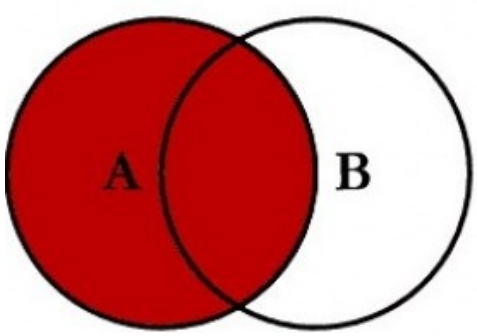
Внутреннее объединение **INNER JOIN** (синоним JOIN, ключевое слово INNER можно опустить). Выбираются только совпадающие данные из объединяемых таблиц.

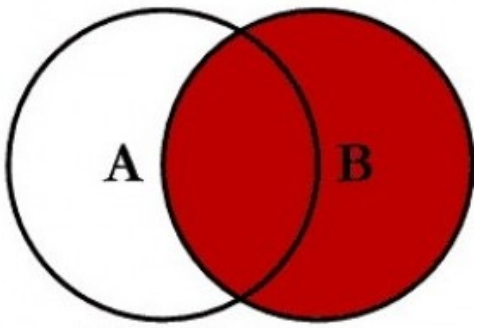



```
SELECT u.id, u.name, d.name AS d_name
FROM users u
INNER JOIN departments d ON u.d_id = d.id
```

Внешнее соединение:

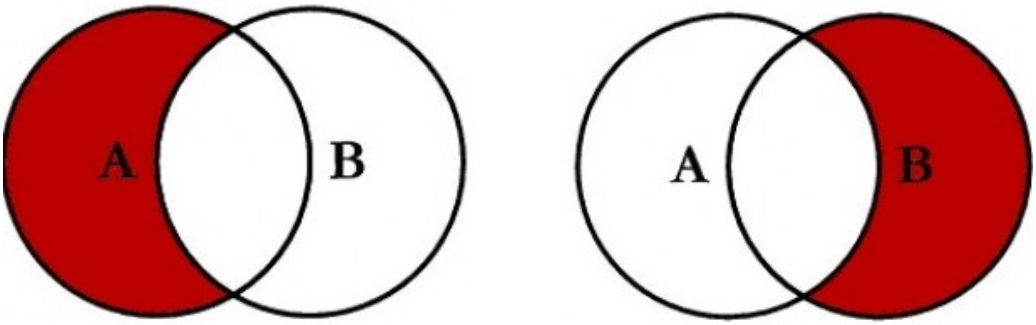
Чтобы получить данные, которые подходят по условию частично, необходимо использовать внешнее объединение **OUTER JOIN - LEFT OUTER JOIN** и **RIGHT OUTER JOIN.**
Работают они одинаково, разница заключается в том что LEFT - указывает что "внешней" таблицей будет находящаяся слева (в нашем примере это таблица users), а RIGTH — правая.
Ключевое слово OUTER можно опустить. Запись LEFT JOIN идентична LEFT OUTER JOIN.





```
SELECT u.id, u.name, d.name AS d_name
FROM users u
LEFT/RIGHT [OUTER] JOIN departments d ON u.d_id = d.id
```

Вопрос в зал: Как получить записи только из 1ой табицы?

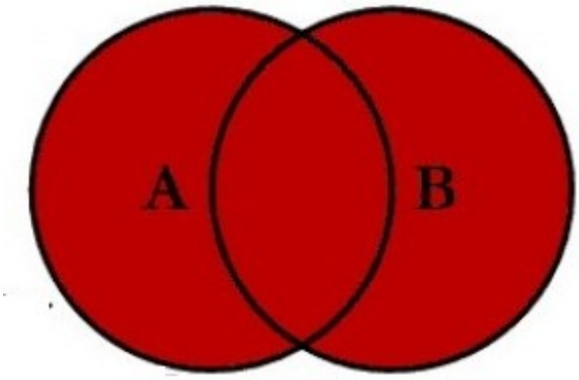



```
SELECT d.id, d.name
FROM users u
LEFT/RIGHT [OUTER] JOIN departments d ON u.d_id = d.id
WHERE u.id IS null
```

Оператор **SQL FULL JOIN** осуществляет формирование таблицы из записей двух или нескольких таблиц. В операторе SQL FULL JOIN не важен порядок следования таблиц, он никак не влияет на окончательный результат, так как оператор является симметричным.

Оператор SQL FULL JOIN можно воспринимать как сочетание операторов SQL INNER JOIN + SQL LEFT JOIN + SQL RIGHT JOIN. Алгоритм его работы следующий:

1. Сначала формируется таблица на основе внутреннего соединения (оператор SQL INNER JOIN).
2. Затем, в таблицу добавляются значения не вошедшие в результат формирования из правой таблицы (оператор SQL LEFT JOIN). Для них, соответствующие записи из правой таблицы заполняются значениями NULL.
3. Наконец, в таблицу добавляются значения не вошедшие в результат формирования из левой таблицы (оператор SQL RIGHT JOIN). Для них, соответствующие записи из левой таблицы заполняются значениями NULL.




```
SELECT u.id, u.name, d.name AS d_name
FROM users u
FULL [OUTER] JOIN departments d ON u.d_id = d.id
```

Элемент **SELF JOIN** используется для объединения таблицы с ней самой таким образом, будто это две разные таблицы, временно переименовывая одну из них.