MySQL

ВЫБОРКА ИЗ НЕСКОЛЬКИХ ТАБЛИЦ: СОЕДИНЕНИЯ И ПОДЗАПРОСЫ

СОДЕРЖАНИЕ

СОЕДИНЕНИЯ

ВВЕДЕНИЕ

- В прошлой главе мы подробно рассмотрели выборку из одной таблицы, однако мы должны помнить, что в базе данных могут одновременно находиться десятки или даже сотни таблиц. Более того, правила нормализации и виды связей ясно дали нам понять, что в разных таблицах могут храниться тесно связанные между собой данные, которые иногда было бы неплохо получить в рамках одного запроса. Конечно, при помощи какого-либо языка программирования можно сначала сделать выборку из одной таблицы, потом сделать выборку необходимых нам значений из других таблиц и в конце концов соединить данные вместе, используя какие-либо общие признаки. Однако делать много запросов для такий целей - крайне неэффективная и затратная по ресурсам стратегия, а приложения, построенные таким образом, будут работать крайне медленно. Поэтому в реляционных базах данных есть способы получить данные из нескольких таблиц при помощи одного запроса.

- Всего мы рассмотрим четыре варианта соединения таблиц - внутреннее соединение (**INNER JOIN**), левое соединение (**LEFT OUTER JOIN**), правое соединение (**RIGHT OUTER JOIN**), перекрестное соединение (**CROSS JOIN**), а также попробуем воспроизвести полное соединение (MySQL сама по себе не разрешает делать полное соединение таблиц, но этого все равно можно добиться с помощью дополнительных манипуляций, например при помощи оператора **UNION**).

- Для того, чтобы начать работать с соединениями, для начала создадим 4 таблицы (employees, passports, clients, projects) и заполним их данными. После создания и всех вставок таблицы должны иметь следующий вид:

Таблица **employees**:

id	**name**	**position**
1	Barbara Schmidt	System Analytic
2	Joe D'Amato	Android Programmer
3	John Smith	Web Programmer

Таблица **passports:**

employee_id	code	nationality	age	sex
1	s8f8798sdf	German	35	woman
2	98dfg87dg8	Italian	22	man
NULL	8s89sf889	Spanish	18	man

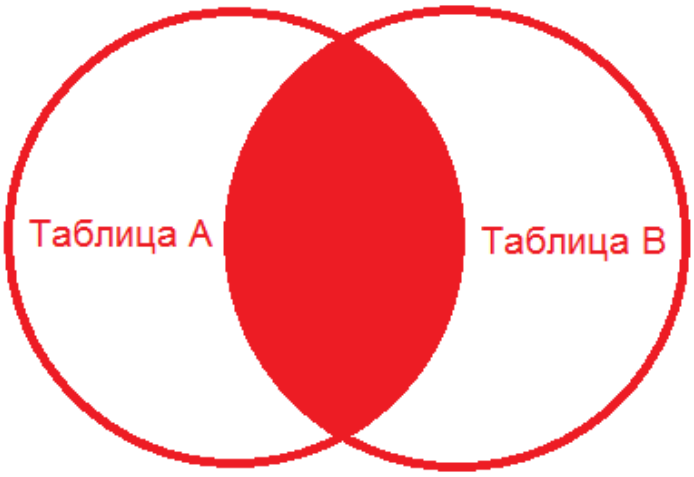
Таблица **clients:**

id	name
1	Google
2	Apple
3	Microsoft

Таблица **projects:**

id	name	employee_id	client_id
1	CMS Development	1	2
2	Mobile App development	2	1
3	Site Development	1	1

ВНУТРЕННЕЕ СОЕДИНЕНИЕ (INNER JOIN)



- При использовании внутреннего соединения таблиц мы хотим получить данные из двух или более таблиц только для тех строк, которые имеют совпадающие данные как в первой, так и во второй таблице (третьей, четвертой и т.д.). Если у некой строки (не важно в какой из таблиц) такие совпадающие данные отсутствуют, то эта строка будет проигнорирована.

- Для реализации внутреннего соединения в запросе после написания названия первой таблицы нам необходимо применить конструкцию **INNER JOIN** или просто **JOIN** (в MySQL **INNER** можно не писать), потом указать название следующей таблицы, а уже после названия поставить ключевое слово **ON** и вписать некое условие, которое должно выполняться для обеих соединяемых таблиц. Приведем пример - мы хотим узнать национальности наших сотрудников. Имена сотрудников хранятся в таблице employees, а национальности - в таблице passports. Общим условием будет равенство значения столбца id в employees и столбца employee_id в passports. Если эти значения равны, данные из обеих таблиц попадут в результирующую таблицу:

```
SELECT name, nationality
FROM employees
INNER JOIN passports ON id = employee_id;
```

Результат закономерный - в первой таблице был проигнорирован John Smith (для его id 3 таблицы employee не нашлось такого же значения в столбце employee_id таблицы passports), в во второй таблице - паспорт для человека с испанской национальностью (в таблице employees нет id равного **NULL**). Кроме того, заметим, что сравнение проводилось по тем столбцам, которые не появились в результирующей таблице (id, employee_id) - значит, их вовсе необязательно включать в выборку (перечислять после **SELECT**):

name	**nationality**
Barbara Schmidt	German
Joe D’Amato	Italian

- Предыдущий пример является абсолютно правильным с точки зрения синтаксиса и логики, но он таит в себе одну

опасность, которая может привести к ошибке в других запросах такого рода. Предположим, что мы хотим узнать названия всех проектов, которые у нас есть, а также имена работников, которые их курируют. Сразу же увидим проблему - имя работника хранится в таблице employees в столбце name, а название проекта хранится в таблице projects в столбце, который тоже имеет название name! В этом случае мы даже не можем дать столбцу псевдоним, т.к. мы точно не будем знать, какой столбец подразумевается первоначально. Кроме того, проявится еще одна ошибка - обе таблицы имеют столбец id, который будет участвовать в соединении. Но мы опять же не можем точно сказать, какой из таблиц он принадлежит.

- Для решения этой проблемы можно прибегнуть к двум способам - первый громоздкий, а второй - более элегантный. Во-первых, перед названием любого столбца можно писать название его таблицы и точку (также для порядка дадим столбцам нормальные псевдонимы):

```
SELECT
    employees.name AS employee_name,
    projects.name AS project_name
FROM employees
INNER JOIN projects ON employees.id = projects.employee_id;
```

Второй способ - дать короткий (уникальный в рамках запроса) псевдоним самим таблицам:

```
SELECT e.name AS employee_name, p.name AS project_name
FROM employees AS e
INNER JOIN projects AS p ON e.id = p.employee_id;
```

В результате получим таблицу с тремя строка - два раза повторится Barbara Schmidt (она курирует сразу два проекта), один раз - Joe D’Amato, а John Smith будет проигнорирован, т.к. он не связан ни с одним проектом:

employee_name	project_name
Barbara Schmidt	CMS Development
Barbara Schmidt	Site Development
Joe D’Amato	Mobile App Development

- Далее рассмотрим пример запроса, в котором участвует сразу три таблицы. Например, попробуем узнать имя работника (таблица employees столбец name), название проекта (который этот работник курирует - таблица projects столбец name) и название клиента, который этот проект заказал (таблица clients столбце name). Тут нет никаких особых моментов, просто **INNER JOIN** будет повторен два раза (соединение employees и projects, а потом к этому соединению добавится clients):

```
SELECT
    e.name AS employee_name,
    p.name AS project_name,
    c.name AS client_name
FROM employees AS e
INNER JOIN projects AS p ON e.id = p.employee_id
INNER JOIN clients AS c ON p.client_id = c.id;
```

  Получим результат, похожий на предыдущий, но добавится столбец client_name с названием клиента. Важный момент - при соединении таблиц в условии после ключевого слова **ON** можно обращаться только к столбцам тех таблиц, которые уже были упомянуты до обращения (в контексте **FROM** или **JOIN**), если обратиться к тем, которые упоминаются ниже - будет ошибка.

employee_name	project_name	client_name
Joe D'Amato	Mobile App Development	Google
Barbara Schmidt	Site Development	Google
Barbara Schmidt	CMS Development	Apple

- При соединении таблиц важно понимать ещё один аспект - в качестве условия может выступать любое количество столбцов и их сравнений, которые мы можем перечислить с помощью ключевых слов **AND** или **OR**. Например, попробуем узнать названия тех проектов, которые заказал Google:

```
SELECT p.name
FROM projects p
INNER JOIN clients c
    ON p.client_id = c.id AND c.name = 'Google';
```

name
Mobile App Development
Site Development

- Несмотря на то, что мы используем условия в соединениях, надо помнить, что стандартные условия (**WHERE**) и сортировки (**ORDER BY**) никто не отменял. В рамках этих конструкций можно использовать любые столбцы (как те, которые перечислены после **SELECT**, так и те, которые там не упомянуты) любых таблиц, но важно не забывать про псевдонимы этих таблиц. Например, получим работников и их возраст, но установим, что возраст должен быть меньше 30 лет. Кроме того, отсортируем результат по возрасту в возрастающем порядке:

```
SELECT e.name, p.age
FROM passports p
INNER JOIN employees e ON p.employee_id = e.id
WHERE p.age < 30
ORDER BY p.age ASC;
```

Получим одну строку с Joe D'Amato, которому 22 года. Barbara Schmidt не подходит, т.к. ей больше 30 лет, а сколько лет John Smith мы не знаем, т.к. у него нет паспорта:

name	age
Joe D'Amato	22

- Упомянет одну особенность, которая напрямую не связана с соединением таблиц. Мы уже помним, что оператор звездочка * позволяет выбрать все значения всех таблиц, участвующих в запросе. Однако * применяется еще в одном контексте - если мы используем псевдоним таблицы, то при при перечислении выбираемых столбцов можно указать звездочку после этого псевдонима и точки - тогда мы получим значения всех столбцов только той таблицы, чей псевдоним мы использовали:

```
SELECT e.*
FROM passports p
INNER JOIN employees e ON p.employee_id = e.id
WHERE p.age < 30
ORDER BY p.age ASC;
```

Получим значения всех столбцов таблицы employees (которые подходят под наш запрос):

id	name	position
2	Joe D'Amato	Android Programmer

- В завершение этой подглавы откроем секрет - внутреннее соединение таблиц можно реализовать и без ключевого слова **JOIN**. Достаточно через запятую перечислить таблицы после FROM, а в контексте **WHERE** указать условия, по которым таблицы мы будем соединять:

```
SELECT
    e.name AS employee_name,
    p.name AS project_name,
    c.name AS client_name
FROM employees AS e, projects AS p, clients AS c
WHERE e.id = p.employee_id AND p.client_id = c.id;
```

employee_name	project_name	client_name
Joe D'Amato	Mobile App Development	Google
Barbara Schmidt	Site Development	Google
Barbara Schmidt	CMS Development	Apple

ЛЕВОЕ ВНЕШНЕЕ СОЕДИНЕНИЕ (LEFT OUTER JOIN)

- Левое внешнее соединение синтаксически мало чем отличается от внутреннего соединения (само собой, вместо **INNER JOIN** следует писать **LEFT OUTER JOIN**, кроме того, OUTER можно игнорировать - достаточно только **LEFT JOIN**), но практически различие очевидно. Данное соединение гарантирует, что строки таблицы (или таблиц), которая была запрошена вначале (условно говоря - слева), будут гарантированно включены в результирующую выборку (хотя бы один раз) - даже в том случае, если значения совпадающих столбцов отсутствуют. Вместо отсутствующих значений столбцов будут возвращены значения **NULL**.

- Для иллюстрации работы левого внешнего соединения воспроизведем первый запрос из предыдущей главы с работниками и паспортами, который вернул нам две строки. На этот раз вместо INNER JOIN воспользуемся LEFT OUTER JOIN (и для порядка укажем псевдонимы - в данном случае это необязательно, но желательно):

```
SELECT e.name, p.nationality
FROM employees e
LEFT OUTER JOIN passports p ON e.id = p.employee_id;
```

  В отличие от запроса с INNER JOIN наш новый запрос учел таблицу слева и вернул работника John Smith даже несмотря на то, что у него нет паспорта. Однако мы не видим национальности Spanish - столбец “национальность” находится в таблице passport, а она находится условно “справа” - именно поэтому **LEFT JOIN** не учитывает ее значения:

name	**nationality**
Barbara Schmidt	German
Joe D’Amato	Italian
John Smith	NULL

- Попробуем слегка изменить запрос - сначала (слева) поставим таблицу паспортов, а затем - таблицу работников:

```
SELECT name, nationality
FROM passports p
LEFT OUTER JOIN employees e ON p.employee_id = e.id;
```

Результирующая таблица закономерна изменилась - John Smith пропал, однако появилась ни к кому не относящаяся национальность Spanish:

name	nationality
Barbara Schmidt	German
Joe D'Amato	Italian
NULL	Spanish

- Далее сделаем запрос **LEFT JOIN** в 3 таблицы (работник, его проекты и клиенты этих проектов) - из них данные John Smith присутствуют (прямо или косвенно) только в первой таблице:

```
SELECT
    e.name AS employee_name,
    p.name AS project_name,
    c.name AS client_name
FROM employees AS e
LEFT JOIN projects AS p ON e.id = p.employee_id
LEFT JOIN clients AS c ON p.client_id = c.id;
```

Несмотря на то, что John Smith появляется только в одном случае из трех, он появится в результирующей выборке. Произошло следующее - сначала соединились employees и projects - John Smith закономерно попадает в результат по праву **LEFT JOIN**. Затем соединяется первое соединение (employees и projects) и clients - опять же данные John Smith присутствуют в первом (левом) соединении, поэтому он опять попадает в конечную таблицу:

employee_name	project_name	client_name
Barbara Schmidt	CMS Development	Apple
Barbara Schmidt	Site Development	Google
Joe D'Amato	Mobile App Development	Google
John Smith	NULL	NULL

- Напоследок попробуем повторить предыдущий запрос, но в соединении с последней таблицей укажем не **LEFT JOIN**, а **INNER**

JOIN - это необходимо для того, чтобы показать, что тип соединения не распространяется на весь запрос, а в случае тех двоих таблиц, где он конкретно применяется:

```
SELECT
    e.name AS employee_name,
    p.name AS project_name,
    c.name AS client_name
FROM employees AS e
LEFT JOIN projects AS p ON e.id = p.employee_id
INNER JOIN clients AS c ON p.client_id = c.id;
```

А вот теперь John Smith не будет включен в финальную таблицу. Первый раунд **LEFT JOIN** с участием таблиц employees и projects он преодолеет, а второй, когда соединяется первое соединение (employees и projects) и таблица clients, он уже не пройдет. Мы помним, что **INNER JOIN** требует чтобы данные совпадали с обеих сторон, а у John Smith нет ни одного проекта - следовательно его нельзя соединить с клиентами:

employee_name	project_name	client_name
Joe D'Amato	Mobile App Development	Google
Barbara Schmidt	Site Development	Google
Barbara Schmidt	CMS Development	Apple

ПРАВОЕ ВНЕШНЕЕ СОЕДИНЕНИЕ (RIGHT OUTER JOIN)

- Как можно понять из названия, правое внешнее соединение является абсолютным антиподом левого внешнего соединения. При соединении таблиц нам гарантируется, что все строки таблицы, которая была запрошена позже (условно “справа”), будут нам возвращены в любом случае (хотя бы один раз) - даже если для них не выполняется условие соединения. Синтаксис очень похож на **INNER JOIN** и **LEFT OUTER JOIN** - сохраняются все предыдущие особенности, но главная конструкция называется **RIGHT OUTER JOIN** (как и ранее, слово **OUTER** можно не писать - в MySQL предоставляется возможность писать только **RIGHT JOIN**).

- Повторим наш запрос из предыдущих глав с именами работников (таблица employees) и их национальностями (таблица passports) в контексте правого соединения. Первой (левой) будет идти таблица работников, а второй таблицей (правой) будет идти таблица паспортов:

```
SELECT e.name, p.nationality
FROM employees e
RIGHT OUTER JOIN passports p ON e.id = p.employee_id;
```

  Результат можно было понять еще до выполнения запроса - мы точно получим все национальности (включая Spanish), т.к. таблица passports запрашивается позднее таблицы (employees), т.е. находится справа. В свою очередь, John Smith из таблицы employees не будет включен в выборку, т.к. у него нет паспорта и для него не выполняется условие соединения:

name	nationality
Barbara Schmidt	German
Joe D’Amato	Italian
NULL	Spanish

- Теперь сделаем тот же самый запрос с правым соединением и с тем же самым условием, но поменяем таблицы местами (сначала обратимся к таблице passports, а уже потом к employees):

```
SELECT e.name, p.nationality
FROM passports p
RIGHT OUTER JOIN employees e ON p.employee_id = e.id;
```

Теперь результат изменился на противоположный - национальность Spanish исчезла, т.к. таблица passports оказалась слева и для испанского паспорта не существует работника, а John Smith появился, т.к. несмотря на то, что у него нет паспорта, его таблица employees расположена справа:

name	nationality
Barbara Schmidt	German
Joe D'Amato	Italian
John Smith	NULL

- Теперь попробуем совместить внутреннее и правое соединение в одном запросе. Для этого представим, что нам надо получить всех работников, которые задействованы на проектах, а также все клиенты, которые эти проекты заказали. Есть обязательное условия - обязательно должны все клиенты (даже не заказавшие проектов). Это можно осуществить следующим образом:

```
SELECT
    e.name AS employee_name,
    p.name AS project_name,
    c.name AS client_name
FROM employees AS e
INNER JOIN projects AS p ON e.id = p.employee_id
RIGHT JOIN clients AS c ON p.client_id = c.id;
```

Теперь мы впервые получим в результате клиента Microsoft, т.к. он находится в таблице clients, а она запрашивается при помощи правого соединения:

employee_name	project_name	client_name
Barbara Schmidt	CMS Development	Apple
Barbara Schmidt	Site Development	Google
Joe D'Amato	Mobile App Development	Google
NULL	NULL	Microsoft

- Важный момент - с теми знаниями, которыми мы обладаем на данный момент, мы не можем гарантированно получить всех

работников (даже без проектов), затем получить всех проекты, а затем получить всех клиентов, которые как заказали эти проекты, так и не заказали. Есть соблазн сделать это, объединив в одном запросе правое и левое соединение сразу:

```
SELECT
    e.name AS employee_name,
    p.name AS project_name,
    c.name AS client_name
FROM employees AS e
LEFT JOIN projects AS p ON e.id = p.employee_id
RIGHT JOIN clients AS c ON p.client_id = c.id;
```

Мы опять получим в результирующей таблице Microsoft в качестве клиента (он не заказывал проектов, но запрошен при помощи правого соединения), но не выполним первое условие - получить всех работников - John Smith будет отсутствовать. Попытаемся это объяснить. Первое соединение (employees и projects) John Smith благополучно переживет - он не курирует проекты, но соединение осуществляется при помощи **LEFT JOIN**, а таблица employees находится как раз слева. А вот из при втором соединении строка с John Smith будет иметь **NULL** в качестве client_id, то есть не будет иметь клиента для проекта. Правое соединение гарантирует присутствие только для тех столбцов, таблицы которых находятся справа. Первое соединение (employees и projects) будет находится слева во время второго соединения, а условие соединения выполняться не будет - поэтому все строки с John Smith будут отброшены:

employee_name	project_name	client_name
Barbara Schmidt	CMS Development	Apple
Barbara Schmidt	Site Development	Google
Joe D’Amato	Mobile App Development	Google
NULL	NULL	Microsoft

ПЕРЕКРЕСТНОЕ СОЕДИНЕНИЕ (CROSS JOIN)

- Далее рассмотрим крайне любопытный вид соединения - перекрестное соединение, которое также известно как “декартово произведение”. Суть его состоит в том, чтобы каждой строке из первой соединяемой таблицы была соотнесена каждая строка из второй соединяемой таблицы - таким образом мы можем получить все комбинации этих строк. Данное соединение крайне редко используется в настоящих проектах - чаще всего просто для генерации большого количества тестовых данных.

- Синтаксис перекрестного соединения представляет собой конструкцию CROSS JOIN, которой можно не прописывать какие либо условия. Для примера перекрестно соединим таблицы employees и passports:

```
SELECT e.name, p.nationality
FROM employees e
CROSS JOIN passports p;
```

  В результате все имена работников будут соединены со всеми национальностями без всяких условий:

name	nationality
John Smith	German
Joe D’Amato	German
Barbara Schmidt	German
John Smith	Italian
Joe D’Amato	Italian
Barbara Schmidt	Italian
John Smith	Spanish
Joe D’Amato	Spanish
Barbara Schmidt	Spanish

- Интересно заметить, что **CROSS JOIN** не запрещает использование условий соединения, просто после внедрения таких условий **CROSS JOIN** ведет себя как **INNER JOIN:**

```
SELECT e.name, p.nationality
FROM employees e
CROSS JOIN passports p ON e.id = p.employee_id;
```

Результат - отброшены все неподходящие строки из всех таблиц:

name	nationality
Barbara Schmidt	German
Joe D'Amato	Italian

- Интересно заметить, что декартово произведение можно осуществить и без конструкции **CROSS JOIN** - всего навсего можно перечислить необходимые таблицы через запятую без всяких условий:

```
SELECT e.name, p.nationality
FROM employees e, passports p;
```

Результат такой же, как у первого запроса в этой подглаве:

name	nationality
John Smith	German
Joe D'Amato	German
Barbara Schmidt	German
John Smith	Italian
Joe D'Amato	Italian
Barbara Schmidt	Italian
John Smith	Spanish
Joe D'Amato	Spanish
Barbara Schmidt	Spanish

ПОЛНОЕ ВНЕШНЕЕ СОЕДИНЕНИЕ

- Иногда нам может потребоваться запрос, в рамках которого мы должны гарантированно получить все строки как из “левой” таблицы, так и из “правой” таблицы (хотя бы один раз). MySQL не имеет конструкций конкретно для этой задачи, но предоставляет оператор **UNION**, который позволяет объединить результаты двух подзапросов. Таким образом, мы можем сначала сделать подзапрос с левым соединением, потом подзапрос с правым соединением, а уже после этого объединить результаты в одну конечную таблицу.

- Применение оператора **UNION** накладываем на нас два ограничения. Во-первых, результаты объединяемых подзапросов должны иметь одинаковое количество и одинаковый порядок выбранных столбцов. Во-вторых, типы выбираемых столбцов должны быть совместимыми (если в первом подзапросе первый столбец имеет тип **INT**, то и во втором подзапросе первый столбец должен иметь тип **INT**, а не **JSON**, **DATETIME** и т.д.).

- Кроме ограничений, оператор **UNION** имеет одну интересную особенность. Если при объединении таблиц в результирующей выборке будут полностью идентичные строки, то **UNION** не будет их показывать все, а покажет только уникальные варианты. Если нам по какой-то причине нужны дублирующиеся строки, то вместо **UNION** следует использовать **UNION ALL**.

- Попробуем в очередной раз получить имена работников и национальности, но уже с применением полного внешнего соединения:

```
(
    SELECT e.name, p.nationality
    FROM employees e
    LEFT JOIN passports p ON e.id = p.employee_id
)
UNION
(
    SELECT e.name, p.nationality
    FROM employees e
    RIGHT JOIN passports p ON e.id = p.employee_id
);
```

В результате получим таблицу где будет и John Smith и национальность Spanish - даже несмотря на то, что у них нет совпадающих данных в противоположных таблицах:

name	nationality
Barbara Schmidt	German
Joe D'Amato	Italian
John Smith	NULL
NULL	Spanish

- Далее проверим работу оператора **UNION ALL** (должны появится одинаковые строки):

```
(
    SELECT e.name, p.nationality
    FROM employees e
    LEFT JOIN passports p ON e.id = p.employee_id
)
UNION ALL
(
    SELECT e.name, p.nationality
    FROM employees e
    RIGHT JOIN passports p ON e.id = p.employee_id
);
```

В конечной таблице мы увидим, что Barbara Schmidt - German и Joe D'Amato - Italian повторяется два раза (они были выбраны как в левом, так и в правом соединении):

name	nationality
Barbara Schmidt	German
Joe D'Amato	Italian
John Smith	NULL
Barbara Schmidt	German
Joe D'Amato	Italian
NULL	Spanish

- Говоря об операторе **UNION**, нельзя не упомянуть о двух его "собратьях" - операторах **INTERSECT** и **EXCEPT**. **INTERSECT** также объединяет таблицы, но оставляет только те строки, одинаковые комбинации которых присутствуют как в результате первого подзапроса, так и в результате второго:

```
(
    SELECT e.name, p.nationality
    FROM employees e
    LEFT JOIN passports p ON e.id = p.employee_id
)
INTERSECT
(
    SELECT e.name, p.nationality
    FROM employees e
    RIGHT JOIN passports p ON e.id = p.employee_id
);
```

В результате останутся Barbara Schmidt - German и Joe D'Amato - Italian - как уже упоминалось, они были выбраны как в левом, так и в правом соединении:

name	nationality
Barbara Schmidt	German
Joe D'Amato	Italian

Оператор **EXCEPT** делает следующее - при объединении двух таблиц он оставляет в конечной выборке только те комбинации строк, которое есть в результате первого подзапроса - если

такая комбинация присутствует и в результате второго подзапроса, то эта строка будет удалена из конечной таблицы:

```
(
    SELECT e.name, p.nationality
    FROM employees e
    LEFT JOIN passports p ON e.id = p.employee_id
)
EXCEPT
(
    SELECT e.name, p.nationality
    FROM employees e
    RIGHT JOIN passports p ON e.id = p.employee_id
);
```

Строки Barbara Schmidt - German и Joe D’Amato - Italian были удалены, т.к. они присутствовали в двух результатах подзапросов, а строка NULL - Spanish была убрана, т.к. она присутствует только во втором подзапросе. Остался только John Smith с нулевой национальностью:

name	nationality
John Smith	NULL

ПОДЗАПРОСЫ

ВВЕДЕНИЕ

- Очень часто складывается ситуация, когда нам необходимо получить данные из какой-то одной таблицы, но условие для выборки строк должно полагаться на данные из другой таблицы. То есть получается, что нам не надо объединять таблицы - нам необходимо получить некий набор промежуточных значений из другой таблицы, чтобы опереться на них при выборке из первой таблицы. Такие запросы во вторую таблицу называются подзапросами.

- Перед тем, как работать с подзапросами, создадим необходимые для этого таблицы и заполним их данными. Условимся, что у нас будут три таблицы - магазины (shops), расположенные в некоторых городах, посетители (customer), которые также живут в каких-то городах, а также заказы (orders), которые могут делать посетители в каких-либо магазинах на определенную сумму. После всех созданий и вставок данных таблицы должны выглядеть следующим образом:

Таблица **shops:**

id	brand	city
1	Maxima	Rīga
2	Lidl	Rīga
3	Rimi	Daugavpils
4	Top	Liepaja

Таблица **customers:**

id	name	debt	city
1	John Smith	120.00	Rīga
2	Brandon Adams	0.00	Liepaja
3	Mary Sue	0.00	Rēzekne
4	Paul George	17.00	Rīga
5	Connor MacLeod	0.00	Daugavpils

Таблица **orders**:

id	customer_id	shop_id	price
1	1	1	999.99
2	1	2	505.69
3	1	2	656.17
4	3	4	15.22
5	3	2	199.76
6	3	2	1059.13
7	4	1	236.87
8	4	1	517.89
9	5	3	1000.00
10	5	1	700.11

ПОДЗАПРОС ДЛЯ ПОЛУЧЕНИЯ ЗНАЧЕНИЯ ДЛЯ СРАВНЕНИЯ

- Представим, что нам надо разослать сообщения для тех людей, которые живут в том городе, в котором расположен некий магазин. Однако мы не знаем, в каком конкретно городе расположен магазин, но знаем его id (например, id = 2). Для этого мы должны сначала получить город магазина (это и будет подзапрос), а потом использовать этот город для сравнения с городами посетителей. Делается это следующим образом:

```
SELECT id, name FROM customers
WHERE city = (SELECT city FROM shops WHERE id = 2);
```

Запрос составлен совершенно правильно, но нам надо всегда помнить две вещи. Во-первых, такой подзапрос не должен возвращать более одной строки (можно 1 или 0 строк), т.к. мы не можем сравнить одиночное значение (в данном случае city) с множеством значений даже в том случае, если какое-либо из них будем правильным. Если мы не уверены, что нам всегда будут выбрано менее 2 строк, мы можем поставить в подзапросе ограничение **LIMIT**. Во-вторых, мы не должны возвращать в подзапросе значение только одного столбца, т.к. опять же мы не можем сравнить значение города с одной стороны и значения города и id с другой. Далее результат выборки:

id	brand
1	John Smith
4	Paul George

ПОДЗАПРОС ДЛЯ ПОЛУЧЕНИЯ НАБОРА ЗНАЧЕНИЙ ДЛЯ СРАВНЕНИЯ

- Иногда нам недостаточно сравнить некое значение с другим значением из подзапроса. Представим, что появилась необходимость получить список имен посетителей, совершивших заказы. Для этого мы должны получить список идентификаторов клиентов из таблицы заказов, а потом выбрать из таблицы посетителей имена тех, чьи идентификаторы встречаются в списке идентификаторов клиентов, полученных из заказов. Простым сравнением мы уже не обойдемся, т.к. сравнивать одинарное значение и множество мы не можем. Однако тут на помощь приходит оператор **IN**, который может возвращать некий набор значений. Совсем необязательно вписывать этот набор значений - набор можно получить как раз с помощью подзапроса:

```
SELECT id, name FROM customers
WHERE id IN (SELECT customer_id FROM orders);
```

  В результате мы увидим, что у заказы имеют четыре человека из пяти возможных. Важно запомнить, что оператор IN позволяет снять ограничение на количество возвращаемых подзапросом строк. Ограничение на столбцы по прежнему остается - в каждой строке, возвращенной подзапросом, должен Быть только один столбец. Далее результаты выборки для клиентов, имеющих заказы:

id	name
1	John Smith
3	Mary Sue
4	Paul George
5	Connor MacLeod

- Заметим, что в **IN** в данном контексте имеет альтернативу. Дело в том, что оператор **IN** предназначен как для получения результатов подзапросов, так может вернуть список значений, поданных в запрос вручную. Мы уже говорили об этом в прошлой главе, но в данном случае проиллюстрирует это на примере:

```
SELECT id, name FROM customers WHERE id IN (1, 2, 3);
```

  В MySQL есть специальный оператор, который так делать не умеет, но он также способен вернуть некий набор значений с помощью подзапроса, кроме того, оа имеет некие дополнительные удобные возможности. Этот оператор называется **ANY** (также он имеет псевдоним **SOME** - запомним, если встретим в коде **ANY** или **SOME**, то это одно и то же), и он всегда используется в связке с операторами =, >, >=, <, <=, =, <>. Например, повторим запрос на получение всех клиентов с заказами:

```
SELECT id, name FROM customers
WHERE id = ANY(SELECT customer_id FROM orders);
```

  В выборке будут все те же четыре человека, какие были получены при помощи оператора **IN**. Можно задать вопрос - зачем нужен **ANY**, если уже есть **IN**? Ответ очевиден - **IN** не умеет пользоваться операторами сравнения, а **ANY** - умеет. Например, получим всех клиентов, которые имеют долг больше, чем цена какого-либо из их предыдущих заказов (клиентов, которые много себе позволяют):

```
SELECT c.id, c.name, c.debt FROM customers c
WHERE c.debt > ANY(
    SELECT price FROM orders o
    WHERE o.customer_id = c.id
);
```

  Заметим, что в данном запросе помимо использования нового оператора **ANY** присутствует еще одна особенность - подзапрос обращается к значению id основного запроса. Мы еще вернемся к подобным видам запросов и подзапросов, но на данный момент запомним, что подзапрос имеет доступ к значениям тех запросов, которое находятся условно “выше” него. А результатом всего запроса будет один злостный должник:

id	name	debt
4	Paul George	300.00

- Используя **IN**, мы должны были вспомнить тот факт, что также существует конструкция **NOT IN**. Не будет сюрпризом, что ее можно использовать и в контексте подзапросов. Например, в начале этой подглавы мы получили всех посетителей, которые имели заказы. Попробуем перевернуть логику, применить **NOT IN** и получить всех посетителей, у которых заказов нет:

```
SELECT id, name FROM customers
WHERE id NOT IN (SELECT customer_id FROM orders);
```

Добавив оператор **NOT** мы сразу же узнаем, что у нас есть всего навсего один посетитель Brandon Adams, который не совершил ни одного заказа:

id	name
2	Brandon Adams

- Нужно заметить, что **ANY** не имеет прямой альтернативы для конструкции **NOT IN**. Если мы смогли повторить при помощи него выборку клиентов с заказами, то результат выборка похожим образом списка клиентов без заказов будет очень неожиданным - нам вернуться все все посетители, хотя должен был вернуться только один Brandon Adams:

```
SELECT id, name FROM customers
WHERE id <> ANY(SELECT customer_id FROM orders);
```

id	name
1	John Smith
2	Brandon Adams
3	Mary Sue
4	Paul George
5	Connor MacLeod

Почему же так произошло? Почему в данном случае **NOT IN** работает, а **<> ANY** работает не так, как мы хотим? Попробуем сначала проговорить логику работы **NOT IN**. Сначала выбираются все id посетителей из заказов, а потом при выборке самих посетителей проверяется, присутствует ли их id в первой выборке. Если есть - посетитель отбрасывается. А теперь поговорим логику работы **<> ANY**. Сначала опять выбираются все id посетителей из заказов, но потом происходит следующее - при выборке посетителей проверяется - есть ли в результате первого подзапроса какой-либо id посетителя, который не равен данному. Он всегда будет, т.к. в нашем случае вернулись id для нескольких людей, которые сделали заказы.

КОРРЕЛИРУЮЩИЕ ПОДЗАПРОСЫ

- Когда мы получали список злостных должников, то мы обратили внимание на получение данных внешнего основного запроса внутри подзапроса. Такой вид подзапроса называется коррелирующим, и он очень часто может привести к очень нехорошим последствиям. Целость данных не пострадает, все будет возвращаться верно, но при большом количестве данных запрос будет осуществляться очень медленно. Дело в том, что при коррелирующем подзапросе этот подзапрос происходит не один раз, а столько раз, сколько у нас строк в основном запросе, т.к. проверка значения осуществляется для каждой строки. Например, если у нас 1000 клиентов, то общее количество будет запросов будет равно 1000. Поэтому такие подзапросы следует делать только в исключительных случая и всеми силами пытаться заменить их на обычные. К сожалению, мы не может исправить наш оригинальный запрос, т.к. это единственный вариант получить всех злостных должников:

```
SELECT c.id, c.name, c.debt FROM customers c
WHERE c.debt > ANY(
    SELECT price FROM orders o
    WHERE o.customer_id = c.id
);
```

- В контексте коррелирующих запросов следует понимать, что корреляция не связана с использованием какой-либо конкретной конструкции (IN, ANY и т.д.). Именно обращение к данным внешнего запроса делает любой подзапрос коррелирующим.

- Чтобы проиллюстрировать работу коррелирующих запросов, обратимся к новому для нас оператору **ALL** - он используется схожим с **ANY** образом, поддерживает операторы сравнения , >=, <, <=, =, <>, требует возвращение только одного столбца, но одновременно считает условие истинным, если значения всех (а не хотя бы какого либо одного, как у **ANY**) строк будут соответствовать проверяемому значению. Например, попробуем выбрать всех посетителей, которые делали заказы в магазинах только тех городов, где они и живут:

```
SELECT c.id, c.name, c.city FROM customers c
WHERE c.city = ALL(
    SELECT city FROM shops s WHERE s.id IN (
        SELECT o.shop_id FROM orders o
        WHERE o.customer_id = c.id
    )
);
```

Этот запрос действительно является особенным. Сразу же заметим, что в данном случае сделано целых два подзапроса - во-первых получены все магазины, в которых конкретный посетитель делал заказы, во-вторых, для этих магазинов получены те города, в которых они расположены, а уже потом с помощью ALL проведена проверка, все ли города равны городу посетителя. Результат тоже несколько необычный - если John Smith и Paul George действительно делали заказы только в магазинах своего города проживания, но Brandon Adams вообще не делал заказов. Но если проговорить условия запроса **ALL**, то мы поймем, почему так произошло. Мы хотим, чтобы количество раз, когда встречается город в ответе, этот город был бы равен городу посетителя. В случае Brandon Adams город вернулся 0 раз и равен городу Liepaja он был равен тоже 0 раз. 0 действительно равен 0, следовательно человек, не совершивший ни одного заказа и был включен в финальный результат:

id	name	city
1	John Smith	Rīga
2	Brandon Adams	Liepaja
4	Paul George	Rīga

- Еще один яркий пример коррелирующего подзапроса можно проиллюстрировать на примере того, как использовать подзапрос не в качестве участника фильтрации выборки, а в качестве присоединенного динамического столбца, т.е. выполним подзапрос сразу после **SELECT**. Для этого попробуем получить в контексте запроса посетителей тот магазин (любой один), который находится в том городе, где посетитель живет.

```
SELECT
    c.name,
    (SELECT s.brand FROM shops s WHERE s.city = c.city LIMIT
1) AS shop_brand
FROM customers c;
```

В подзапросе мы запрашивает названием магазина (один столбец, т.к. в данном случае мы можем вставить только 1 значение), а также принудительно ограничиваем количество строк выборки, т.к. в одном и том же городе может быть два магазина. Еще отметим, что Mary Sue в качестве рекомендованного магазина получила пустое значение NULL, т.к. в ее городе магазинов нет (в рамках записей в нашей базе данных):

name	city
John Smith	Maxima
Brandon Adams	Top
Mary Sue	NULL
Paul George	Rīga
Connor MacLeod	Rimi

- В конце наших рассуждений рассмотрим еще один оператор, который позволяет осуществлять подзапрос - **EXISTS**. Ему вообще не нужно значение для сравнения, не нужно следить за количеством столбцов или строк - условие будет считаться выполненным, если **EXISTS** найдет хотя бы одну любую строку. Но **EXISTS**, как и все остальные операторы, рассмотренные в нашем уроке, имеет доступ к значениям верхних запросов, поэтому он позволяет осуществлять коррелирующие запросы. Добавим, что у **EXISTS** есть конструкция-антипод **NOT EXISTS**, которая считает условие выполненным, если подзапрос не

вернул ни одной строки. Попробуем использовать оператор **EXISTS**, чтобы получить тех посетителей, у которых есть заказы на сумму 1000 евро или более:

```
SELECT c.id, c.name FROM customers c
WHERE EXISTS(SELECT * FROM orders o WHERE o.customer_id =
c.id AND o.price >= 1000);
```

id	name
3	Mary Sue
5	Connor MacLeod

Теперь сделаем наоборот - при помощи **NOT EXISTS** получим тех посетителей без заказов на сумму 1000 или более евро:

```
SELECT c.id, c.name FROM customers c
WHERE NOT EXISTS(SELECT * FROM orders o WHERE o.customer_id =
c.id AND o.price >= 1000);
```

id	name
1	John Smith
2	Brandon Adams
4	Paul George

Кстати, оба предыдущих запроса сделаны коррелирующими только в целях знакомства с конструкциями **EXISTS / NOT EXISTS** - их можно переписать и без корреляции:

```
-- есть заказы на 1000 или более 1000 евро
SELECT id, name FROM customers
WHERE id IN(SELECT customer_id FROM orders WHERE price >=
1000);

-- нет заказов на 1000 или более 1000 евро
SELECT id, name FROM customers
WHERE id NOT IN(SELECT customer_id FROM orders WHERE price >=
1000);
```