**Подзапросы**

**Подзапросы в SELECT**

В выражении **SELECT** мы можем вводить подзапросы четырьмя способами:

1. В условии в выражении **WHERE**
2. В условии в выражении **HAVING**
3. В качестве таблицы для выборки в выражении **FROM**
4. В качестве спецификации столбца в выражении **SELECT**

Рассмотрим некоторые из этих случаев. Например, получим все товары, у которых цена выше средней:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Products  WHERE Price > (SELECT AVG(Price) FROM Products) |

Чтобы получить нужные товары, нам вначале надо выполнить подзапрос на получение средней цены товара: SELECT AVG(Price) FROM Products.

**Оператор IN**

Нередко подзапросы применяются **вместе с оператором IN**, который выбирает из набора значений. И подзапрос как раз может предоставить требуемый набор значений. Например, выберем все товары из таблицы Products, на которые есть заказы в таблице Orders:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Products  WHERE Id IN (SELECT ProductId FROM Orders) |

То есть подзапрос в данном случае выбирает все идентификаторы товаров из Orders, затем по этим идентификаторам извлекаютя товары из Products.

Добавив оператор NOT, мы можем выбрать те товары, на которые нет заказов в таблице Orders:

|  |  |
| --- | --- |
| 1  2 | SELECT \* FROM Products  WHERE Id NOT IN (SELECT ProductId FROM Orders) |

Стоит отметить, что это не самый эффективный способ для извлечения связанных данных из других таблиц, так как для сведения данных из разных таблиц можно использовать оператор JOIN, который рассматривается в следующей главе.

**Получение набора значений**

При использовании в операторах сравнения подзапросы должны возвращать одно скалярное значение. Но иногда возникает необходимость получить набор значений. Чтобы при использовании в операторах сравнения подзапрос мог возвращать набор значений, перед ним необходимо использовать один из операторов: **ALL**, **SOME** или **ANY**.

При использовании ключевого слова ALL условие в операции сравнения должно быть верно для всех значений, которые возвращаются подзапросом. Например, найдем все товары, цена которых меньше чем у любого товара фирмы Apple:

|  |  |
| --- | --- |
|  | SELECT \* FROM Products  WHERE Price < ALL(SELECT Price FROM Products WHERE Manufacturer='Apple') |

Если бы мы в данном случае опустили бы ключевое слово ALL, то мы бы столкнулись с ошибкой.

Допустим, если данный подзапрос возвращает значения vаl1, val2 и val3, то условие фильтрации фактически было бы аналогично объединению этих значений через оператор AND:

|  |  |
| --- | --- |
| 1 | WHERE Price < val1 AND Price < val2 AND Price < val3 |

В тоже время подобный запрос гораздо проще переписать другим образом:

|  |  |
| --- | --- |
|  | SELECT \* FROM Products  WHERE Price < (SELECT MIN(Price) FROM Products WHERE Manufacturer='Apple') |

Как работает оператор ALL:

* x > ALL (1, 2) эквивалентно x > 2
* x < ALL (1, 2) эквивалентно x < 1
* x = ALL (1, 2) эквивалентно (x = 1) AND (x = 2)
* x <> ALL (1, 2) эквивалентно x NOT IN (1, 2)

Операторы ANY и SOME условие в операции сравнения должно быть истинным для хотя бы одного из значений, возвращаемых подзапросом. По своему действию оба этих оператора аналогичны, поэтому можно применять любой из них. Например, в следующем случае получим товары, которые стоят меньше самого дорогого товара компании Apple:

|  |  |
| --- | --- |
|  | SELECT \* FROM Products  WHERE Price < ANY(SELECT Price FROM Products WHERE Manufacturer='Apple') |

И также стоит отметить, что данный запрос можно сделать проще, переписав следующим образом:

|  |  |
| --- | --- |
|  | SELECT \* FROM Products  WHERE Price < (SELECT MAX(Price) FROM Products WHERE Manufacturer='Apple') |

Как работает оператор ANY (а также SOME):

* x > ANY (1, 2) эквивалентно x > 1
* x < ANY (1, 2) эквивалентно x < 2
* x = ANY (1, 2) эквивалентно x IN (1, 2)
* x <> ANY (1, 2) эквивалентно (x <> 1) OR (x <> 2)

**Подзапрос как спецификация столбца**

Результат подзапроса может представлять отдельный столбец в выборке. Например, выберем все заказы и добавим к ним информацию о названии товара:

|  |  |
| --- | --- |
|  | SELECT \*,  (SELECT ProductName FROM Products WHERE Id=Orders.ProductId) AS Product  FROM Orders |

**Подзапросы в команде INSERT**

В команде **INSERT** подзапросы могут применяться для определения значения, которое вставляется в один из столбцов:

|  |  |
| --- | --- |
|  | INSERT INTO Orders (ProductId, CreatedAt, ProductCount, Price)  VALUES  (      (SELECT Id FROM Products WHERE ProductName='Galaxy S8'),      '2018-05-23',      2,      (SELECT Price FROM Products WHERE ProductName='Galaxy S8')  ) |

**Подзапросы в команде UPDATE**

В команде **UPDATE** подзапросы могут применяться:

1. В качестве устанавливаемого значения после оператора **SET**
2. Как часть условия в выражении **WHERE**

Так, увеличим в таблице Orders количество купленных товаров компании Apple на 2:

|  |  |
| --- | --- |
|  | UPDATE Orders  SET ProductCount = ProductCount + 2  WHERE ProductId IN (SELECT Id FROM Products WHERE Manufacturer='Apple'); |

Или установим для заказа цену товара, полученную в результате подзапроса:

|  |  |
| --- | --- |
|  | UPDATE Orders  SET Price = (SELECT Price FROM Products WHERE Id=Orders.ProductId) + 3000  WHERE Id=1; |

**Подзапросы в команде DELETE**

В команде **DELETE** подзапросы также применяются как часть условия. Так, удалим все заказы на Galaxy S8:

|  |  |
| --- | --- |
|  | DELETE FROM Orders  WHERE ProductId=(SELECT Id FROM Products WHERE ProductName='Galaxy S8'); |

**Агрегатные функции**

Агрегатные функции вычисляют некоторые скалярные значения в наборе строк. В MySQL есть следующие агрегатные функции:

* AVG: вычисляет среднее значение
* SUM: вычисляет сумму значений
* MIN: вычисляет наименьшее значение
* MAX: вычисляет наибольшее значение
* COUNT: вычисляет количество строк в запросе

Все агрегатные функции принимают в качестве параметра выражение, которое представляет критерий для определения значений. Зачастую, в качестве выражения выступает название столбца, над значениями которого надо проводить вычисления.

Выражения в функциях AVG и SUM должно представлять числовое значение (например, столбец, который хранит числовые значения). Выражение в функциях MIN, MAX и COUNT может представлять числовое или строковое значение или дату.

Все агрегатные функции **за исключением COUNT(\*)** игнорируют значения NULL.

**Avg**

Функция Avg возвращает среднее значение на диапазоне значений столбца таблицы.

Например, пусть есть следующая таблица товаров Products:

|  |  |
| --- | --- |
|  | CREATE TABLE Products  (      Id INT AUTO\_INCREMENT PRIMARY KEY,      ProductName VARCHAR(30) NOT NULL,      Manufacturer VARCHAR(20) NOT NULL,      ProductCount INT DEFAULT 0,      Price DECIMAL NOT NULL  );    INSERT INTO Products(ProductName, Manufacturer, ProductCount, Price)  VALUES  ('iPhone X', 'Apple', 3, 76000),  ('iPhone 8', 'Apple', 2, 51000),  ('iPhone 7', 'Apple', 5, 32000),  ('Galaxy S9', 'Samsung', 2, 56000),  ('Galaxy S8', 'Samsung', 1, 46000),  ('Honor 10', 'Huawei', 5, 28000),  ('Nokia 8', 'HMD Global', 6, 38000) |

Найдем среднюю цену товаров из базы данных:

|  |  |
| --- | --- |
|  | SELECT AVG(Price) AS Average\_Price FROM Products |

Для поиска среднего значения в качестве выражения в функцию передается столбец Price. Для получаемого значения устанавливается псевдоним Average\_Price, хотя в принципе устанавливать псевдоним необязательно.

На этапе выборки можно применять фильтрацию. Например, найдем среднюю цену для товаров определенного производителя:

|  |  |
| --- | --- |
|  | SELECT AVG(Price) FROM Products  WHERE Manufacturer='Apple' |

Также можно находить среднее значение для более сложных выражений. Например, найдем среднюю сумму всех товаров, учитывая их количество:

|  |  |
| --- | --- |
|  | SELECT AVG(Price \* ProductCount) FROM Products |
|  |  |

**Count**

Функция Count вычисляет количество строк в выборке. Есть две формы этой функции. Первая форма COUNT(\*) подсчитывает число строк в выборке:

|  |  |
| --- | --- |
|  | SELECT COUNT(\*) FROM Products |

Вторая форма функции вычисляет количество строк по определенному столбцу, при этом строки со значениями NULL игнорируются:

|  |  |
| --- | --- |
|  | SELECT COUNT(Manufacturer) FROM Products |
|  |  |

**Min и Max**

Функции Min и Max вычисляют минимальное и максимальное значение по столбцу соответственно. Например, найдем минимальную цену среди товаров:

|  |  |
| --- | --- |
|  | SELECT MIN(Price), MAX(Price) FROM Products |

Данные функции также игнорируют значения NULL и не учитывают их при подсчете.

**Sum**

Функция Sum вычисляет сумму значений столбца. Например, подсчитаем общее количество товаров:

|  |  |
| --- | --- |
|  | SELECT SUM(ProductCount) FROM Products |
|  |  |

Также вместо имени столбца может передаваться вычисляемое выражение. Например, найдем общую стоимость всех имеющихся товаров:

|  |  |
| --- | --- |
|  | SELECT SUM(ProductCount \* Price) FROM Products |
|  |  |

**All и Distinct**

По умолчанию все вышеперечисленных пять функций учитывают все строки выборки для вычисления результата. Но выборка может содержать повторяющие значения. Если необходимо выполнить вычисления только над уникальными значениями, исключив из набора значений повторяющиеся данные, то для этого применяется оператор DISTINCT.

|  |  |
| --- | --- |
|  | SELECT COUNT(DISTINCT Manufacturer) FROM Products |
|  |  |

По умолчанию вместо DISTINCT применяется оператор ALL, который выбирает все строки:

|  |  |
| --- | --- |
|  | SELECT COUNT(ALL Manufacturer) FROM Products |
|  |  |

В данном случае мы видим, что производители могут повторяться в таблице, так как некоторые товары могут иметь одних и тех же производителей. Поэтому чтобы подсчитать количество уникальных производителей, необходимо использовать оператор DISTINCT.

Так как этот оператор неявно подразумевается при отсутствии DISTINCT, то его можно не указывать.

**Комбинирование функций**

Объединим применение нескольких функций:

|  |  |
| --- | --- |
|  | SELECT COUNT(\*) AS ProdCount,         SUM(ProductCount) AS TotalCount,         MIN(Price) AS MinPrice,         MAX(Price) AS MaxPrice,         AVG(Price) AS AvgPrice  FROM Products |

**Union**

Оператор UNION позволяет обединить две однотипных выборки. Эти выборки могут быть из разных таблиц или из одной и той же таблицы. Формальный синтаксис объединения:

|  |  |
| --- | --- |
|  | SELECT\_выражение1  UNION [ALL] SELECT\_выражение2  [UNION [ALL] SELECT\_выражениеN] |
|  |  |

Например, пусть в базе данных будут две отдельные таблицы для клиентов банка (таблица Customers) и для сотрудников банка (таблица Employees):

|  |  |
| --- | --- |
|  | CREATE TABLE Customers  (      Id INT AUTO\_INCREMENT PRIMARY KEY,      FirstName VARCHAR(20) NOT NULL,      LastName VARCHAR(20) NOT NULL,      AccountSum DECIMAL  );  CREATE TABLE Employees  (      Id INT AUTO\_INCREMENT PRIMARY KEY,      FirstName VARCHAR(20) NOT NULL,      LastName VARCHAR(20) NOT NULL  );    INSERT INTO Customers(FirstName, LastName, AccountSum)  VALUES  ('Tom', 'Smith', 2000),  ('Sam', 'Brown', 3000),  ('Mark', 'Adams', 2500),  ('Paul', 'Ins', 4200),  ('John', 'Smith', 2800),  ('Tim', 'Cook', 2800);    INSERT INTO Employees(FirstName, LastName)  VALUES  ('Homer', 'Simpson'),  ('Tom', 'Smith'),  ('Mark', 'Adams'),  ('Nick', 'Svensson'); |

Здесь мы можем заметить, что обе таблицы, несмотря на наличие различных данных, могут характеризоваться двумя общими атрибутами - именем (FirstName) и фамилией (LastName). Выберем сразу всех клиентов банка и его сотрудников из обеих таблиц:

|  |  |
| --- | --- |
|  | SELECT FirstName, LastName  FROM Customers  UNION SELECT FirstName, LastName FROM Employees; |
|  |  |

Здесь из первой таблицы выбираются два значения - имя и фамилия клиента. Из второй таблицы Employees также выбираются два значения - имя и фамилия сотрудников. То есть при объединении количество выбираемых столбцов и их тип совпадают для обеих выборок.

При этом названия столбцов объединенной выборки будут совпадать с названия столбцов первой выборки. И если мы захотим при этом еще произвести сортировку, то в выражениях ORDER BY необходимо ориентироваться именно на названия столбцов первой выборки:

|  |  |
| --- | --- |
|  | SELECT FirstName AS FName, LastName  FROM Customers  UNION SELECT FirstName, LastName  FROM Employees  ORDER BY FName DESC; |

В данном случае каждая выборка имеет по столбцу FName из первой выборки. Тем не менее при сортировке будет учитываться и значение столбца FirstName из второй выборки:

Если же в одной выборке больше столбцов, чем в другой, то они не смогут быть объединены. Например, в следующем случае объединение завершится с ошибкой:

|  |  |
| --- | --- |
|  | SELECT FirstName, LastName, AccountSum  FROM Customers  UNION SELECT FirstName, LastName  FROM Employees; |

Если оба объединяемых набора содержат в строках идентичные значения, то при объединении повторяющиеся строки удаляются. Например, в случае с таблицами Customers и Employees сотрудники банка могут быть одновременно его клиентами и содержаться в обеих таблицах. При объединении в примерах выше всех дублирующиеся строки удалялись. Если же необходимо при объединении сохранить все, в том числе повторяющиеся строки, то для этого необходимо использовать оператор ALL:

|  |  |
| --- | --- |
|  | SELECT FirstName, LastName  FROM Customers  UNION ALL SELECT FirstName, LastName  FROM Employees  ORDER BY FirstName; |

Объединять выборки можно и из одной и той же таблицы. Например, в зависимости от суммы на счете клиента нам надо начислять ему определенные проценты:

|  |  |
| --- | --- |
|  | SELECT FirstName, LastName, AccountSum + AccountSum \* 0.1 AS TotalSum  FROM Customers WHERE AccountSum < 3000  UNION SELECT FirstName, LastName, AccountSum + AccountSum \* 0.3 AS TotalSum  FROM Customers WHERE AccountSum >= 3000; |

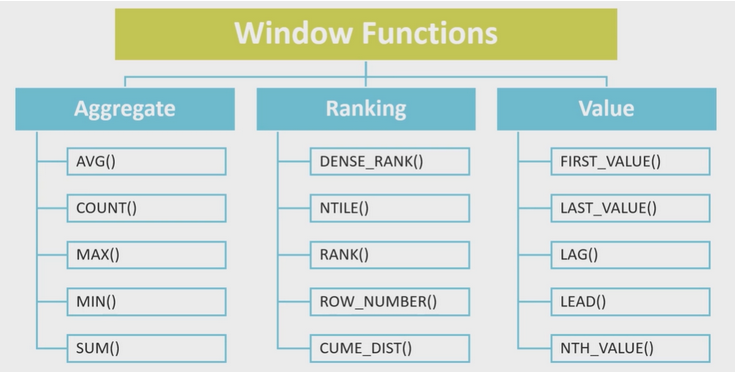
В данном случае если сумма меньше 3000, то начисляются проценты в размере 10% от суммы на счете. Если на счете больше 3000, то проценты увеличиваются до 30%.

**Анал функции**

Аналитические функции вычисляют агрегированное значение, основанное на группе строк. ОТЛИЧИЕ агрегатной от аналитической функции - аналитическая функция возвращает не 1 строку результата для группы, а множество строк для группы.  
ИМЯ\_ФУНКЦИИ(<аргумент>,< аргумент >, . . . )  
OVER  
(<конструкция\_фрагментации> <конструкция\_упорядочения> <конструкция\_окна>)  
**OVER** — ключевое слово, идентифицирующее эту функцию как аналитическую.Конструкция после ключевого слова OVER описывает срез данных, по которому будет вычисляться аналитическая функция.  
PARTITION BY логически разбивает результирующее множество на группы по критериям, задаваемым выражениями секционирования. Аналитические функции применяются к каждой группе независимо, — для каждой новой группы они сбрасываются. Если не указать конструкцию секционирования, все результирующее множество считается одной группой.  
ORDER BY задает критерий сортировки данных в каждой группе (в каждой секции)  
SELECT empno, deptno, sal,  
 AVG(sal) OVER (PARTITION BY deptno) AS avg\_dept\_sal  
FROM emp;

Множество оконных функций можно разделять на 3 класса:

* Агрегирующие (Aggregate)
* Ранжирующие (Ranking)
* Функции смещения (Value)



В ранжирующих функция под ключевым словом OVER обязательным идет указание условия ORDER BY, по которому будет происходить сортировка ранжирования.

**ROW\_NUMBER()** - функция вычисляет последовательность ранг (порядковый номер) строк внутри партиции, НЕЗАВИСИМО от того, есть ли в строках повторяющиеся значения или нет.

**RANK()** - функция вычисляет ранг каждой строки внутри партиции. Если есть повторяющиеся значения, функция возвращает одинаковый ранг для таких строчек, пропуская при этом следующий числовой ранг.

**DENSE\_RANK()** - то же самое что и RANK, только в случае одинаковых значений DENSE\_RANK не пропускает следующий числовой ранг, а идет последовательно.

Для SQL пустые NULL значения будут определяться одинаковым рангом

**Рекурсия**

Рекурсивны SQL запросы являются одним из способов решения проблемы дерева и других проблем, требующих рекурсивную обработку.

Рекурсивные запросы используют довольно редко, прежде всего, из-за их сложного и непонятного синтаксиса:

with [recursive] <имя\_алиаса\_запроса> [ (<список столбцов>) ]  
as (<запрос>)   
<основной запрос>

Проще разобрать на примере. Предположим, есть таблица:

create table tree\_sample (  
  id integer not null primary key,   
  id\_parent integer foreign key references tree\_sample (id),   
  nm varchar(31) )

id – идентификатор  
id\_parent – ссылка на родитель  
nm – название.  
  
Для вывода дерева:

with recursive tree (nm, id, level, pathstr)  
as (select nm, id, 0, cast('' as text)   
   from tree\_sample  
   where id\_parent is null   
union all  
   select tree\_sample.nm, tree\_sample.id, t.level + 1, tree.pathstr + tree\_sample.nm  
   from tree\_sample   
     inner join tree on tree.id = tree\_sample.id\_parent)   
select id, space( level ) + nm as nm   
from tree   
order by pathstr

Этот пример выведет дерево по таблице с отступами. Первый запрос из tree\_sample этот запрос выдаст все корни дерева. Второй запрос соединяет между собой таблицу tree\_sample и tree, которая определяется этим же запросом. Этот запрос дополняет таблицу узлами дерева.   
  
Сначала выполняется первый запрос. Потом к его результатам добавляются результаты второго запроса, где данные таблица tree – это результат первого запроса. Затем снова выполняется второй запрос, но данные таблицы tree – это уже результат предыдущего выполнения второго запроса. И так далее. На самом деле база данных работает не совсем так, но результат будет таким же, как результат работы описанного алгоритма.  
  
После этого данные этой таблицы можно использовать в основном запросе как обычно.

**НЕДОЧЁТЫ**

**1**

Столбец должен быть указан в предложении FROM инструкции SELECT, но не обязательно должен присутствовать в списке SELECT. Каждый столбец таблицы или представления в любом нестатистическом выражении в списке <select> должен быть включен в список GROUP BY:

Следующие инструкции являются допустимыми.

SQL

SELECT ColumnA, ColumnB FROM T GROUP BY ColumnA, ColumnB;

SELECT ColumnA + ColumnB FROM T GROUP BY ColumnA, ColumnB;

SELECT ColumnA + ColumnB FROM T GROUP BY ColumnA + ColumnB;

SELECT ColumnA + ColumnB + constant FROM T GROUP BY ColumnA, ColumnB;

Следующие инструкции не являются допустимыми.

SQL

SELECT ColumnA, ColumnB FROM T GROUP BY ColumnA + ColumnB;

SELECT ColumnA + constant + ColumnB FROM T GROUP BY ColumnA + ColumnB;

Выражение столбца не может содержать:

* псевдоним столбца, определенный в списке SELECT. Выражение может использовать псевдоним столбца для производной таблицы, определенной в предложении FROM.
* столбец типа **text**, **ntext**, или **image**. Однако столбец типа text, ntext или image можно использовать как аргумент для функции, возвращающей значение допустимого типа данных. Например, выражение может использовать SUBSTRING() и CAST(). Это также относится к выражениям в предложении HAVING.
* методы типа данных xml. Сюда может входить определяемой пользователем функции, которая использует методы типа данных xml. Сюда может входить вычисляемый столбец, который использует методы типа данных xml.
* вложенный запрос. Возвращается ошибка 144.
* столбец из индексированного представления.

2

**Common Table Expression** (CTE) — результаты запроса, которые можно использовать множество раз в других запросах. То есть, запросом мы достаем данные, и они помещаются в пространство памяти, аналогично временному представлению, которое физически не сохраняется в виде объектов. Далее мы работаем с получившейся конструкцией как с таблицей, используя такие конструкции как select, update, insert и delete.

Выведем количество сотрудников, устроившихся на работу, в разбивке по годам:

WITH TABLE\_CTE (FIO, ID\_DEPART, BEGIN\_YEAR) AS ( SELECT FIO, ID\_DEPART, YEAR(DATE\_BEGIN) as BEGIN\_YEAR FROM EMPL ) SELECT count(FIO) as COUNT\_SOTR, BEGIN\_YEAR FROM TABLE\_CTE group by BEGIN\_YEAR

Еще обобщенное табличное выражение можно составить из результатов нескольких запросов. Последний результирующий запрос обращается к данным нижнего CTE(TABLE\_CTE2), но может и к любому из них:

WITH TABLE\_CTE1(FIO, YEAR\_EMPL) AS ( SELECT FIO, YEAR(DATE\_BEGIN) as YEAR\_EMPL FROM EMPL ), TABLE\_CTE2 (COUNT\_FIO, YEAR\_EMPL) AS ( SELECT count(FIO) as COUNT\_FIO, YEAR\_EMPL FROM TABLE\_CTE1 group by YEAR\_EMPL ) SELECT \* FROM TABLE\_CTE2

Основные **способы** использования:

* для улучшения читаемости запроса в случае сложных запросов (разительно уменьшают размер кода);
* в случаях, когда нужно много раз обращаться к одним и тем же таблицам/выборкам из таблиц;
* для создания представлений (VIEW) в части select;
* для написания рекурсивных запросов.

**Отличия** от вложенного запроса:

* вложенный запрос повторяется для каждой строки из нашей выборки, что повышает стоимость выполнения запроса.

**Отличия** от временной таблицы:

* заполнение временной таблицы при больших объемах создает нагрузку на диск;
* исполнение запросов с использованием временной таблицы увеличивает время их выполнения из-за места хранения данного типа таблиц (tempdb).

3

Когда вы используете подзапросы в SQL, вы можете обратиться к внут-

реннему запросу таблицы в предложении внешнего запроса FROM , сформи-

ровав - соотнесенный подзапрос. Когда вы делаете это, подзапрос выпол-

няется неоднократно, по одному разу для каждой строки таблицы основно-

го запроса. Соотнесенный подзапрос - один из большого количества тон-

ких понятий в SQL из-за сложности в его оценке. Если вы сумеете овла-

деть им, вы найдете что он очень мощный, потому что может выполнять

сложные функции с помощью очень лаконичных указаний.

Например, имеется один способ найти всех заказчиков в порядках на

3-е Октября ( вывод показывается в Рисунке 11.1 ):

SELECT \*

FROM Customers outer

WHERE 10/03/1990 IN

( SELECT odate

FROM Orders inner

WHERE outer.cnum = inner.cnum );

КАК РАБОТАЕТ СООТНЕСЕННЫЙ ПОДЗАПРОС

В вышеупомянутом примере, "внутренний"(inner) и "внешний"(outer),

это псевдонимы, подобно обсужденным в Главе 9. Мы выбрали эти имена

для большей ясности; они отсылают к значениям внутренних и внешних

запросов, соответственно. Так как значение в поле cnum внешнего запро-

са меняется, внутренний запрос должен выполняться отдельно для каждой

строки внешнего запроса. Строка внешнего запроса для которого внутрен-

=============== SQL Execution Log ============

| |

| SELECT \* |

| FROM Customers outer |

| WHERE 10/03/1990 IN |

| (SELECT odate |

| FROM Orders inner |

| WHERE outer.cnum = inner.cnum); |

| ============================================= |

| cnum cname city rating snum |

| ----- -------- ---- ------ ----- |

| 2001 Hoffman London 100 1001 |

| 2003 Liu San Jose 200 1002 |

| 2008 Cisneros San Jose 300 1007 |

| 2007 Pereira Rome 100 1004 |

=============================================

Рисунок 11.1: Использование соотнесенного подзапроса

ний запрос каждый раз будет выполнен, называется - текущей стро-

кой-кандидатом. Следовательно, процедура оценки выполняемой соотнесен-

ным подзапросом - это:

1. Выбрать строку из таблицы именованной в внешнем запросе. Это будет

текущая строка-кандидат.

2. Сохранить значения из этой строки-кандидата в псевдониме с именем в

предложении FROM внешнего запроса.

3. Выполнить подзапрос. Везде, где псевдоним данный для внешнего зап-

роса найден ( в этом случае "внешний" ), использовать значение для

текущей строки-кандидата. Использование значения из строки-кандида-

та внешнего запроса в подзапросе называется - внешней ссылкой.

4. Оценить предикат внешнего запроса на основе результатов подзапроса

выполняемого в шаге 3. Он определяеть - выбирается ли строка-канди-

дат для вывода.

5. Повторить процедуру для следующей строки-кандидата таблицы, и так

далее пока все строки таблицы не будут проверены.

В вышеупомянутом примере, SQL осуществляет следующую процедуру:

1. Он выбирает строку Hoffman из таблицы Заказчиков.

2. Сохраняет эту строку как текущую строку-кандидат под псевдонимом -

"внешним".

3. Затем он выполняет подзапрос. Подзапрос просматривает всю таблицу

Порядков чтобы найти строки где значение cnum поле - такое же как

значение outer.cnum, которое в настоящее время равно 2001, - поле

cnum строки Hoffmanа. Затем он извлекает поле odate из каждой стро-

ки таблицы Порядков для которой это верно, и формирует набор значе-

ний поля odate.

4. Получив набор всех значений поля odate, для поля cnum = 2001, он

проверяет предикат основного запроса чтобы видеть имеется ли значе-

ние на 3 Октября в этом наборе. Если это так(а это так), то он вы-

бирает строку Hoffmanа для вывода ее из основного запроса.

5. Он повторяет всю процедуру, используя строку Giovanni как стро-

ку-кандидата, и затем сохраняет повторно пока каждая строка таблицы

Заказчиков не будет проверена.

Как вы можете видеть, вычисления которые SQL выполняет с помощью

этих простых инструкций - это полный комплекс. Конечно, вы могли бы

решить ту же самую проблему используя обьединение, следующего вида (

вывод для этого запроса показывается в Рисунке 11.2 ):

SELECT \*

FROM Customers first, Orders second

WHERE first.cnum = second.cnum

AND second.odate = 10/03/1990;

Обратите внимание что Cisneros был выбран дважды, по одному разу для

каждого порядка который он имел для данной даты. Мы могли бы устранить

это используя SELECT DISTINCT вместо просто SELECT. Но это необяза-

тельно в варианте подзапроса. Оператор IN, используемый в варианте

подзапроса, не делает никакого различия между значениями которые выби-

раются подзапросом один раз и значениями которые выбираются неоднок-

ратно. Следовательно DISTINCT необязателен.

=============== SQL Execution Log ============

| |

| SELECT \* |

| FROM Customers first, Orders second |

| WHERE first.cnum = second.cnum |

| (SELECT COUNT (\*) |

| FROM Customers |

| WHERE snum = main.snum; |

| ============================================= |

| cnum cname |

| ----- -------- |

| 1001 Peel |

| 1002 Serres |

=============================================

Рисунок 11. 2 Использование обьединения вместо соотнесенного

подзапроса

Предположим что мы хотим видеть имена и номера всех продавцов кото-

рые имеют более одного заказчика. Следующий запрос выполнит это для

вас ( вывод показывается в Рисунке 11.3 ):

SELECT snum, sname

FROM Salespeople main

WHERE 1 <

( SELECT COUNT (\*)

FROM Customers

WHERE snum = main.snum );

Обратите внимание что предложение FROM подзапроса в этом примере не

использует псевдоним. При отсутствии имени таблицы или префикса псев-

донима, SQL может для начала принять, что любое поле выводится из таб-

лицы с именем указанным в предложении FROM текущего запроса. Если поле

с этим именем отсутствует( в нашем случае - snum ) в той таблице, SQL

будет проверять внешние запросы. Именно поэтому, префикс имени таблицы

обычно необходим в соотнесенных подзапросах - для отмены этого предпо-

ложения. Псевдонимы также часто запрашиваются чтобы давать вам возмож-

ность ссылаться к той же самой таблице во внутреннем и внешнем запросе

без какой-либо неоднозначности.

=============== SQL Execution Log ============

| |

| SELECT snum sname |

| FROM Salespeople main |

| WHERE 1 < |

| AND second.odate = 10/03/1990; |

| ============================================= |

| cnum cname city rating snum |

| ----- -------- ---- ------ ----- |

| 2001 Hoffman London 100 1001 |

| 2003 Liu San Jose 200 1002 |

| 2008 Cisneros San Jose 300 1007 |

| 2007 Pereira Rome 100 1004 |

=============================================

Рисунок 11.3: Нахождение продавцов с многочислеными заказчиками

ИСПОЛЬЗОВАНИЕ СООТНЕСЕННЫХ ПОДЗАПРОСОВ

ДЛЯ НАХОЖДЕНИЯ ОШИБОК

Иногда полезно выполнять запросы которые разработаны специально так

чтобы находить ошибки. Это всегда возможно при дефектной информации

которую можно ввести в вашу базу данных, и, если она введена, бывает

трудно ее определить. Следующий запрос не должен производить никакого

вывода. Он просматривает таблицу Порядков чтобы видеть совпадают ли

поля snum и cnum в каждой строке таблицы Заказчиков и выводит каждую

строку где этого совпадения нет. Другими словами, запрос выясняет, тот

ли продавец кредитовал каждую продажу ( он воспринимает поле cnum, как

первичный ключ таблицы Заказчиков, который не будет иметь никаких

двойных значений в этой таблице ).

SELECT \*

FROM Orders main

WHERE NOT snum =

( SELECT snum

FROM Customers

WHERE cnum = main.cnum );

При использовании механизма справочной целостности ( обсужденного в

Главе 19 ), вы можете быть гарантированы от некоторых ошибок такого

вида. Этот механизм не всегда доступен, хотя его использование жела-

тельно во всех случаях, причем поиск ошибки запроса описанный выше,

может быть еще полезнее.

СРАВНЕНИЕ ТАБЛИЦЫ С СОБОЙ

Вы можете также использовать соотнесенный подзапрос основанный на

той же самой таблице что и основной запрос. Это даст вам возможность

извлечть определенные сложные формы произведенной информации. Напри-

мер, мы можем найти все порядки со значениями сумм приобретений выше

среднего для их заказчиков ( вывод показан в Рисунке 11.4 ):

SELECT \*

FROM Orders outer

WHERE amt >

( SELECT AVG amt

FROM Orders inter

WHERE inner.cnum = outer.cnum );

=============== SQL Execution Log ==============

| |

| SELECT \* |

| FROM Orders outer |

| WHERE amt > |

| (SELECT AVG (amt) |

| FROM Orders inner |

| WHERE inner.cnum = outer.cnum |

| =============================================== |

| onum amt odate cnum snum |

| ----- -------- ---------- ----- ------ |

| 3006 1098.19 10/03/1990 2008 1007 |

| 3010 1309.00 10/06/1990 2004 1002 |

| 3011 9891.88 10/06/1990 2006 1001 |

================================================

Рисунок 11.4: Соотнесение таблицы с собой

Конечно, в нашей маленькой типовой таблице, где большиство заказчи-

ков имеют только один порядок, большинство значений являются одновре-

менно средними и следовательно не выбираются. Давайте введем команду

другим способом ( вывод показывается в Рисунке 11.5 ):

SELECT \*

FROM Orders outer

WHERE amt > =

( SELECT AVG (amt)

FROM Orders inner

WHERE inner.cnum = outer.cnum );

=============== SQL Execution Log ==============

| |

| SELECT \* |

| FROM Orders outer |

| WHERE amt > = |

| (SELECT AVG (amt) |

| FROM Orders inner |

| WHERE inner.cnum = outer.cnum); |

| =============================================== |

| onum amt odate cnum snum |

| ----- -------- ---------- ----- ------ |

| 3003 767.19 10/03/1990 2001 1001 |

| 3002 1900.10 10/03/1990 2007 1004 |

| 3005 5160.45 10/03/1990 2003 1002 |

| 3006 1098.19 10/03/1990 2008 1007 |

| 3009 1713.23 10/04/1990 2002 1003 |

| 3010 1309.95 10/06/1990 2004 1002 |

| 3011 9891.88 10/06/1990 2006 1001 |

================================================

Рисунок 11.5: Выбераются порядки которые > = средней сумме приобрете-

ний для их заказчиков.

Различие, конечно, в том, что реляционный оператор основного преди-

ката включает значения которые равняются среднему ( что обычно означа-

ет что они - единственые порядки для данных заказчиков ).

СООТНЕСЕННЫЕ ПОДЗАПРОСЫ В ПРЕДЛОЖЕНИИ HAVING

Также как предложение HAVING может брать подзапросы, он может брать

и соотнесенные подзапросы. Когда вы используете соотнесенный подзапрос

в предложении HAVING, вы должны ограничивать внешние ссылки к позициям

которые могли бы непосредственно использоваться в самом предложении

HAVING. Вы можете вспомнить из Главы 6 что предложение HAVING может

использовать только агрегатные функции которые указаны в их предложе-

нии SELECT или поля используемые в их предложении GROUP BY. Они явля-

ются только внешними ссылками, которые вы можете делать. Все это пото-

му, что предикат предложения HAVING оценивается для каждой группы из

внешнего запроса, а не для каждой строки. Следовательно, подзапрос бу-

дет выполняться один раз для каждой группы выведеной из внешнего зап-

роса, а не для каждой строки.

Предположим что вы хотите суммировать значения сумм приобретений по-

купок из таблицы Порядков, сгруппировав их по датам, удалив все даты

где бы SUM не был по крайней мере на 2000.00 выше максимальной ( MAX )

суммы:

SELECT odate, SUM (amt)

FROM Orders a

GROUP BY odate

HAVING SUM (amt) >

( SELECT 2000.00 + MAX (amt)

FROM Orders b

WHERE a.odate = b.odate );

Подзапрос вычисляет значение MAX для всех строк с той же самой датой

что и у текущей агрегатной группы основного запроса. Это должно быть

выполнено, как и ранее, с испошльзованием предложения WHERE. Сам под-

запрос не должен использовать предложения GROUP BY или HAVING.

СООТНЕСЕННЫЕ ПОДЗАПРОСЫ И ОБЬЕДИНЕНИЯ

Как вы и могли предположить, соотнесенные подзапросы по природе

близки к обьединениям - они оба включают проверку каждой строки одной

таблицы с каждой строкой другой ( или псевдонимом из той же ) таблицы.

Вы найдете что большинство операций которые могут выполняться с одним

из них будут также работать и с другим.

Однако имеется различие в прикладной программе между ними, такое как

вышеупомянутая потребность в использовании DISTINCT с обьединением и

его необязательность с подзапросом. Имеются также некоторые вещи кото-

рые каждый может делать так, как этого не может другой. Подзапросы,

например, могут использовать агрегатную функцию в предикате, делая

возможным выполнение операций типа нашего предыдущего примера в кото-

ром мы извлекли порядки усредненные для их заказчиков. Обьединения, с

другой стороны, могут выводить строки из обеих сравниваемых таблиц, в

то время как вывод подзапросов используется только в предикатах внеш-

них запросов. Как правило, форма запроса которая кажется наиболее ин-

туитивной будет вероятно лучшей в использовании, но при этом хорошо бы

знать обе техники для тех ситуаций когда та или иная могут не рабо-

тать.

|  |  |
| --- | --- |
| Коррелирующие подзапросы |  |
|  | Коррелирующие подзапросы позволяют иногда очень кратко написать запросы, которые могут выглядеть весьма громоздко при использовании других языковых средств. Напомним, что коррелирующий подзапрос — это подзапрос, который содержит ссылку на столбцы из включающего его запроса (назовем его основным). Таким образом, коррелирующий подзапрос будет выполняться для каждой строки основного запроса, так как значения столбцов основного запроса будут меняться. |

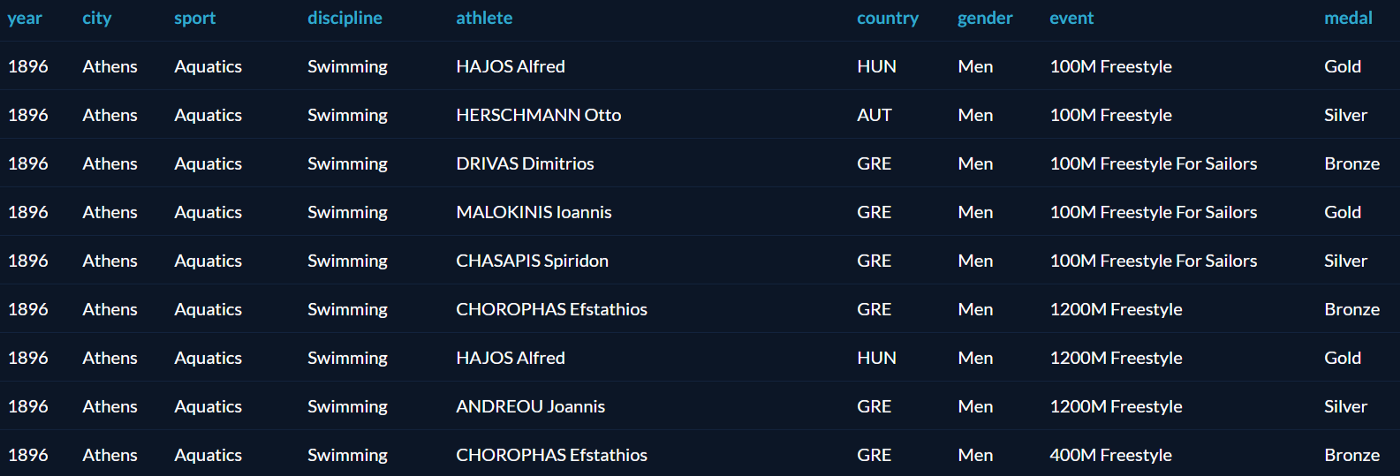
## Почему не GROUP BY и не JOIN

Сразу проясним, что оконные функции — это не то же самое, что GROUP BY. Они не уменьшают количество строк, а возвращают столько же значений, сколько получили на вход. Во-вторых, в отличие от GROUP BY, OVER может обращаться к другим строкам. И в-третьих, они могут считать скользящие средние и кумулятивные суммы.

Примечание Оконные функции не изменяют выборку, а только добавляют некоторую дополнительную информацию о ней. Для простоты понимания можно считать, что SQL сначала выполняет весь запрос (кроме сортировки и limit), а уже потом считает значения окна.

Окей, с GROUP BY разобрались. Но в SQL практически всегда можно пойти несколькими путями. К примеру, может возникнуть желание использовать подзапросы или JOIN. Конечно, JOIN по производительности предпочтительнее подзапросов, а производительность конструкций JOIN и OVER окажется одинаковой. Но OVER даёт больше свободы, чем жёсткий JOIN. Да и объём кода в итоге окажется гораздо меньше.

## Для начала

Оконные функции начинаются с оператора OVER и настраиваются с помощью трёх других операторов: PARTITION BY, ORDER BY и ROWS. Про ORDER BY, PARTITION BY и его вспомогательные операторы LAG, LEAD, RANK мы расскажем подробнее.  
Все примеры будут основаны на датасете олимпийских медалистов от Datacamp. Таблица называется summer\_medals и содержит результаты Олимпиад с 1896 по 2010: [](https://media.tproger.ru/uploads/2020/06/window_functions_6.png)

## ROW\_NUMBER и ORDER BY

Как уже говорилось выше, оператор OVER создаёт оконную функцию. Начнём с простой функции ROW\_NUMBER, которая присваивает номер каждой выбранной записи:

SELECT

athlete,

event,

ROW\_NUMBER() OVER() AS row\_number

FROM Summer\_Medals

ORDER BY row\_number ASC;

[](https://media.tproger.ru/uploads/2020/06/window_functions_5.png)  
Каждая пара «спортсмен — вид спорта» получила номер, причём к этим номерам можно обращаться по имени row\_number.  
ROW\_NUMBER можно объединить с ORDER BY, чтобы определить, в каком порядке строки будут нумероваться. Выберем с помощью DISTINCT все имеющиеся виды спорта и пронумеруем их в алфавитном порядке:

SELECT

sport,

ROW\_NUMBER() OVER(ORDER BY sport ASC) AS Row\_N

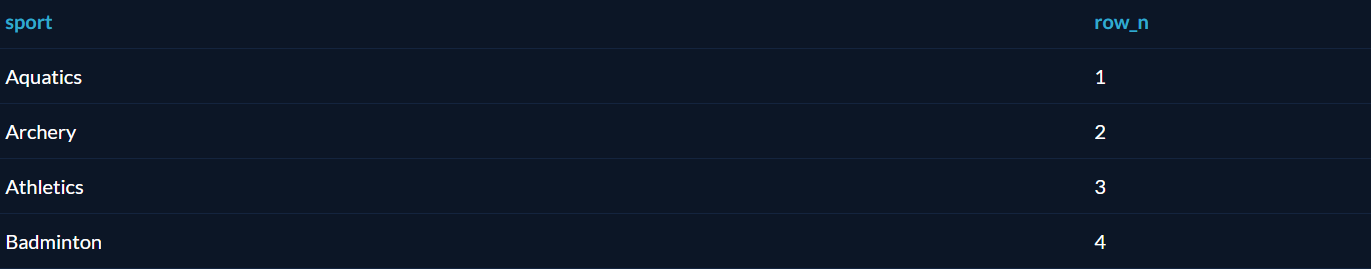
FROM (

SELECT DISTINCT sport

FROM Summer\_Medals

) AS sports

ORDER BY sport ASC;

[](https://media.tproger.ru/uploads/2020/06/window_functions_4.png)

## PARTITION BY и LAG, LEAD и RANK

PARTITION BY позволяет сгруппировать строки по значению определённого столбца. Это полезно, если данные логически делятся на какие-то категории и нужно что-то сделать с данной строкой с учётом других строк той же группы (скажем, сравнить теннисиста с остальными теннисистами, но не с бегунами или пловцами). Этот оператор работает только с оконными функциями типа LAG, LEAD, RANK и т. д.

### LAG

Функция LAG берёт строку и возвращает ту, которая шла перед ней. Например, мы хотим найти всех олимпийских чемпионов по теннису (мужчин и женщин отдельно), начиная с 2004 года, и для каждого из них выяснить, кто был предыдущим чемпионом.  
Решение этой задачи требует нескольких шагов. Сначала надо создать табличное выражение, которое сохранит результат запроса «чемпионы по теннису с 2004 года» как временную именованную структуру для дальнейшего анализа. А затем разделить их по полу и выбрать предыдущего чемпиона с помощью LAG:

-- Табличное выражение ищет теннисных чемпионов и выбирает нужные столбцы

WITH Tennis\_Gold AS (

SELECT

Athlete,

Gender,

Year,

Country

FROM

Summer\_Medals

WHERE

Year >= 2004 AND

Sport = 'Tennis' AND

event = 'Singles' AND

Medal = 'Gold')

-- Оконная функция разделяет по полу и берёт чемпиона из предыдущей строки

SELECT

Athlete as Champion,

Gender,

Year,

LAG(Athlete) OVER (PARTITION BY gender

ORDER BY Year ASC) AS Last\_Champion

FROM Tennis\_Gold

ORDER BY Gender ASC, Year ASC;

[](https://media.tproger.ru/uploads/2020/06/window_functions_3.png)  
Функция PARTITION BY в таблице вернула сначала всех мужчин, потом всех женщин. Для победителей 2008 и 2012 года приведён предыдущий чемпион; так как данные есть только за 3 олимпиады, у чемпионов 2004 года нет предшественников, поэтому в соответствующих полях стоит null.

### LEAD

Функция LEAD похожа на LAG, но вместо предыдущей строки возвращает следующую. Можно узнать, кто стал следующим чемпионом после того или иного спортсмена:

-- Табличное выражение ищет теннисных чемпионов и выбирает нужные столбцы

WITH Tennis\_Gold AS (

SELECT

Athlete,

Gender,

Year,

Country

FROM

Summer\_Medals

WHERE

Year >= 2004 AND

Sport = 'Tennis' AND

event = 'Singles' AND

Medal = 'Gold')

-- Оконная функция разделяет по полу и берёт чемпиона из следующей строки

SELECT

Athlete as Champion,

Gender,

Year,

LEAD(Athlete) OVER (PARTITION BY gender

ORDER BY Year ASC) AS Future\_Champion

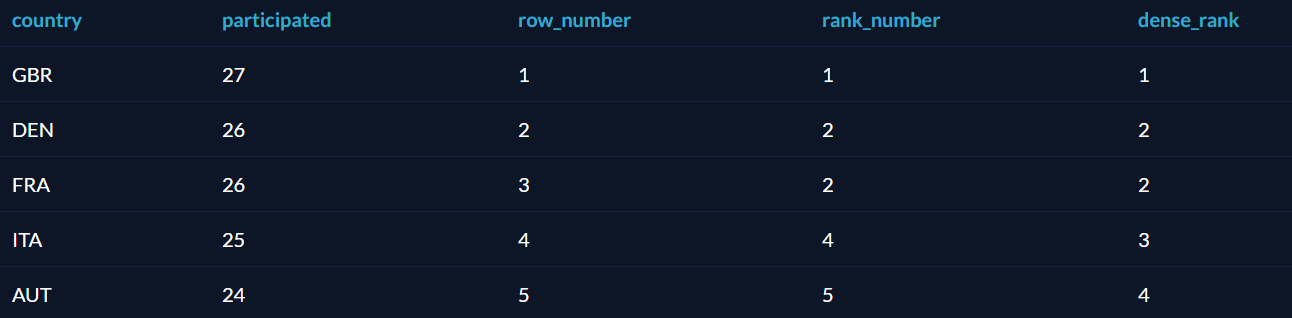
FROM Tennis\_Gold

ORDER BY Gender ASC, Year ASC;

[](https://media.tproger.ru/uploads/2020/06/window_functions_2.png)

### RANK

Оператор RANK похож на ROW\_NUMBER, но присваивает одинаковые номера строкам с одинаковыми значениями, а «лишние» номера пропускает. Есть также DENSE\_RANK, который не пропускает номеров. Звучит запутанно, так что проще показать на примере. Вот ранжирование стран по числу олимпиад, в которых они участвовали, разными операторами:

[](https://media.tproger.ru/uploads/2020/06/window_functions_1.png)

* Row\_number — ничего интересного, строки просто пронумерованы по возрастанию.
* Rank\_number — строки ранжированы по возрастанию, но нет номера 3. Вместо этого, 2 строки делят номер 2, а за ними сразу идёт номер 4.
* Dense\_rank — то же самое, что и rank\_number, но номер 3 не пропущен. Номера идут подряд, но зато никто не оказался пятым из пяти.

Вот код:

-- Табличное выражение выбирает страны и считает годы

WITH countries AS (

SELECT

Country,

COUNT(DISTINCT year) AS participated

FROM

Summer\_Medals

WHERE

Country in ('GBR', 'DEN', 'FRA', 'ITA','AUT')

GROUP BY

Country)

-- Разные оконные функции ранжируют страны

SELECT

Country,

participated,

ROW\_NUMBER()

OVER(ORDER BY participated DESC) AS Row\_Number,

RANK()

OVER(ORDER BY participated DESC) AS Rank\_Number,

DENSE\_RANK()

OVER(ORDER BY participated DESC) AS Dense\_Rank

FROM countries

ORDER BY participated DESC;