Бесплатная электронная книга

УЧУСЬ SQL

Free unaffiliated eBook created from
Stack Overflow contributors.

#sql

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sql

It is an unofficial and free SQL ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official SQL.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с SQL

замечания

SQL - это структурированный запрос Язык, используемый для управления данными в системе реляционных баз данных. Различные производители улучшили язык и имеют множество вариантов для языка.

NB: этот тег явно ссылается на **стандарт ISO / ANSI SQL** ; а не какой-либо конкретной реализации этого стандарта.

Версии

Версия	Короткое имя	стандарт	Дата выхода
1986	SQL-86	ANSI X3.135-1986, ISO 9075: 1987	1986-01-01
1989	SQL-89	ANSI X3.135-1989, ISO / IEC 9075: 1989	1989-01-01
1992	SQL-92	ISO / IEC 9075: 1992	1992-01-01
1999	SQL: 1999	ISO / IEC 9075: 1999	1999-12-16
2003	SQL: 2003	ISO / IEC 9075: 2003	2003-12-15
2006	SQL: 2006	ISO / IEC 9075: 2006	2006-06-01
2008	SQL: 2008	ISO / IEC 9075: 2008	2008-07-15
2011	SQL: 2011	ISO / IEC 9075: 2011	2011-12-15
2016	SQL: 2016	ISO / IEC 9075: 2016	2016-12-01

Examples

обзор

Язык структурированных запросов (SQL) - это специальный язык программирования, предназначенный для управления данными, хранящимися в системе реляционных баз данных (RDBMS). SQL-подобные языки также могут использоваться в реляционных системах управления потоками данных (RDSMS) или в базах данных «не только SQL» (NoSQL).

SQL состоит из 3 основных подязыков:

1. Язык определения данных (DDL): создание и изменение структуры базы данных;
2. Язык манипулирования данными (DML): выполнять операции чтения, вставки, обновления и удаления данных базы данных;
3. Язык управления данными (DCL): контроль доступа к данным, хранящимся в базе данных.

Статья в SQL в Википедии

Основными операциями DML являются Create, Read, Update и Delete (CRUD для краткости), которые выполняются операторами `INSERT` , `SELECT` , `UPDATE` и `DELETE` .
Существует также (недавно добавленный) оператор `MERGE` который может выполнять все 3 операции записи (INSERT, UPDATE, DELETE).

Статья CRUD в Википедии

Многие базы данных SQL реализованы как системы клиент / сервер; термин «сервер SQL» описывает такую базу данных.
В то же время Microsoft создает базу данных с именем «SQL Server». Хотя эта база данных говорит на диалекте SQL, информация, относящаяся к этой базе данных, не относится к теме в этом теге, но принадлежит к документации SQL Server .

Прочитайте Начало работы с SQL онлайн: https://riptutorial.com/ru/sql/topic/184/начало-работы-с-sql

глава 2: ALTER TABLE

Вступление

Команда ALTER в SQL используется для изменения столбца / ограничения в таблице

Синтаксис

- ALTER TABLE [имя_таблицы] ADD [имя_столбца] [тип данных]

Examples

Добавить колонку (и)

```
ALTER TABLE Employees
ADD StartingDate date NOT NULL DEFAULT GetDate(),
    DateOfBirth date NULL
```

В приведенном выше утверждении будут добавлены столбцы с именем `StartingDate` которые не могут быть NULL со значением по умолчанию в качестве текущей даты и `DateOfBirth` которые могут быть NULL в таблице Employees .

Колонка падения

```
ALTER TABLE Employees
DROP COLUMN salary;
```

Это не только удалит информацию из этого столбца, но и снизит зарплату столбца от сотрудников таблицы (колонка больше не будет существовать).

Ограничение падения

```
ALTER TABLE Employees
DROP CONSTRAINT DefaultSalary
```

Это удаляет ограничение, называемое DefaultSalary, из определения таблицы сотрудников.

Примечание. - Перед сбросом столбца убедитесь, что ограничения столбца упакованы.

Добавить ограничение

```
ALTER TABLE Employees
ADD CONSTRAINT DefaultSalary DEFAULT ((100)) FOR [Salary]
```

Это добавляет ограничение, называемое DefaultSalary, которое задает значение по умолчанию для столбца Зарплата.

Ограничение может быть добавлено на уровне таблицы.

Типы ограничений

- Первичный ключ - предотвращает дублирование записи в таблице
- Внешний ключ - указывает на первичный ключ из другой таблицы
- Not Null - предотвращает ввод нулевых значений в столбец
- Уникальный - уникально идентифицирует каждую запись в таблице
- Значение по умолчанию - задает значение по умолчанию
- Check - ограничивает диапазоны значений, которые могут быть помещены в столбец

Дополнительные сведения о ограничениях см. В документации Oracle .

Изменить колонку

```
ALTER TABLE Employees
ALTER COLUMN StartingDate DATETIME NOT NULL DEFAULT (GETDATE())
```

Этот запрос будет изменить тип данных столбца `StartingDate` и изменить его с простой `date` в `datetime` и установить по умолчанию для текущей даты.

Добавить основной ключ

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID)
```

Это добавит основной ключ к таблице Employees в поле `ID` . Включение более чем одного имени столбца в круглые скобки вместе с идентификатором создаст составной первичный ключ. При добавлении нескольких столбцов имена столбцов должны быть разделены запятыми.

```
ALTER TABLE EMPLOYEES ADD pk_EmployeeID PRIMARY KEY (ID, FName)
```

Прочитайте ALTER TABLE онлайн: https://riptutorial.com/ru/sql/topic/356/alter-table

глава 3: CREATE Database

Синтаксис

- CREATE DATABASE dbname;

Examples

CREATE Database

База данных создается со следующей командой SQL:

```
CREATE DATABASE myDatabase;
```

Это создало бы пустую базу данных с именем myDatabase, где вы могли бы создавать таблицы.

Прочитайте CREATE Database онлайн: https://riptutorial.com/ru/sql/topic/2744/create-database

глава 4: DROP или DELETE Database

Синтаксис

- Синтаксис MSSQL:
- DROP DATABASE [IF EXISTS] {имя_базы_базы | database_snapshot_name} [, ... n] [;]
- Синтаксис MySQL:
- DROP {БАЗЫ ДАННЫХ | SCHEMA} [IF EXISTS] db_name

замечания

`DROP DATABASE` используется для удаления базы данных из SQL. Обязательно создайте резервную копию своей базы данных, прежде чем отбрасывать ее, чтобы предотвратить случайную потерю информации.

Examples

База данных DROP

Удаление базы данных - это простой оператор с одним вкладышем. База данных Drop удалит базу данных, поэтому всегда обеспечивайте резервное копирование базы данных, если это необходимо.

Ниже приведена команда удалить базу данных сотрудников

```
DROP DATABASE [dbo].[Employees]
```

Прочитайте DROP или DELETE Database онлайн: https://riptutorial.com/ru/sql/topic/3974/drop-или-delete-database

глава 5: GRANT и REVOKE

Синтаксис

- GRANT [привилегия1] [, [привилегия2] ...] ON [таблица] TO [грантополучатель1] [, [grantee2] ...] [WITH GRANT OPTION]
- REVOKE [привилегия1] [, [привилегия2] ...] ON [таблица] FROM [grantee1] [, [grantee2] ...]

замечания

Предоставьте разрешения для пользователей. Если указан параметр `WITH GRANT OPTION` , грантополучатель дополнительно получает привилегию предоставить данное разрешение или отменять ранее предоставленные разрешения.

Examples

Предоставить привилегии / аннулировать привилегии

```
GRANT SELECT, UPDATE
ON Employees
TO User1, User2;
```

Предоставить `User1` и `User2` разрешение выполнять операции `SELECT` и `UPDATE` в таблице `Employees` .

```
REVOKE SELECT, UPDATE
ON Employees
FROM User1, User2;
```

Отменить из `User1` и `User2` разрешение на выполнение операций `SELECT` и `UPDATE` в таблице Employees.

Прочитайте GRANT и REVOKE онлайн: https://riptutorial.com/ru/sql/topic/5574/grant-и-revoke

глава 6: IN

Examples

Простая статья IN

Чтобы получить записи, имеющие **какой-либо** из указанных `id`

```
select *
from products
where id in (1,8,3)
```

Вышеуказанный запрос равен

```
select *
from products
where id = 1
   or id = 8
   or id = 3
```

Использование предложения IN с подзапросом

```
SELECT *
FROM customers
WHERE id IN (
    SELECT DISTINCT customer_id
    FROM orders
);
```

Вышесказанное даст вам всех клиентов, у которых есть заказы в системе.

Прочитайте IN онлайн: https://riptutorial.com/ru/sql/topic/3169/in

глава 7: MERGE

Вступление

MERGE (часто также называемый UPSERT для «обновления или вставки») позволяет вставлять новые строки или, если строка уже существует, обновлять существующую строку. Суть заключается в том, чтобы выполнить весь набор операций атомарно (чтобы гарантировать, что данные остаются согласованными) и предотвращать накладные расходы на связь для нескольких операторов SQL в системе клиент / сервер.

Examples

MERGE, чтобы создать Target match Source

```
MERGE INTO targetTable t
    USING sourceTable s
        ON t.PKID = s.PKID
    WHEN MATCHED AND NOT EXISTS (
            SELECT s.ColumnA, s.ColumnB, s.ColumnC
            INTERSECT
            SELECT t.ColumnA, t.ColumnB, s.ColumnC
            )
        THEN UPDATE SET
            t.ColumnA = s.ColumnA
            ,t.ColumnB = s.ColumnB
            ,t.ColumnC = s.ColumnC
    WHEN NOT MATCHED BY TARGET
        THEN INSERT (PKID, ColumnA, ColumnB, ColumnC)
        VALUES (s.PKID, s.ColumnA, s.ColumnB, s.ColumnC)
    WHEN NOT MATCHED BY SOURCE
        THEN DELETE
    ;
```

Примечание. Часть `AND NOT EXISTS` позволяет обновлять записи, которые не изменились. Использование конструкции `INTERSECT` позволяет сравнивать столбцы с `INTERSECT` значениями без специальной обработки.

MySQL: подсчет пользователей по имени

Предположим, мы хотим знать, сколько пользователей имеют одно и то же имя. Давайте создадим `users` таблицы следующим образом:

```
create table users(
    id int primary key auto_increment,
    name varchar(8),
    count int,
    unique key name(name)
);
```

Теперь мы просто обнаружили нового пользователя по имени Джо и хотели бы его принять во внимание. Для этого нам нужно определить, существует ли существующая строка с его именем, и если да, обновите ее, чтобы увеличить счетчик; с другой стороны, если нет существующей строки, мы должны ее создать.

MySQL использует следующий синтаксис: insert ... при дублировании ключевого обновления .... В этом случае:

```
insert into users(name, count)
       values ('Joe', 1)
       on duplicate key update count=count+1;
```

PostgreSQL: подсчет пользователей по имени

Предположим, мы хотим знать, сколько пользователей имеют одно и то же имя. Давайте создадим `users` таблицы следующим образом:

```
create table users(
    id serial,
    name varchar(8) unique,
    count int
);
```

Теперь мы просто обнаружили нового пользователя по имени Джо и хотели бы его принять во внимание. Для этого нам нужно определить, существует ли существующая строка с его именем, и если да, обновите ее, чтобы увеличить счетчик; с другой стороны, если нет существующей строки, мы должны ее создать.

PostgreSQL использует следующий синтаксис: insert ... on conflict ... do update .... В этом случае:

```
insert into users(name, count)
    values('Joe', 1)
    on conflict (name) do update set count = users.count + 1;
```

Прочитайте MERGE онлайн: https://riptutorial.com/ru/sql/topic/1470/merge

глава 8: SKIP TAKE (Pagination)

Examples

Пропуск ряда строк из результата

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 42424242424242;
-- skips 20 for take use very large number that is more than rows in table
```

Oracle:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber > 20
```

PostgreSQL:

```
SELECT * FROM TableName OFFSET 20;
```

SQLite:

```
SELECT * FROM TableName LIMIT -1 OFFSET 20;
```

Ограничение количества результатов

ISO / ANSI SQL:

```
SELECT * FROM TableName FETCH FIRST 20 ROWS ONLY;
```

MySQL; PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20;
```

Oracle:

```
SELECT Id,
   Col1
FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber <= 20
```

SQL Server:

```
SELECT TOP 20 *
FROM dbo.[Sale]
```

Пропуская затем некоторые результаты (разбиение на страницы)

ISO / ANSI SQL:

```
SELECT Id, Col1
FROM TableName
ORDER BY Id
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

MySQL:

```
SELECT * FROM TableName LIMIT 20, 20; -- offset, limit
```

Oracle; SQL Server:

```
SELECT Id,
   Col1
 FROM (SELECT Id,
           Col1,
           row_number() over (order by Id) RowNumber
      FROM TableName)
WHERE RowNumber BETWEEN 21 AND 40
```

PostgreSQL; SQLite:

```
SELECT * FROM TableName LIMIT 20 OFFSET 20;
```

Прочитайте SKIP TAKE (Pagination) онлайн: https://riptutorial.com/ru/sql/topic/2927/skip-take--pagination-

глава 9: SQL CURSOR

Examples

Пример курсора, который запрашивает все строки по индексу для каждой базы данных

Здесь курсор используется для прокрутки всех баз данных.
Более того, курсор из динамического sql используется для запроса каждой базы данных, возвращаемой первым курсором.
Это должно продемонстрировать область подключения курсора.

```
DECLARE @db_name nvarchar(255)
DECLARE @sql nvarchar(MAX)

DECLARE @schema nvarchar(255)
DECLARE @table nvarchar(255)
DECLARE @column nvarchar(255)




DECLARE db_cursor CURSOR FOR
SELECT name FROM sys.databases


OPEN db_cursor
FETCH NEXT FROM db_cursor INTO @db_name

WHILE @@FETCH_STATUS = 0
BEGIN
    SET @sql = 'SELECT * FROM ' + QUOTENAME(@db_name) + '.information_schema.columns'
    PRINT ''
    PRINT ''
    PRINT ''
    PRINT @sql
    -- EXECUTE(@sql)



    -- For each database

    DECLARE @sqlstatement nvarchar(4000)
    --move declare cursor into sql to be executed
    SET @sqlstatement = 'DECLARE  columns_cursor CURSOR FOR SELECT TABLE_SCHEMA, TABLE_NAME,
COLUMN_NAME FROM ' + QUOTENAME(@db_name) + '.information_schema.columns ORDER BY TABLE_SCHEMA,
TABLE_NAME, ORDINAL_POSITION'



    EXEC sp_executesql @sqlstatement


    OPEN columns_cursor
```

```
    FETCH NEXT FROM columns_cursor
    INTO @schema, @table, @column

    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT @schema + '.' + @table + '.' + @column
        --EXEC asp_DoSomethingStoredProc @UserId

    FETCH NEXT FROM columns_cursor --have to fetch again within loop
    INTO @schema, @table, @column

    END
    CLOSE columns_cursor
    DEALLOCATE columns_cursor

    -- End for each database




    FETCH NEXT FROM db_cursor INTO @db_name
END

CLOSE db_cursor
DEALLOCATE db_cursor
```

Прочитайте SQL CURSOR онлайн: https://riptutorial.com/ru/sql/topic/8895/sql-cursor

глава 10: SQL Group By vs Distinct

Examples

Разница между GROUP BY и DISTINCT

`GROUP BY` используется в сочетании с функциями агрегации. Рассмотрим следующую таблицу:

номер заказа	Идентификатор пользователя	название магазина	Ценность заказа	Дата заказа
1	43	Магазин A	25	20-03-2016
2	57	Магазин B	50	22-03-2016
3	43	Магазин A	30	25-03-2016
4	82	Магазин C	10	26-03-2016
5	21	Магазин A	45	29-03-2016

В нижеследующем запросе используется `GROUP BY` для выполнения агрегированных вычислений.

```
SELECT
    storeName,
    COUNT(*) AS total_nr_orders,
    COUNT(DISTINCT userId) AS nr_unique_customers,
    AVG(orderValue) AS average_order_value,
    MIN(orderDate) AS first_order,
    MAX(orderDate) AS lastOrder
FROM
    orders
GROUP BY
    storeName;
```

и вернет следующую информацию

название магазина	total_nr_orders	nr_unique_customers	average_order_value	первый заказ	последни заказ
Магазин A	3	2	33,3	20-03-2016	29-03-201
Магазин B	1	1	50	22-03-2016	22-03-201
Магазин C	1	1	10	26-03-2016	26-03-201

Хотя `DISTINCT` используется для отображения уникальной комбинации различных значений для указанных столбцов.

```
SELECT DISTINCT
    storeName,
    userId
FROM
    orders;
```

название магазина	Идентификатор пользователя
Магазин A	43
Магазин B	57
Магазин C	82
Магазин A	21

Прочитайте SQL Group By vs Distinct онлайн: https://riptutorial.com/ru/sql/topic/2499/sql-group-by-vs-distinct

глава 11: SQL-инъекция

Вступление

SQL-инъекция - попытка получить доступ к таблицам базы данных веб-сайта, введя SQL в поле формы. Если веб-сервер не защищает от атак SQL-инъекций, хакер может обмануть базу данных для запуска дополнительного кода SQL. Выполняя собственный код SQL, хакеры могут обновлять свой доступ к учетной записи, просматривать чужую личную информацию или делать какие-либо другие изменения в базе данных.

Examples

Образец инъекции SQL

Предполагая, что вызов обработчика входа вашего веб-приложения выглядит так:

```
https://somepage.com/ajax/login.ashx?username=admin&password=123
```

Теперь в login.ashx вы читаете эти значения:

```
strUserName = getHttpsRequestParameterString("username");
strPassword = getHttpsRequestParameterString("password");
```

и запросите свою базу данных, чтобы определить, существует ли пользователь с этим паролем.

Итак, вы строите строку запроса SQL:

```
txtSQL = "SELECT * FROM Users WHERE username = '" + strUserName + "' AND password = '"+
strPassword +"'";
```

Это будет работать, если имя пользователя и пароль не содержат цитаты.

Однако, если один из параметров содержит цитату, SQL, который отправляется в базу данных, будет выглядеть так:

```
-- strUserName = "d'Alambert";
txtSQL = "SELECT * FROM Users WHERE username = 'd'Alambert' AND password = '123'";
```

Это приведет к синтаксической ошибке, потому что цитата после `d` в `d'Alambert` заканчивает строку SQL.

Вы можете исправить это, ускорив кавычки в имени пользователя и пароле, например:

```
strUserName = strUserName.Replace("'", "''");
strPassword = strPassword.Replace("'", "''");
```

Однако более целесообразно использовать параметры:

```
cmd.CommandText = "SELECT * FROM Users WHERE username = @username AND password = @password";

cmd.Parameters.Add("@username", strUserName);
cmd.Parameters.Add("@password", strPassword);
```

Если вы не используете параметры и не забываете заменить цитату хотя бы одним из значений, злоумышленник (ака хакер) может использовать это для выполнения SQL-команд в вашей базе данных.

Например, если злоумышленник злодей, он / она установит пароль для

```
lol'; DROP DATABASE master; --
```

и тогда SQL будет выглядеть так:

```
"SELECT * FROM Users WHERE username = 'somebody' AND password = 'lol'; DROP DATABASE master; --'";
```

К сожалению, для вас это действительный SQL, и БД выполнит это!

Этот тип эксплойта называется SQL-инъекцией.

Есть много других вещей, которые мог бы совершить злонамеренный пользователь, например, кражи каждого адреса электронной почты каждого пользователя, кражи пароля каждого пользователя, кражи номеров кредитных карт, кражи любого количества данных в вашей базе данных и т. Д.

Вот почему вам всегда нужно избегать ваших строк.
И тот факт, что вы непременно забудете сделать это рано или поздно, именно поэтому вы должны использовать параметры. Потому что, если вы используете параметры, ваша система языка программирования сделает все необходимое для вас.

простой образец для инъекций

Если инструкция SQL построена следующим образом:

```
SQL = "SELECT * FROM Users WHERE username = '" + user + "' AND password ='" + pw + "'";
db.execute(SQL);
```

Затем хакер может получить ваши данные, указав пароль как `pw' or '1'='1` ; результирующий оператор SQL будет выглядеть следующим образом:

```
SELECT * FROM Users WHERE username = 'somebody' AND password ='pw' or '1'='1'
```

Это проверит проверку пароля для всех строк в таблице « `Users` потому что `'1'='1'` всегда истинно.

Чтобы предотвратить это, используйте параметры SQL:

```
SQL = "SELECT * FROM Users WHERE username = ? AND password = ?";
db.execute(SQL, [user, pw]);
```

Прочитайте SQL-инъекция онлайн: https://riptutorial.com/ru/sql/topic/3517/sql-инъекция

глава 12: TRUNCATE

Вступление

Оператор TRUNCATE удаляет все данные из таблицы. Это похоже на DELETE без фильтра, но, в зависимости от программного обеспечения базы данных, имеет определенные ограничения и оптимизации.

Синтаксис

- TRUNCATE TABLE имя_таблицы;

замечания

TRUNCATE - это команда DDL (Data Definition Language), и поэтому существуют значительные различия между ней и DELETE (язык манипулирования данными, команда DML). Хотя TRUNCATE может быть средством быстрого удаления больших объемов записей из базы данных, эти различия следует понимать, чтобы решить, подходит ли использование команды TRUNCATE в вашей конкретной ситуации.

- TRUNCATE - это операция с данными. Поэтому триггеры DML (ON DELETE), связанные с таблицей, не срабатывают при выполнении операции TRUNCATE. Хотя это позволит сэкономить большое количество времени для массовых операций удаления, однако вам может понадобиться вручную удалить связанные данные.
- TRUNCATE освободит пространство на диске, используемое удаленными строками, DELETE освободит место
- Если усеченная таблица использует столбцы идентификаторов (MS SQL Server), то семя сбрасывается командой TRUNCATE. Это может привести к проблемам ссылочной целостности
- В зависимости от используемых ролей безопасности и используемого варианта SQL у вас могут не быть необходимых разрешений для выполнения команды TRUNCATE

Examples

Удаление всех строк из таблицы Employee

```
TRUNCATE TABLE Employee;
```

Использование таблицы усечений часто бывает лучше, чем использование DELETE TABLE, поскольку оно игнорирует все индексы и триггеры и просто удаляет все.

Удалить таблицу - это операция на основе строки, это означает, что каждая строка удаляется. Таблица усечения - это операция страницы с данными, перераспределенная на всю страницу данных. Если у вас есть таблица с миллионом строк, будет намного быстрее обрезать таблицу, чем было бы использовать оператор таблицы удалений.

Хотя мы можем удалять определенные строки с помощью DELETE, мы не можем TRUNCATE определенные строки, мы можем только TRUNCATE все записи сразу. Удаление всех строк, а затем добавление новой записи будет продолжать добавление значения «Приоритет» с автоматическим добавлением из ранее вставленного значения, где, как и в Truncate, значение первичного ключа Auto Incremental также будет сброшено и начнется с 1.

Обратите внимание, что при усечении таблицы **внешние ключи не должны присутствовать** , иначе вы получите сообщение об ошибке.

Прочитайте TRUNCATE онлайн: https://riptutorial.com/ru/sql/topic/1466/truncate

глава 13: XML

Examples

Запрос из типа данных XML

```
DECLARE @xmlIN XML = '<TableData>
<aaa Main="First">
  <row name="a" value="1" />
  <row name="b" value="2" />
  <row name="c" value="3" />
</aaa>
<aaa Main="Second">
  <row name="a" value="3" />
  <row name="b" value="4" />
  <row name="c" value="5" />
</aaa>
<aaa Main="Third">
  <row name="a" value="10" />
  <row name="b" value="20" />
  <row name="c" value="30" />
</aaa>
</TableData>'

SELECT t.col.value('../@Main', 'varchar(10)') [Header],
t.col.value('@name', 'VARCHAR(25)') [name],
t.col.value('@value',  'VARCHAR(25)') [Value]
FROM    @xmlIn.nodes('//TableData/aaa/row') AS t (col)
```

Результаты

```
Header    name    Value
First      a        1
First      b        2
First      c        3
Second     a        3
Second     b        4
Second     c        5
Third      a        10
Third      b        20
Third      c        30
```

Прочитайте XML онлайн: https://riptutorial.com/ru/sql/topic/4421/xml

глава 14: Блоки выполнения

Examples

Использование BEGIN ... END

```
BEGIN
  UPDATE Employees SET PhoneNumber = '5551234567' WHERE Id = 1;
  UPDATE Employees SET Salary = 650 WHERE Id = 3;
END
```

Прочитайте Блоки выполнения онлайн: https://riptutorial.com/ru/sql/topic/1632/блоки-выполнения

глава 15: ВСТАВИТЬ

Синтаксис

- INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);
- INSERT INTO table_name (column1, column2 ...) SELECT value1, value2 ... from other_table

Examples

Вставить новую строку

```
INSERT INTO Customers
VALUES ('Zack', 'Smith', 'zack@example.com', '7049989942', 'EMAIL');
```

Этот оператор добавит новую строку в таблицу `Customers` . Обратите внимание, что значение не было указано для столбца `Id` , поскольку оно будет добавлено автоматически. Однако должны быть указаны все другие значения столбцов.

Вставить только указанные столбцы

```
INSERT INTO Customers (FName, LName, Email, PreferredContact)
VALUES ('Zack', 'Smith', 'zack@example.com', 'EMAIL');
```

Этот оператор добавит новую строку в таблицу `Customers` . Данные будут вставлены только в указанные столбцы - обратите внимание, что для столбца `PhoneNumber` не было указано `PhoneNumber` . Обратите внимание, однако, что все столбцы, помеченные как `not null` должны быть включены.

INSERT данные из другой таблицы с помощью SELECT

```
INSERT INTO Customers (FName, LName, PhoneNumber)
SELECT FName, LName, PhoneNumber FROM Employees
```

В этом примере в таблицу Customers будут вставляться все сотрудники . Поскольку у двух таблиц есть разные поля, и вы не хотите переместить все поля, вам нужно указать, какие поля вставлять и какие поля выбрать. Имена полей корреляции не нужно называть одной и той же, но тогда они должны быть одного и того же типа данных. В этом примере предполагается, что поле Id имеет набор идентификационных данных и будет автоматически увеличиваться.

Если у вас есть две таблицы с одинаковыми именами полей и просто хотите переместить

все записи, вы можете использовать:

```
INSERT INTO Table1
SELECT * FROM Table2
```

Вставить сразу несколько строк

Несколько строк могут быть вставлены с помощью одной команды вставки:

```
INSERT INTO tbl_name (field1, field2, field3)

VALUES (1,2,3), (4,5,6), (7,8,9);
```

Для одновременного ввода больших объемов данных (объемная вставка) существуют специфические функции и рекомендации СУБД.

MySQL - LOAD DATA INFILE

MSSQL - BULK INSERT

Прочитайте ВСТАВИТЬ онлайн: https://riptutorial.com/ru/sql/topic/465/вставить

глава 16: ВЫБРАТЬ

Вступление

Оператор SELECT лежит в основе большинства SQL-запросов. Он определяет, какой результат должен быть возвращен запросом, и почти всегда используется в сочетании с предложением FROM, которое определяет, какую часть (ы) базы данных следует запрашивать.

Синтаксис

- SELECT [DISTINCT] [column1] [, [column2] ...]
  FROM [таблица]
  [ГДЕ условие]
  [GROUP BY [column1] [, [column2] ...]

  [HAVING [column1] [, [column2] ...]

  [ORDER BY ASC | DESC]

замечания

SELECT определяет, какие данные столбцов будут возвращены и в каком порядке из данной таблицы (при условии, что они соответствуют другим требованиям в вашем запросе конкретно - где и с фильтрами и объединениями).

```
SELECT Name, SerialNumber
FROM ArmyInfo
```

будут возвращать результаты только из столбцов `Name` и `Serial Number` , но не из столбца `Rank` , например

```
SELECT *
FROM ArmyInfo
```

указывает, что **все** столбцы будут возвращены. Однако учтите, что это низкая практика `SELECT *` поскольку вы буквально возвращаете все столбцы таблицы.

Examples

Использование символа подстановки для выбора всех столбцов в запросе.

Рассмотрим базу данных со следующими двумя таблицами.

Таблица сотрудников:

Я бы	FName	LName	DeptId
1	Джеймс	кузнец	3
2	Джон	Джонсон	4

Таблица отделов:

Я бы	название
1	Продажи
2	маркетинг
3	финансов
4	ЭТО

Операция простого выбора

`*` - это **символ подстановки,** используемый для выбора всех доступных столбцов в таблице.

При использовании в качестве замены явных имен столбцов он возвращает все столбцы во всех таблицах, которые запрос выбирает `FROM` . Этот эффект применяется ко **всем таблицам,** к которым обращается запрос через его предложения `JOIN` .

Рассмотрим следующий запрос:

```
SELECT * FROM Employees
```

Он вернет все поля всех строк таблицы `Employees` :

Я бы	FName	LName	DeptId
1	Джеймс	кузнец	3
2	Джон	Джонсон	4

Точечная нотация

Чтобы выбрать все значения из конкретной таблицы, подстановочный знак можно применить к таблице с *точечной нотацией* .

Рассмотрим следующий запрос:

```
SELECT
    Employees.*,
    Departments.Name
FROM
    Employees
JOIN
    Departments
    ON Departments.Id = Employees.DeptId
```

Это вернет набор данных со всеми полями в таблице `Employee` , а затем просто поле `Name` в таблице `Departments` :

Я бы	FName	LName	DeptId	название
1	Джеймс	кузнец	3	финансов
2	Джон	Джонсон	4	ЭТО

Предупреждения против использования

Как правило, рекомендуется использовать `*` в производственном коде, где это возможно, поскольку это может вызвать ряд потенциальных проблем, в том числе:

1. Избыток ввода-вывода, загрузка сети, использование памяти и т. Д. Из-за того, что данные базы данных не считывают данные и не передают их интерфейсу. Это особенно опасно, когда могут быть большие поля, такие как те, которые используются для хранения длинных заметок или прикрепленных файлов.
2. Дальнейшая избыточная загрузка ввода-вывода, если базе данных необходимо спутать внутренние результаты на диск как часть обработки запроса более сложным, чем `SELECT <columns> FROM <table>` .
3. Дополнительная обработка (и / или даже больше ввода-вывода), если некоторые из ненужных столбцов:
    - вычисляемые столбцы в базах данных, которые их поддерживают
    - в случае выбора из представления столбцы из таблицы / представления, которые оптимизатор запросов мог бы в противном случае оптимизировать
4. Потенциал неожиданных ошибок, если столбцы добавляются к таблицам и представлениям позже, приводит к неоднозначным именам столбцов. Например, `SELECT * FROM orders JOIN people ON people.id = orders.personid ORDER BY displayname` - если столбец столбца с именем `displayname` добавлен в таблицу заказов, чтобы пользователи могли давать свои заказы значимых имен для будущей ссылки, тогда появится имя столбца дважды на выходе, поэтому предложение `ORDER BY` будет

неоднозначным, что может привести к ошибкам («неоднозначное имя столбца» в последних версиях MS SQL Server), и если не в этом примере, ваш код приложения может начать отображать имя заказа, в котором имя человека потому что новый столбец является первым из этого имени, и так далее.

Когда вы можете использовать `*` , принимая предупреждение выше?

Несмотря на то, что в производственном коде лучше всего избегать, использование `*` отлично подходит для выполнения ручных запросов к базе данных для исследования или работы прототипа.

Иногда проектные решения в вашей заявке делают ее неизбежной (в таких обстоятельствах предпочитают `tablealias.*` по возможности `*` возможности).

При использовании `EXISTS` , таких как `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)` , мы не возвращаем никаких данных из B. Таким образом, соединение не нужно, и двигатель знает, что никакие значения из B не должны быть возвращены, поэтому при использовании `*` . Аналогично `COUNT(*)` является прекрасным, так как он также фактически не возвращает ни один из столбцов, поэтому нужно только читать и обрабатывать те, которые используются для целей фильтрации.

Выбор с условием

Основной синтаксис предложения SELECT с предложением WHERE:

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

[Condition] может быть любым выражением SQL, указанным с использованием сравнительных или логических операторов, таких как>, <, =, <>,> =, <=, LIKE, NOT, IN, BETWEEN и т. Д.

Следующий оператор возвращает все столбцы из таблицы «Автомобили», где столбец состояния «ГОТОВ»:

```
SELECT * FROM Cars WHERE status = 'READY'
```

См. WHERE и HAVING для получения дополнительных примеров.

Выберите отдельные столбцы

```
SELECT
    PhoneNumber,
    Email,
    PreferredContact
```

```
FROM Customers
```

Этот оператор возвращает столбцы `PhoneNumber` , `Email` и `PreferredContact` из всех строк таблицы `Customers` . Также столбцы будут возвращены в последовательности, в которой они отображаются в предложении `SELECT` .

Результатом будет:

Номер телефона	Эл. адрес	PreferredContact
3347927472	william.jones@example.com	ТЕЛЕФОН
2137921892	dmiller@example.net	ЭЛ. АДРЕС
НОЛЬ	richard0123@example.com	ЭЛ. АДРЕС

Если несколько таблиц объединены вместе, вы можете выбрать столбцы из определенных таблиц, указав имя таблицы перед именем столбца: `[table_name].[column_name]`

```
SELECT
    Customers.PhoneNumber,
    Customers.Email,
    Customers.PreferredContact,
    Orders.Id AS OrderId
FROM
    Customers
LEFT JOIN
    Orders ON Orders.CustomerId = Customers.Id
```

* `AS OrderId` означает, что поле `Id` таблицы `Orders` будет возвращено в виде столбца с именем `OrderId` . Для получения дополнительной информации см. Выбор с помощью псевдонима столбца .

Чтобы избежать использования имен длинных таблиц, вы можете использовать псевдонимы таблиц. Это уменьшает боль при написании длинных имен таблиц для каждого поля, которое вы выбираете в объединениях. Если вы выполняете самостоятельное соединение (соединение между двумя экземплярами одной и той *же* таблицы), вы должны использовать псевдонимы таблиц, чтобы отличать ваши таблицы. Мы можем написать псевдоним таблицы, такой как `Customers c` или `Customers AS c` . Здесь `c` работает как псевдоним для `Customers` и мы можем выбрать, скажем, по `Email` : `c.Email` .

```
SELECT
    c.PhoneNumber,
    c.Email,
    c.PreferredContact,
    o.Id AS OrderId
FROM
    Customers c
LEFT JOIN
```

```
    Orders o ON o.CustomerId = c.Id
```

SELECT Использование псевдонимов столбцов

Алиасы столбцов используются, главным образом, для сокращения кода и создания более удобных имен столбцов.

Код становится короче, как длинные имена таблиц и ненужная идентификация столбцов *(например, в таблице может быть 2 идентификатора, но только один из них используется в инструкции)* . Наряду с псевдонимами таблиц это позволяет использовать более длинные описательные имена в структуре вашей базы данных, сохраняя при этом запросы в этой структуре краткими.

Кроме того, они иногда *требуются* , например, в представлениях, чтобы назвать вычисленные выходы.

Все версии SQL

Псевдонимы могут быть созданы во всех версиях SQL с использованием двойных кавычек (`"`).

```
SELECT
    FName AS "First Name",
    MName AS "Middle Name",
    LName AS "Last Name"
FROM Employees
```

Различные версии SQL

Вы можете использовать одинарные кавычки (`'`), двойные кавычки (`"`) и квадратные скобки (`[]`) для создания псевдонима в Microsoft SQL Server.

```
SELECT
    FName AS "First Name",
    MName AS 'Middle Name',
    LName AS [Last Name]
FROM Employees
```

Оба приведут к:

Имя	Второе имя	Фамилия
Джеймс	Джон	кузнец
Джон	Джеймс	Джонсон

Имя	Второе имя	Фамилия
Майкл	Маркус	Williams

Этот оператор возвращает столбцы `FName` и `LName` с заданным именем (псевдоним). Это достигается с помощью оператора `AS` за которым следует псевдоним, или просто записывая псевдоним непосредственно после имени столбца. Это означает, что следующий запрос имеет тот же результат, что и выше.

```
SELECT
    FName "First Name",
    MName "Middle Name",
    LName "Last Name"
FROM Employees
```

Имя	Второе имя	Фамилия
Джеймс	Джон	кузнец
Джон	Джеймс	Джонсон
Майкл	Маркус	Williams

Однако явная версия (т. Е. Использование оператора `AS`) более читаема.

Если псевдоним имеет одно слово, которое не является зарезервированным словом, мы можем записать его без одинарных кавычек, двойных кавычек или скобок:

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM Employees
```

Имя	Фамилия
Джеймс	кузнец
Джон	Джонсон
Майкл	Williams

Еще один вариант, доступный в MS SQL Server, - это `<alias> = <column-or-calculation>` , например:

```
SELECT FullName = FirstName + ' ' + LastName,
       Addr1    = FullStreetAddress,
       Addr2    = TownName
FROM CustomerDetails
```

что эквивалентно:

```
SELECT FirstName + ' ' + LastName As FullName
       FullStreetAddress          As Addr1,
       TownName                   As Addr2
FROM CustomerDetails
```

Оба приведут к:

ФИО	Addr1	Addr2
Джеймс Смит	123 AnyStreet	TownVille
Джон Джонсон	668 MyRoad	Anytown
Майкл Уильямс	999 High End Dr	Williamsburgh

Некоторые находят использование `=` вместо того, чтобы читать `As` легче, хотя многие рекомендуют этот формат, главным образом потому, что он не является стандартным, поэтому не поддерживается всеми базами данных. Это может привести к путанице с другими использованиями символа `=` .

Все версии SQL

Кроме того, если вам *нужно* использовать зарезервированные слова, вы можете использовать скобки или кавычки для выхода:

```
SELECT
    FName as "SELECT",
    MName as "FROM",
    LName as "WHERE"
FROM Employees
```

Различные версии SQL

Аналогично, вы можете избежать ключевых слов в MSSQL со всеми различными подходами:

```
SELECT
    FName AS "SELECT",
    MName AS 'FROM',
    LName AS [WHERE]
FROM Employees
```

ВЫБРАТЬ	ОТ	ГДЕ
Джеймс	Джон	кузнец

ВЫБРАТЬ	ОТ	ГДЕ
Джон	Джеймс	Джонсон
Майкл	Маркус	Williams

Кроме того, псевдоним столбца может использоваться любым из заключительных предложений одного и того же запроса, например `ORDER BY` :

```
SELECT
    FName AS FirstName,
    LName AS LastName
FROM
    Employees
ORDER BY
    LastName DESC
```

Однако вы *не* можете использовать

```
SELECT
    FName AS SELECT,
    LName AS FROM
FROM
    Employees
ORDER BY
    LastName DESC
```

Чтобы создать псевдоним из этих зарезервированных слов (`SELECT` и `FROM`).

Это вызовет многочисленные ошибки при выполнении.

Выбор с отсортированными результатами

```
SELECT * FROM Employees ORDER BY LName
```

Этот оператор вернет все столбцы из таблицы `Employees` .

Я бы	FName	LName	Номер телефона
2	Джон	Джонсон	2468101214
1	Джеймс	кузнец	1234567890
3	Майкл	Williams	1357911131

```
SELECT * FROM Employees ORDER BY LName DESC
```

Или же

```
 SELECT * FROM Employees ORDER BY LName ASC
```

Это заявление изменяет направление сортировки.

Можно также указать несколько столбцов сортировки. Например:

```
SELECT * FROM Employees ORDER BY LName ASC, FName ASC
```

Этот пример сначала сортирует результаты по `LName` а затем для записей, имеющих одно и то же `LName` , сортирует по `FName` . Это даст вам результат, аналогичный тому, что вы найдете в телефонной книге.

Чтобы сохранить переименование имени столбца в предложении `ORDER BY` , вместо него можно использовать номер столбца. Обратите внимание, что номера столбцов начинаются с 1.

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY 3
```

Вы также можете включить оператор `CASE` в предложение `ORDER BY` .

```
SELECT Id, FName, LName, PhoneNumber FROM Employees ORDER BY CASE WHEN LName='Jones` THEN 0
ELSE 1 END ASC
```

Это приведет к сортировке результатов, чтобы все записи были с `LName` «Jones» вверху.

Выберите столбцы с именами из зарезервированных ключевых слов

Когда имя столбца соответствует зарезервированному ключевому слову, стандартный SQL требует, чтобы вы заключили его в двойные кавычки:

```
SELECT
    "ORDER",
    ID
FROM ORDERS
```

Обратите внимание, что это делает имя столбца чувствительным к регистру.

Некоторые СУБД имеют собственные способы цитирования имен. Например, для этой цели SQL Server использует квадратные скобки:

```
SELECT
    [Order],
    ID
FROM ORDERS
```

в то время как MySQL (и MariaDB) по умолчанию используют обратные ссылки:

```
SELECT
    `Order`,
    id
FROM orders
```

Выбор указанного количества записей

Стандарт SQL 2008 определяет предложение `FETCH FIRST` , чтобы ограничить количество возвращенных записей.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
FETCH FIRST 10 ROWS ONLY
```

Этот стандарт поддерживается только в последних версиях некоторых RDMS. Специфический нестандартный синтаксис, предоставляемый поставщиком, предоставляется в других системах. Прогресс OpenEdge 11.x также поддерживает синтаксис `FETCH FIRST <n> ROWS ONLY` .

Кроме того, `OFFSET <m> ROWS` перед `FETCH FIRST <n> ROWS ONLY` позволяет пропускать строки перед извлечением строк.

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
OFFSET 5 ROWS
FETCH FIRST 10 ROWS ONLY
```

Следующий запрос поддерживается в SQL Server и MS Access:

```
SELECT TOP 10 Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
```

Чтобы сделать то же самое в MySQL или PostgreSQL, необходимо использовать ключевое слово `LIMIT` :

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
ORDER BY UnitPrice DESC
LIMIT 10
```

В Oracle то же самое можно сделать с `ROWNUM` :

```
SELECT Id, ProductName, UnitPrice, Package
FROM Product
WHERE ROWNUM <= 10
ORDER BY UnitPrice DESC
```

Результаты : **10** записей.

```
Id    ProductName               UnitPrice             Package
38    Côte de Blaye             263.50                12 - 75 cl bottles
29    Thüringer Rostbratwurst   123.79                50 bags x 30 sausgs.
9    Mishi Kobe Niku            97.00                 18 - 500 g pkgs.
20    Sir Rodney's Marmalade    81.00                 30 gift boxes
18    Carnarvon Tigers          62.50                 16 kg pkg.
59    Raclette Courdavault      55.00                 5 kg pkg.
51    Manjimup Dried Apples     53.00                 50 - 300 g pkgs.
62    Tarte au sucre            49.30                 48 pies
43    Ipoh Coffee               46.00                 16 - 500 g tins
28    Rössle Sauerkraut         45.60                 25 - 825 g cans
```

Нюансы поставщика:

Важно отметить, что `TOP` в Microsoft SQL работает после `WHERE` и возвращает указанное количество результатов, если они существуют в любой точке таблицы, в то время как `ROWNUM` работает как часть `WHERE` , поэтому, если другие условия не существуют в указанное количество строк в начале таблицы, вы получите нулевые результаты, когда могут быть найдены другие.

Выбор с псевдонимом таблицы

```
SELECT e.Fname, e.LName
FROM Employees e
```

Таблице Employees присваивается псевдоним «e» непосредственно после имени таблицы. Это помогает устранить неоднозначность в сценариях, где несколько таблиц имеют одинаковое имя поля, и вам нужно указать, в какой таблице вы хотите вернуть данные.

```
SELECT e.Fname, e.LName, m.Fname AS ManagerFirstName
FROM Employees e
    JOIN Managers m ON e.ManagerId = m.Id
```

Обратите внимание, что как только вы определяете псевдоним, вы больше не можете использовать имя канонической таблицы. т.е.

```
SELECT e.Fname, Employees.LName, m.Fname AS ManagerFirstName
FROM Employees e
JOIN Managers m ON e.ManagerId = m.Id
```

будет выдавать ошибку.

Стоит отметить табличные псевдонимы - более формально «переменные диапазона» - были введены в язык SQL для решения проблемы дублированных столбцов, вызванных `INNER JOIN` . Стандарт SQL 1992 года исправил этот ранее недостаток дизайна, введя `NATURAL JOIN` (реализованный в mySQL, PostgreSQL и Oracle, но еще не в SQL Server), результат которого никогда не имеет повторяющихся имен столбцов. Вышеприведенный

пример интересен тем, что таблицы объединяются в столбцы с разными именами (`Id` и `ManagerId`), но не должны соединяться в столбцах с тем же именем (`LName` , `FName`), требуя переименования столбцов, которые должны быть выполнены *перед* присоединением:

```
SELECT Fname, LName, ManagerFirstName
FROM Employees
     NATURAL JOIN
     ( SELECT Id AS ManagerId, Fname AS ManagerFirstName
       FROM Managers ) m;
```

Обратите внимание, что хотя переменная alias / range должна быть объявлена для таблицы dervied (иначе SQL будет выдавать ошибку), никогда не имеет смысла фактически использовать ее в запросе.

Выбор строк из нескольких таблиц

```
SELECT *
FROM
    table1,
    table2
```

```
SELECT
    table1.column1,
    table1.column2,
    table2.column1
FROM
    table1,
    table2
```

Это называется кросс-продуктом в SQL, оно аналогично перекрестному продукту в наборах

Эти операторы возвращают выбранные столбцы из нескольких таблиц в одном запросе.

Между столбцами, возвращаемыми из каждой таблицы, нет никакой конкретной связи.

Выбор с помощью функций агрегации

Средний

Функция агрегации `AVG()` вернет среднее значение выбранных значений.

```
SELECT AVG(Salary) FROM Employees
```

Совокупные функции также могут быть объединены с предложением where.

```
SELECT AVG(Salary) FROM Employees where DepartmentId = 1
```

Совокупные функции также могут быть объединены с предложением group by.

Если сотрудник разделен на несколько отделов, и мы хотим найти среднюю зарплату для каждого отдела, мы можем использовать следующий запрос.

```
SELECT AVG(Salary) FROM Employees GROUP BY DepartmentId
```

минимальный

Функция агрегата `MIN()` возвращает минимальное количество выбранных значений.

```
SELECT MIN(Salary) FROM Employees
```

максимальная

Функция агрегации `MAX()` возвращает максимум выбранных значений.

```
SELECT MAX(Salary) FROM Employees
```

подсчитывать

Функция агрегации `COUNT()` возвращает количество выбранных значений.

```
SELECT Count(*) FROM Employees
```

Его также можно комбинировать с условиями, чтобы получить количество строк, удовлетворяющих определенным условиям.

```
SELECT Count(*) FROM Employees where ManagerId IS NOT NULL
```

Конкретные столбцы также могут быть указаны для получения количества значений в столбце. Обратите внимание, что значения `NULL` не учитываются.

```
Select Count(ManagerId) from Employees
```

Граф также может быть объединен с отдельным ключевым словом для отдельного счета.

```
Select Count(DISTINCT DepartmentId) from Employees
```

сумма

Функция агрегата `SUM()` возвращает сумму значений, выбранных для всех строк.

```
SELECT SUM(Salary) FROM Employees
```

Выбор с нулевым

```
SELECT Name FROM Customers WHERE PhoneNumber IS NULL
```

Выбор с нулями принимает другой синтаксис. Не используйте `=` , используйте `IS NULL` или `IS NOT NULL` .

Выбор с помощью CASE

Когда результаты должны иметь некоторую логику, применяемую «на лету», можно использовать оператор CASE для ее реализации.

```
SELECT CASE WHEN Col1 < 50 THEN 'under' ELSE 'over' END threshold
FROM TableName
```

также может быть прикован

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         WHEN Col1 > 50 AND Col1 <100 THEN 'between'
         ELSE 'over'
    END threshold
FROM TableName
```

один может также иметь `CASE` внутри другого оператора `CASE`

```
SELECT
    CASE WHEN Col1 < 50 THEN 'under'
         ELSE
            CASE WHEN Col1 > 50 AND Col1 <100 THEN Col1
            ELSE 'over' END
    END threshold
FROM TableName
```

Выбор без блокировки таблицы

Иногда, когда таблицы используются в основном (или только) для чтения, индексирование больше не помогает, и каждый бит подсчитывается, можно использовать select без LOCK для повышения производительности.

SQL Server

```
SELECT * FROM TableName WITH (nolock)
```

MySQL

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

оракул

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM TableName;
```

DB2

```
SELECT * FROM TableName WITH UR;
```

где `UR` означает «незафиксированное чтение».

Если вы используете таблицу, которая имеет изменения в записи, может иметь непредсказуемые результаты.

Выберите отдельные (только уникальные значения)

```
SELECT DISTINCT ContinentCode
FROM Countries;
```

Этот запрос вернет все значения `DISTINCT` (уникальные, разные) из столбца `ContinentCode` из таблицы `Countries`

ContinentCode
ОС
Евросоюз
КАК
Не Доступно
AF

Демоверсия SQLFiddle

Выбрать с условием нескольких значений из столбца

```
SELECT * FROM Cars  WHERE status IN ( 'Waiting', 'Working' )
```

Это семантически эквивалентно

```
SELECT * FROM Cars  WHERE ( status = 'Waiting' OR status = 'Working' )
```

т.е. `value IN ( <value list> )` является сокращением для дизъюнкции (логическое `OR`).

Получить агрегированный результат для групп строк

Подсчет строк на основе определенного значения столбца:

```
SELECT category, COUNT(*) AS item_count
FROM item
GROUP BY category;
```

Получение среднего дохода по отделу:

```
SELECT department, AVG(income)
FROM employees
GROUP BY department;
```

Важно выбрать только столбцы, указанные в предложении `GROUP BY` или используемые с агрегатными функциями .

Предложение `WHERE` также может использоваться с `GROUP BY` , но `WHERE` фильтрует записи *до того, как* будет выполнена какая-либо группировка:

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department;
```

Если вам нужно отфильтровать результаты после завершения группировки, например, чтобы увидеть только отделы, средний доход которых превышает 1000, вам нужно использовать предложение `HAVING` :

```
SELECT department, AVG(income)
FROM employees
WHERE department <> 'ACCOUNTING'
GROUP BY department
HAVING avg(income) > 1000;
```

Выбор с более чем 1 условием.

`AND` ключевое слово используется для добавления дополнительных условий для запроса.

название	Возраст	Пол
Сэм	18	M
Джон	21	M
боб	22	M
Мэри	23	F

```
SELECT name FROM persons WHERE gender = 'M' AND age > 20;
```

Это вернет:

название
Джон
боб

использование ключевого слова `OR`

```
SELECT name FROM persons WHERE gender = 'M' OR age < 20;
```

Это вернет:

название
Сэм
Джон
боб

Эти ключевые слова могут быть объединены для более сложных сочетаний критериев:

```
SELECT name
FROM persons
WHERE (gender = 'M' AND age < 20)
   OR (gender = 'F' AND age > 20);
```

Это вернет:

название
Сэм

название
Мэри

Прочитайте ВЫБРАТЬ онлайн: https://riptutorial.com/ru/sql/topic/222/выбрать

глава 17: ГРУППА ПО

Вступление

Результаты запроса SELECT можно сгруппировать по одному или нескольким столбцам с помощью оператора `GROUP BY` : все результаты с одинаковым значением в сгруппированных столбцах агрегируются вместе. Это генерирует таблицу частичных результатов вместо одного результата. GROUP BY может использоваться совместно с функциями агрегации с использованием оператора `HAVING` для определения того, как агрегируются негрупповые столбцы.

Синтаксис

- ГРУППА ПО {
  Колонка выражение
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBE (<group_by_expression> [, ... n])
  | ГРУППОВЫЕ НАБОРЫ ([, ... n])
  | () - вычисляет общую сумму
  } [, ... n]
- <group_by_expression> :: =
  Колонка выражение
  | (column-expression [, ... n])
- <grouping_set> :: =
  () - вычисляет общую сумму
  | <Grouping_set_item>
  | (<grouping_set_item> [, ... n])
- <grouping_set_item> :: =
  <Group_by_expression>
  | ROLLUP (<group_by_expression> [, ... n])
  | CUBE (<group_by_expression> [, ... n])

Examples

ИСПОЛЬЗУЙТЕ GROUP BY, чтобы указать количество строк для каждой уникальной записи в данном столбце

Предположим, вы хотите генерировать подсчеты или промежуточные итоги для заданного значения в столбце.

Учитывая эту таблицу, «Вестеросианцы»:

название	GreatHouseAllegience
Arya	застывший
Cercei	Lannister
Myrcella	Lannister
Yara	Грейджый
Catelyn	застывший
Sansa	застывший

Без GROUP BY COUNT просто вернет общее количество строк:

```
SELECT Count(*) Number_of_Westerosians
FROM Westerosians
```

возвращается ...

Number_of_Westerosians
6

Но добавив GROUP BY, мы можем ЗАПИСАТЬ пользователей для каждого значения в данном столбце, чтобы вернуть число людей в данном Великом Доме, скажем:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
```

возвращается ...

жилой дом	Number_of_Westerosians
застывший	3
Грейджый	1
Lannister	2

Обычно для объединения результатов GROUP BY с ORDER BY можно сортировать результаты по самой большой или наименьшей категории:

```
SELECT GreatHouseAllegience House, Count(*) Number_of_Westerosians
FROM Westerosians
GROUP BY GreatHouseAllegience
ORDER BY Number_of_Westerosians Desc
```

возвращается ...

жилой дом	Number_of_Westerosians
застывший	3
Lannister	2
Грейджый	1

Результаты фильтрации GROUP BY с использованием предложения HAVING

Предложение HAVING фильтрует результаты выражения GROUP BY. Примечание. В следующих примерах используется база данных примеров библиотеки .

Примеры:

Верните всех авторов, написавших более одной книги (живой пример).

```
SELECT
  a.Id,
  a.Name,
  COUNT(*) BooksWritten
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
GROUP BY
  a.Id,
  a.Name
HAVING COUNT(*) > 1    -- equals to HAVING BooksWritten > 1
;
```

Верните все книги, в которых есть более трех авторов (живой пример).

```
SELECT
  b.Id,
  b.Title,
  COUNT(*) NumberOfAuthors
FROM BooksAuthors ba
  INNER JOIN Books b ON b.id = ba.bookid
GROUP BY
  b.Id,
  b.Title
HAVING COUNT(*) > 3    -- equals to HAVING NumberOfAuthors > 3
;
```

Пример базовой GROUP BY

Это может быть проще, если вы думаете о GROUP BY как «для каждого» для объяснения. Запрос ниже:

```
SELECT EmpID, SUM (MonthlySalary)
FROM Employee
GROUP BY EmpID
```

говорит:

«Дайте мне сумму MonthlySalary **для каждого** EmpID»

Поэтому, если ваша таблица выглядит так:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Результат:

```
+-+---+
|1|200|
+-+---+
|2|300|
+-+---+
```

Сумма не будет ничего делать, потому что сумма одного числа - это число. С другой стороны, если бы это выглядело так:

```
+-----+-------------+
|EmpID|MonthlySalary|
+-----+-------------+
|1    |200          |
+-----+-------------+
|1    |300          |
+-----+-------------+
|2    |300          |
+-----+-------------+
```

Результат:

```
+-+---+
|1|500|
+-+---+
|2|300|
+-+---+
```

Тогда это будет потому, что есть два EmpID 1 для суммирования.

Агрегация ROLAP (интеллектуальный анализ данных)

Описание

Стандарт SQL предоставляет два дополнительных оператора агрегата. Они используют полиморфное значение «ALL» для обозначения набора всех значений, которые может принимать атрибут. Двумя операторами являются:

- `with data cube` что он предоставляет все возможные комбинации, чем атрибуты аргумента этого предложения.
- `with roll up` что он обеспечивает агрегаты, полученные путем рассмотрения атрибутов в порядке слева направо, по сравнению с тем, как они перечислены в аргументе предложения.

Стандартные версии SQL, поддерживающие эти функции: 1999,2003,2006,2008,2011.

Примеры

Рассмотрим эту таблицу:

питание	марка	Итого
Макаронные изделия	Brand1	100
Макаронные изделия	brand2	250
Пицца	brand2	300

С кубом

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with cube
```

питание	марка	Итого
Макаронные изделия	Brand1	100
Макаронные изделия	brand2	250
Макаронные изделия	BCE	350
Пицца	brand2	300

питание	марка	Итого
Пицца	ВСЕ	300
ВСЕ	Brand1	100
ВСЕ	brand2	550
ВСЕ	ВСЕ	650

С рулоном

```
select Food,Brand,Total_amount
from Table
group by Food,Brand,Total_amount with roll up
```

питание	марка	Итого
Макаронные изделия	Brand1	100
Макаронные изделия	brand2	250
Пицца	brand2	300
Макаронные изделия	ВСЕ	350
Пицца	ВСЕ	300
ВСЕ	ВСЕ	650

Прочитайте ГРУППА ПО онлайн: https://riptutorial.com/ru/sql/topic/627/группа-по

глава 18: ДЕЛО

Вступление

Выражение CASE используется для реализации логики if-then.

Синтаксис

- CASE input_expression
  КОГДА compare1 THEN result1
  [WHEN compare2 THEN result2] ...
  [ELSE resultX]
  КОНЕЦ
- ДЕЛО
  КОГДА условие1 THEN result1
  [КОГДА условие2 THEN result2] ...
  [ELSE resultX]
  КОНЕЦ

замечания

Простое выражение CASE возвращает первый результат, `compareX` значение `compareX` равно `input_expression` .

Выбранное выражение CASE возвращает первый результат, `conditionX` которого истинно.

Examples

Искал CASE в SELECT (Соответствует логическому выражению)

Выбранный CASE возвращает результаты, когда *логическое* выражение TRUE.

(Это отличается от простого случая, который может проверять только на эквивалентность ввода.)

```
SELECT Id, ItemId, Price,
  CASE WHEN Price < 10 THEN 'CHEAP'
       WHEN Price < 20 THEN 'AFFORDABLE'
       ELSE 'EXPENSIVE'
  END AS PriceRating
FROM ItemSales
```

Я бы	ItemId	Цена	PriceRating
1	100	34,5	ДОРОГО
2	145	2,3	ДЕШЕВЫЕ
3	100	34,5	ДОРОГО
4	100	34,5	ДОРОГО
5	145	10	ДОСТУПНЫЙ

Используйте CASE для COUNT, количество строк в столбце соответствует условию.

Случай использования

`CASE` может использоваться совместно с `SUM` для возврата количества только тех элементов, которые соответствуют предварительно определенному условию. (Это похоже на `COUNTIF` в Excel.)

Хитрость заключается в возврате двоичных результатов, указывающих совпадения, поэтому возвращаемые «1» для сопоставления записей могут быть суммированы для подсчета общего количества совпадений.

Учитывая эту таблицу `ItemSales` , скажем, вы хотите узнать общее количество предметов, которые были классифицированы как «Дорогие»:

Я бы	ItemId	Цена	PriceRating
1	100	34,5	ДОРОГО
2	145	2,3	ДЕШЕВЫЕ
3	100	34,5	ДОРОГО
4	100	34,5	ДОРОГО
5	145	10	ДОСТУПНЫЙ

запрос

```
SELECT
    COUNT(Id) AS ItemsCount,
    SUM ( CASE
            WHEN PriceRating = 'Expensive' THEN 1
            ELSE 0
          END
```

```
        ) AS ExpensiveItemsCount
FROM ItemSales
```

Результаты:

ItemsCount	ExpensiveItemsCount
5	3

Альтернатива:

```
SELECT
    COUNT(Id) as ItemsCount,
    SUM (
        CASE PriceRating
            WHEN 'Expensive' THEN 1
            ELSE 0
        END
       ) AS ExpensiveItemsCount
FROM ItemSales
```

Сокращенный CASE в SELECT

Сокращенный вариант `CASE` оценивает выражение (обычно столбца) относительно ряда значений. Этот вариант немного короче и сохраняет повторение оцененного выражения снова и снова. Предложение `ELSE` все еще можно использовать:

```
SELECT Id, ItemId, Price,
  CASE Price WHEN 5  THEN 'CHEAP'
             WHEN 15 THEN 'AFFORDABLE'
             ELSE         'EXPENSIVE'
  END as PriceRating
FROM ItemSales
```

Слово предостережения. Важно понимать, что при использовании короткого варианта весь оператор оценивается в каждом `WHEN` . Поэтому следующее утверждение:

```
SELECT
    CASE ABS(CHECKSUM(NEWID())) % 4
        WHEN 0 THEN 'Dr'
        WHEN 1 THEN 'Master'
        WHEN 2 THEN 'Mr'
        WHEN 3 THEN 'Mrs'
    END
```

может дать результат `NULL` . Это происходит потому, что каждый `WHEN NEWID()` снова вызывается с новым результатом. Эквивалент:

```
SELECT
    CASE
```

```
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 0 THEN 'Dr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 1 THEN 'Master'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 2 THEN 'Mr'
        WHEN ABS(CHECKSUM(NEWID())) % 4 = 3 THEN 'Mrs'
    END
```

Поэтому он может пропустить все случаи `WHEN` и результат как `NULL` .

CASE в предложении ORDER BY

Мы можем использовать 1,2,3 .. для определения типа порядка:

```
SELECT * FROM DEPT
ORDER BY
CASE DEPARTMENT
      WHEN 'MARKETING' THEN  1
      WHEN 'SALES' THEN 2
      WHEN 'RESEARCH' THEN 3
      WHEN 'INNOVATION' THEN 4
      ELSE        5
      END,
      CITY
```

Я БЫ	ОБЛАСТЬ, КРАЙ	ГОРОД	ОТДЕЛ	EMPLOYEES_NUMBER
12	Новая Англия	Бостон	МАРКЕТИНГ	9
15	запад	Сан-Франциско	МАРКЕТИНГ	12
9	Средний Запад	Чикаго	ПРОДАЖИ	8
14	Mid-Atlantic	Нью-Йорк	ПРОДАЖИ	12
5	запад	Лос-Анджелес	ИССЛЕДОВАНИЕ	11
10	Mid-Atlantic	Филадельфия	ИССЛЕДОВАНИЕ	13
4	Средний Запад	Чикаго	ИННОВАЦИИ	11
2	Средний Запад	Детройт	ОТДЕЛ КАДРОВ	9

Использование CASE в UPDATE

образец повышения цен:

```
UPDATE ItemPrice
SET Price = Price *
  CASE ItemId
```

```
    WHEN 1 THEN 1.05
    WHEN 2 THEN 1.10
    WHEN 3 THEN 1.15
    ELSE 1.00
  END
```

Использование CASE для значений NULL, заказанных последними

таким образом, «0», представляющий известные значения, занимает первое место, «1», представляющее значения NULL, сортируется по последнему:

```
SELECT ID
      ,REGION
      ,CITY
      ,DEPARTMENT
      ,EMPLOYEES_NUMBER
  FROM DEPT
  ORDER BY
  CASE WHEN REGION IS NULL THEN 1
  ELSE 0
  END,
  REGION
```

Я БЫ	ОБЛАСТЬ, КРАЙ	ГОРОД	ОТДЕЛ	EMPLOYEES_NUMBER
10	Mid-Atlantic	Филадельфия	ИССЛЕДОВАНИЕ	13
14	Mid-Atlantic	Нью-Йорк	ПРОДАЖИ	12
9	Средний Запад	Чикаго	ПРОДАЖИ	8
12	Новая Англия	Бостон	МАРКЕТИНГ	9
5	запад	Лос-Анджелес	ИССЛЕДОВАНИЕ	11
15	НОЛЬ	Сан-Франциско	МАРКЕТИНГ	12
4	НОЛЬ	Чикаго	ИННОВАЦИИ	11
2	НОЛЬ	Детройт	ОТДЕЛ КАДРОВ	9

CASE в предложении ORDER BY для сортировки записей по наименьшему значению из 2 столбцов

Представьте, что вам нужны записи сортировки по наименьшему значению одного из двух столбцов. Некоторые базы данных могут использовать неагрегированную функцию `MIN()` или `LEAST()` для этого (`... ORDER BY MIN(Date1, Date2)`), но в стандартном SQL вы должны

использовать выражение `CASE` .

Выражение `CASE` в нижеследующем `Date1 Date2` столбцам `Date1` и `Date2` , проверяет, какой столбец имеет меньшее значение, и сортирует записи в зависимости от этого значения.

Пример данных

Я бы	Date1	Дата2
1	2017-01-01	2017-01-31
2	2017-01-31	2017-01-03
3	2017-01-31	2017-01-02
4	2017-01-06	2017-01-31
5	2017-01-31	2017-01-05
6	2017-01-04	2017-01-31

запрос

```
SELECT Id, Date1, Date2
FROM YourTable
ORDER BY CASE
           WHEN COALESCE(Date1, '1753-01-01') < COALESCE(Date2, '1753-01-01') THEN Date1
           ELSE Date2
         END
```

Результаты

Я бы	Date1	Дата2
1	**2017-01-01**	2017-01-31
3	2017-01-31	**2017-01-02**
2	2017-01-31	**2017-01-03**
6	**2017-01-04**	2017-01-31
5	2017-01-31	**2017-01-05**
4	**2017-01-06**	2017-01-31

объяснение

Поскольку сначала вы видите строку с `Id = 1` , потому что `Date1` имеет самую низкую запись из всей таблицы `2017-01-01` , строка, где `Id = 3` является второй, потому что `Date2` равна `2017-01-02` что является вторым самым низким значением из таблицы и так далее.

Таким образом, мы отсортировали записи с `2017-01-01` по `2017-01-06` возрастанию и не заботились о `Date1` `Date2` являются эти столбцы `Date1` или `Date2` .

Прочитайте ДЕЛО онлайн: https://riptutorial.com/ru/sql/topic/456/дело

глава 19: Дизайн стола

замечания

Открытый университет (1999) Реляционные системы баз данных: блок 2 Реляционная теория, Милтон Кейнс, Открытый университет.

Examples

Свойства хорошо спроектированной таблицы.

Истинная реляционная база данных должна выходить за рамки передачи данных в несколько таблиц и написания некоторых операторов SQL, чтобы вытащить эти данные. В лучшем случае плохо спроектированная структура таблицы замедлит выполнение запросов и может сделать невозможным работу базы данных по назначению.

Таблица базы данных не должна рассматриваться как просто другая таблица; он должен следовать ряду правил, которые считаются по-настоящему реляционными. В академическом смысле это обозначается как «отношение», чтобы сделать различие.

Пять правил реляционной таблицы:

1. Каждое значение является *атомарным* ; значение в каждом поле в каждой строке должно быть единственным значением.
2. Каждое поле содержит значения, относящиеся к одному типу данных.
3. Каждый заголовок поля имеет уникальное имя.
4. Каждая строка в таблице должна иметь как минимум одно значение, которое делает его уникальным среди других записей в таблице.
5. Порядок строк и столбцов не имеет значения.

Таблица, соответствующая пяти правилам:

Я бы	название	дата рождения	Менеджер
1	Фред	11/02/1971	3
2	Фред	11/02/1971	3
3	Сью	08/07/1975	2

- Правило 1: Каждое значение является атомарным. `Id` , `Name` , `DOB` и `Manager` содержат только одно значение.
- Правило 2: `Id` содержит только целые числа, `Name` содержит текст (мы можем

добавить, что это текст из четырех символов или меньше), `DOB` содержит даты действительного типа, а `Manager` содержит целые числа (мы могли бы добавить, что соответствует полю первичного ключа в менеджерах Таблица).
- Правило 3: `Id` , `Name` , `DOB` и `Manager` являются уникальными именами заголовков в таблице.
- Правило 4: включение поля `Id` гарантирует, что каждая запись отличается от любой другой записи в таблице.

Плохо спроектированная таблица:

Я бы	название	дата рождения	название
1	Фред	11/02/1971	3
1	Фред	11/02/1971	3
3	Сью	Пятница, 18 июля 1975 года	2, 1

- Правило 1: второе поле имени содержит два значения - 2 и 1.
- Правило 2: Поле DOB содержит даты и текст.
- Правило 3: Есть два поля, называемых «имя».
- Правило 4: Первая и вторая записи абсолютно одинаковы.
- Правило 5: Это правило не нарушается.

Прочитайте Дизайн стола онлайн: https://riptutorial.com/ru/sql/topic/2515/дизайн-стола

глава 20: Идентификатор

Вступление

В этом разделе описываются идентификаторы, т. Е. Правила синтаксиса для имен таблиц, столбцов и других объектов базы данных.

При необходимости примеры должны охватывать варианты, используемые различными реализациями SQL, или идентифицировать реализацию SQL в примере.

Examples

Идентификаторы без кавычек

Идентификаторы без кавычек могут использовать буквы (`a` - `z`), цифры (`0` - `9`) и подчеркивание (`_`) и должны начинаться с буквы.

В зависимости от реализации SQL и / или параметров базы данных могут допускаться другие символы, некоторые даже в качестве первого символа, например

- MS SQL: `@` , `$` , `#` и другие буквы Unicode *(источник)*
- MySQL: `$` *(источник)*
- Oracle: `$` , `#` и другие буквы из набора символов базы данных *(источник)*
- PostgreSQL: `$` и другие символы Unicode *(источник)*

Идентификаторы без кавычек нечувствительны к регистру. То, как это обрабатывается, во многом зависит от реализации SQL:

- MS SQL: сохранение в случае, чувствительность, определяемая набором символов базы данных, поэтому может учитываться регистр.
- MySQL: сохранение событий, чувствительность зависит от настроек базы данных и базовой файловой системы.
- Oracle: преобразован в верхний регистр, а затем обрабатывается как цитируемый идентификатор.
- PostgreSQL: преобразован в нижний регистр, а затем обрабатывается как котируемый идентификатор.
- SQLite: сохранение случая; нечувствительность к регистру только для символов ASCII.

Прочитайте Идентификатор онлайн: https://riptutorial.com/ru/sql/topic/9677/идентификатор

глава 21: Индексы

Вступление

Индексы представляют собой структуру данных, содержащую указатели на содержимое таблицы, упорядоченной в определенном порядке, чтобы помочь оптимизировать запросы базы данных. Они похожи на индекс книги, где страницы (строки таблицы) индексируются по их номеру страницы.

Существует несколько типов индексов и могут быть созданы на столе. Когда индекс существует в столбцах, используемых в предложении WHERE запроса, в предложении JOIN или в предложении ORDER BY, он может существенно повысить производительность запросов.

замечания

Индексы являются способом ускорения запросов на чтение путем сортировки строк таблицы в соответствии с столбцом.

Эффект индекса не заметен для небольших баз данных, таких как пример, но если имеется большое количество строк, это может значительно повысить производительность. Вместо проверки каждой строки таблицы сервер может выполнять двоичный поиск по индексу.

Компромисс для создания индекса - скорость записи и размер базы данных. Хранение индекса занимает пробел. Кроме того, каждый раз, когда выполняется INSERT или обновляется столбец, индекс должен обновляться. Это не такая дорогостоящая операция, как сканирование всей таблицы в запросе SELECT, но это все еще нужно иметь в виду.

Examples

Создание индекса

```
CREATE INDEX ix_cars_employee_id ON Cars (EmployeeId);
```

Это создаст индекс для столбца *EmployeeId* в таблице *Cars* . Этот индекс улучшит скорость запросов, запрашивающих сервер для сортировки или выбора значений в *EmployeeId* , например:

```
SELECT * FROM Cars WHERE EmployeeId = 1
```

Индекс может содержать более 1 столбца, как в следующем;

```
CREATE INDEX ix_cars_e_c_o_ids ON Cars (EmployeeId, CarId, OwnerId);
```

В этом случае индекс был бы полезен для запросов, запрашивающих сортировку или выбор всех включенных столбцов, если набор условий упорядочен таким же образом. Это означает, что при извлечении данных он может найти строки, которые будут извлекаться с использованием индекса, а не просматривать всю таблицу.

Например, в следующем случае будет использоваться второй индекс;

```
SELECT * FROM Cars WHERE EmployeeId = 1 Order by CarId DESC
```

Однако, если порядок отличается, индекс не имеет таких же преимуществ, как в следующем;

```
SELECT * FROM Cars WHERE OwnerId = 17 Order by CarId DESC
```

Индекс не так полезен, потому что база данных должна извлекать весь индекс по всем значениям EmployeeId и CarID, чтобы определить, какие элементы имеют `OwnerId = 17` .

(Индекс все еще может быть использован, возможно, оптимизатор запросов обнаруживает, что извлечение индекса и фильтрация на `OwnerId` , а затем получение только необходимых строк выполняется быстрее, чем извлечение полной таблицы, особенно если таблица большая.)

Кластерные, уникальные и отсортированные индексы

Индексы могут иметь несколько характеристик, которые могут быть установлены либо при создании, либо путем изменения существующих индексов.

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

Вышеупомянутый оператор SQL создает новый кластерный индекс для Employees. Кластеризованные индексы - это индексы, которые определяют фактическую структуру таблицы; сама таблица сортируется в соответствии со структурой индекса. Это означает, что на таблице может быть не более одного кластеризованного индекса. Если кластерный индекс уже существует в таблице, вышеуказанный оператор не будет выполнен. (Таблицы без кластеризованных индексов также называются кучами.)

```
CREATE UNIQUE INDEX uq_customers_email ON Customers(Email);
```

Это создаст уникальный индекс для столбца *Email* в таблице *Customers* . Этот индекс, наряду с ускорением запросов, таких как нормальный индекс, также заставит каждый адрес электронной почты в этом столбце быть уникальным. Если строка вставлена или обновлена с нестандартным значением *электронной почты* , вставка или обновление по умолчанию будут неудачными.

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

Это создает индекс для клиентов, который также создает ограничение таблицы, в котором EmployeeID должен быть уникальным. (Это не удастся, если столбец в настоящее время не уникален - в этом случае, если есть сотрудники, которые имеют идентификатор).

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

Это создает индекс, который сортируется в порядке убывания. По умолчанию индексы (по крайней мере, на сервере MSSQL) возрастают, но их можно изменить.

Вставка с уникальным индексом

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1;
```

Это приведет к сбою, если в столбце *Электронная почта Клиентов* установлен уникальный индекс. Однако для этого случая можно определить альтернативное поведение:

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1 ON DUPLICATE KEY;
```

SAP ASE: индекс падения

Эта команда уменьшит индекс в таблице. Он работает на сервере `SAP ASE` .

Синтаксис:

```
DROP INDEX [table name].[index name]
```

Пример:

```
DROP INDEX Cars.index_1
```

Сортированный указатель

Если вы используете индекс, который сортируется так, как вы его извлекли, `SELECT` не будет выполнять дополнительную сортировку при поиске.

```
CREATE INDEX ix_scoreboard_score ON scoreboard (score DESC);
```

Когда вы выполняете запрос

```
SELECT * FROM scoreboard ORDER BY score DESC;
```

Система базы данных не будет выполнять дополнительную сортировку, поскольку она

может выполнять поиск по индексу в этом порядке.

Удаление индекса, или отключение и восстановление его

```
DROP INDEX ix_cars_employee_id ON Cars;
```

Мы можем использовать команду `DROP` для удаления нашего индекса. В этом примере мы будем `DROP` индексом *ix_cars_employee_id* на столе *Cars* .

Это полностью исключает индекс, и если индекс кластеризуется, удаляется любая кластеризация. Он не может быть перестроен без воссоздания индекса, который может быть медленным и дорогостоящим. В качестве альтернативы индекс можно отключить:

```
ALTER INDEX ix_cars_employee_id ON Cars DISABLE;
```

Это позволяет таблице сохранить структуру вместе с метаданными об индексе.

Критически это сохраняет статистику индекса, так что можно легко оценить изменение. Если это оправдано, индекс затем может быть перестроен, а не полностью восстановлен;

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

Уникальный индекс, который позволяет NULLS

```
CREATE UNIQUE INDEX idx_license_id
   ON Person(DrivingLicenseID) WHERE DrivingLicenseID IS NOT NULL
GO
```

Эта схема допускает отношения 0..1 - люди могут иметь ноль или одну водительскую лицензию, и каждая лицензия может принадлежать только одному человеку

Перестроить индекс

С течением времени индексы B-Tree могут стать фрагментированными из-за обновления / удаления / вставки данных. В терминологии SQLServer у нас может быть внутренняя (индексная страница, наполовину пустая) и внешняя (логический порядок страниц не соответствует физическому порядку). Индекс перестройки очень похож на падение и воссоздание.

Мы можем перестроить индекс с помощью

```
ALTER INDEX index_name REBUILD;
```

По умолчанию индекс перестройки - это автономная операция, которая блокирует таблицу и предотвращает DML, но многие RDBMS позволяют осуществлять онлайн-перестройку.

Кроме того, некоторые поставщики БД предлагают альтернативы перестройке индекса, такие как `REORGANIZE` (SQLServer) или `COALESCE` / `SHRINK SPACE` (Oracle).

Кластерный индекс

При использовании кластерного индекса строки таблицы сортируются по столбцу, к которому применяется кластерный индекс. Поэтому в таблице может быть только один кластерный индекс, потому что вы не можете упорядочить таблицу двумя разными столбцами.

Как правило, лучше использовать кластерный индекс при выполнении чтения в больших таблицах данных. Домен кластеризованного индекса заключается в том, чтобы писать в таблицу, и данные необходимо реорганизовать (прибегать).

Пример создания кластерного индекса в таблице Сотрудники в столбце Employee_Surname:

```
CREATE CLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

Некомпилированный индекс

Некластеризованные индексы хранятся отдельно от таблицы. Каждый индекс в этой структуре содержит указатель на строку в таблице, которую он представляет.

Эти указатели называются локаторами строк. Структура локатора строк зависит от того, хранятся ли страницы данных в куче или кластеризованной таблице. Для кучи указатель строки является указателем на строку. Для кластерной таблицы локатор строк представляет собой кластерный индексный ключ.

Пример создания некластеризованного индекса в таблице Сотрудники и столбец Employee_Surname:

```
CREATE NONCLUSTERED INDEX ix_employees_name ON Employees(Employee_Surname);
```

В таблице может быть несколько некластеризованных индексов. Операции чтения, как правило, медленнее с некластеризованными индексами, чем с кластеризованными индексами, так как вам нужно сначала индексировать, а не таблицу. Однако никаких ограничений в операциях записи нет.

Частичный или отфильтрованный указатель

SQL Server и SQLite позволяют создавать индексы, которые содержат не только подмножество столбцов, но и подмножество строк.

Рассмотрим постоянное возрастающее количество заказов с `order_state_id` равным

завершенному (2), и стабильное количество заказов с `order_state_id equal` (1).

Если ваш бизнес использует такие запросы:

```
SELECT id, comment
  FROM orders
 WHERE order_state_id =  1
   AND product_id = @some_value;
```

Частичная индексация позволяет вам ограничить индекс, включая только незавершенные заказы:

```
CREATE INDEX Started_Orders
          ON orders(product_id)
       WHERE order_state_id = 1;
```

Этот индекс будет меньше, чем нефильтрованный индекс, что экономит место и снижает стоимость обновления индекса.

Прочитайте Индексы онлайн: https://riptutorial.com/ru/sql/topic/344/индексы

глава 22: Иностранные ключи

Examples

Создание таблицы с внешним ключом

В этом примере у нас есть существующая таблица `SuperHeros` .

В этой таблице содержится `ID` первичного ключа.

Мы добавим новую таблицу, чтобы сохранить полномочия каждого супергероя:

```
CREATE TABLE HeroPowers
(
    ID int NOT NULL PRIMARY KEY,
    Name nvarchar(MAX) NOT NULL,
    HeroId int REFERENCES SuperHeros(ID)
)
```

Столбец `HeroId` является **внешним ключом** к таблице `SuperHeros` .

Внешние ключи

Ограничения внешних ключей обеспечивают целостность данных, применяя эти значения в одной таблице, чтобы соответствовать значениям в другой таблице.

Примером того, где требуется внешний ключ, является: В университете курс должен принадлежать отделу. Код для этого сценария:

```
CREATE TABLE Department (
    Dept_Code        CHAR (5)     PRIMARY KEY,
    Dept_Name        VARCHAR (20) UNIQUE
);
```

Вставьте значения со следующим утверждением:

```
INSERT INTO Department VALUES ('CS205', 'Computer Science');
```

Следующая таблица будет содержать информацию о предметах, предлагаемых отраслью информатики:

```
CREATE TABLE Programming_Courses (
    Dept_Code       CHAR(5),
    Prg_Code        CHAR(9) PRIMARY KEY,
    Prg_Name        VARCHAR (50) UNIQUE,
    FOREIGN KEY (Dept_Code) References Department(Dept_Code)
);
```

(Тип данных внешнего ключа должен соответствовать типу данных ссылочного ключа.)

Ограничение внешнего ключа в столбце `Dept_Code` допускает значения только в том случае, если они уже существуют в ссылочной таблице, `Department` . Это означает, что если вы попытаетесь вставить следующие значения:

```
INSERT INTO Programming_Courses Values ('CS300', 'FDB-DB001', 'Database Systems');
```

база данных вызовет ошибку нарушения внешнего ключа, потому что `CS300` не существует в таблице `Department` . Но когда вы пробуете ключевое значение, которое существует:

```
INSERT INTO Programming_Courses VALUES ('CS205', 'FDB-DB001', 'Database Systems');
INSERT INTO Programming_Courses VALUES ('CS205', 'DB2-DB002', 'Database Systems II');
```

то база данных допускает эти значения.

Несколько советов по использованию внешних ключей

- Внешний ключ должен ссылаться на UNIQUE (или PRIMARY) ключ в родительской таблице.
- Ввод значения NULL в столбце «Внешний ключ» не вызывает ошибки.
- Ограничения внешнего ключа могут ссылаться на таблицы в одной базе данных.
- Ограничения внешнего ключа могут ссылаться на другой столбец в той же таблице (самореклама).

Прочитайте Иностранные ключи онлайн: https://riptutorial.com/ru/sql/topic/1533/иностранные-ключи

глава 23: Информационная схема

Examples

Поиск базовой информации

Одним из наиболее полезных запросов для конечных пользователей больших РСУБД является поиск информационной схемы.

Такой запрос позволяет пользователям быстро находить таблицы базы данных, содержащие столбцы, представляющие интерес, например, при попытке связать данные из 2 таблиц косвенно через третью таблицу без каких-либо знаний о том, какие таблицы могут содержать ключи или другие полезные столбцы, общие с целевыми таблицами ,

Используя T-SQL для этого примера, можно запросить информационную схему базы данных следующим образом:

```
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE COLUMN_NAME LIKE '%Institution%'
```

Результат содержит список совпадающих столбцов, имена их таблиц и другую полезную информацию.

Прочитайте Информационная схема онлайн: https://riptutorial.com/ru/sql/topic/3151/информационная-схема

глава 24: Каскадное удаление

Examples

УДАЛИТЬ КАСКАД

Предположим, у вас есть приложение, которое администрирует комнаты.
Предположим также, что ваше приложение работает на основе клиента (арендатора).
У вас несколько клиентов.
Таким образом, ваша база данных будет содержать одну таблицу для клиентов и одну для комнат.

Теперь у каждого клиента есть N номеров.

Это должно означать, что у вас есть внешний ключ на вашем столе комнаты, ссылаясь на таблицу клиентов.

```
ALTER TABLE dbo.T_Room  WITH CHECK ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
GO
```

Предполагая, что клиент переходит к другому программному обеспечению, вам придется удалить его данные в своем программном обеспечении. Но если вы это сделаете

```
DELETE FROM T_Client WHERE CLI_ID = x
```

Затем вы получите нарушение внешнего ключа, потому что вы не можете удалить клиента, когда у него все еще есть комнаты.

Теперь вы должны написать код в своем приложении, который удаляет комнаты клиента до того, как он удалит клиента. Предположим далее, что в будущем в вашей базе данных будет добавлено гораздо больше зависимостей от внешнего ключа, поскольку функциональность вашего приложения расширяется. Какой ужас. Для каждой модификации в вашей базе данных вам придется адаптировать код приложения в N местах. Возможно, вам придется адаптировать код и в других приложениях (например, интерфейсы к другим системам).

Есть лучшее решение, чем делать это в вашем коде.
Вы можете просто добавить `ON DELETE CASCADE` к вашему внешнему ключу.

```
ALTER TABLE dbo.T_Room  -- WITH CHECK -- SQL-Server can specify WITH CHECK/WITH NOCHECK
ADD  CONSTRAINT FK_T_Room_T_Client FOREIGN KEY(RM_CLI_ID)
REFERENCES dbo.T_Client (CLI_ID)
ON DELETE CASCADE
```

Теперь вы можете сказать

```
DELETE FROM T_Client WHERE CLI_ID = x
```

и номера автоматически удаляются при удалении клиента.
Проблема решена - без изменений кода приложения.

Одно слово предостережения: в Microsoft SQL-Server это не сработает, если у вас есть таблица, которая ссылается сама. Поэтому, если вы попытаетесь определить каскад delete в рекурсивной древовидной структуре, например:

```
IF NOT EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation]  WITH CHECK ADD  CONSTRAINT
[FK_T_FMS_Navigation_T_FMS_Navigation] FOREIGN KEY([NA_NA_UID])
REFERENCES [dbo].[T_FMS_Navigation] ([NA_UID])
ON DELETE CASCADE
GO

IF  EXISTS (SELECT * FROM sys.foreign_keys WHERE object_id =
OBJECT_ID(N'[dbo].[FK_T_FMS_Navigation_T_FMS_Navigation]') AND parent_object_id =
OBJECT_ID(N'[dbo].[T_FMS_Navigation]'))
ALTER TABLE [dbo].[T_FMS_Navigation] CHECK CONSTRAINT [FK_T_FMS_Navigation_T_FMS_Navigation]
GO
```

это не сработает, потому что Microsoft-SQL-сервер не позволяет вам установить внешний ключ с `ON DELETE CASCADE` на рекурсивную древовидную структуру. Одной из причин этого является то, что дерево возможно циклическое, и это может привести к тупиковой ситуации.

PostgreSQL, с другой стороны, может это сделать;
необходимо, чтобы дерево было нециклическим.
Если дерево циклически, вы получите ошибку времени выполнения.
В этом случае вам просто нужно реализовать функцию удаления самостоятельно.

Предупреждение:
Это означает, что вы больше не можете просто удалять и повторно вставлять таблицу клиентов, потому что, если вы сделаете это, он удалит все записи в «T_Room» ... (без дополнительных обновлений без дельта)

Прочитайте Каскадное удаление онлайн: https://riptutorial.com/ru/sql/topic/3518/каскадное-удаление

глава 25: Комментарии

Examples

Однострочные комментарии

Одиночным комментариям предшествует `--` и идите до конца строки:

```
SELECT *
FROM Employees -- this is a comment
WHERE FName = 'John'
```

Многострочные комментарии

Комментарии к нескольким линиям заключены в `/* ... */` :

```
/* This query
   returns all employees */
SELECT *
FROM Employees
```

Также можно вставить такой комментарий в середину строки:

```
SELECT /* all columns: */ *
FROM Employees
```

Прочитайте Комментарии онлайн: https://riptutorial.com/ru/sql/topic/1597/комментарии

глава 26: крест применяется, наружный применяется

Examples

Основы CROSS APPLY и OUTER APPLY

Применить будет использоваться, когда функция table value в правильном выражении.

создайте таблицу отдела для хранения информации о департаментах. Затем создайте таблицу Employee, в которой хранятся сведения о сотрудниках. Обратите внимание: каждый сотрудник принадлежит отделу, поэтому таблица Employee имеет ссылочную целостность с таблицей Департамента.

Первый запрос выбирает данные из таблицы Department и использует CROSS APPLY для оценки таблицы Employee для каждой записи таблицы Department. Второй запрос просто присоединяется к таблице Департамента с таблицей Employee, и все соответствующие записи создаются.

```
SELECT *
FROM Department D
CROSS APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
INNER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
```

Если вы посмотрите на результаты, которые они произвели, это точно такой же результат; Чем он отличается от JOIN и как он помогает в написании более эффективных запросов.

Первый запрос в сценарии # 2 выбирает данные из таблицы Department и использует OUTER APPLY для оценки таблицы Employee для каждой записи таблицы Department. Для тех строк, для которых нет совпадений в таблице Employee, эти строки содержат значения NULL, как вы можете видеть в случае строк 5 и 6. Второй запрос просто использует LEFT OUTER JOIN между таблицей Департамента и таблицей Employee. Как и ожидалось, запрос возвращает все строки из таблицы Department; даже для тех строк, для которых нет совпадений в таблице Employee.

```
SELECT *
FROM Department D
```

```
OUTER APPLY (
    SELECT *
    FROM Employee E
    WHERE E.DepartmentID = D.DepartmentID
) A
GO
SELECT *
FROM Department D
LEFT OUTER JOIN Employee E
  ON D.DepartmentID = E.DepartmentID
GO
```

Несмотря на то, что вышеупомянутые два запроса возвращают одну и ту же информацию, план выполнения будет бит другим. Но стоить разумно, не будет большой разницы.

Теперь настало время посмотреть, где действительно нужен оператор APPLY. В сценарии №3 я создаю табличную функцию, которая принимает параметр DepartmentID как параметр и возвращает всех сотрудников, принадлежащих этому отделу. Следующий запрос выбирает данные из таблицы Department и использует CROSS APPLY для объединения с созданной нами функцией. Он передает идентификатор отдела для каждой строки из выражения внешней таблицы (в нашем случае таблица Департамента) и оценивает функцию для каждой строки, подобной коррелированному подзапросу. Следующий запрос использует OUTER APPLY вместо CROSS APPLY и, следовательно, в отличие от CROSS APPLY, который возвращает только коррелированные данные, OUTER APPLY также возвращает некоррелированные данные, помещая NULL в отсутствующие столбцы.

```
CREATE FUNCTION dbo.fn_GetAllEmployeeOfADepartment (@DeptID AS int)
RETURNS TABLE
AS
  RETURN
  (
  SELECT
    *
  FROM Employee E
  WHERE E.DepartmentID = @DeptID
  )
GO
SELECT
  *
FROM Department D
CROSS APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
SELECT
  *
FROM Department D
OUTER APPLY dbo.fn_GetAllEmployeeOfADepartment(D.DepartmentID)
GO
```

Итак, теперь, если вам интересно, можем ли мы использовать простое соединение вместо вышеуказанных запросов? Тогда ответ НЕТ, если вы замените CROSS / OUTER APPLY в вышеуказанных запросах INNER JOIN / LEFT OUTER JOIN, укажите предложение ON (что-

то как 1 = 1) и запустите запрос, вы получите «Идентификатор с несколькими частями», D.DepartmentID "не может быть связан". ошибка. Это связано с тем, что с JOINs контекст выполнения внешнего запроса отличается от контекста выполнения функции (или производной таблицы), и вы не можете привязать значение / переменную от внешнего запроса к функции в качестве параметра. Следовательно, для таких запросов требуется оператор APPLY.

Прочитайте крест применяется, наружный применяется онлайн:
https://riptutorial.com/ru/sql/topic/2516/крест-применяется--наружный-применяется

глава 27: КРОМЕ

замечания

`EXCEPT` возвращает любые различные значения из набора данных слева от оператора EXCEPT, которые также не возвращаются из нужного набора данных.

Examples

Выберите набор данных, кроме тех значений, которые находятся в этом другом наборе данных

```
--dataset schemas must be identical
SELECT 'Data1' as 'Column' UNION ALL
SELECT 'Data2' as 'Column' UNION ALL
SELECT 'Data3' as 'Column' UNION ALL
SELECT 'Data4' as 'Column' UNION ALL
SELECT 'Data5' as 'Column'
EXCEPT
SELECT 'Data3' as 'Column'
--Returns Data1, Data2, Data4, and Data5
```

Прочитайте КРОМЕ онлайн: https://riptutorial.com/ru/sql/topic/4082/кроме

глава 28: Материализованные виды

Вступление

Материализованное представление представляет собой представление, результаты которого физически хранятся и должны периодически обновляться, чтобы оставаться текущим. Поэтому они полезны для хранения результатов сложных, длительных запросов, когда результаты в реальном времени не требуются. Материализованные представления могут быть созданы в Oracle и PostgreSQL. Другие системы баз данных предлагают аналогичные функции, такие как индексированные представления SQL Server или материализованные таблицы запросов DB2.

Examples

Пример PostgreSQL

```
CREATE TABLE mytable (number INT);
INSERT INTO mytable VALUES (1);

CREATE MATERIALIZED VIEW myview AS SELECT * FROM mytable;

SELECT * FROM myview;
 number
--------
      1
(1 row)

INSERT INTO mytable VALUES(2);

SELECT * FROM myview;
 number
--------
      1
(1 row)

REFRESH MATERIALIZED VIEW myview;

SELECT * FROM myview;
 number
--------
      1
      2
(2 rows)
```

Прочитайте Материализованные виды онлайн: https://riptutorial.com/ru/sql/topic/8367/материализованные-виды

глава 29: НОЛЬ

Вступление

`NULL` в SQL, а также программирование в целом означает буквально «ничего». В SQL это легче понять как «отсутствие какой-либо ценности».

Важно отличить его от кажущихся пустых значений, таких как пустая строка `''` или число `0` , ни один из которых на самом деле не `NULL` .

Также важно быть осторожным, чтобы не заключать `NULL` в кавычки, например `'NULL'` , который разрешен в столбцах, которые принимают текст, но не является `NULL` и может вызывать ошибки и неправильные наборы данных.

Examples

Фильтрация для NULL в запросах

Синтаксис фильтрации для `NULL` (т.е. отсутствия значения) в блоках `WHERE` несколько отличается от фильтрации для определенных значений.

```
SELECT * FROM Employees WHERE ManagerId IS NULL ;
SELECT * FROM Employees WHERE ManagerId IS NOT NULL ;
```

Обратите внимание, что поскольку `NULL` не равен чему-либо, даже самому себе, используя операторы равенства `= NULL` или `<> NULL` (или `!= NULL`), всегда будет давать значение правды `UNKNOWN` которое будет отклонено `WHERE` .

`WHERE` фильтрует все строки, что условие `FALSE` или `UKNOWN` и сохраняет только строки, что условие `TRUE` .

Столбцы в таблицах

При создании таблиц можно объявить столбец как nullable или non-nullable.

```
CREATE TABLE MyTable
(
    MyCol1 INT NOT NULL, -- non-nullable
    MyCol2 INT NULL      -- nullable
) ;
```

По умолчанию каждый столбец (кроме ограничений в отношении первичного ключа) имеет значение NULL, если мы явно не устанавливаем ограничение `NOT NULL` .

Попытка присвоить `NULL` столбцу, не допускающему null, приведет к ошибке.

```
INSERT INTO MyTable (MyCol1, MyCol2) VALUES (1, NULL) ;  -- works fine

INSERT INTO MyTable (MyCol1, MyCol2) VALUES (NULL, 2) ;
        -- cannot insert
        -- the value NULL into column 'MyCol1', table 'MyTable';
        -- column does not allow nulls. INSERT fails.
```

Обновление полей до NULL

Установка поля в `NULL` работает точно так же, как и с любым другим значением:

```
UPDATE Employees
SET ManagerId = NULL
WHERE Id = 4
```

Вставка строк с полями NULL

Например, добавив сотрудника в номер таблицы « Сотрудники без номера телефона» и «нет менеджера»:

```
INSERT INTO Employees
    (Id, FName, LName, PhoneNumber, ManagerId, DepartmentId, Salary, HireDate)
VALUES
    (5, 'Jane', 'Doe', NULL, NULL, 2, 800, '2016-07-22') ;
```

Прочитайте НОЛЬ онлайн: https://riptutorial.com/ru/sql/topic/3421/ноль

глава 30: Номер строки

Синтаксис

- ROW_NUMBER ()
- OVER ([PARTITION BY value_expression, ... [n]] order_by_clause)

Examples

Номера строк без разделов

Включите номер строки в соответствии с указанным порядком.

```
SELECT
  ROW_NUMBER() OVER(ORDER BY Fname ASC) AS RowNumber,
  Fname,
  LName
FROM Employees
```

Номера строк с разделами

Использует критерии раздела для группировки нумерации строк в соответствии с ним.

```
SELECT
  ROW_NUMBER() OVER(PARTITION BY DepartmentId ORDER BY DepartmentId ASC) AS RowNumber,
  DepartmentId, Fname, LName
FROM Employees
```

Удалить все, кроме последней записи (от 1 до многих)

```
WITH cte AS (
  SELECT ProjectID,
         ROW_NUMBER() OVER (PARTITION BY ProjectID ORDER BY InsertDate DESC) AS rn
  FROM ProjectNotes
)
DELETE FROM cte WHERE rn > 1;
```

Прочитайте Номер строки онлайн: https://riptutorial.com/ru/sql/topic/1977/номер-строки

глава 31: ОБНОВИТЬ

Синтаксис

- *Таблица* UPDATE
  SET *column_name* = *значение* , *column_name2* = *значение_2* , ..., *column_name_n* = *значение_n*
  Условие WHERE (*логическое* условие *operator_n*)

Examples

Обновление всех строк

В этом примере таблица Cars используется в примерах баз данных.

```
UPDATE Cars
SET Status = 'READY'
```

Этот оператор установит столбец «status» всех строк таблицы «Cars» в «READY», потому что у него нет `WHERE` для фильтрации набора строк.

Обновление заданных строк

В этом примере таблица Cars используется в примерах баз данных.

```
UPDATE
    Cars
SET
    Status = 'READY'
WHERE
    Id = 4
```

Это утверждение установит статус строки «Автомобили» с id 4 на «READY».

Предложение `WHERE` содержит логическое выражение, которое оценивается для каждой строки. Если строка соответствует критериям, ее значение обновляется. В противном случае строка остается неизменной.

Изменение существующих значений

В этом примере таблица Cars используется в примерах баз данных.

```
UPDATE Cars
SET TotalCost = TotalCost + 100
WHERE Id = 3 or Id = 4
```

Операции обновления могут включать текущие значения в обновленной строке. В этом простом примере `TotalCost` увеличивается на 100 для двух строк:

- Общая стоимость автомобиля №3 увеличена с 100 до 200
- Общая стоимость автомобиля №4 увеличена с 1254 до 1354

Новое значение столбца может быть получено из его предыдущего значения или любого значения другого столбца в той же таблице или объединенной таблице.

UPDATE с данными из другой таблицы

Приведенные ниже примеры заполняют `PhoneNumber` для любого Сотрудника, который также является `Customer` и в настоящее время не имеет номера телефона, установленного в таблице `Employees` .

(В этих примерах используются таблицы Employees and Customers из примеров баз данных.)

Стандартный SQL

Обновление с использованием коррелированного подзапроса:

```
UPDATE
    Employees
SET PhoneNumber =
    (SELECT
         c.PhoneNumber
     FROM
         Customers c
     WHERE
         c.FName = Employees.FName
         AND c.LName = Employees.LName)
WHERE Employees.PhoneNumber IS NULL
```

SQL: 2003

Обновление с помощью `MERGE` :

```
MERGE INTO
    Employees e
USING
    Customers c
ON
    e.FName = c.Fname
    AND e.LName = c.LName
    AND e.PhoneNumber IS NULL
WHEN MATCHED THEN
   UPDATE
```

```
      SET PhoneNumber = c.PhoneNumber
```

SQL Server

Обновление с помощью `INNER JOIN` :

```
UPDATE
    Employees
SET
    PhoneNumber = c.PhoneNumber
FROM
    Employees e
INNER JOIN Customers c
        ON e.FName = c.FName
        AND e.LName = c.LName
WHERE
    PhoneNumber IS NULL
```

Захват обновленных записей

Иногда хочется записать только что обновленные записи.

```
CREATE TABLE #TempUpdated(ID INT)

Update TableName SET Col1 = 42
    OUTPUT inserted.ID INTO #TempUpdated
    WHERE Id > 50
```

Прочитайте ОБНОВИТЬ онлайн: https://riptutorial.com/ru/sql/topic/321/обновить

глава 32: Общие выражения таблицы

Синтаксис

- WITH QueryName [(ColumnName, ...)] AS (
  ВЫБРАТЬ ...
)
  SELECT ... FROM QueryName ...;

- С RECURSIVE QueryName [(ColumnName, ...)] AS (
  ВЫБРАТЬ ...
  СОЮЗ [ВСЕ]
  SELECT ... FROM QueryName ...
)
  SELECT ... FROM QueryName ...;

замечания

Официальная документация: статья WITH

Выражение общей таблицы - это временный набор результатов, и это может быть результатом сложного суб-запроса. Он определяется с помощью предложения WITH. CTE улучшает читаемость и создается в памяти, а не в базе данных TempDB, где создается переменная Temp Table и Table.

Ключевые понятия общих табличных выражений:

- Может использоваться для разбивки сложных запросов, особенно сложных объединений и подзапросов.
- Является способом инкапсуляции определения запроса.
- Сохранять только до тех пор, пока не будет запущен следующий запрос.
- Правильное использование может привести к улучшению как качества кода, так и поддерживаемости и скорости.
- Может использоваться для ссылки на результирующую таблицу несколько раз в одном выражении (исключить дублирование в SQL).
- Может быть заменой для представления, когда общее использование представления не требуется; то есть вам не нужно сохранять определение в метаданных.
- Будет выполняться при вызове, а не при определении. Если CTE используется несколько раз в запросе, он будет запускаться несколько раз (возможно, с разными результатами).

Examples

Временный запрос

Они ведут себя так же, как вложенные подзапросы, но с другим синтаксисом.

```
WITH ReadyCars AS (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
)
SELECT ID, Model, TotalCost
FROM ReadyCars
ORDER BY TotalCost;
```

Я БЫ	модель	Общая стоимость
1	Ford F-150	200
2	Ford F-150	230

Эквивалентный синтаксис подзапроса

```
SELECT ID, Model, TotalCost
FROM (
  SELECT *
  FROM Cars
  WHERE Status = 'READY'
) AS ReadyCars
ORDER BY TotalCost
```

рекурсивно поднимающийся в дерево

```
WITH RECURSIVE ManagersOfJonathon AS (
    -- start with this row
    SELECT *
    FROM Employees
    WHERE ID = 4

    UNION ALL

    -- get manager(s) of all previously selected rows
    SELECT Employees.*
    FROM Employees
    JOIN ManagersOfJonathon
        ON Employees.ID = ManagersOfJonathon.ManagerID
)
SELECT * FROM ManagersOfJonathon;
```

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID
4	Джонатон	кузнец	1212121212	**2**	1
2	Джон	Джонсон	2468101214	**1**	1

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID
1	Джеймс	кузнец	1234567890	НОЛЬ	1

генерирующие значения

Большинство баз данных не имеют собственного способа генерации серии чисел для ad-hoc-использования; однако общие выражения таблицы могут использоваться с рекурсией для эмуляции этого типа функции.

В следующем примере генерируется общее табличное выражение, называемое `Numbers` с столбцом `i` которое имеет строку для чисел 1-5:

```
--Give a table name `Numbers" and a column `i` to hold the numbers
WITH Numbers(i) AS (
    --Starting number/index
    SELECT 1
    --Top-level UNION ALL operator required for recursion
    UNION ALL
    --Iteration expression:
    SELECT i + 1
    --Table expression we first declared used as source for recursion
    FROM Numbers
    --Clause to define the end of the recursion
    WHERE i < 5
)
--Use the generated table expression like a regular table
SELECT i FROM Numbers;
```

я
1
2
3
4
5

Этот метод можно использовать с любым интервалом числа, а также с другими типами данных.

рекурсивно перечисляя поддерево

```
WITH RECURSIVE ManagedByJames(Level, ID, FName, LName) AS (
    -- start with this row
    SELECT 1, ID, FName, LName
    FROM Employees
    WHERE ID = 1
```

```
    UNION ALL

    -- get employees that have any of the previously selected rows as manager
    SELECT ManagedByJames.Level + 1,
           Employees.ID,
           Employees.FName,
           Employees.LName
    FROM Employees
    JOIN ManagedByJames
        ON Employees.ManagerID = ManagedByJames.ID

    ORDER BY 1 DESC   -- depth-first search
)
SELECT * FROM ManagedByJames;
```

уровень	Я БЫ	FName	LName
1	1	Джеймс	кузнец
2	2	Джон	Джонсон
3	4	Джонатон	кузнец
2	3	Майкл	Williams

Функциональность Oracle CONNECT BY с рекурсивными CTE

Функциональность CONNECT BY от Oracle обеспечивает множество полезных и нетривиальных функций, которые не встроены при использовании стандартных рекурсивных CTE SQL. Этот пример реплицирует эти функции (с несколькими дополнениями для полноты), используя синтаксис SQL Server. Для разработчиков Oracle очень полезно найти многие функции, отсутствующие в их иерархических запросах в других базах данных, но также служит для демонстрации того, что можно сделать с помощью иерархического запроса в целом.

```
  WITH tbl AS (
       SELECT id, name, parent_id
         FROM mytable)
     , tbl_hierarchy AS (
       /* Anchor */
       SELECT 1 AS "LEVEL"
            --, 1 AS CONNECT_BY_ISROOT
            --, 0 AS CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , 0 AS CONNECT_BY_ISCYCLE
            , '/' + CAST(t.id   AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_id
            , '/' + CAST(t.name AS VARCHAR(MAX)) + '/' AS SYS_CONNECT_BY_PATH_name
            , t.id AS root_id
            , t.*
         FROM tbl t
        WHERE t.parent_id IS NULL                              -- START WITH parent_id IS NULL
```

```
       UNION ALL
       /* Recursive */
       SELECT th."LEVEL" + 1 AS "LEVEL"
            --, 0 AS CONNECT_BY_ISROOT
            --, CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 1 ELSE 0 END AS
CONNECT_BY_ISBRANCH
            , CASE WHEN t.id IN (SELECT parent_id FROM tbl) THEN 0 ELSE 1 END AS
CONNECT_BY_ISLEAF
            , CASE WHEN th.SYS_CONNECT_BY_PATH_id LIKE '%/' + CAST(t.id AS VARCHAR(MAX)) +
'/%' THEN 1 ELSE 0 END AS CONNECT_BY_ISCYCLE
            , th.SYS_CONNECT_BY_PATH_id   + CAST(t.id   AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_id
            , th.SYS_CONNECT_BY_PATH_name + CAST(t.name AS VARCHAR(MAX)) + '/' AS
SYS_CONNECT_BY_PATH_name
            , th.root_id
            , t.*
         FROM tbl t
              JOIN tbl_hierarchy th ON (th.id = t.parent_id) -- CONNECT BY PRIOR id =
parent_id
        WHERE th.CONNECT_BY_ISCYCLE = 0)                     -- NOCYCLE
SELECT th.*
     --, REPLICATE(' ', (th."LEVEL" - 1) * 3) + th.name AS tbl_hierarchy
  FROM tbl_hierarchy th
       JOIN tbl CONNECT_BY_ROOT ON (CONNECT_BY_ROOT.id = th.root_id)
 ORDER BY th.SYS_CONNECT_BY_PATH_name;                       -- ORDER SIBLINGS BY name
```

CONNECT BY, показанные выше, с пояснениями:

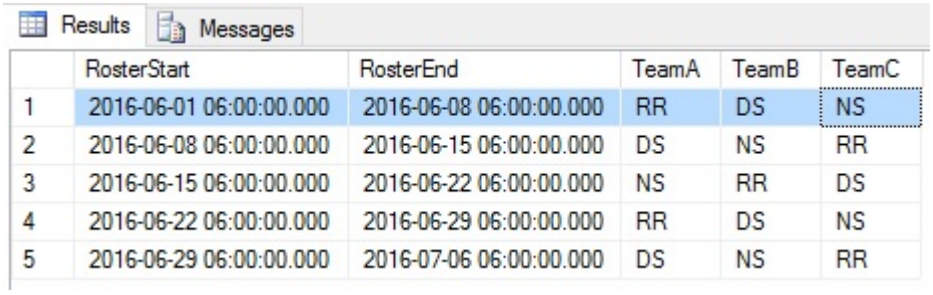
- Статьи
  - CONNECT BY: Определяет взаимосвязь, определяющую иерархию.
  - START WITH: указывает корневые узлы.
  - ORDER SIBLINGS BY: Заказывает результат правильно.
- параметры
  - NOCYCLE: останавливает обработку ветки при обнаружении цикла. Допустимые иерархии - это ациклические графики, а круговые ссылки нарушают эту конструкцию.
- операторы
  - ПРИОР: Получает данные от родителя узла.
  - CONNECT_BY_ROOT: Получает данные из корня узла.
- псевдостолбцы
  - LEVEL: указывает расстояние до узла от его корня.
  - CONNECT_BY_ISLEAF: указывает узел без детей.
  - CONNECT_BY_ISCYCLE: указывает узел с круглой ссылкой.
- функции
  - SYS_CONNECT_BY_PATH: Возвращает сжатое / конкатенированное представление пути к узлу из его корня.

Рекурсивно генерировать даты, расширенные для включения в список команд в качестве примера

```
DECLARE @DateFrom DATETIME = '2016-06-01 06:00'
DECLARE @DateTo DATETIME = '2016-07-01 06:00'
DECLARE @IntervalDays INT = 7

-- Transition Sequence = Rest & Relax into Day Shift into Night Shift
-- RR (Rest & Relax) = 1
-- DS (Day Shift) = 2
-- NS (Night Shift) = 3

;WITH roster AS
(
   SELECT @DateFrom AS RosterStart, 1 AS TeamA, 2 AS TeamB, 3 AS TeamC
   UNION ALL
   SELECT DATEADD(d, @IntervalDays, RosterStart),
          CASE TeamA WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamA,
          CASE TeamB WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamB,
          CASE TeamC WHEN 1 THEN 2 WHEN 2 THEN 3 WHEN 3 THEN 1 END AS TeamC
   FROM roster WHERE RosterStart < DATEADD(d, -@IntervalDays, @DateTo)
)

SELECT RosterStart,
       ISNULL(LEAD(RosterStart) OVER (ORDER BY RosterStart), RosterStart + @IntervalDays) AS
RosterEnd,
       CASE TeamA WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamA,
       CASE TeamB WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamB,
       CASE TeamC WHEN 1 THEN 'RR' WHEN 2 THEN 'DS' WHEN 3 THEN 'NS' END AS TeamC
FROM roster
```

Результат

Т.е. за неделю 1 TeamA находится на R & R, TeamB находится в режиме Day Shift, а TeamC - в ночном режиме.


Results Messages

	RosterStart	RosterEnd	TeamA	TeamB	TeamC
1	2016-06-01 06:00:00.000	2016-06-08 06:00:00.000	RR	DS	NS
2	2016-06-08 06:00:00.000	2016-06-15 06:00:00.000	DS	NS	RR
3	2016-06-15 06:00:00.000	2016-06-22 06:00:00.000	NS	RR	DS
4	2016-06-22 06:00:00.000	2016-06-29 06:00:00.000	RR	DS	NS
5	2016-06-29 06:00:00.000	2016-07-06 06:00:00.000	DS	NS	RR

Рефакторинг запроса для использования общих выражений таблицы

Предположим, мы хотим получить все категории продуктов с общим объемом продаж более 20.

Вот запрос без общих выражений таблицы:

```
SELECT category.description, sum(product.price) as total_sales
FROM sale
LEFT JOIN product on sale.product_id = product.id
LEFT JOIN category on product.category_id = category.id
GROUP BY category.id, category.description
HAVING sum(product.price) > 20
```

И эквивалентный запрос с использованием общих табличных выражений:

```
WITH all_sales AS (
  SELECT product.price, category.id as category_id, category.description as
category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
, sales_by_category AS (
  SELECT category_description, sum(price) as total_sales
  FROM all_sales
  GROUP BY category_id, category_description
)
SELECT * from sales_by_category WHERE total_sales > 20
```

Пример сложного SQL с общим выражением таблицы

Предположим, мы хотим запросить «самые дешевые продукты» из «лучших категорий».

Ниже приведен пример запроса с использованием общих табличных выражений

```
-- all_sales: just a simple SELECT with all the needed JOINS
WITH all_sales AS (
  SELECT
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM sale
  LEFT JOIN product on sale.product_id = product.id
  LEFT JOIN category on product.category_id = category.id
)
-- Group by category
, sales_by_category AS (
  SELECT category_id, category_description,
  sum(product_price) as total_sales
  FROM all_sales
  GROUP BY category_id, category_description
)
-- Filtering total_sales > 20
, top_categories AS (
  SELECT * from sales_by_category WHERE total_sales > 20
)
-- all_products: just a simple SELECT with all the needed JOINS
, all_products AS (
  SELECT
  product.id as product_id,
  product.description as product_description,
  product.price as product_price,
  category.id as category_id,
  category.description as category_description
  FROM product
  LEFT JOIN category on product.category_id = category.id
)
-- Order by product price
, cheapest_products AS (
  SELECT * from all_products
  ORDER by product_price ASC
```

```
)
-- Simple inner join
, cheapest_products_from_top_categories AS (
  SELECT product_description, product_price
  FROM cheapest_products
  INNER JOIN top_categories ON cheapest_products.category_id = top_categories.category_id
)
--The main SELECT
SELECT * from cheapest_products_from_top_categories
```

Прочитайте Общие выражения таблицы онлайн: https://riptutorial.com/ru/sql/topic/747/общие-выражения-таблицы

глава 33: ОБЪЯСНЕНИЕ И ОПИСАНИЕ

Examples

DESCRIBE tablename;

`DESCRIBE` и `EXPLAIN` являются синонимами. `DESCRIBE` в tablename возвращает определение столбцов.

```
DESCRIBE tablename;
```

Exmple Результат:

```
COLUMN_NAME     COLUMN_TYPE     IS_NULLABLE     COLUMN_KEY     COLUMN_DEFAULT    EXTRA
id              int(11)         NO              PRI            0
auto_increment
test            varchar(255)    YES                            (null)
```

Здесь вы видите имена столбцов, за которыми следует тип столбцов. Он показывает, разрешено ли `null` в столбце, и если столбец использует индекс. также отображается значение по умолчанию, и если таблица содержит какое-либо особое поведение, такое как `auto_increment` .

Выберите EXPLAIN Выберите запрос

`Explain` infront запроса `select` показывает вам, как будет выполняться запрос. Таким образом, вы увидите, использует ли запрос индекс или если вы можете оптимизировать свой запрос, добавив индекс.

Пример запроса:

```
explain select * from user join data on user.test = data.fk_user;
```

Пример результата:

```
id  select_type  table   type    possible_keys  key     key_len ref       rows  Extra
1   SIMPLE       user    index   test           test    5       (null)    1     Using where;
Using index
1   SIMPLE       data    ref     fk_user        fk_user 5       user.test 1     (null)
```

по `type` вы видите, использовался ли индекс. В столбце `possible_keys` вы видите, может ли план выполнения выбрать из разных индексов, если он не существует. `key` указывает вам используемый acutal индекс. `key_len` показывает размер в байтах для одного элемента индекса. Чем ниже это значение, тем больше элементов индекса вписывается в один и тот же размер памяти, и их можно быстрее обрабатывать. `rows` показывают ожидаемое

количество строк, которые требуется сканировать для запроса, чем ниже, тем лучше.

Прочитайте ОБЪЯСНЕНИЕ И ОПИСАНИЕ онлайн: https://riptutorial.com/ru/sql/topic/2928/объяснение-и-описание

глава 34: Оператор LIKE

Синтаксис

- **Wild Card c%:** SELECT * FROM [table] WHERE [имя_столбца] Как «% Value%»

  Wild Card with _: SELECT * FROM [table] WHERE [column_name] Как 'V_n%'

  Wild Card c [charlist]: SELECT * FROM [table] WHERE [column_name] Как 'V [abc] n%'

замечания

Условие LIKE в предложении WHERE используется для поиска значений столбцов, соответствующих данному шаблону. Шаблоны формируются с использованием следующих двух подстановочных знаков

- % (Процентный символ) - Используется для представления ноль или более символов
- _ (Underscore) - используется для представления одного символа

Examples

Матч открытого шаблона

Подстановочный знак ％ добавленный к началу или концу (или обеим) строки, будет содержать 0 или более символов до начала или после окончания шаблона.

Использование «%» в середине позволит совместить 0 или более символов между двумя частями шаблона.

Мы собираемся использовать эту таблицу сотрудников:

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Оплата труда	Дата приема на работу
1	Джон	Джонсон	2468101214	1	1	400	23-03-2005
2	Софи	Amudsen	2479100211	1	1	400	11-01-2010
3	Ronny	кузнец	2462544026	2	1	600	06-08-2015

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Оплата труда	Дата приема на работу
4	Джон	Sanchez	2454124602	1	1	400	23-03-2005
5	Хильде	сук	2468021911	2	1	800	01-01-2000

Следующие утверждения соответствуют всем записям, содержащим FName, **содержащие** строку «on» из таблицы Employees.

```
SELECT * FROM Employees WHERE FName LIKE '%on%';
```

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Оплата труда	Дата приема на работу
3	R **на** ny	кузнец	2462544026	2	1	600	06-08-2015
4	J **on**	Sanchez	2454124602	1	1	400	23-03-2005

Следующий оператор соответствует всем записям, имеющим PhoneNumber, **начиная со** строки «246» от Employees.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '246%';
```

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Оплата труда	Дата приема на работу
1	Джон	Джонсон	**246** 8101214	1	1	400	23-03-2005
3	Ronny	кузнец	**246** 2544026	2	1	600	06-08-2015
5	Хильде	сук	**246** 8021911	2	1	800	01-01-2000

Следующий оператор соответствует всем записям с номером PhoneNumber, **заканчивающимся на** строку «11» от Employees.

```
SELECT * FROM Employees WHERE PhoneNumber LIKE '%11'
```

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Оплата труда	Дата приема на работу
2	Софи	Amudsen	24791002**11**	1	1	400	11-01-2010
5	Хильде	сук	24680219**11**	2	1	800	01-01-2000

Все записи, где **3-й символ** Fname - «n» от сотрудников.

```
SELECT * FROM Employees WHERE FName LIKE '__n%';
```

(два символа подчеркивания используются до «n», чтобы пропустить первые 2 символа)

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Оплата труда	Дата приема на работу
3	Ronny	кузнец	2462544026	2	1	600	06-08-2015
4	Джон	Sanchez	2454124602	1	1	400	23-03-2005

Совпадение одного символа

Чтобы расширить выбор оператора структурированного запроса (SQL-SELECT), можно использовать подстановочные знаки, знак процента (%) и подчеркивание (_).

Символ _ (подчеркивание) может использоваться в качестве подстановочного знака для любого отдельного символа в совпадении с шаблоном.

Найдите всех сотрудников, чье Fname начинается с «j» и заканчивается на «n» и имеет ровно 3 символа в Fname.

```
SELECT * FROM Employees WHERE FName LIKE 'j_n'
```

_ (подчеркивание) также может использоваться более одного раза в качестве дикой карты для соответствия шаблонам.

Например, этот шаблон будет соответствовать «jon», «jan», «jen» и т. Д.

Эти имена не будут отображаться «jn», «john», «jordan», «justin», «jason», «julian», «jillian», «joann», потому что в нашем запросе используется один знак подчеркивания, и он может пропустить точно один символ, поэтому результат должен иметь 3 символа Fname.

Например, этот шаблон будет соответствовать «LaSt», «LoSt», «HaLt» и т. Д.

```
SELECT * FROM Employees WHERE FName LIKE '_A_T'
```

Соответствие диапазону или набору

Сопоставьте любой отдельный символ в указанном диапазоне (например: `[af]`) или установите (например: `[abcdef]`).

Этот шаблон диапазона будет соответствовать «gary», но не «mary»:

```
SELECT * FROM Employees WHERE FName LIKE '[a-g]ary'
```

Этот шаблон будет соответствовать «mary», но не «gary»:

```
SELECT * FROM Employees WHERE Fname LIKE '[lmnop]ary'
```

Диапазон или набор можно также отменить, добавив `^` каретку перед диапазоном или установить:

Этот шаблон диапазона *не* будет соответствовать «gary», но будет соответствовать « mary»:

```
SELECT * FROM Employees WHERE FName LIKE '[^a-g]ary'
```

Этот шаблон набора *не* будет соответствовать «mary», но будет соответствовать «gary»:

```
SELECT * FROM Employees WHERE Fname LIKE '[^lmnop]ary'
```

Матч ЛЮБОЙ против ВСЕХ

Совпадение:
Необходимо совместить хотя бы одну строку. В этом примере тип продукта должен быть «электроникой», «книгами» или «видео».

```
SELECT *
FROM   purchase_table
```

```
WHERE  product_type LIKE ANY ('electronics', 'books', 'video');
```

Все совпадение (должно соответствовать всем требованиям).
В этом примере должны быть согласованы как «объединенное королевство», так *и* « лондон» *и* «восточная дорога» (включая вариации).

```
SELECT *
FROM   customer_table
WHERE  full_address LIKE ALL ('%united kingdom%', '%london%', '%eastern road%');
```

Отрицательный выбор:
Используйте ВСЕ, чтобы исключить все элементы.
В этом примере приводятся все результаты, когда тип продукта не является «электроникой», а не «книгами», а не «видео».

```
SELECT *
FROM   customer_table
WHERE  product_type NOT LIKE ALL ('electronics', 'books', 'video');
```

Поиск диапазона символов

Следующий оператор соответствует всем записям, имеющим FName, которое начинается с буквы от A до F из таблицы Employees .

```
SELECT * FROM Employees WHERE FName LIKE '[A-F]%'
```

Выписка ESCAPE в LIKE-запросе

Если вы реализуете текстовый поиск как `LIKE` -query, вы обычно делаете это так:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%')
```

Однако (помимо того, что вы не должны использовать `LIKE` когда вы можете использовать полнотекстовый поиск), это создает проблему, когда кто-то вводит текст типа «50%» или « a_b».

Таким образом (вместо перехода на полнотекстовый поиск) вы можете решить эту проблему с помощью инструкции `LIKE` -escape:

```
SELECT *
FROM T_Whatever
WHERE SomeField LIKE CONCAT('%', @in_SearchText, '%') ESCAPE '\'
```

Это означает, что `\` теперь будет рассматриваться как символ ESCAPE. Это означает, что теперь вы можете просто добавить `\` к каждому символу в строке, которую вы ищете, и

результаты будут корректными, даже если пользователь вводит специальный символ, например `%` или _ .

например

```
string stringToSearch = "abc_def 50%";
string newString = "";
foreach(char c in stringToSearch)
     newString += @"\" + c;

sqlCmd.Parameters.Add("@in_SearchText", newString);
// instead of sqlCmd.Parameters.Add("@in_SearchText", stringToSearch);
```

Примечание. Вышеупомянутый алгоритм предназначен только для демонстрационных целей. Он не будет работать в случаях, когда 1 графема состоит из нескольких символов (utf-8). например `string stringToSearch = "Les Mise\u0301rables";` Вам нужно будет сделать это для каждой графемы, а не для каждого персонажа. Вы не должны использовать вышеуказанный алгоритм, если имеете дело с азиатскими / восточно-азиатскими / южноазиатскими языками. Вернее, если вам нужен правильный код для начала, вы должны просто сделать это для каждого graphemeCluster.

См. Также ReverseString, вопрос интервью с C #

Подстановочные знаки

подстановочные символы используются с оператором SQL LIKE. Подстановочные знаки SQL используются для поиска данных в таблице.

Подстановочные знаки в SQL:%, _, [charlist], [^ charlist]

% - заменить ноль или более символов

```
   Eg:  //selects all customers with a City starting with "Lo"
        SELECT * FROM Customers
        WHERE City LIKE 'Lo%';

       //selects all customers with a City containing the pattern "es"
      SELECT * FROM Customers
       WHERE City LIKE '%es%';
```

_ - заменить один символ

```
Eg://selects all customers with a City starting with any character, followed by "erlin"
SELECT * FROM Customers
WHERE City LIKE '_erlin';
```

[charlist] - устанавливает и диапазоны символов для соответствия

```
Eg://selects all customers with a City starting with "a", "d", or "l"
```

```
SELECT * FROM Customers
WHERE City LIKE '[adl]%';

//selects all customers with a City starting with "a", "d", or "l"
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

[^ charlist] - Соответствует только символу, не указанному в скобках

```
Eg://selects all customers with a City starting with a character that is not "a", "p", or "l"
SELECT * FROM Customers
WHERE City LIKE '[^apl]%';

or

SELECT * FROM Customers
WHERE City NOT LIKE '[apl]%' and city like '_%';
```

Прочитайте Оператор LIKE онлайн: https://riptutorial.com/ru/sql/topic/860/оператор-like

глава 35: Операторы AND & OR

Синтаксис

1. SELECT * FROM table WHERE (условие1) AND (условие2);
2. SELECT * FROM table WHERE (условие 1) ИЛИ (условие2);

Examples

И ИЛИ Пример

Стол

название	Возраст	город
боб	10	Париж
Мат	20	Берлин
Мэри	24	Прага

```
select Name from table where Age>10 AND City='Prague'
```

дает

название
Мэри

```
select Name from table where Age=10 OR City='Prague'
```

дает

название
боб
Мэри

Прочитайте Операторы AND & OR онлайн: https://riptutorial.com/ru/sql/topic/1386/операторы-and--amp--or

глава 36: операции

замечания

Транзакция - это логическая единица работы, содержащая один или несколько шагов, каждая из которых должна быть успешно завершена, чтобы транзакция зафиксировала базу данных. Если есть ошибки, то все изменения данных стираются, и база данных возвращается в исходное состояние в начале транзакции.

Examples

Простая транзакция

```
BEGIN TRANSACTION
    INSERT INTO DeletedEmployees(EmployeeID, DateDeleted, User)
        (SELECT 123, GetDate(), CURRENT_USER);
    DELETE FROM Employees WHERE EmployeeID = 123;
COMMIT TRANSACTION
```

Откат транзакции

Когда что-то не удается в коде транзакции и вы хотите его отменить, вы можете отменить транзакцию:

```
BEGIN TRY
    BEGIN TRANSACTION
        INSERT INTO Users(ID, Name, Age)
        VALUES(1, 'Bob', 24)

        DELETE FROM Users WHERE Name = 'Todd'
   COMMIT TRANSACTION
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION
END CATCH
```

Прочитайте операции онлайн: https://riptutorial.com/ru/sql/topic/2424/операции

глава 37: Очистить код в SQL

Вступление

Как писать хорошие, читаемые SQL-запросы и примеры передового опыта.

Examples

Форматирование и правописание ключевых слов и имен

Таблицы / Имена столбцов

Двумя распространенными способами форматирования имен таблиц и столбцов являются `CamelCase` и `snake_case` :

```
SELECT FirstName, LastName
FROM Employees
WHERE Salary > 500;
```

```
SELECT first_name, last_name
FROM employees
WHERE salary > 500;
```

Имена должны описывать, что хранится в их объекте. Это означает, что имена столбцов обычно должны быть сингулярными. Должны ли имена таблиц использовать сингулярное или множественное число, является широко обсуждаемым вопросом, но на практике чаще всего используется множество имен таблиц.

Добавление префиксов или суффиксов, таких как `tbl` или `col` уменьшает читаемость, поэтому избегайте их. Однако они иногда используются для предотвращения конфликтов с ключевыми словами SQL и часто используются с триггерами и индексами (имена которых обычно не упоминаются в запросах).

Ключевые слова

Ключевые слова SQL не чувствительны к регистру. Тем не менее, обычно их пишут в верхнем регистре.

ВЫБРАТЬ *

`SELECT *` возвращает все столбцы в том же порядке, который определен в таблице.

При использовании `SELECT *` данные, возвращаемые запросом, могут меняться при изменении определения таблицы. Это увеличивает риск того, что разные версии вашего приложения или вашей базы данных несовместимы друг с другом.

Кроме того, чтение большего количества столбцов, чем необходимо, может увеличить объем дискового и сетевого ввода-вывода.

Поэтому вы всегда должны явно указывать столбцы (столбцы), которые вы действительно хотите получить:

```
--SELECT *                                  don't
  SELECT ID, FName, LName, PhoneNumber  -- do
  FROM Emplopees;
```

(При выполнении интерактивных запросов эти соображения не применяются.)

Однако `SELECT *` не повредит в подзапросе оператора EXISTS, потому что EXISTS все равно игнорирует фактические данные (он проверяет, только если найдена хотя бы одна строка). По той же причине не имеет смысла перечислять какие-либо конкретные столбцы для EXISTS, поэтому `SELECT *` имеет смысл:

```
-- list departments where nobody was hired recently
SELECT ID,
       Name
FROM Departments
WHERE NOT EXISTS (SELECT *
                  FROM Employees
                  WHERE DepartmentID = Departments.ID
                    AND HireDate >= '2015-01-01');
```

Отступ

Нет общепринятого стандарта. Все согласны с тем, что сжатие всего в одну строку плохое:

```
SELECT d.Name, COUNT(*) AS Employees FROM Departments AS d JOIN Employees AS e ON d.ID =
e.DepartmentID WHERE d.Name != 'HR' HAVING COUNT(*) > 10 ORDER BY COUNT(*) DESC;
```

Как минимум, поместите каждое предложение в новую строку и разделите строки, если они станут слишком длинными в противном случае:

```
SELECT d.Name,
       COUNT(*) AS Employees
FROM Departments AS d
JOIN Employees AS e ON d.ID = e.DepartmentID
WHERE d.Name != 'HR'
HAVING COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

Иногда все после того, как ключевое слово SQL, вводящее предложение, имеет отступ в

одном столбце:

```
SELECT   d.Name,
         COUNT(*) AS Employees
FROM     Departments AS d
JOIN     Employees AS e ON d.ID = e.DepartmentID
WHERE    d.Name != 'HR'
HAVING   COUNT(*) > 10
ORDER BY COUNT(*) DESC;
```

(Это можно сделать и при правильном выравнивании ключевых слов SQL.)

Другим распространенным стилем является включение важных ключевых слов в их собственные строки:

```
SELECT
    d.Name,
    COUNT(*) AS Employees
FROM
    Departments AS d
JOIN
    Employees AS e
    ON d.ID = e.DepartmentID
WHERE
    d.Name != 'HR'
HAVING
    COUNT(*) > 10
ORDER BY
    COUNT(*) DESC;
```

Вертикальное выравнивание нескольких похожих выражений улучшает читаемость:

```
SELECT Model,
       EmployeeID
FROM Cars
WHERE CustomerID = 42
  AND Status     = 'READY';
```

Использование нескольких строк затрудняет внедрение SQL-команд в другие языки программирования. Однако многие языки имеют механизм для многострочных строк, например, `@"..."` в C #, `"""..."""` в Python или `R"(...)"` в C ++.

присоединяется

Явные соединения всегда должны использоваться; неявные объединения имеют несколько проблем:

- Условие соединения находится где-то в предложении WHERE, смешанном с любыми другими условиями фильтра. Это затрудняет понимание того, какие таблицы объединены и как.

- Из-за вышеизложенного существует более высокий риск ошибок, и более вероятно, что они будут найдены позже.
- В стандартном SQL явные соединения являются единственным способом использования внешних соединений :

```
SELECT d.Name,
       e.Fname || e.LName AS EmpName
FROM      Departments AS d
LEFT JOIN Employees   AS e ON d.ID = e.DepartmentID;
```

- Явные объединения позволяют использовать условие USING:

```
SELECT RecipeID,
       Recipes.Name,
       COUNT(*) AS NumberOfIngredients
FROM      Recipes
LEFT JOIN Ingredients USING (RecipeID);
```

(Это требует, чтобы обе таблицы использовали одно и то же имя столбца. ИСПОЛЬЗОВАНИЕ автоматически удаляет дубликат столбца из результата, например, соединение в этом запросе возвращает один столбец из `RecipeID` .)

Прочитайте Очистить код в SQL онлайн: https://riptutorial.com/ru/sql/topic/9843/очистить-код-в-sql

глава 38: Первичные ключи

Синтаксис

- MySQL: CREATE TABLE Сотрудники (Id int NOT NULL, PRIMARY KEY (Id), ...);
- Другие: CREATE TABLE Сотрудники (Id int NOT NULL PRIMARY KEY, ...);

Examples

Создание первичного ключа

```
CREATE TABLE Employees (
    Id int NOT NULL,
    PRIMARY KEY (Id),
    ...
);
```

Это создаст таблицу Employees с ключевым словом «Id». Первичный ключ может использоваться для однозначной идентификации строк таблицы. Только один первичный ключ разрешен для каждой таблицы.

Ключ также может быть составлен одним или несколькими полями, так называемым составным ключом, со следующим синтаксисом:

```
CREATE TABLE EMPLOYEE (
    e1_id INT,
    e2_id INT,
    PRIMARY KEY (e1_id, e2_id)
)
```

Использование автоматического увеличения

Многие базы данных позволяют автоматически увеличивать значение первичного ключа при добавлении нового ключа. Это гарантирует, что каждый ключ отличается.

MySQL

```
CREATE TABLE Employees (
    Id int NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (Id)
);
```

PostgreSQL

```
CREATE TABLE Employees (
```

```
    Id SERIAL PRIMARY KEY
);
```

SQL Server

```
CREATE TABLE Employees (
    Id int NOT NULL IDENTITY,
    PRIMARY KEY (Id)
);
```

SQLite

```
CREATE TABLE Employees (
    Id INTEGER PRIMARY KEY
);
```

Прочитайте Первичные ключи онлайн: https://riptutorial.com/ru/sql/topic/505/первичные-ключи

глава 39: подзапросов

замечания

Подзапросы могут отображаться в разных предложениях внешнего запроса или в заданной операции.

Они должны быть заключены в круглые скобки `()` . Если результат подзапроса сравнивается с чем-то другим, количество столбцов должно совпадать. Табличные псевдонимы необходимы для подзапросов в предложении FROM, чтобы назвать временную таблицу.

Examples

Подзапрос в предложении WHERE

Используйте подзапрос для фильтрации набора результатов. Например, это вернет всех сотрудников с зарплатой, равной самому высокооплачиваемому сотруднику.

```
SELECT *
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees)
```

Подзапрос в предложении FROM

Подзапрос в предложении `FROM` действует аналогично временной таблице, которая генерируется во время выполнения запроса и впоследствии теряется.

```
SELECT Managers.Id, Employees.Salary
FROM (
  SELECT Id
  FROM Employees
  WHERE ManagerId IS NULL
) AS Managers
JOIN Employees ON Managers.Id = Employees.Id
```

Подзапрос в предложении SELECT

```
SELECT
  Id,
  FName,
  LName,
  (SELECT COUNT(*) FROM Cars WHERE Cars.CustomerId = Customers.Id) AS NumberOfCars
FROM Customers
```

Подзапросы в предложении FROM

Вы можете использовать подзапросы для определения временной таблицы и использовать ее в предложении FROM «внешнего» запроса.

```
SELECT * FROM (SELECT city, temp_hi - temp_lo AS temp_var FROM weather) AS w
WHERE temp_var > 20;
```

Вышеуказанные местоположения обнаруживают города из таблицы погоды, чье ежедневное изменение температуры больше 20. В результате получается:

город	temp_var
СВЯТОЙ ЛУИ	21
ЛОС-АНДЖЕЛЕС	31
ЛОС-АНДЖЕЛЕС	23
ЛОС-АНДЖЕЛЕС	31
ЛОС-АНДЖЕЛЕС	27
ЛОС-АНДЖЕЛЕС	28
ЛОС-АНДЖЕЛЕС	28
ЛОС-АНДЖЕЛЕС	32

,

Подзапросы в предложении WHERE

В следующем примере найдены города (из примера городов), население которых находится ниже средней температуры (полученной через sub-qquery):

```
SELECT name, pop2000 FROM cities
WHERE pop2000 < (SELECT avg(pop2000)  FROM cities);
```

Здесь: подзапрос (SELECT avg (pop2000) FROM города) используется для указания условий в предложении WHERE. Результат:

название	pop2000
Сан-Франциско	776733
СВЯТОЙ ЛУИ	348189

название	pop2000
Канзас-Сити	146866

Подзапросы в предложении SELECT

Подзапросы могут также использоваться в части `SELECT` внешнего запроса. Следующий запрос показывает все столбцы таблицы погоды с соответствующими состояниями из таблицы городов .

```
SELECT w.*,  (SELECT c.state FROM cities AS c WHERE c.name = w.city ) AS state
FROM weather AS w;
```

Фильтрация результатов запроса с использованием запроса в другой таблице

Этот запрос выбирает всех сотрудников не в таблице Supervisors.

```
SELECT *
FROM Employees
WHERE EmployeeID not in (SELECT EmployeeID
                         FROM Supervisors)
```

Те же результаты могут быть достигнуты с использованием LEFT JOIN.

```
SELECT *
FROM Employees AS e
LEFT JOIN Supervisors AS s ON s.EmployeeID=e.EmployeeID
WHERE s.EmployeeID is NULL
```

Коррелированные подзапросы

Коррелированные (также называемые синхронизированными или скоординированными) подзапросами - это вложенные запросы, которые ссылаются на текущую строку их внешнего запроса:

```
SELECT EmployeeId
    FROM Employee AS eOuter
    WHERE Salary > (
       SELECT AVG(Salary)
       FROM Employee eInner
       WHERE eInner.DepartmentId = eOuter.DepartmentId
    )
```

Подзапрос `SELECT AVG(Salary) ...` *коррелирован,* потому что он ссылается на строку `Employee` row `eOuter` из внешнего запроса.

Прочитайте подзапросов онлайн: https://riptutorial.com/ru/sql/topic/1606/подзапросов

глава 40: Поиск дубликатов в подмножестве столбцов с деталями

замечания

- Чтобы выбрать строки без дубликатов, измените предложение WHERE на «RowCnt = 1»
- Чтобы выбрать одну строку из каждого набора, используйте Rank () вместо Sum () и измените внешнее предложение WHERE, чтобы выбрать строки с Rank () = 1

Examples

Студенты с одинаковым именем и датой рождения

```
WITH CTE (StudentId, Fname, LName, DOB, RowCnt)
as (
SELECT StudentId, FirstName, LastName, DateOfBirth as DOB, SUM(1) OVER (Partition By
FirstName, LastName, DateOfBirth) as RowCnt
FROM tblStudent
)
SELECT * from CTE where RowCnt > 1
ORDER BY DOB, LName
```

В этом примере используется выражение Common Table и функция Window, чтобы показать все повторяющиеся строки (в подмножестве столбцов) рядом.

Прочитайте Поиск дубликатов в подмножестве столбцов с деталями онлайн: https://riptutorial.com/ru/sql/topic/1585/поиск-дубликатов-в-подмножестве-столбцов-с-деталями

глава 41: ПОПРОБУЙ ПОЙМАТЬ

замечания

TRY / CATCH - это языковая конструкция, специфичная для T-SQL MS SQL Server.

Он позволяет обрабатывать ошибки в T-SQL, подобно тому, как это видно в коде .NET.

Examples

Транзакция В TRY / CATCH

Это приведет к откату обеих вставок из-за недопустимого datetime:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, 'not a date', 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Это зафиксирует обе вставки:

```
BEGIN TRANSACTION
BEGIN TRY
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    INSERT INTO dbo.Sale(Price, SaleDate, Quantity)
    VALUES (5.2, GETDATE(), 1)
    COMMIT TRANSACTION
END TRY
BEGIN CATCH
    THROW
    ROLLBACK TRANSACTION
END CATCH
```

Прочитайте ПОПРОБУЙ ПОЙМАТЬ онлайн: https://riptutorial.com/ru/sql/topic/4420/попробуй-поймать

глава 42: Порядок исполнения

Examples

Логический порядок обработки запросов в SQL

```
/*(8)*/  SELECT /*9*/ DISTINCT /*11*/ TOP
/*(1)*/  FROM
/*(3)*/         JOIN
/*(2)*/       ON
/*(4)*/  WHERE
/*(5)*/  GROUP BY
/*(6)*/  WITH {CUBE | ROLLUP}
/*(7)*/  HAVING
/*(10)*/ ORDER BY
/*(11)*/ LIMIT
```

Порядок обработки запроса и описание каждого раздела.

VT обозначает «Виртуальную таблицу» и показывает, как различные данные создаются по мере обработки запроса

1. FROM: Декартово произведение (кросс-соединение) выполняется между двумя первыми двумя таблицами в предложении FROM, и в результате создается виртуальная таблица VT1.

2. ON: фильтр ВКЛ применяется к VT1. Только строки, для которых TRUE, вставлены в VT2.

3. OUTER (join): Если указан OUTER JOIN (в отличие от CROSS JOIN или INNER JOIN), строки из сохраненной таблицы или таблиц, для которых совпадение не было найдено, добавляются к строкам из VT2 в качестве внешних строк, генерируя VT3. Если в предложении FROM указано более двух таблиц, шаги 1 - 3 применяются повторно между результатом последнего соединения и следующей таблицей в предложении FROM до тех пор, пока не будут обработаны все таблицы.

4. ГДЕ: Фильтр WHERE применяется к VT3. Только строки, для которых TRUE, вставлены в VT4.

5. GROUP BY: Строки из VT4 расположены в группах на основе списка столбцов, указанного в предложении GROUP BY. VT5 генерируется.

6. CUBE | ROLLUP: Супергруппы (группы групп) добавляются к строкам из VT5, генерируя VT6.

7. HAVING: Фильтр HAVING применяется к VT6. Только группы, для которых TRUE,

вставлены в VT7.

8. SELECT: обрабатывается список SELECT, генерирующий VT8.
9. DISTINCT: Дублированные строки удаляются из VT8. Генерируется VT9.
10. ORDER BY: строки из VT9 сортируются в соответствии с списком столбцов, указанным в предложении ORDER BY. Создается курсор (VC10).
11. TOP: указанное число или процент строк выбирается с начала VC10. Таблица VT11 генерируется и возвращается вызывающему абоненту. LIMIT имеет ту же функциональность, что и TOP в некоторых SQL-диалектах, таких как Postgres и Netezza.

Прочитайте Порядок исполнения онлайн: https://riptutorial.com/ru/sql/topic/3671/порядок-исполнения

глава 43: Последовательность

Examples

Создать последовательность

```
CREATE SEQUENCE orders_seq
START WITH     1000
INCREMENT BY   1;
```

Создает последовательность с начальным значением 1000, которое увеличивается на 1.

Использование последовательностей

ссылка на *seq_name* .NEXTVAL используется для получения следующего значения в последовательности. Один оператор может генерировать только одно значение последовательности. Если в инструкции есть несколько ссылок на NEXTVAL, они будут использовать тот же сгенерированный номер.

NEXTVAL можно использовать для INSERTS

```
INSERT INTO Orders (Order_UID, Customer)
        VALUES (orders_seq.NEXTVAL, 1032);
```

Он может использоваться для ОБНОВЛЕНИЯ

```
UPDATE Orders
SET Order_UID = orders_seq.NEXTVAL
WHERE Customer = 581;
```

Он также может использоваться для SELECTS

```
SELECT Order_seq.NEXTVAL FROM dual;
```

Прочитайте Последовательность онлайн: https://riptutorial.com/ru/sql/topic/1586/последовательность

глава 44: Примеры баз данных и таблиц

Examples

База данных автозагрузки

В следующем примере - База данных для бизнеса автомагазина, у нас есть список отделов, сотрудников, клиентов и автомобилей клиентов. Мы используем внешние ключи для создания связей между различными таблицами.

Пример Live: скрипт SQL

Отношения между таблицами

- У каждого Департамента может быть 0 или более сотрудников
- У каждого сотрудника может быть 0 или 1 менеджер
- У каждого Клиента может быть 0 или более автомобилей

ведомства

Я бы	название
1	HR
2	Продажи
3	Технология

Операторы SQL для создания таблицы:

```
CREATE TABLE Departments (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(25) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Departments
    ([Id], [Name])
VALUES
    (1, 'HR'),
    (2, 'Sales'),
    (3, 'Tech')
;
```

Сотрудники

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Оплата труда	Дата приема на работу
1	Джеймс	кузнец	1234567890	НОЛЬ	1	1000	01-01-2002
2	Джон	Джонсон	2468101214	1	1	400	23-03-2005
3	Майкл	Williams	1357911131	1	2	600	12-05-2009
4	Джонатон	кузнец	1212121212	2	1	500	24-07-2016

Операторы SQL для создания таблицы:

```
CREATE TABLE Employees (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    PhoneNumber VARCHAR(11),
    ManagerId INT,
    DepartmentId INT NOT NULL,
    Salary INT NOT NULL,
    HireDate DATETIME NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (ManagerId) REFERENCES Employees(Id),
    FOREIGN KEY (DepartmentId) REFERENCES Departments(Id)
);

INSERT INTO Employees
    ([Id], [FName], [LName], [PhoneNumber], [ManagerId], [DepartmentId], [Salary], [HireDate])
VALUES
    (1, 'James', 'Smith', 1234567890, NULL, 1, 1000, '01-01-2002'),
    (2, 'John', 'Johnson', 2468101214, '1', 1, 400, '23-03-2005'),
    (3, 'Michael', 'Williams', 1357911131, '1', 2, 600, '12-05-2009'),
    (4, 'Johnathon', 'Smith', 1212121212, '2', 1, 500, '24-07-2016')
;
```

Клиенты

Я бы	FName	LName	Эл. адрес	Номер телефона	PreferredContact
1	Уильям	Джонс	william.jones@example.com	3347927472	ТЕЛЕФОН
2	Дэвид	мельник	dmiller@example.net	2137921892	ЭЛ. АДРЕС
3	Ричард	Дэвис	richard0123@example.com	НОЛЬ	ЭЛ. АДРЕС

Операторы SQL для создания таблицы:

```
CREATE TABLE Customers (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    Email varchar(100) NOT NULL,
    PhoneNumber VARCHAR(11),
    PreferredContact VARCHAR(5) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Customers
    ([Id], [FName], [LName], [Email], [PhoneNumber], [PreferredContact])
VALUES
    (1, 'William', 'Jones', 'william.jones@example.com', '3347927472', 'PHONE'),
    (2, 'David', 'Miller', 'dmiller@example.net', '2137921892', 'EMAIL'),
    (3, 'Richard', 'Davis', 'richard0123@example.com', NULL, 'EMAIL')
;
```

Машины

Я бы	Пользовательский ИД	EmployeeID	модель	Статус	Общая стоимость
1	1	2	Ford F-150	ГОТОВЫ	230
2	1	2	Ford F-150	ГОТОВЫ	200
3	2	1	Ford Mustang	ОЖИДАНИЯ	100
4	3	3	Toyota Prius	ЗА РАБОТОЙ	1254

Операторы SQL для создания таблицы:

```
CREATE TABLE Cars (
    Id INT NOT NULL AUTO_INCREMENT,
    CustomerId INT NOT NULL,
```

```
    EmployeeId INT NOT NULL,
    Model varchar(50) NOT NULL,
    Status varchar(25) NOT NULL,
    TotalCost INT NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (CustomerId) REFERENCES Customers(Id),
    FOREIGN KEY (EmployeeId) REFERENCES Employees(Id)
);

INSERT INTO Cars
    ([Id], [CustomerId], [EmployeeId], [Model], [Status], [TotalCost])
VALUES
    ('1', '1', '2', 'Ford F-150', 'READY', '230'),
    ('2', '1', '2', 'Ford F-150', 'READY', '200'),
    ('3', '2', '1', 'Ford Mustang', 'WAITING', '100'),
    ('4', '3', '3', 'Toyota Prius', 'WORKING', '1254')
;
```

База данных библиотек

В этой базе данных для библиотеки есть таблицы *авторов* , *книг* и *книг* .

Пример Live: скрипт SQL

Авторы и *книги* известны как **базовые таблицы** , поскольку они содержат определение столбцов и данные для реальных объектов в реляционной модели. *BooksAuthors* известна как **таблица отношений** , так как эта таблица определяет взаимосвязь между таблицей « *Книги* и *авторы*» .

Отношения между таблицами

- У каждого автора может быть 1 или более книг
- В каждой книге может быть 1 или более авторов

Авторы

(*таблица просмотра*)

Я бы	название	Страна
1	Дж. Д. Сэлинджер	Соединенные Штаты Америки
2	Ф. Скотт. Fitzgerald	Соединенные Штаты Америки
3	Джейн Остин	Соединенное Королевство
4	Скотт Гензельман	Соединенные Штаты Америки

Я бы	название	Страна
5	Джейсон Н. Гейлорд	Соединенные Штаты Америки
6	Пранав Растоги	Индия
7	Тодд Миранда	Соединенные Штаты Америки
8	Кристиан Венц	Соединенные Штаты Америки

SQL для создания таблицы:

```
CREATE TABLE Authors (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(70) NOT NULL,
    Country VARCHAR(100) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Authors
    (Name, Country)
VALUES
    ('J.D. Salinger', 'USA'),
    ('F. Scott. Fitzgerald', 'USA'),
    ('Jane Austen', 'UK'),
    ('Scott Hanselman', 'USA'),
    ('Jason N. Gaylord', 'USA'),
    ('Pranav Rastogi', 'India'),
    ('Todd Miranda', 'USA'),
    ('Christian Wenz', 'USA')
;
```

КНИГИ

(*таблица просмотра*)

Я бы	заглавие
1	Ловец во ржи
2	Девять историй
3	Фрэнни и Зои
4	Великий Гэтсби
5	Тендерный идентификатор Ночь
6	Гордость и предубеждение
7	Профессиональный ASP.NET 4.5 в C # и VB

SQL для создания таблицы:

```
CREATE TABLE Books (
    Id INT NOT NULL AUTO_INCREMENT,
    Title VARCHAR(50) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Books
    (Id, Title)
VALUES
    (1, 'The Catcher in the Rye'),
    (2, 'Nine Stories'),
    (3, 'Franny and Zooey'),
    (4, 'The Great Gatsby'),
    (5, 'Tender id the Night'),
    (6, 'Pride and Prejudice'),
    (7, 'Professional ASP.NET 4.5 in C# and VB')
;
```

BooksAuthors

(*таблица просмотра*)

BookID	AuthorID
1	1
2	1
3	1
4	2
5	2
6	3
7	4
7	5
7	6
7	7
7	8

SQL для создания таблицы:

```
CREATE TABLE BooksAuthors (
```

```
    AuthorId INT NOT NULL,
    BookId  INT NOT NULL,
    FOREIGN KEY (AuthorId) REFERENCES Authors(Id),
    FOREIGN KEY (BookId) REFERENCES Books(Id)
);

INSERT INTO BooksAuthors
    (BookId, AuthorId)
VALUES
    (1, 1),
    (2, 1),
    (3, 1),
    (4, 2),
    (5, 2),
    (6, 3),
    (7, 4),
    (7, 5),
    (7, 6),
    (7, 7),
    (7, 8)
;
```

Примеры

Просмотреть всех авторов (просмотреть живой пример):

```
SELECT * FROM Authors;
```

Просмотр всех названий книг (просмотр живого примера):

```
SELECT * FROM Books;
```

Просмотреть все книги и их авторов (посмотреть живой пример):

```
SELECT
  ba.AuthorId,
  a.Name AuthorName,
  ba.BookId,
  b.Title BookTitle
FROM BooksAuthors ba
  INNER JOIN Authors a ON a.id = ba.authorid
  INNER JOIN Books b ON b.id = ba.bookid
;
```

Таблица стран

В этом примере у нас есть таблица **стран** . Таблица для стран имеет много применений, особенно в финансовых приложениях, связанных с валютами и обменными курсами.

Пример Live: скрипт SQL

Некоторые программные приложения для данных рынка, такие как Bloomberg и Reuters, требуют, чтобы вы предоставили свой API код страны или 2 символа страны вместе с кодом валюты. Следовательно, эта таблица примеров содержит как 2- `ISO3` столбец `ISO` кода, так и 3 символьных `ISO3` кода `ISO3` .

страны

(*таблица просмотра*)

Я бы	ISO	ISO3	ISONumeric	Название страны	Капитал	ContinentCode	Код вал
1	AU	AUS	36	Австралия	Канберра	OC	AUD
2	Делавэр	DEU	276	Германия	Берлин	Евросоюз	евр
2	В	IND	356	Индия	Нью-Дели	КАК	INR
3	Луизиана	ЛАО	418	Лаос	Вьентьян	КАК	LAK
4	НАС	Соединенные Штаты Америки	840	Соединенные Штаты	Вашингтон	Не Доступно	дол США
5	ZW	ZWE	716	Зимбабве	Хараре	AF	ZWL

SQL для создания таблицы:

```
CREATE TABLE Countries (
    Id INT NOT NULL AUTO_INCREMENT,
    ISO VARCHAR(2) NOT NULL,
    ISO3 VARCHAR(3) NOT NULL,
    ISONumeric INT NOT NULL,
    CountryName VARCHAR(64) NOT NULL,
    Capital VARCHAR(64) NOT NULL,
    ContinentCode VARCHAR(2) NOT NULL,
    CurrencyCode VARCHAR(3) NOT NULL,
    PRIMARY KEY(Id)
)
;

INSERT INTO Countries
    (ISO, ISO3, ISONumeric, CountryName, Capital, ContinentCode, CurrencyCode)
VALUES
    ('AU', 'AUS', 36, 'Australia', 'Canberra', 'OC', 'AUD'),
    ('DE', 'DEU', 276, 'Germany', 'Berlin', 'EU', 'EUR'),
    ('IN', 'IND', 356, 'India', 'New Delhi', 'AS', 'INR'),
    ('LA', 'LAO', 418, 'Laos', 'Vientiane', 'AS', 'LAK'),
    ('US', 'USA', 840, 'United States', 'Washington', 'NA', 'USD'),
    ('ZW', 'ZWE', 716, 'Zimbabwe', 'Harare', 'AF', 'ZWL')
```

```
;
```

Прочитайте Примеры баз данных и таблиц онлайн: https://riptutorial.com/ru/sql/topic/280/примеры-баз-данных-и-таблиц

глава 45: ПРИСОЕДИНИТЬСЯ

Вступление

JOIN - это метод объединения (объединения) информации из двух таблиц. Результатом является сшитое множество столбцов из обеих таблиц, определяемое типом объединения (INNER / OUTER / CROSS и LEFT / RIGHT / FULL, объяснено ниже) и критерии присоединения (как связаны строки из обеих таблиц).

Таблица может быть присоединена к себе или к любой другой таблице. Если требуется получить доступ к информации из более чем двух таблиц, в предложении FROM можно указать несколько объединений.

Синтаксис

- `[ { INNER | { { LEFT | RIGHT | FULL } [ OUTER ] } } ] JOIN`

замечания

Соединения, как следует из их названия, являются способом одновременного запроса данных из нескольких таблиц, причем строки отображают столбцы, взятые из более чем одной таблицы.

Examples

Основное явное внутреннее соединение

Базовое соединение (также называемое «внутреннее соединение») запрашивает данные из двух таблиц с их отношением, определенным в предложении `join` .

В следующем примере будут выбраны имена сотрудников (FName) из таблицы Employees и название отдела, в котором они работают (имя) из таблицы Departments:

```
SELECT Employees.FName, Departments.Name
FROM   Employees
JOIN   Departments
ON Employees.DepartmentId = Departments.Id
```

Это вернет из базы данных примера следующее :

Employees.FName	Departments.Name
Джеймс	HR

Employees.FName	Departments.Name
Джон	HR
Ричард	Продажи

Неявное присоединение

Присоединяется также может быть выполнено при наличии нескольких таблиц в `from` пункта, разделенных запятыми `,` и определения отношений между ними в `where` п. Этот метод называется Implicit Join (поскольку он фактически не содержит предложение `join`).

Все РСУБД поддерживают его, но синтаксис обычно не рекомендуется. Причинами плохого использования этого синтаксиса являются:

- Можно получить случайные кросс-соединения, которые затем возвращают неверные результаты, особенно если у вас много запросов в запросе.
- Если вы намеревались перекрестное соединение, то это не ясно из синтаксиса (вместо этого выпишите CROSS JOIN), и кто-то, вероятно, изменит его во время обслуживания.

В следующем примере будут отобраны имена сотрудников и имена отделов, в которых они работают:

```
SELECT e.FName, d.Name
FROM   Employee e, Departments d
WHERE  e.DeptartmentId = d.Id
```

Это вернет из базы данных примера следующее :

e.FName	d.Name
Джеймс	HR
Джон	HR
Ричард	Продажи

Левая внешняя связь

Левая внешняя регистрация (также известная как «Левая регистрация» или «Внешняя связь») - это объединение, которое обеспечивает отображение всех строк из левой таблицы; если не существует подходящей строки из правой таблицы, соответствующие поля - `NULL` .

В следующем примере будут отобраны все отделы и первое имя сотрудников, работающих

в этом отделе. Отделы без сотрудников все еще возвращаются в результатах, но будут иметь NULL для имени сотрудника:

```
SELECT          Departments.Name, Employees.FName
FROM            Departments
LEFT OUTER JOIN Employees
ON              Departments.Id = Employees.DepartmentId
```

Это вернет из базы данных примера следующее :

Departments.Name	Employees.FName
HR	Джеймс
HR	Джон
HR	Джонатон
Продажи	Майкл
Технология	НОЛЬ

Так как же это работает?

В предложении FROM есть две таблицы:

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Оплата труда	Дата приема на работу
1	Джеймс	кузнец	1234567890	НОЛЬ	1	1000	01-01-2002
2	Джон	Джонсон	2468101214	1	1	400	23-03-2005
3	Майкл	Williams	1357911131	1	2	600	12-05-2009
4	Джонатон	кузнец	1212121212	2	1	500	24-07-2016

а также

Я бы	название
1	HR
2	Продажи
3	Технология

Сначала из двух таблиц создается *декартово* произведение, дающее промежуточную таблицу.
Записи, соответствующие критериям соединения (*Departments.Id = Employees.DepartmentId*) выделены жирным шрифтом; они передаются на следующий этап запроса.

Поскольку это LEFT OUTER JOIN, все записи возвращаются со стороны LEFT соединения (Departments), в то время как любые записи на стороне RIGHT имеют маркер NULL, если они не соответствуют критериям соединения. В приведенной ниже таблице возвращается **Tech** с `NULL`

Я бы	название	Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Опла труд
1	**HR**	**1**	**Джеймс**	**кузнец**	**1234567890**	**НОЛЬ**	**1**	**1000**
1	**HR**	**2**	**Джон**	**Джонсон**	**2468101214**	**1**	**1**	**400**
1	HR	3	Майкл	Williams	1357911131	1	2	600
1	**HR**	**4**	**Джонатон**	**кузнец**	**1212121212**	**2**	**1**	**500**
2	Продажи	1	Джеймс	кузнец	1234567890	НОЛЬ	1	1000
2	Продажи	2	Джон	Джонсон	2468101214	1	1	400
2	**Продажи**	**3**	**Майкл**	**Williams**	**1357911131**	**1**	**2**	**600**
2	Продажи	4	Джонатон	кузнец	1212121212	2	1	500
3	Технология	1	Джеймс	кузнец	1234567890	НОЛЬ	1	1000

Я бы	название	Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Опла труд
3	Технология	2	Джон	Джонсон	2468101214	1	1	400
3	Технология	3	Майкл	Williams	1357911131	1	2	600
3	Технология	4	Джонатон	кузнец	1212121212	2	1	500

Наконец, каждое выражение, используемое в предложении **SELECT** , вычисляется для возврата нашей итоговой таблицы:

Departments.Name	Employees.FName
HR	Джеймс
HR	Джон
Продажи	Ричард
Технология	НОЛЬ

Self Join

Таблица может быть соединена с самим собой, с разными строками, соответствующими друг другу. В этом случае использования следует использовать псевдонимы для того, чтобы отличить два вхождения таблицы.

В приведенном ниже примере для каждого сотрудника в таблице базы данных примера базы данных возвращается запись, содержащая имя первого сотрудника вместе с соответствующим первым именем менеджера сотрудника. Поскольку менеджеры также являются сотрудниками, таблица объединяется с собой:

```
SELECT
    e.FName AS "Employee",
    m.FName AS "Manager"
FROM
    Employees e
JOIN
    Employees m
    ON e.ManagerId = m.Id
```

Этот запрос вернет следующие данные:

Работник	Менеджер
Джон	Джеймс
Майкл	Джеймс
Джонатон	Джон

Так как же это работает?

Исходная таблица содержит следующие записи:

Я бы	FName	LName	Номер телефона	ManagerID	DepartmentID	Оплата труда	Дата приема на работу
1	Джеймс	кузнец	1234567890	НОЛЬ	1	1000	01-01-2002
2	Джон	Джонсон	2468101214	1	1	400	23-03-2005
3	Майкл	Williams	1357911131	1	2	600	12-05-2009
4	Джонатон	кузнец	1212121212	2	1	500	24-07-2016

Первое действие - создать *декартово* произведение всех записей в таблицах, используемых в предложении **FROM** . В этом случае это таблица Employees дважды, поэтому промежуточная таблица будет выглядеть так (я удалил все поля, не используемые в этом примере):

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
1	Джеймс	НОЛЬ	1	Джеймс	НОЛЬ
1	Джеймс	НОЛЬ	2	Джон	1
1	Джеймс	НОЛЬ	3	Майкл	1
1	Джеймс	НОЛЬ	4	Джонатон	2

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
2	Джон	1	1	Джеймс	НОЛЬ
2	Джон	1	2	Джон	1
2	Джон	1	3	Майкл	1
2	Джон	1	4	Джонатон	2
3	Майкл	1	1	Джеймс	НОЛЬ
3	Майкл	1	2	Джон	1
3	Майкл	1	3	Майкл	1
3	Майкл	1	4	Джонатон	2
4	Джонатон	2	1	Джеймс	НОЛЬ
4	Джонатон	2	2	Джон	1
4	Джонатон	2	3	Майкл	1
4	Джонатон	2	4	Джонатон	2

Следующее действие заключается только в том, чтобы сохранить записи, соответствующие критериям **JOIN** , поэтому любые записи, где aliased `e` table `ManagerId` равен `Id` таблицы aliased `m` :

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
2	Джон	1	1	Джеймс	НОЛЬ
3	Майкл	1	1	Джеймс	НОЛЬ
4	Джонатон	2	2	Джон	1

Затем каждое выражение, используемое в предложении **SELECT** , вычисляется для возврата этой таблицы:

e.FName	m.FName
Джон	Джеймс
Майкл	Джеймс
Джонатон	Джон

Наконец, имена столбцов `e.FName` и `m.FName` заменяются именами их псевдонимов, назначенными оператору AS :

Работник	Менеджер
Джон	Джеймс
Майкл	Джеймс
Джонатон	Джон

ПЕРЕКРЕСТНЫЙ ПРИСОЕДИНЕНИЕ

Cross join делает декартово произведение двух членов. Декартово произведение означает, что каждая строка из одной таблицы объединяется с каждой строкой второй таблицы в соединении. Например, если `TABLEA` имеет 20 строк и `TABLEB` имеет 20 строк, результатом будет `20*20 = 400` строк вывода.

Использование примерной базы данных

```
SELECT d.Name, e.FName
FROM   Departments d
CROSS JOIN Employees e;
```

Что возвращает:

d.Name	e.FName
HR	Джеймс
HR	Джон
HR	Майкл
HR	Джонатон
Продажи	Джеймс
Продажи	Джон
Продажи	Майкл
Продажи	Джонатон
Технология	Джеймс
Технология	Джон

d.Name	e.FName
Технология	Майкл
Технология	Джонатон

Рекомендуется написать явный CROSS JOIN, если вы хотите сделать декартовое соединение, чтобы подчеркнуть, что это то, что вы хотите.

Присоединение к подзапросу

Объединение подзапроса часто используется, когда вы хотите получить агрегированные данные из таблицы child / details и отображать их вместе с записями из таблицы parent / header. Например, вы можете захотеть получить количество дочерних записей, среднее число числовых столбцов в дочерних записях или верхнюю или нижнюю строку на основе поля даты или числа. В этом примере используются псевдонимы, которые, возможно, упрощают чтение запросов при использовании нескольких таблиц. Вот как выглядит довольно типичное подзапрос. В этом случае мы извлекаем все строки из родительской таблицы «Заказы на поставку» и извлекаем только первую строку для каждой родительской записи дочерней таблицы PurchaseOrderLineItems.

```
SELECT po.Id, po.PODate, po.VendorName, po.Status, item.ItemNo,
  item.Description, item.Cost, item.Price
FROM PurchaseOrders po
LEFT JOIN
     (
       SELECT l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price, Min(l.id) as Id
       FROM PurchaseOrderLineItems l
       GROUP BY l.PurchaseOrderId, l.ItemNo, l.Description, l.Cost, l.Price
     ) AS item ON item.PurchaseOrderId = po.Id
```

КРЕСТ ПРИМЕНЯЕТСЯ И ПОСЛЕДНЕЕ СОЕДИНЕНИЕ

Очень интересным типом JOIN является LATERAL JOIN (новый в PostgreSQL 9.3+), который также известен как CROSS APPLY / OUTER APPLY в SQL-Server & Oracle.

Основная идея заключается в том, что для каждой присоединяемой строки применяется табличная функция (или встроенный подзапрос).

Это позволяет, например, присоединить только первую совпадающую запись в другой таблице.
Разница между нормальным и боковым соединением заключается в том, что вы можете использовать столбец, который вы ранее вложили **в подзапрос,** который вы «CROSS APPLY».

Синтаксис:

PostgreSQL 9.3+

левый | право | внутреннее соединение **ЛАТЕРАЛЬНОГО**

SQL-сервер:

CROSS | ВНЕШНИЕ **ПРИМЕНЕНИЯ**

`INNER JOIN LATERAL` - это то же самое, что и `CROSS APPLY`
и `LEFT JOIN LATERAL` - это то же самое, что и `OUTER APPLY`

Пример использования (PostgreSQL 9.3+):

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989


LEFT JOIN LATERAL
(
    SELECT
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (__in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (__in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
   LIMIT 1
) AS FirstOE
```

И для SQL-Server

```
SELECT * FROM T_Contacts

--LEFT JOIN T_MAP_Contacts_Ref_OrganisationalUnit ON MAP_CTCOU_CT_UID = T_Contacts.CT_UID AND
MAP_CTCOU_SoftDeleteStatus = 1
--WHERE T_MAP_Contacts_Ref_OrganisationalUnit.MAP_CTCOU_UID IS NULL -- 989

-- CROSS APPLY -- = INNER JOIN
OUTER APPLY    -- = LEFT JOIN
(
    SELECT TOP 1
         --MAP_CTCOU_UID
         MAP_CTCOU_CT_UID
        ,MAP_CTCOU_COU_UID
```

```
        ,MAP_CTCOU_DateFrom
        ,MAP_CTCOU_DateTo
   FROM T_MAP_Contacts_Ref_OrganisationalUnit
   WHERE MAP_CTCOU_SoftDeleteStatus = 1
   AND MAP_CTCOU_CT_UID = T_Contacts.CT_UID

    /*
    AND
    (
        (@in_DateFrom <= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateTo)
        AND
        (@in_DateTo >= T_MAP_Contacts_Ref_OrganisationalUnit.MAP_KTKOE_DateFrom)
    )
    */
   ORDER BY MAP_CTCOU_DateFrom
) AS FirstOE
```

ПОЛНОЕ СОЕДИНЕНИЕ

Один тип JOIN, который менее известен, - это ПОЛНЫЙ ПРИСОЕДИНЕНИЕ.
(Примечание: FULL JOIN не поддерживается MySQL в соответствии с 2016)

FULL OUTER JOIN возвращает все строки из левой таблицы и все строки из правой таблицы.

Если в левой таблице есть строки, которые не имеют совпадений в правой таблице, или если в правой таблице есть строки, которые не имеют совпадений в левой таблице, то эти строки также будут перечислены.

Пример 1:

```
SELECT * FROM Table1

FULL JOIN Table2
     ON 1 = 2
```

Пример 2:

```
SELECT
     COALESCE(T_Budget.Year, tYear.Year) AS RPT_BudgetInYear
    ,COALESCE(T_Budget.Value, 0.0) AS RPT_Value
FROM T_Budget

FULL JOIN tfu_RPT_All_CreateYearInterval(@budget_year_from, @budget_year_to) AS tYear
      ON tYear.Year = T_Budget.Year
```

Обратите внимание: если вы используете soft-delete, вам нужно будет снова проверить статус soft-delete в предложении WHERE (потому что FULL JOIN ведет себя вроде как UNION);
Легко упустить этот маленький факт, поскольку вы добавляете AP_SoftDeleteStatus = 1 в предложение join.

Кроме того, если вы выполняете ПОЛНЫЙ ПРИСОЕДИНЯЙТЕСЬ, вам обычно нужно разрешить NULL в предложении WHERE; забыв позволить NULL по значению, будет иметь те же эффекты, что и INNER join, что вам не нужно, если вы выполняете ПОЛНЫЙ JOIN.

Пример:

```
SELECT
     T_AccountPlan.AP_UID
    ,T_AccountPlan.AP_Code
    ,T_AccountPlan.AP_Lang_EN
    ,T_BudgetPositions.BUP_Budget
    ,T_BudgetPositions.BUP_UID
    ,T_BudgetPositions.BUP_Jahr
FROM T_BudgetPositions

FULL JOIN T_AccountPlan
    ON T_AccountPlan.AP_UID = T_BudgetPositions.BUP_AP_UID
    AND T_AccountPlan.AP_SoftDeleteStatus = 1

WHERE (1=1)
AND (T_BudgetPositions.BUP_SoftDeleteStatus = 1 OR T_BudgetPositions.BUP_SoftDeleteStatus IS
NULL)
AND (T_AccountPlan.AP_SoftDeleteStatus = 1 OR T_AccountPlan.AP_SoftDeleteStatus IS NULL)
```

Рекурсивные СОБЫТИЯ

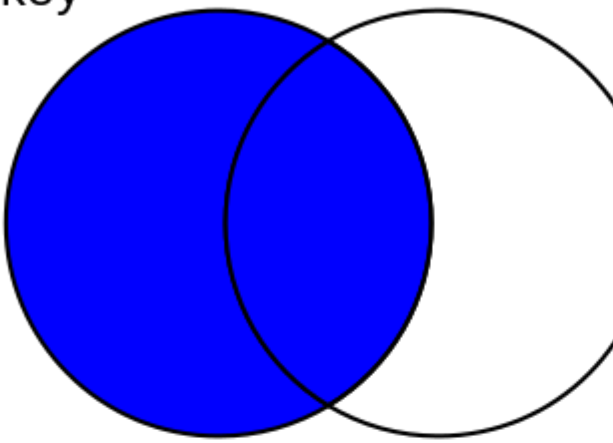
Рекурсивные объединения часто используются для получения данных родитель-ребенка. В SQL они реализованы с использованием рекурсивных общих табличных выражений , например:

```
WITH RECURSIVE MyDescendants AS (
    SELECT Name
    FROM People
    WHERE Name = 'John Doe'

    UNION ALL

    SELECT People.Name
    FROM People
    JOIN MyDescendants ON People.Name = MyDescendants.Parent
)
SELECT * FROM MyDescendants;
```

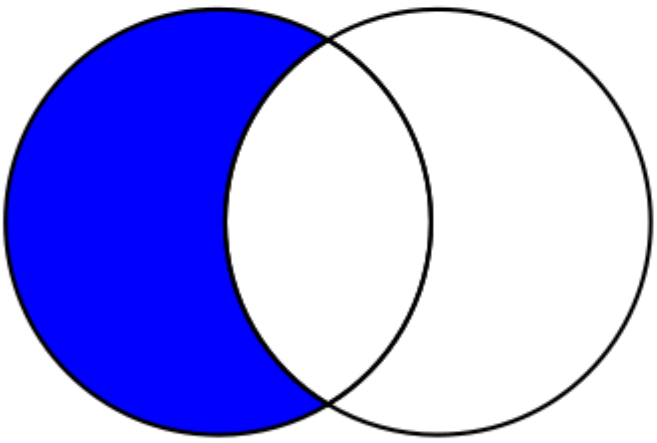
Различия между внутренними / внешними соединениями

SQL имеет различные типы соединений, чтобы указать, включены ли в результат (не) совпадающие строки: `INNER JOIN` , `LEFT OUTER JOIN` , `RIGHT OUTER JOIN` и `FULL OUTER JOIN` (ключевые слова `INNER` и `OUTER` являются необязательными). На следующем рисунке показаны различия между этими типами объединений: синяя область представляет результаты, возвращаемые соединением, а белая область представляет результаты, которые соединение не будет возвращено.

```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
```



```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
WHERE B.key IS NULL
```



являются общими, и (5,6) являются единственными для B.

Внутреннее соединение

Внутреннее соединение, использующее любой из эквивалентных запросов, дает пересечение двух таблиц, то есть двух строк, которые они имеют вместе:

```
select * from a INNER JOIN b on a.a = b.b;
select a.*,b.* from a,b where a.a = b.b;

a | b
--+--
3 | 3
4 | 4
```

Левое внешнее соединение

Левое внешнее соединение даст все строки в A плюс любые общие строки в B:

```
select * from a LEFT OUTER JOIN b on a.a = b.b;

a |  b
--+-----
1 | null
2 | null
3 |    3
4 |    4
```

Правое внешнее соединение

Точно так же правое внешнее соединение даст все строки в B плюс любые общие строки в A:

```
select * from a RIGHT OUTER JOIN b on a.a = b.b;

a    |  b
-----+----
3    |  3
4    |  4
null |  5
null |  6
```

Полное внешнее соединение

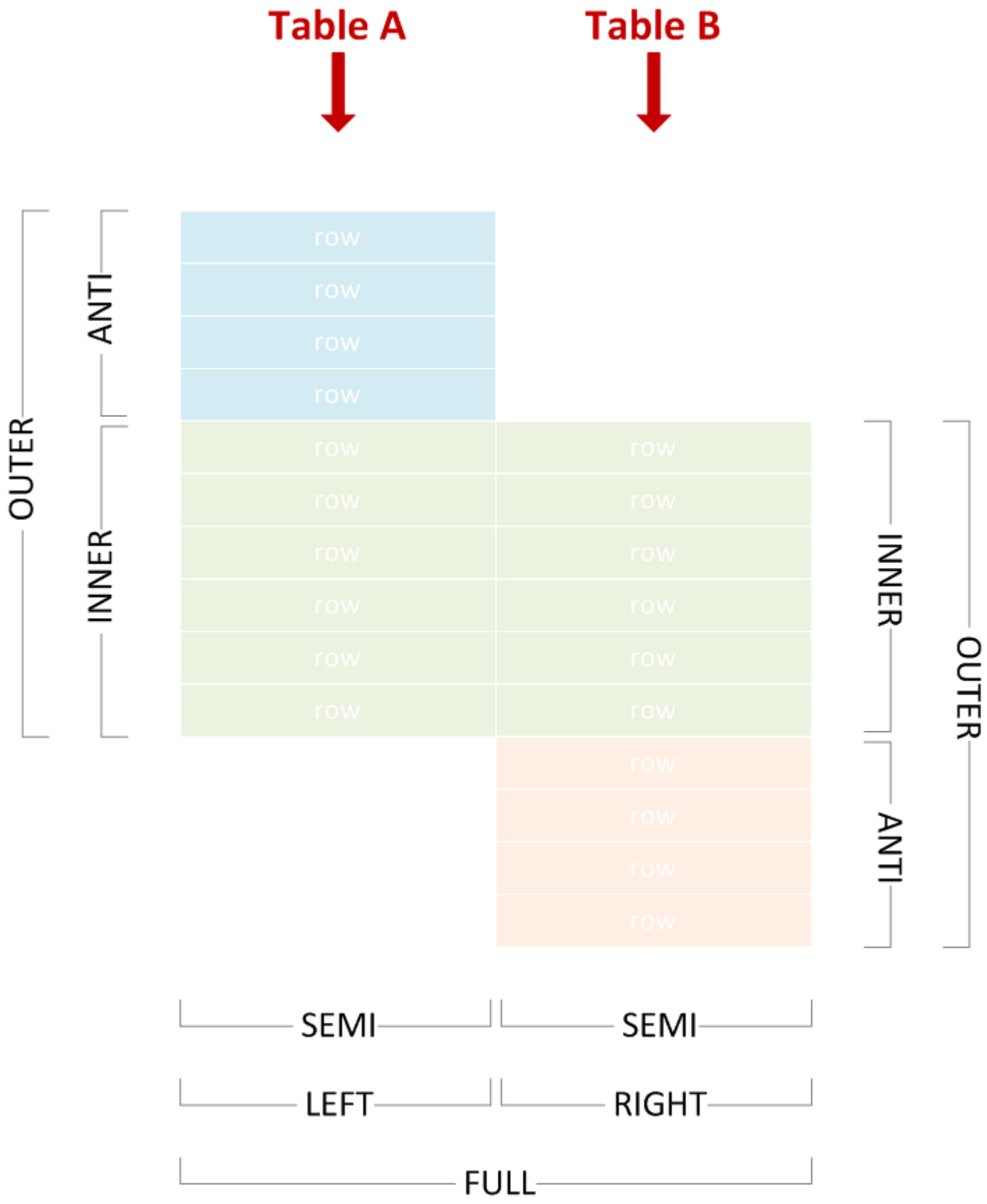
Полное внешнее соединение даст вам объединение A и B, т. Е. Все строки в A и все строки

в B. Если что-то в A не имеет соответствующей базы данных в B, то B-часть имеет значение NULL и наоборот.

```
select * from a FULL OUTER JOIN b on a.a = b.b;

 a    |  b
------+-----
    1 | null
    2 | null
    3 |    3
    4 |    4
null |    6
null |    5
```

JOIN Терминология: Внутренняя, Наружная, Полу, Анти ...

Допустим, у нас есть две таблицы (A и B), и некоторые из их строк соответствуют (относительно заданного условия JOIN, что бы это ни было в конкретном случае):



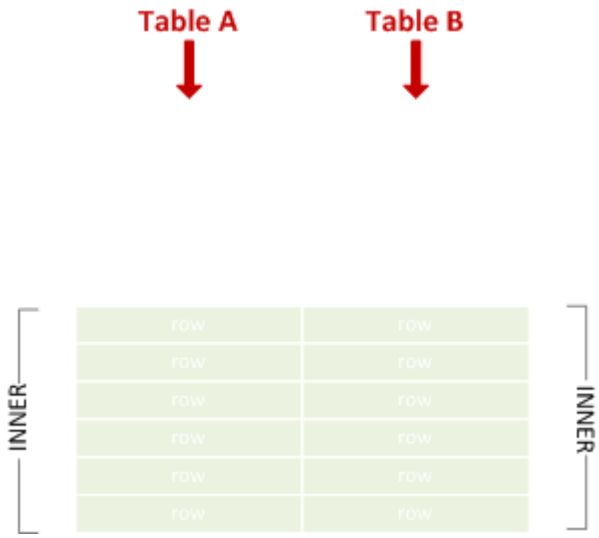
Мы можем использовать различные типы соединений для включения или исключения совпадающих или несогласованных строк с любой стороны и правильно называть объединение, выбирая соответствующие термины из приведенной выше диаграммы.

В приведенных ниже примерах используются следующие тестовые данные:

```
CREATE TABLE A (
    X varchar(255) PRIMARY KEY
);

CREATE TABLE B (
    Y varchar(255) PRIMARY KEY
);

INSERT INTO A VALUES
    ('Amy'),
    ('John'),
    ('Lisa'),
    ('Marco'),
    ('Phil');

INSERT INTO B VALUES
    ('Lisa'),
    ('Marco'),
    ('Phil'),
    ('Tim'),
    ('Vincent');
```

Внутреннее соединение

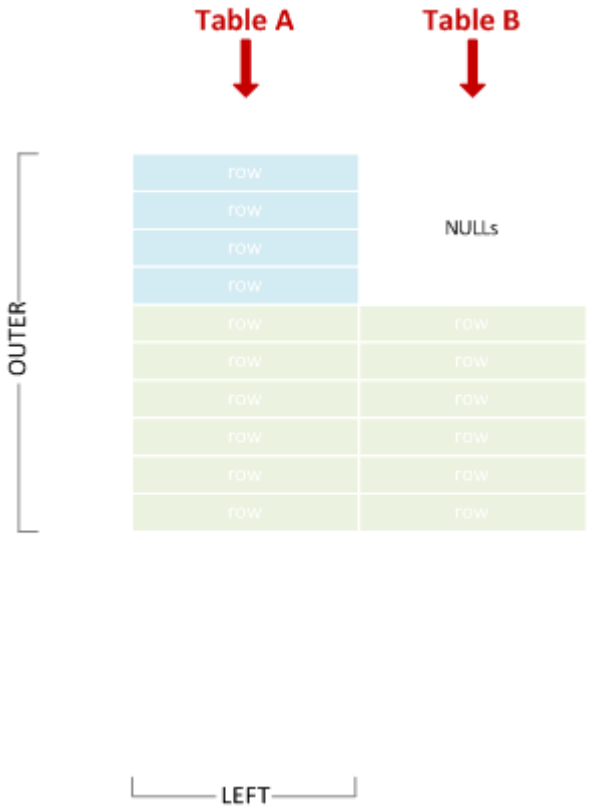
Объединяет левую и правую строки, которые соответствуют.




```
SELECT * FROM A JOIN B ON X = Y;

X      Y
------ -----
Lisa   Lisa
Marco  Marco
Phil   Phil
```

Левая внешняя связь

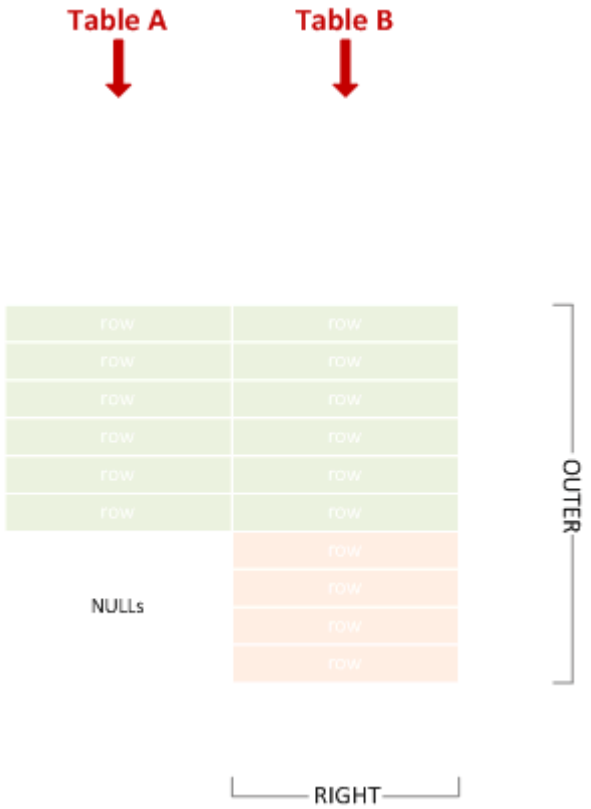
Иногда сокращается до «left join». Объединяет левую и правую строки, которые соответствуют, и включает несогласованные левые строки.




```
SELECT * FROM A LEFT JOIN B ON X = Y;

X      Y
-----  -----
Amy    NULL
John   NULL
Lisa   Lisa
Marco  Marco
Phil   Phil
```

Правостороннее соединение

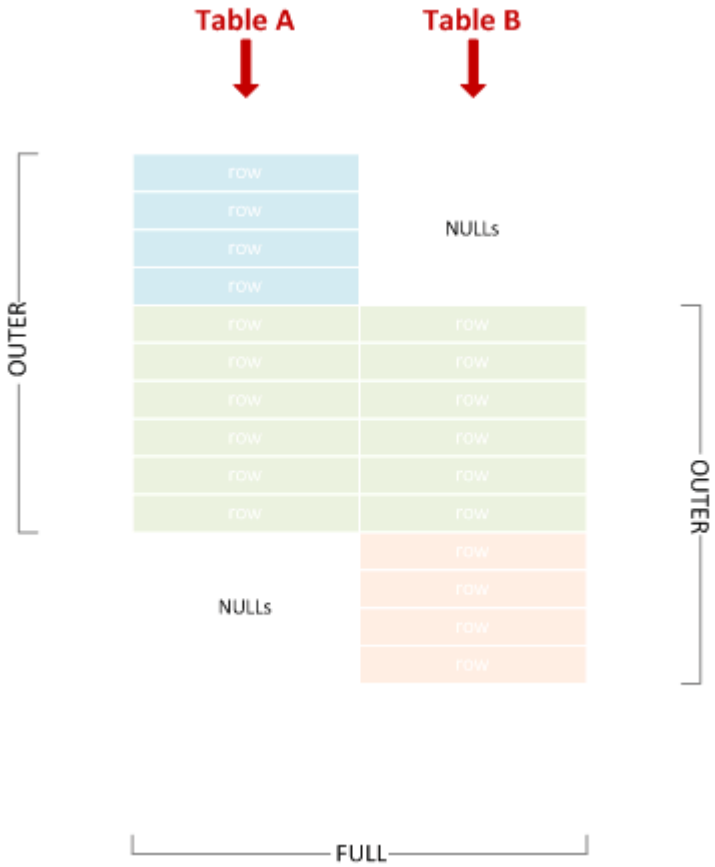
Иногда сокращается «правое соединение». Объединяет левую и правую строки, которые соответствуют, и включает несогласованные правые строки.




```
SELECT * FROM A RIGHT JOIN B ON X = Y;

X       Y
-----   -------
Lisa    Lisa
Marco   Marco
Phil    Phil
NULL    Tim
NULL    Vincent
```

Полная внешняя связь

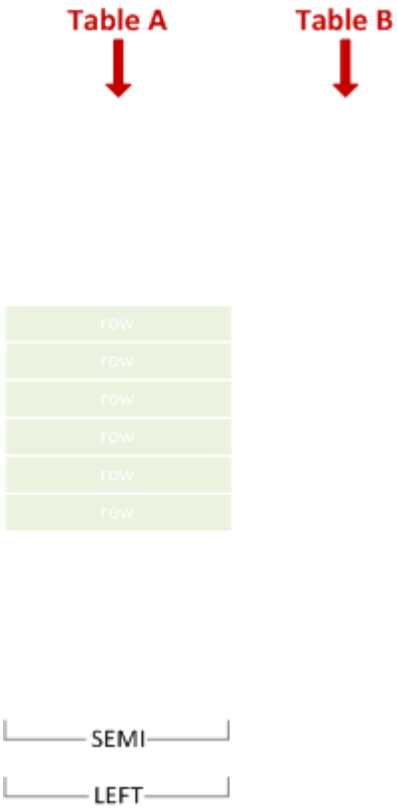
Иногда сокращается до «полного соединения». Союз левого и правого внешнего соединения.




```
SELECT * FROM A FULL JOIN B ON X = Y;

X       Y
-----   -------
Amy     NULL
John    NULL
Lisa    Lisa
Marco   Marco
Phil    Phil
NULL    Tim
NULL    Vincent
```

Левая полуось

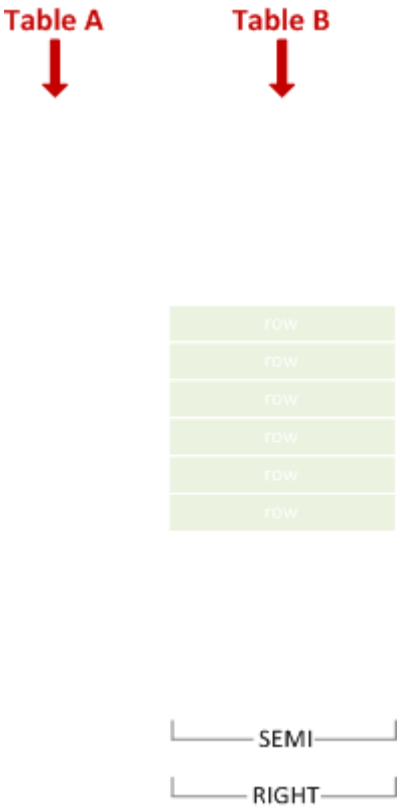
Включает левые строки, соответствующие строкам справа.




```
SELECT * FROM A WHERE X IN (SELECT Y FROM B);

X
-----
Lisa
Marco
Phil
```

Правая полупрофессионал

Включает правые строки, которые соответствуют левым строкам.

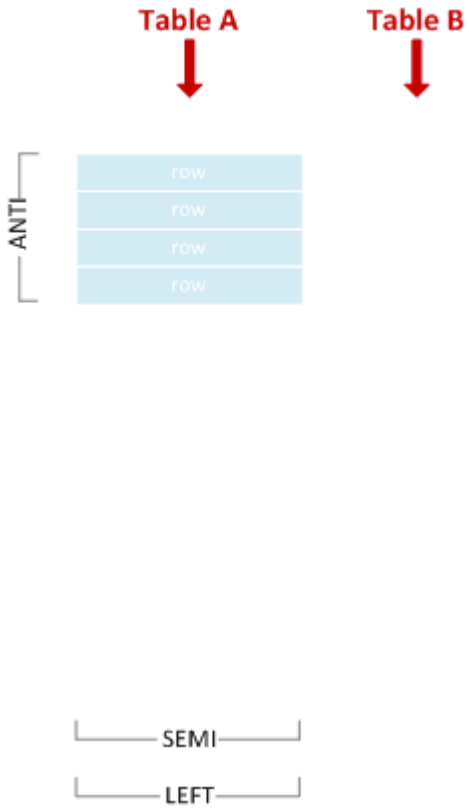



```
SELECT * FROM B WHERE Y IN (SELECT X FROM A);

Y
-----
Lisa
Marco
Phil
```

Как вы можете видеть, нет специального синтаксиса IN для левого и правого полусоединения - мы достигаем эффекта просто путем переключения позиций таблицы в тексте SQL.

Left Anti Semi Join

Включает левые строки, которые **не** соответствуют строкам справа.

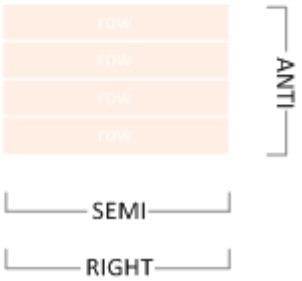



```
SELECT * FROM A WHERE X NOT IN (SELECT Y FROM B);

X
----
Amy
John
```

ПРЕДУПРЕЖДЕНИЕ. Будьте осторожны, если вы используете NOT IN в столбце NULL! Подробнее здесь .

Right Anti Semi Join

Включает правые строки, которые **не** соответствуют левым строкам.




```
SELECT * FROM B WHERE Y NOT IN (SELECT X FROM A);

Y
-------
Tim
Vincent
```

Как вы можете видеть, нет специального синтаксиса NOT IN для левого и правого анти-полусоединения - мы достигаем эффекта просто путем переключения позиций таблицы в тексте SQL.

Крест

Декартово произведение всех левых со всеми правыми строками.

```
SELECT * FROM A CROSS JOIN B;

X      Y
-----  -------
Amy    Lisa
John   Lisa
Lisa   Lisa
Marco  Lisa
Phil   Lisa
Amy    Marco
John   Marco
Lisa   Marco
Marco  Marco
```

```
Phil   Marco
Amy    Phil
John   Phil
Lisa   Phil
Marco  Phil
Phil   Phil
Amy    Tim
John   Tim
Lisa   Tim
Marco  Tim
Phil   Tim
Amy    Vincent
John   Vincent
Lisa   Vincent
Marco  Vincent
Phil   Vincent
```

Кросс-соединение эквивалентно внутреннему соединению с условием соединения, которое всегда совпадает, поэтому следующий запрос вернул бы тот же результат:

```
SELECT * FROM A JOIN B ON 1 = 1;
```

Автообъединение

Это просто означает, что таблица соединяется сама с собой. Самосоединением может быть любой из типов соединений, рассмотренных выше. Например, это внутреннее самосоединение:

```
SELECT * FROM A A1 JOIN A A2 ON LEN(A1.X) < LEN(A2.X);

X     X
----  -----
Amy   John
Amy   Lisa
Amy   Marco
John  Marco
Lisa  Marco
Phil  Marco
Amy   Phil
```

Прочитайте ПРИСОЕДИНИТЬСЯ онлайн: https://riptutorial.com/ru/sql/topic/261/присоединиться

глава 46: Просмотры

Examples

Простые виды

Представление может фильтровать некоторые строки из базовой таблицы или проекта только из нескольких столбцов:

```
CREATE VIEW new_employees_details AS
SELECT E.id, Fname, Salary, Hire_date
FROM Employees E
WHERE hire_date > date '2015-01-01';
```

Если вы выберете форму вида:

```
select * from new_employees_details
```

Я бы	FName	Оплата труда	Дата приема на работу
4	Джонатон	500	24-07-2016

Сложные виды

Представление может представлять собой действительно сложный запрос (агрегации, объединения, подзапросы и т. Д.). Просто убедитесь, что вы добавляете имена столбцов для всего, что вы выбираете:

```
Create VIEW dept_income AS
SELECT d.Name as DepartmentName, sum(e.salary) as TotalSalary
FROM Employees e
JOIN Departments d on e.DepartmentId = d.id
GROUP BY d.Name;
```

Теперь вы можете выбрать его из любой таблицы:

```
SELECT *
FROM dept_income;
```

Название отдела	TotalSalary
HR	1900
Продажи	600

Прочитайте Просмотры онлайн: https://riptutorial.com/ru/sql/topic/766/просмотры

глава 47: Реляционная алгебра

Examples

обзор

Реляционная алгебра не является полномасштабным языком *SQL* , а скорее способ получить теоретическое понимание реляционной обработки. Поэтому он не должен ссылаться на физические объекты, такие как таблицы, записи и поля; он должен ссылаться на абстрактные конструкции, такие как отношения, кортежи и атрибуты. Говоря это, я не буду использовать академические термины в этом документе и буду придерживаться более широко известных терминов непрофессионала - таблиц, записей и полей.

Пара правил реляционной алгебры, прежде чем мы начнем:

- Операторы, используемые в реляционной алгебре, работают на целых таблицах, а не на отдельных записях.
- Результатом реляционного выражения всегда будет таблица (это называется *свойством закрытия*)

Во всем этом документе я буду ссылаться на следующие две таблицы:

Departments

ID	Dept
1	Production
2	Quality Control

People

ID	PersonName	StartYear	ManagerID	DepartmentID
1	Darren	2005		1
2	David	2006	1	1
3	Burt	2006	1	1
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

ВЫБРАТЬ

Оператор **select** возвращает подмножество основной таблицы.
выберите *<table>*, ***где*** *<условие>*

Например, рассмотрите выражение:

выберите Люди, **где** DepartmentID = 2

Это можно записать так:

$$\sigma_{\text{DepartmentID} = 2}(\text{People})$$

Это приведет к тому, что таблица, записи которой состоят из всех записей в таблице *People,* где значение *DepartmentID* равно 2:

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

Условия также могут быть объединены для дальнейшего ограничения выражения:

выберите « Люди, **где** StartYear> 2005 **и** DepartmentID = 2

приведет к следующей таблице:

ID	PersonName	StartYear	ManagerID	DepartmentID
5	Fred	2008	4	2

ПРОЕКТ

Оператор **проекта** возвращает значения поля из таблицы.
project *<table>* ***over*** *<список полей>*

Например, рассмотрите следующее выражение:
проект Люди **старше** StartYear

Это можно записать так:

$$\Pi_{\text{StartYear}}(\text{People})$$

Это приведет к таблице, состоящей из отдельных значений, *хранящихся в* поле *StartYear* таблицы *People* .

StartYear
2005
2006
2004
2008

Дублирующие значения удаляются из результирующей таблицы из-за *свойства закрытия,* создающего реляционную таблицу: все записи в реляционной таблице должны быть различны.

Если *список полей* содержит более одного поля, тогда результирующая таблица является отдельной версией этих полей.
проект People **over** StartYear, DepartmentID вернется:

StartYear	DepartmentID
2005	1
2006	1
2004	2
2008	2
2005	2

Одна запись удаляется из-за дублирования 2006 *StartYear* и 1 *DepartmentID* .

ПРЕДОСТАВЛЕНИЕ

Реляционные выражения можно связать вместе, называя отдельные выражения с помощью ключевого слова **предоставления** или вставляя одно выражение в другое.

*<выражение реляционной алгебры>, **дающее** <имя псевдонима>*

Например, рассмотрим следующие выражения:
выберите People **where** DepartmentID = 2, **дающий** A
проект A **над** PersonName, **дающий** B

Это приведет к таблице B ниже, а таблица A будет результатом первого выражения.

A

ID	PersonName	StartYear	ManagerID	DepartmentID
4	Sarah	2004		2
5	Fred	2008	4	2
6	Joanne	2005	4	2

B

PersonName
Sarah
Fred
Joanne

Первое выражение оценивается, и результирующей таблице присваивается псевдоним A. Эта таблица затем используется во втором выражении, чтобы дать финальной таблице псевдоним B.

Другим способом написания этого выражения является замена имени псевдонима таблицы во втором выражении на весь текст первого выражения, заключенного в скобки:
проекта (**выберите** « Люди, **где** DepartmentID = 2»), **с** именем PersonName, которое **дает** B

Это называется *вложенным выражением* .

ПРИРОДНОЕ СОЕДИНЕНИЕ

Естественное объединение объединяет две таблицы, используя общее поле, разделяемое между таблицами.

***join** <table 1> **и** <table 2>, **где** <field 1> = <field 2>*
предполагая, что <поле 1> находится в <table 1>, а <field 2> находится в <table 2>.

Например, следующее выражение объединения присоединяется к *людям* и *отделам на* основе столбцов *DepartmentID* и *ID* в соответствующих таблицах:
присоединиться к людям **и** департаментам, **где** DepartmentID = ID

ID	PersonName	StartYear	ManagerID	DepartmentID	Dept
1	Darren	2005		1	Production
2	David	2006	1	1	Production
3	Burt	2006	1	1	Production
4	Sarah	2004		2	Quality Control
5	Fred	2008	4	2	Quality Control
6	Joanne	2005	4	2	Quality Control

Обратите внимание, что отображается только *идентификатор отдела* из таблицы « *Люди»,* а не *идентификатор* из таблицы « *Департамент»* . Должно быть показано только одно из сравниваемых полей, которое обычно является именем поля из первой таблицы в операции объединения.

Хотя это не показано в этом примере, возможно, что объединение таблиц может привести к тому, что два поля будут иметь один заголовок. Например, если я использовал заголовок *Name* для идентификации полей *PersonName* и *Dept* (т.е. для идентификации имени лица и имени отдела). Когда возникает такая ситуация, мы используем имя таблицы для определения имен полей с использованием точечной нотации: *People.Name* и *Departments.Name*

объединение в сочетании с **select** и **проектом** может использоваться вместе для получения информации:

присоединиться к *людям* ***и*** *отделам,* ***где*** *DepartmentID = ID,* ***дающий*** *A*
выберите *A,* ***где*** *StartYear = 2005* ***и*** *Dept = 'Production',* ***давая*** *B*
проект *B* ***через*** *PersonName,* ***дающий*** *C*

или как комбинированное выражение:

проект *(* ***выберите*** *(* ***присоединитесь к*** *людям* ***и*** *отделениям,* ***где*** *DepartmentID = ID),* ***где*** *StartYear = 2005* ***и*** *Dept = 'Production')* ***над*** *PersonName,* ***дающим*** *C*

Это приведет к этой таблице:

PersonName
Darren

ALIAS

ДЕЛИТЬ

UNION

INTERSECTION

РАЗНИЦА

UPDATE (: =)

TIMES

Прочитайте Реляционная алгебра онлайн: https://riptutorial.com/ru/sql/topic/7311/реляционная-алгебра

глава 48: Синонимы

Examples

Создать синоним

```
CREATE SYNONYM EmployeeData
FOR MyDatabase.dbo.Employees
```

Прочитайте Синонимы онлайн: https://riptutorial.com/ru/sql/topic/2518/синонимы

глава 49: СОЗДАТЬ ТАБЛИЦУ

Вступление

Оператор CREATE TABLE используется для создания новой таблицы в базе данных. Определение таблицы состоит из списка столбцов, их типов и любых ограничений целостности.

Синтаксис

- CREATE TABLE tableName ([ColumnName1] [datatype1] [, [ColumnName2] [datatype2] ...])

параметры

параметр	подробности
TABLENAME	Название таблицы
столбцы	Содержит «перечисление» всех столбцов таблицы. Подробнее см. « **Создание новой таблицы**» .

замечания

Имена таблиц должны быть уникальными.

Examples

Создать новую таблицу

Таблица основных `Employees` , содержащая идентификатор, а также имя и фамилию сотрудника вместе с их номером телефона могут быть созданы с использованием

```
CREATE TABLE Employees(
    Id int identity(1,1) primary key not null,
    FName varchar(20) not null,
    LName varchar(20) not null,
    PhoneNumber varchar(10) not null
);
```

Этот пример относится к Transact-SQL

`CREATE TABLE` создает новую таблицу в базе данных, за которой следует имя таблицы, `Employees`

Затем следует список имен столбцов и их свойств, таких как идентификатор

```
Id int identity(1,1) not null
```

Значение	Имея в виду
`Id`	имя столбца.
`int`	это тип данных.
`identity(1,1)`	указывает, что столбец будет иметь автоматически сгенерированные значения, начиная с 1 и увеличивая на 1 для каждой новой строки.
`primary key`	что все значения в этом столбце будут иметь уникальные значения
`not null`	заявляет, что этот столбец не может иметь нулевые значения

Создать таблицу из списка

Вы можете создать дубликат таблицы:

```
CREATE TABLE ClonedEmployees AS SELECT * FROM Employees;
```

Вы можете использовать любые другие функции оператора SELECT для изменения данных перед передачей их в новую таблицу. Столбцы новой таблицы автоматически создаются в соответствии с выбранными строками.

```
CREATE TABLE ModifiedEmployees AS
SELECT Id, CONCAT(FName," ",LName) AS FullName FROM Employees
WHERE Id > 10;
```

Дублировать таблицу

Чтобы дублировать таблицу, выполните следующие действия:

```
CREATE TABLE newtable LIKE oldtable;
INSERT newtable SELECT * FROM oldtable;
```

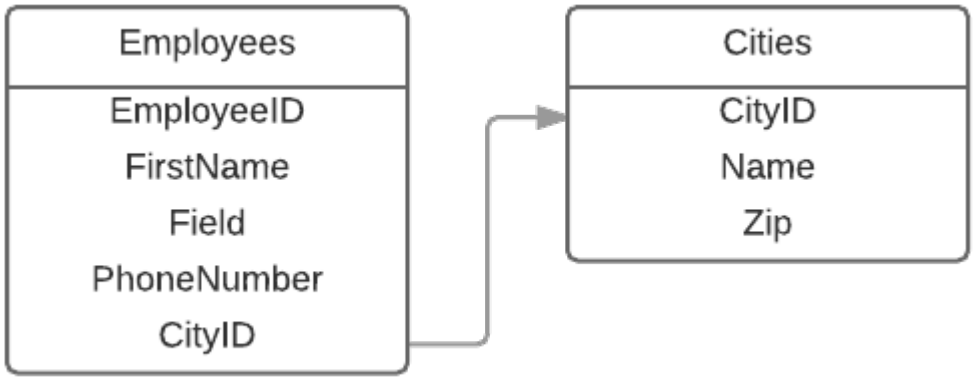
СОЗДАТЬ ТАБЛИЦУ С ИНОСТРАННЫМ КЛЮЧОМ

Ниже вы можете найти таблицу `Employees` со ссылкой на таблицу `Cities` .

```
CREATE TABLE Cities(
    CityID INT IDENTITY(1,1) NOT NULL,
    Name VARCHAR(20) NOT NULL,
    Zip VARCHAR(10) NOT NULL
);
```

```
CREATE TABLE Employees(
    EmployeeID INT IDENTITY (1,1) NOT NULL,
    FirstName VARCHAR(20) NOT NULL,
    LastName VARCHAR(20) NOT NULL,
    PhoneNumber VARCHAR(10) NOT NULL,
    CityID INT FOREIGN KEY REFERENCES Cities(CityID)
);
```

Здесь вы можете найти диаграмму базы данных.



Столбец `CityID` таблицы `Employees` будут ссылаться на столбец `CityID` таблицы `Cities` . Ниже вы можете найти синтаксис, чтобы сделать это.

```
CityID INT FOREIGN KEY REFERENCES Cities(CityID)
```

Значение	Имея в виду
`CityID`	Название столбца
`int`	тип столбца
`FOREIGN KEY`	Делает внешний ключ *(необязательно)*
`REFERENCES Cities(CityID)`	Делает ссылку к столбцу « `Cities CityID`

Важно: вы не могли ссылаться на таблицу, которая не существует в базе данных. Будьте источником, чтобы сделать сначала таблицу `Cities` а затем таблицу `Employees` . Если вы сделаете это наоборот, это вызовет ошибку.

Создание временной или внутренней памяти

PostgreSQL и SQLite

Чтобы создать временную таблицу, локальную для сеанса:

```
CREATE TEMP TABLE MyTable(...);
```

SQL Server

Чтобы создать временную таблицу, локальную для сеанса:

```
CREATE TABLE #TempPhysical(...);
```

Чтобы создать временную таблицу для всех:

```
CREATE TABLE ##TempPhysicalVisibleToEveryone(...);
```

Чтобы создать таблицу в памяти:

```
DECLARE @TempMemory TABLE(...);
```

Прочитайте СОЗДАТЬ ТАБЛИЦУ онлайн: https://riptutorial.com/ru/sql/topic/348/создать-таблицу

глава 50: СОЗДАТЬ ФУНКЦИЮ

Синтаксис

- CREATE FUNCTION function_name ([list_of_paramenters]) RETURNS return_data_type AS BEGIN function_body RETURN scalar_expression END

параметры

аргументация	Описание
function_name	имя функции
list_of_paramenters	параметры, которые функция принимает
return_data_type	тип, который возвращает функция. Некоторые типы данных SQL
function_body	код функции
scalar_expression	скалярное значение, возвращаемое функцией

замечания

CREATE FUNCTION создает пользовательскую функцию, которая может использоваться при выполнении запросов SELECT, INSERT, UPDATE или DELETE. Функции могут быть созданы для возврата одной переменной или отдельной таблицы.

Examples

Создать новую функцию

```
CREATE FUNCTION FirstWord (@input varchar(1000))
RETURNS varchar(1000)
AS
BEGIN
    DECLARE @output varchar(1000)
    SET @output = SUBSTRING(@input, 0, CASE CHARINDEX(' ', @input)
        WHEN 0 THEN LEN(@input) + 1
        ELSE CHARINDEX(' ', @input)
    END)

    RETURN @output
END
```

В этом примере создается функция с именем **FirstWord** , которая принимает параметр

varchar и возвращает другое значение varchar.

Прочитайте СОЗДАТЬ ФУНКЦИЮ онлайн: https://riptutorial.com/ru/sql/topic/2437/создать-функцию

глава 51: СОРТИРОВАТЬ ПО

Examples

Используйте ORDER BY с TOP, чтобы вернуть верхние строки x на основе значения столбца

В этом примере мы можем использовать GROUP BY определяется не только *вид* из возвращаемых строк, но и какие строки *будут* возвращены, так как мы используем TOP ограничить набор результатов.

Предположим, мы хотим вернуть лучших 5 наивысших пользователей репутации из неназванного популярного сайта Q & A.

Без ORDER BY

Этот запрос возвращает верхние 5 строк, упорядоченных по умолчанию, в этом случае это «Id», первый столбец в таблице (хотя это не столбец, показанный в результатах).

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

возвращается ...

Отображаемое имя	репутации
сообщество	1
Джефф Далгас	12567
Джаррод Диксон	11739
Джефф Этвуд	37628
Джоэл Спольский	25784

С ORDER BY

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
ORDER BY Reputation desc
```

возвращается ...

Отображаемое имя	репутации
JonSkeet	**865023**
Дарин Димитров	**661741**
BalusC	**650237**
Ханс Пассант	**625870**
Марк Гравелл	**601636**

замечания

Некоторые версии SQL (например, MySQL) используют предложение `LIMIT` в конце `SELECT` , а не `TOP` в начале, например:

```
SELECT DisplayName, Reputation
FROM Users
ORDER BY Reputation DESC
LIMIT 5
```

Сортировка по нескольким столбцам

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY JoinDate, Reputation
```

Отображаемое имя	Дате вступления	репутации
сообщество	**2008-09-15**	**1**
Джефф Этвуд	**2008-09-16**	**25784**
Джоэл Спольский	2008-09-16	**37628**
Джаррод Диксон	**2008-10-03**	**11739**
Джефф Далгас	2008-10-03	**12567**

Сортировка по столбцу (вместо имени)

Вы можете использовать номер столбца (где самый левый столбец равен «1»), чтобы указать, какой столбец будет содержать сортировку, вместо описания столбца по его имени.

Pro: Если вы считаете, что скорее всего, вы можете изменить имена столбцов позже, это не нарушит этот код.

Con: Это, как правило, снижает читаемость запроса (сразу же понятно, что означает « ORDER BY Reputation», в то время как «ORDER BY 14» требует некоторого подсчета, возможно, пальцем на экране.)

Этот запрос сортирует результат по информации в относительной позиции столбца `3` из оператора select вместо имени столбца « `Reputation` .

```
SELECT DisplayName, JoinDate, Reputation
FROM Users
ORDER BY 3
```

Отображаемое имя	Дате вступления	репутации
сообщество	2008-09-15	**1**
Джарред Диксон	2008-10-03	**11739**
Джефф Далгас	2008-10-03	**12567**
Джоэл Спольский	2008-09-16	**25784**
Джефф Этвуд	2008-09-16	**37628**

Сортировать по Alias

Из-за логического порядка обработки запросов псевдоним может использоваться по порядку.

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY jd, rep
```

И можете использовать относительный порядок столбцов в инструкции select. Рассмотрим тот же пример, что и выше, и вместо использования псевдонима используйте относительный порядок, например, для отображаемого имени: 1, для Jd - 2 и так далее

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY 2, 3
```

Настроенный порядок сортировки

Чтобы отсортировать эту таблицу `Employee` отдела, вы должны использовать `ORDER BY Department` . Однако, если вам нужен другой порядок сортировки, который не является алфавитным, вам необходимо сопоставить значения `Department` с разными значениями, которые сортируются правильно; это можно сделать с помощью выражения CASE:

название	отдел
Hasan	ЭТО
Юсуф	HR
Hillary	HR
Джо	ЭТО
веселый	HR
кругозор	бухгалтер

```
SELECT *
FROM Employee
ORDER BY CASE Department
         WHEN 'HR'         THEN 1
         WHEN 'Accountant' THEN 2
         ELSE                   3
         END;
```

название	отдел
Юсуф	**HR**
Hillary	**HR**
веселый	**HR**
кругозор	**бухгалтер**
Hasan	**ЭТО**
Джо	**ЭТО**

Прочитайте СОРТИРОВАТЬ ПО онлайн: https://riptutorial.com/ru/sql/topic/620/сортировать-по

глава 52: СОЮЗ / СОЮЗ ВСЕ

Вступление

Ключевое слово ***UNION*** в SQL используется для объединения результатов ***SELECT*** с любым дубликатом. Чтобы использовать UNION и комбинировать результаты, оператор SELECT должен иметь одинаковое количество столбцов с одинаковым типом данных в том же порядке, но длина столбца может быть разной.

Синтаксис

- SELECT column_1 [, column_2] FROM table_1 [, table_2] [Условие WHERE]
  СОЮЗ | **СОЮЗ ВСЕ**
  SELECT column_1 [, column_2] FROM table_1 [, table_2] [Условие WHERE]

замечания

`UNION` и `UNION ALL` объединяют результирующий набор из двух или более идентично структурированных операторов SELECT в единый результат / таблицу.

Оба столбца и типы столбцов для каждого запроса должны совпадать, чтобы работать с `UNION` / `UNION ALL` .

Разница между запросом `UNION` и `UNION ALL` заключается в том, что предложение `UNION` удалит любые повторяющиеся строки в результате, когда `UNION ALL` не будет.

Это отличное удаление записей может значительно замедлить запросы, даже если нет четких строк, которые должны быть удалены из-за этого, если вы знаете, что не будет никаких дубликатов (или не важно) всегда по умолчанию для `UNION ALL` для более оптимизированного запроса.

Examples

Основной запрос UNION ALL

```
CREATE TABLE HR_EMPLOYEES
(
    PersonID int,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
    Position VARCHAR(30)
);

CREATE TABLE FINANCE_EMPLOYEES
```

```
(
    PersonID INT,
    LastName VARCHAR(30),
    FirstName VARCHAR(30),
    Position VARCHAR(30)
);
```

Предположим, мы хотим извлечь имена всех `managers` из наших отделов.

Используя `UNION` мы можем получить всех сотрудников из отдела кадров и финансов, которые занимают `position manager`

```
SELECT
    FirstName, LastName
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

Оператор `UNION` удаляет повторяющиеся строки из результатов запроса. Поскольку в обоих отделах есть люди с одинаковым именем и позицией, мы используем `UNION ALL` , чтобы не удалять дубликаты.

Если вы хотите использовать псевдоним для каждого столбца вывода, вы можете просто поместить их в первый оператор select, как показано ниже:

```
SELECT
    FirstName as 'First Name', LastName as 'Last Name'
FROM
    HR_EMPLOYEES
WHERE
    Position = 'manager'
UNION ALL
SELECT
    FirstName, LastName
FROM
    FINANCE_EMPLOYEES
WHERE
    Position = 'manager'
```

Простое объяснение и пример

Проще говоря:

- `UNION` объединяет 2 набора результатов при удалении дубликатов из набора результатов

- `UNION ALL` объединяет 2 набора результатов, не пытаясь удалить дубликаты

  Одна из ошибок, которую делают многие люди, - использовать `UNION` когда им не нужно удалять дубликаты. Дополнительные затраты по сравнению с большими наборами результатов могут быть очень значительными.

Когда вам понадобится `UNION`

Предположим, вам нужно отфильтровать таблицу с двумя разными атрибутами, и вы создали отдельные некластеризованные индексы для каждого столбца. `UNION` позволяет вам использовать оба индекса при одновременном предотвращении дублирования.

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1
UNION
SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

Это упрощает настройку производительности, поскольку для оптимального выполнения этих запросов необходимы простые индексы. Вы даже можете обойтись с небольшим количеством некластеризованных индексов, улучшая общую производительность записи по сравнению с исходной таблицей.

Когда вам может понадобиться `UNION ALL`

Предположим, вам по-прежнему необходимо отфильтровать таблицу по двум атрибутам, но вам не нужно фильтровать повторяющиеся записи (либо потому, что это не имеет значения, либо ваши данные не будут создавать дубликатов во время объединения из-за вашей модели данных).

```
SELECT C1 FROM Table1
UNION ALL
SELECT C1 FROM Table2
```

Это особенно полезно при создании представлений, которые объединяют данные, которые предназначены для физического разделения по нескольким таблицам (возможно, по соображениям производительности, но по-прежнему хотят свертывать записи). Так как данные уже разделены, то с помощью механизма удаления базы данных дубликаты не добавляют значения и просто добавляют дополнительное время обработки к запросам.

Прочитайте СОЮЗ / СОЮЗ ВСЕ онлайн: https://riptutorial.com/ru/sql/topic/349/союз---союз-все

глава 53: Строковые функции

Вступление

Строковые функции выполняют операции с строковыми значениями и возвращают либо числовые, либо строковые значения.

Используя строковые функции, вы можете, например, комбинировать данные, извлекать подстроку, сравнивать строки или преобразовывать строку ко всем строчным или строчным символам.

Синтаксис

- CONCAT (string_value1, string_value2 [, string_valueN])
- LTRIM (символьное выражение)
- RTRIM (character_expression)
- SUBSTRING (выражение, начало, длина)
- ASCII (символьное выражение)
- REPLICATE (string_expression, integer_expression)
- REVERSE (string_expression)
- ВЕРХНИЙ (символьное выражение)
- TRIM (строка [characters FROM])
- STRING_SPLIT (строка, разделитель)
- STUFF (character_expression, start, length, replaceWith_expression)
- REPLACE (string_expression, string_pattern, string_replacement)

замечания

Ссылка на строковые функции для Transact-SQL / Microsoft

Ссылка на строковые функции для MySQL

Ссылка на String для PostgreSQL

Examples

Обрезать пустые пространства

Trim используется для удаления пространства записи в начале или в конце выбора

В MSSQL нет единого `TRIM()`

```
SELECT LTRIM('  Hello  ') --returns 'Hello  '
SELECT RTRIM('  Hello  ') --returns '  Hello'
SELECT LTRIM(RTRIM('  Hello  ')) --returns 'Hello'
```

MySql и Oracle

```
SELECT TRIM('  Hello  ') --returns 'Hello'
```

Объединить

В стандарте ANSI / ISO SQL оператор для конкатенации строк имеет значение `||` , Этот синтаксис поддерживается всеми основными базами данных, кроме SQL Server:

```
SELECT 'Hello' || 'World' || '!'; --returns HelloWorld!
```

Многие базы данных поддерживают функцию `CONCAT` для объединения строк:

```
SELECT CONCAT('Hello', 'World'); --returns 'HelloWorld'
```

Некоторые базы данных поддерживают использование `CONCAT` для объединения более двух строк (Oracle не работает):

```
SELECT CONCAT('Hello', 'World', '!'); --returns 'HelloWorld!'
```

В некоторых базах данных должны быть отлиты или конвертированы нестроковые типы:

```
SELECT CONCAT('Foo', CAST(42 AS VARCHAR(5)), 'Bar'); --returns 'Foo42Bar'
```

Некоторые базы данных (например, Oracle) выполняют неявные конверсии без потерь. Например, `CONCAT` на `CLOB` и `NCLOB` дает `NCLOB` . `CONCAT` на число и `varchar2` приводит к `varchar2` и т. Д .:

```
SELECT CONCAT(CONCAT('Foo', 42), 'Bar') FROM dual; --returns Foo42Bar
```

Некоторые базы данных могут использовать нестандартный `+` оператор (но в большинстве `+` работает только для чисел):

```
SELECT 'Foo' + CAST(42 AS VARCHAR(5)) + 'Bar';
```

В SQL Server <2012, где `CONCAT` не поддерживается, `+` - единственный способ присоединиться к строкам.

Верхний и нижний регистр

```
SELECT UPPER('HelloWorld') --returns 'HELLOWORLD'
```

```
SELECT LOWER('HelloWorld') --returns 'helloworld'
```

Substring

Синтаксис: `SUBSTRING ( string_expression, start, length )` . Обратите внимание, что строки SQL являются 1-индексированными.

```
SELECT SUBSTRING('Hello', 1, 2) --returns 'He'
SELECT SUBSTRING('Hello', 3, 3) --returns 'llo'
```

Это часто используется в сочетании с функцией `LEN()` чтобы получить последние `n` символов строки неизвестной длины.

```
DECLARE @str1 VARCHAR(10) = 'Hello', @str2 VARCHAR(10) = 'FooBarBaz';
SELECT SUBSTRING(@str1, LEN(@str1) - 2, 3) --returns 'llo'
SELECT SUBSTRING(@str2, LEN(@str2) - 2, 3) --returns 'Baz'
```

Трещина

Разделяет строковое выражение с помощью разделителя символов. Обратите внимание, что `STRING_SPLIT()` является табличной функцией.

```
SELECT value FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' ');
```

Результат:

```
value
-----
Lorem
ipsum
dolor
sit
amet.
```

дрянь

Поместите строку в другую, заменив 0 или более символов в определенной позиции.

Примечание: `start` позиция 1-индексация (вы начинаете индексирование с 1, а не 0).

Синтаксис:

```
STUFF ( character_expression , start , length , replaceWith_expression )
```

Пример:

```
SELECT STUFF('FooBarBaz', 4, 3, 'Hello') --returns 'FooHelloBaz'
```

длина

SQL Server

LEN не считает конечное пространство.

```
SELECT LEN('Hello') -- returns 5

SELECT LEN('Hello '); -- returns 5
```

DATALENGTH подсчитывает конечное пространство.

```
SELECT DATALENGTH('Hello') -- returns 5

SELECT DATALENGTH('Hello '); -- returns 6
```

Следует отметить, однако, что DATALENGTH возвращает длину базового байтового представления строки, которая зависит, ia, от кодировки, используемой для хранения строки.

```
DECLARE @str varchar(100) = 'Hello ' --varchar is usually an ASCII string, occupying 1 byte
per char
SELECT DATALENGTH(@str) -- returns 6

DECLARE @nstr nvarchar(100) = 'Hello ' --nvarchar is a unicode string, occupying 2 bytes per
char
SELECT DATALENGTH(@nstr) -- returns 12
```

оракул

Синтаксис: Length (char)

Примеры:

```
SELECT Length('Bible') FROM dual; --Returns 5
SELECT Length('righteousness') FROM dual; --Returns 13
SELECT Length(NULL) FROM dual; --Returns NULL
```

См. Также: LengthB, LengthC, Length2, Length4

замещать

Синтаксис:

`REPLACE(` String для поиска , String для поиска и замены , String для размещения в исходной строке `)`

Пример:

```
SELECT REPLACE( 'Peter Steve Tom', 'Steve', 'Billy' ) --Return Values: Peter Billy Tom
```

ЛЕВО ПРАВО

Синтаксис:
LEFT (строковое выражение, целое число)
RIGHT (строковое выражение, целое число)

```
SELECT LEFT('Hello',2)  --return He
SELECT RIGHT('Hello',2) --return lo
```

Oracle SQL не имеет функций LEFT и RIGHT. Их можно эмулировать с помощью SUBSTR и LENGTH.
SUBSTR (строковое выражение, 1, целое число)
SUBSTR (string-expression, length (string-expression) -integer + 1, integer)

```
SELECT SUBSTR('Hello',1,2)  --return He
SELECT SUBSTR('Hello',LENGTH('Hello')-2+1,2) --return lo
```

ЗАДНИЙ ХОД

Синтаксис: REVERSE (строковое выражение)

```
SELECT REVERSE('Hello') --returns olleH
```

REPLICATE

Функция `REPLICATE` объединяет строку с самим собой определенное количество раз.

Синтаксис: REPLICATE (string-expression, integer)

```
SELECT REPLICATE ('Hello',4) --returns 'HelloHelloHelloHello'
```

REGEXP

MySQL 3.19

Проверяет, соответствует ли строка регулярному выражению (определенному другой строкой).

```
SELECT 'bedded' REGEXP '[a-f]' -- returns True

SELECT 'beam' REGEXP '[a-f]' -- returns False
```

Заменить функцию в SQL-запросе Select and Update

Функция Replace в SQL используется для обновления содержимого строки. Вызов функции REPLACE () для MySQL, Oracle и SQL Server.

Синтаксис функции Replace:

```
REPLACE (str, find, repl)
```

Следующий пример заменяет появление таблицы `South` with `Southern` in Employees:

Имя	Адрес
Джеймс	Южный Нью-Йорк
Джон	Южный Бостон
Майкл	Южный Сан-Диего

Выберите Заявление:

Если мы применим следующую функцию Replace:

```
SELECT
    FirstName,
    REPLACE (Address, 'South', 'Southern') Address
FROM Employees
ORDER BY FirstName
```

Результат:

Имя	Адрес
Джеймс	Южный Нью-Йорк
Джон	Южный Бостон
Майкл	Южный Сан-Диего

Заявление о обновлении:

Мы можем использовать функцию замены для внесения постоянных изменений в нашу таблицу с помощью следующего подхода.

```
Update Employees
Set city = (Address, 'South', 'Southern');
```

Более общий подход заключается в использовании этого в сочетании с предложением WHERE следующим образом:

```
Update Employees
Set Address = (Address, 'South', 'Southern')
Where Address LIKE 'South%';
```

ParseName

БАЗЫ ДАННЫХ : SQL Server

Функция **PARSENAME** возвращает определенную часть заданной строки (имя объекта). имя объекта может содержать строку, такую как имя объекта, имя владельца, имя базы данных и имя сервера.

Подробнее `MSDN: PARSENAME`

Синтаксис

```
PARSENAME('NameOfStringToParse',PartIndex)
```

пример

Чтобы получить имя объекта, используйте индекс детали `1`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',1)  // returns `ObjectName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',1)     // returns `Student`
```

Чтобы получить имя схемы, используйте параметр part `2`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',2)  // returns `SchemaName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',2)     // returns `school`
```

Чтобы получить имя базы данных, используйте индекс детали `3`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',3) // returns `DatabaseName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',3)    // returns `SchoolDatabase`
```

Чтобы получить имя сервера, используйте индекс детали `4`

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',4)  // returns `ServerName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',4)     // returns `[1012-1111]`
```

PARSENAME возвращает null указанная часть отсутствует в заданной строке имени объекта

INSTR

Возвращает индекс первого вхождения подстроки (ноль, если не найден)

Синтаксис: INSTR (строка, подстрока)

```
SELECT INSTR('FooBarBar', 'Bar') -- return 4
SELECT INSTR('FooBarBar', 'Xar') -- return 0
```

Прочитайте Строковые функции онлайн: https://riptutorial.com/ru/sql/topic/1120/строковые-функции

глава 54: СУЩЕСТВУЕТ СЛОЖНОСТЬ

Examples

СУЩЕСТВУЕТ СЛОЖНОСТЬ

Таблица клиентов

Я бы	Имя	Фамилия
1	Ozgur	Ozturk
2	Юсеф	Medi
3	Генри	Tai

Таблица заказов

Я бы	Пользовательский ИД	Количество
1	2	123,50
2	3	14,80

Получите всех клиентов с минимальным заказом

```
SELECT * FROM Customer WHERE EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId=Customer.Id
)
```

Результат

Я бы	Имя	Фамилия
2	Юсеф	Medi
3	Генри	Tai

Получить всех клиентов без заказа

```
SELECT * FROM Customer WHERE NOT EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId = Customer.Id
)
```

Результат

Я бы	Имя	Фамилия
1	Ozgur	Ozturk

Цель

`EXISTS` , `IN` и `JOIN` могут когда-то использоваться для одного и того же результата, однако они не равны:

- `EXISTS` следует использовать, чтобы проверить, существует ли значение в другой таблице
- `IN` должен использоваться для статического списка
- `JOIN` следует использовать для извлечения данных из других таблиц (ов)

Прочитайте СУЩЕСТВУЕТ СЛОЖНОСТЬ онлайн: https://riptutorial.com/ru/sql/topic/7933/существует-сложность

глава 55: Таблица DROP

замечания

DROP TABLE удаляет определение таблицы из схемы вместе с строками, индексами, разрешениями и триггерами.

Examples

Простое падение

```
Drop Table MyTable;
```

Проверьте наличие перед тем, как сбросить

MySQL 3.19

```
DROP TABLE IF EXISTS MyTable;
```

PostgreSQL 8.x

```
DROP TABLE IF EXISTS MyTable;
```

SQL Server 2005

```
If Exists(Select * From Information_Schema.Tables
          Where Table_Schema = 'dbo'
            And Table_Name = 'MyTable')
  Drop Table dbo.MyTable
```

SQLite 3.0

```
DROP TABLE IF EXISTS MyTable;
```

Прочитайте Таблица DROP онлайн: https://riptutorial.com/ru/sql/topic/1832/таблица-drop

глава 56: Типы данных

Examples

DECIMAL и NUMERIC

Фиксированная точность и масштабирование десятичных чисел. `DECIMAL` и `NUMERIC` функционально эквивалентны.

Синтаксис:

```
DECIMAL ( precision [ , scale] )
NUMERIC ( precision [ , scale] )
```

Примеры:

```
SELECT CAST(123 AS DECIMAL(5,2)) --returns 123.00
SELECT CAST(12345.12 AS NUMERIC(10,5)) --returns 12345.12000
```

FLOAT и REAL

Типы данных приблизительного числа для использования с числовыми данными с плавающей запятой.

```
SELECT CAST( PI() AS FLOAT) --returns 3.14159265358979
SELECT CAST( PI() AS REAL) --returns 3.141593
```

Целые

Типы данных точного числа, которые используют целочисленные данные.

Тип данных	Спектр	Место хранения
BIGINT	-2 ^ 63 (-9,223,372,036,854,775,808) до 2 ^ 63-1 (9,223,372,036,854,775,807)	8 байт
ИНТ	-2 ^ 31 (-2,147,483,648) до 2 ^ 31-1 (2,147,483,647)	4 байта
SMALLINT	-2 ^ 15 (-32,768) до 2 ^ 15-1 (32,767)	2 байта
TINYINT	От 0 до 255	1 байт

ДЕНЬГИ И МАЛОМОННИ

Типы данных, которые представляют собой денежные или валютные значения.

Тип данных	Спектр	Место хранения
Деньги	-922,337,203,685,477,5808 до 922,337,203,685,477.5807	8 байт
smallmoney	-214,748.3648 до 214,748.3647	4 байта

БИНАРЫ И ВАРИАНТЫ

Бинарные типы данных как фиксированной длины, так и переменной длины.

Синтаксис:

```
BINARY [ ( n_bytes ) ]
VARBINARY [ ( n_bytes | max ) ]
```

`n_bytes` может быть любым числом от 1 до 8000 байт. `max` указывает, что максимальное пространство для хранения составляет 2 ^ 31-1.

Примеры:

```
SELECT CAST(12345 AS BINARY(10)) -- 0x0000000000000003039
SELECT CAST(12345 AS VARBINARY(10)) -- 0x00003039
```

CHAR и VARCHAR

Строковые типы данных как фиксированной длины, так и переменной длины.

Синтаксис:

```
CHAR [ ( n_chars ) ]
VARCHAR [ ( n_chars ) ]
```

Примеры:

```
SELECT CAST('ABC' AS CHAR(10)) -- 'ABC       ' (padded with spaces on the right)
SELECT CAST('ABC' AS VARCHAR(10)) -- 'ABC' (no padding due to variable character)
SELECT CAST('ABCDEFGHIJKLMNOPQRSTUVWXYZ' AS CHAR(10))  -- 'ABCDEFGHIJ' (truncated to 10
characters)
```

NCHAR и NVARCHAR

Строковые типы данных UNICODE фиксированной длины или переменной длины.

Синтаксис:

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

Используйте `MAX` для очень длинных строк, которые могут превышать 8000 символов.

УНИКАЛЬНЫЙ ИДЕНТИФИКАТОР

16-байтовый GUID / UUID.

```
DECLARE @GUID UNIQUEIDENTIFIER = NEWID();
SELECT @GUID -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
DECLARE @bad_GUID_string VARCHAR(100) = 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
SELECT
    @bad_GUID_string,   -- 'E28B3BD9-9174-41A9-8508-899A78A33540_foobarbaz'
    CONVERT(UNIQUEIDENTIFIER, @bad_GUID_string) -- 'E28B3BD9-9174-41A9-8508-899A78A33540'
```

Прочитайте Типы данных онлайн: https://riptutorial.com/ru/sql/topic/1166/типы-данных

глава 57: Триггеры

Examples

СОЗДАТЬ ТРИГГЕР

В этом примере создается триггер, который вставляет запись во вторую таблицу (MyAudit) после того, как запись вставлена в таблицу, на которой установлен триггер (MyTable). Здесь «вставленная» таблица является специальной таблицей, используемой Microsoft SQL Server для хранения затронутых строк во время инструкций INSERT и UPDATE; существует также специальная «удаленная» таблица, которая выполняет ту же функцию для операторов DELETE.

```
CREATE TRIGGER MyTrigger
    ON MyTable
    AFTER INSERT

AS

BEGIN
    -- insert audit record to MyAudit table
    INSERT INTO MyAudit(MyTableId, User)
    (SELECT MyTableId, CURRENT_USER FROM inserted)
END
```

Используйте Trigger для управления «корзиной» для удаленных элементов

```
CREATE TRIGGER BooksDeleteTrigger
    ON MyBooksDB.Books
    AFTER DELETE
AS
  INSERT INTO BooksRecycleBin
    SELECT *
    FROM deleted;
GO
```

Прочитайте Триггеры онлайн: https://riptutorial.com/ru/sql/topic/1432/триггеры

глава 58: УДАЛЯТЬ

Вступление

Оператор DELETE используется для удаления записей из таблицы.

Синтаксис

1. DELETE FROM *TableName* [WHERE *Condition*] [LIMIT *count*]

Examples

УДАЛИТЬ некоторые строки с ГДЕ

Это приведет к удалению всех строк, соответствующих критериям `WHERE` .

```
DELETE FROM Employees
WHERE FName = 'John'
```

УДАЛИТЬ все строки

Опускание предложения `WHERE` приведет к удалению всех строк из таблицы.

```
DELETE FROM Employees
```

См. Документацию TRUNCATE для получения подробной информации о том, как производительность TRUNCATE может быть лучше, поскольку она игнорирует триггеры, индексы и журналы, чтобы просто удалять данные.

Предложение TRUNCATE

Используйте это, чтобы сбросить таблицу до состояния, в котором она была создана. Это удаляет все строки и сбрасывает такие значения, как автоинкремент. Он также не регистрирует удаление каждой отдельной строки.

```
TRUNCATE TABLE Employees
```

УДАЛИТЬ некоторые строки на основе сравнений с другими таблицами

Можно `DELETE` данные из таблицы, если они соответствуют (или не соответствуют) определенным данным в других таблицах.

Предположим, мы хотим `DELETE` данные из Источника после его загрузки в Целевой.

```
DELETE FROM Source
WHERE  EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

Наиболее распространенные реализации RDBMS (например, MySQL, Oracle, PostgresSQL, Teradata) позволяют объединять таблицы во время `DELETE` что позволяет более сложное сравнение в компактном синтаксисе.

Добавив сложность к исходному сценарию, предположим, что Aggregate построен из Target один раз в день и не содержит одного и того же идентификатора, но содержит ту же дату. Предположим также, что мы хотим удалить данные из Источника *только* после того, как совокупность будет заполнена в течение дня.

В MySQL, Oracle и Teradata это можно сделать, используя:

```
DELETE FROM Source
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.Date = AggregateSchema.Aggregate.Date
```

В PostgreSQL используйте:

```
DELETE FROM Source
USING  TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Это по существу приводит к ВНУТРЕННЫМ СОЕДИНЕНИЯМ между Source, Target и Aggregate. Удаление выполняется для Источника, когда те же идентификаторы существуют в Целевом И дата, присутствующая в Целевом для этих идентификаторов, также существуют в Агрегате.

Тот же запрос также может быть написан (в MySQL, Oracle, Teradata) следующим образом:

```
DELETE Source
FROM   Source, TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Явные объединения могут упоминаться в `Delete` в некоторых реализациях СУБД (например, Oracle, MySQL), но не поддерживаются на всех платформах (например, Teradata не поддерживает их)

Сравнение может быть спроектировано для проверки сценариев несоответствия, а не для сопоставления со всеми стилями синтаксиса (см. Ниже `NOT EXISTS`)

```
DELETE FROM Source
WHERE NOT EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
                FROM Target
                Where Source.ID = Target.ID )
```

Прочитайте УДАЛЯТЬ онлайн: https://riptutorial.com/ru/sql/topic/1105/удалять

глава 59: Фильтровать результаты, используя WHERE и HAVING

Синтаксис

- SELECT column_name
  FROM table_name
  WHERE значение оператора column_name
- SELECT column_name, aggregate_function (column_name)
  FROM table_name
  GROUP BY column_name
  ИМЕЕТ значение оператора aggregate_function (column_name)

Examples

Предложение WHERE возвращает только строки, соответствующие его критериям

Steam имеет игры под секцией $ 10 на своей странице магазина. Где-то глубоко в центре их систем, вероятно, есть запрос, который выглядит примерно так:

```
SELECT *
FROM Items
WHERE Price < 10
```

Используйте IN для возврата строк со значением, содержащимся в списке

В этом примере используется таблица автомобилей из примерных баз данных.

```
SELECT *
FROM Cars
WHERE TotalCost IN (100, 200, 300)
```

Этот запрос вернет Car # 2, который стоит 200 и Car # 3, который стоит 100. Обратите внимание, что это эквивалентно использованию нескольких предложений с `OR` , например:

```
SELECT *
FROM Cars
WHERE TotalCost = 100 OR TotalCost = 200 OR TotalCost = 300
```

Используйте LIKE, чтобы найти соответствующие строки и подстроки

См. Полную документацию по оператору LIKE .

В этом примере таблица Employees используется в примерах баз данных.

```
SELECT *
FROM Employees
WHERE FName LIKE 'John'
```

Этот запрос возвращает только Employee # 1, чье имя точно совпадает с «John».

```
SELECT *
FROM Employees
WHERE FName like 'John%'
```

Добавление `%` позволяет вам искать подстроку:

- `John%` - вернет любого Сотрудника, чье имя начинается с «Джона», за которым следует любое количество символов
- `%John` - вернет любого Сотрудника, чье имя заканчивается на «Джон», которое исходит из любого количества символов
- `%John%` - вернет любого Сотрудника, чье имя содержит «Джон» в любом месте значения

В этом случае запрос вернет Employee # 2, чье имя «John», а также Employee # 4, чье имя « Johnathon».

Предложение WHERE с NULL / NOT NULL значениями

```
SELECT *
FROM Employees
WHERE ManagerId IS NULL
```

Этот оператор возвращает все записи Employee, где значение столбца `ManagerId` равно `NULL` .

Результатом будет:

```
Id    FName    LName    PhoneNumber    ManagerId    DepartmentId
1     James    Smith    1234567890     NULL         1
```

```
SELECT *
FROM Employees
WHERE ManagerId IS NOT NULL
```

Этот оператор вернет все записи Employee, где значение `ManagerId` *не* `NULL` .

Результатом будет:

```
Id    FName       LName     PhoneNumber    ManagerId    DepartmentId
2     John        Johnson   2468101214     1            1
3     Michael     Williams  1357911131     1            2
4     Johnathon   Smith     1212121212     2            1
```

Примечание. Тот же запрос не будет возвращать результаты, если вы измените предложение WHERE на `WHERE ManagerId = NULL` или `WHERE ManagerId <> NULL` .

Использование HAVING с агрегатными функциями

В отличие от `WHERE` , `HAVING` может использоваться с совокупными функциями.

> Агрегатная функция - это функция, в которой значения нескольких строк группируются вместе как входные данные по определенным критериям, чтобы сформировать одно значение более значимого значения или измерения (Википедия).

Общие агрегированные функции включают `COUNT()` , `SUM()` , `MIN()` и `MAX()` .

В этом примере используется таблица автомобилей из примерных баз данных.

```
SELECT CustomerId, COUNT(Id) AS [Number of Cars]
FROM Cars
GROUP BY CustomerId
HAVING COUNT(Id) > 1
```

Этот запрос вернет счетчик `CustomerId` и `Number of Cars` любого клиента, у которого есть более одного автомобиля. В этом случае единственным клиентом, у которого есть более одного автомобиля, является Клиент № 1.

Результаты будут выглядеть так:

Пользовательский ИД	Количество автомобилей
1	2

Используйте BETWEEN для фильтрации результатов

В следующих примерах используются типовые базы данных Sales and Customers .

> Примечание: МЕЖДУ оператора включительно.

Использование оператора BETWEEN с номерами:

```
SELECT * From ItemSales
WHERE Quantity BETWEEN 10 AND 17
```

В этом запросе будут возвращены все записи `ItemSales` , количество которых больше или равно 10 и меньше или равно 17. Результаты будут выглядеть так:

Я бы	Дата продажи	ItemId	Количество	Цена
1	2013-07-01	100	10	34,5
4	2013-07-23	100	15	34,5
5	2013-07-24	145	10	34,5

Использование оператора BETWEEN с значениями даты:

```
SELECT * From ItemSales
WHERE SaleDate BETWEEN '2013-07-11' AND '2013-05-24'
```

Этот запрос вернет все записи `ItemSales` с помощью `SaleDate` который больше или равен 11 июля 2013 года и меньше или равен 24 мая 2013 года.

Я бы	Дата продажи	ItemId	Количество	Цена
3	2013-07-11	100	20	34,5
4	2013-07-23	100	15	34,5
5	2013-07-24	145	10	34,5

> При сравнении значений даты и времени вместо дат вам может потребоваться преобразовать значения даты и времени в значения даты или добавить или вычесть 24 часа, чтобы получить правильные результаты.

Использование оператора BETWEEN с текстовыми значениями:

```
SELECT Id, FName, LName FROM Customers
WHERE LName BETWEEN 'D' AND 'L';
```

Пример Live: скрипт SQL

Этот запрос возвращает всех клиентов, имя которых в алфавитном порядке находится между буквами «D» и «L». В этом случае возвращаются Клиент №1 и №3. Клиент № 2, имя которого начинается с «M», не будет включено.

Я бы	FName	LName
1	Уильям	Джонс

Я бы	FName	LName
3	Ричард	Дэвис

равенство

```
SELECT * FROM Employees
```

Этот оператор вернет все строки из таблицы `Employees` .

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Использование `WHERE` в конце вашего `SELECT` позволяет ограничить возвращаемые строки условием. В этом случае, когда существует точное совпадение с помощью знака `=` :

```
SELECT * FROM Employees WHERE DepartmentId = 1
```

Вернет только строки, в которых значение параметра `DepartmentId` равно `1` :

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

И И ИЛИ

Вы также можете объединить несколько операторов для создания более сложных условий `WHERE` . В следующих примерах используется таблица `Employees` :

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
1    James     Smith    1234567890    NULL        1               1000    01-01-2002    01-01-
2002    01-01-2002
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
```

```
2016    01-01-2002
```

А ТАКЖЕ

```
SELECT * FROM Employees WHERE DepartmentId = 1 AND ManagerId = 1
```

Вернусь:

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
2    John      Johnson  2468101214    1           1               400     23-03-2005    23-03-
2005    01-01-2002
```

ИЛИ ЖЕ

```
SELECT * FROM Employees WHERE DepartmentId = 2 OR ManagerId = 2
```

Вернусь:

```
Id   FName     LName    PhoneNumber   ManagerId   DepartmentId    Salary  Hire_date
CreatedDate   ModifiedDate
3    Michael   Williams 1357911131    1           2               600     12-05-2009    12-05-
2009    NULL
4    Johnathon Smith    1212121212    2           1               500     24-07-2016    24-07-
2016    01-01-2002
```

Используйте HAVING для проверки нескольких условий в группе

Таблица заказов

Пользовательский ИД	Код товара	Количество	Цена
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

Чтобы проверить клиентов, которые заказали оба продукта - ProductID 2 и 3, можно использовать HAVING

```
select customerId
from orders
```

```
where productID in (2,3)
group by customerId
having count(distinct productID) = 2
```

Возвращаемое значение:

Пользовательский ИД
1

Запрос выбирает только записи с идентификаторами productID в вопросах и с проверкой предложения HAVING для групп, имеющих 2 productIds, а не только один.

Другая возможность

```
select customerId
from orders
group by customerId
having sum(case when productID = 2 then 1 else 0 end) > 0
   and sum(case when productID = 3 then 1 else 0 end) > 0
```

В этом запросе выбираются только группы, имеющие хотя бы одну запись с идентификатором productID 2 и по меньшей мере с идентификатором productID 3.

Где EXISTS

Выберет записи в `TableName` , имеющие записи, соответствующие в `TableName1` .

```
SELECT * FROM TableName t WHERE EXISTS (
    SELECT 1 FROM TableName1 t1 where t.Id = t1.Id)
```

Прочитайте Фильтровать результаты, используя WHERE и HAVING онлайн:
https://riptutorial.com/ru/sql/topic/636/фильтровать-результаты--используя-where-и-having

глава 60: Функции (Scalar / Single Row)

Вступление

SQL предоставляет несколько встроенных скалярных функций. Каждая скалярная функция принимает одно значение в качестве входных данных и возвращает одно значение в качестве вывода для каждой строки в результирующем наборе.

Вы используете скалярные функции везде, где выражение допускается в инструкции T-SQL.

Синтаксис

- CAST (выражение AS data_type [(length)])
- CONVERT (data_type [(длина)], выражение [, style])
- PARSE (string_value AS data_type [ИСПОЛЬЗОВАНИЕ культуры])
- DATENAME (datepart, date)
- GETDATE ()
- DATEDIFF (datepart, startdate, enddate)
- DATEADD (datepart, number, date)
- ВЫБЕРИТЕ (индекс, val_1, val_2 [, val_n])
- IIF (boolean_expression, true_value, false_value)
- SIGN (числовое выражение)
- POWER (float_expression, y)

замечания

Скалярные или однорядные функции используются для управления каждой строкой данных в результирующем наборе, в отличие от агрегатных функций, которые работают со всем набором результатов.

Существует десять типов скалярных функций.

1. Функции конфигурации предоставляют информацию о конфигурации текущего экземпляра SQL.
2. Функции преобразования преобразуют данные в правильный тип данных для данной операции. Например, эти типы функций могут переформатировать информацию путем преобразования строки в дату или число, чтобы можно было сравнить два разных типа.
3. Функции даты и времени управляют полями, содержащими значения даты и времени. Они могут возвращать числовые, даты или строковые значения. Например, вы можете использовать функцию для извлечения текущего дня недели или года или для

получения только года с даты.

Значения, возвращаемые функциями даты и времени, зависят от даты и времени, установленных для операционной системы компьютера, на котором запущен экземпляр SQL.

4. Логическая функция, выполняющая операции с использованием логических операторов. Он оценивает набор условий и возвращает единственный результат.
5. Математические функции выполняют математические операции или вычисления для числовых выражений. Этот тип функции возвращает одно числовое значение.
6. Функции метаданных извлекают информацию о указанной базе данных, такую как ее имя и объекты базы данных.
7. Функции безопасности предоставляют информацию, которую вы можете использовать для управления безопасностью базы данных, например информацию о пользователях и ролях пользователей.
8. Строковые функции выполняют операции с строковыми значениями и возвращают либо числовые, либо строковые значения.

Используя строковые функции, вы можете, например, комбинировать данные, извлекать подстроку, сравнивать строки или преобразовывать строку ко всем строчным или строчным символам.

9. Системные функции выполняют операции и возвращают информацию о значениях, объектах и настройках для текущего экземпляра SQL
10. Статистические функции системы предоставляют различные статистические данные о текущем экземпляре SQL - например, чтобы вы могли отслеживать текущие уровни производительности системы.

Examples

Изменение персонажей

Функции модификации символов включают преобразование символов в символы верхнего или нижнего регистра, преобразование чисел в форматированные числа, выполнение манипуляций с символами и т. Д.

Функция `lower(char)` преобразует заданный параметр символа в нижние символы.

```
SELECT customer_id, lower(customer_last_name) FROM customer;
```

вернет фамилию клиента, измененную с «SMITH» на «smith».

Дата и время

В SQL вы используете типы данных даты и времени для хранения информации календаря.

Эти типы данных включают время, дату, smalldatetime, datetime, datetime2 и datetimeoffset. Каждый тип данных имеет определенный формат.

Тип данных	Формат
время	чч: мм: сс [.nnnnnnn]
Дата	YYYY-MM-DD
smalldatetime	ГГГГ-ММ-ДД чч: мм: сс
Дата и время	ГГГГ-ММ-ДД hh: mm: ss [.nnn]
datetime2	ГГГГ-ММ-ДД hh: mm: ss [.nnnnnnn]
DateTimeOffset	ГГГГ-ММ-ДД hh: mm: ss [.nnnnnnn] [+/-] hh: mm

Функция `DATENAME` возвращает имя или значение определенной части даты.

```
SELECT DATENAME (weekday,'2017-01-14') as Datename
```

Datename
суббота

Вы используете функцию `GETDATE` для определения текущей даты и времени компьютера, на котором запущен текущий экземпляр SQL. Эта функция не включает разницу в часовых поясах.

```
SELECT GETDATE() as Systemdate
```

Systemdate
2017-01-14 11: 11: 47.7230728

Функция `DATEDIFF` возвращает разницу между двумя датами.

В синтаксисе datepart является параметром, который указывает, какую часть даты вы хотите использовать для вычисления разницы. Datepart может быть год, месяц, неделя, день, час, минута, секунда или миллисекунда. Затем вы указываете дату начала в параметре startdate и дату окончания в параметре enddate, для которого вы хотите найти разницу.

```
SELECT SalesOrderID, DATEDIFF(day, OrderDate, ShipDate)
AS 'Processing time'
FROM Sales.SalesOrderHeader
```

SalesOrderID	Время обработки
43659	7
43660	7
43661	7
43662	7

Функция `DATEADD` позволяет добавить интервал к определенной дате.

```
SELECT DATEADD (day, 20, '2017-01-14') AS Added20MoreDays
```

Added20MoreDays
2017-02-03 00: 00: 00.000

Конфигурация и функция преобразования

Примером функции конфигурации в SQL является функция `@@SERVERNAME` . Эта функция предоставляет имя локального сервера, на котором запущен SQL.

```
SELECT @@SERVERNAME AS 'Server'
```

сервер
SQL064

В SQL большинство преобразований данных происходит неявно, без вмешательства пользователя.

Чтобы выполнить любые преобразования, которые не могут быть выполнены неявно, вы можете использовать функции `CAST` или `CONVERT` .

`CAST` синтаксис функции проще , чем `CONVERT` синтаксис функции, но ограничен в том, что он может сделать.

Здесь мы используем функции `CAST` и `CONVERT` для преобразования типа данных datetime в тип данных `varchar` .

Функция `CAST` всегда использует настройку стиля по умолчанию. Например, он будет представлять даты и время с использованием формата YYYY-MM-DD.

Функция `CONVERT` использует указанный вами стиль даты и времени. В этом случае 3 задает формат даты dd / mm / yy.

```
USE AdventureWorks2012
GO
SELECT FirstName + ' ' + LastName + ' was hired on ' +
       CAST(HireDate AS varchar(20)) AS 'Cast',
       FirstName + ' ' + LastName + ' was hired on ' +
       CONVERT(varchar, HireDate, 3) AS 'Convert'
FROM Person.Person AS p
JOIN HumanResources.Employee AS e
ON p.BusinessEntityID = e.BusinessEntityID
GO
```

В ролях	Перерабатывать
Дэвид Хэмилтон был нанят в 2003-02-04	Дэвид Хэмилтон был нанят 04/02/03

Другим примером функции преобразования является функция `PARSE` . Эта функция преобразует строку в указанный тип данных.

В синтаксисе для функции вы указываете строку, которая должна быть преобразована, ключевое слово `AS` , а затем требуемый тип данных. При желании вы также можете указать культуру, в которой строковое значение должно быть отформатировано. Если вы не указали это, используется язык для сеанса.

Если строковое значение не может быть преобразовано в числовой, дату или формат времени, это приведет к ошибке. Затем вам понадобится использовать `CAST` или `CONVERT` для преобразования.

```
SELECT PARSE('Monday, 13 August 2012' AS datetime2 USING 'en-US') AS 'Date in English'
```

Дата на английском языке
2012-08-13 00: 00: 00.0000000

Логическая и математическая функция

SQL имеет две логические функции: `CHOOSE` и `IIF` .

Функция `CHOOSE` возвращает элемент из списка значений на основе его позиции в списке. Эта позиция указана индексом.

В синтаксисе параметр index указывает элемент и представляет собой целое число или целое число. Параметр val_1 ... val_n идентифицирует список значений.

```
SELECT CHOOSE(2, 'Human Resources', 'Sales', 'Admin', 'Marketing' ) AS Result;
```

Результат
Продажи

В этом примере вы используете функцию `CHOOSE` для возврата второй записи в список отделов.

Функция `IIF` возвращает одно из двух значений, основанное на определенном условии. Если условие истинно, оно вернет истинное значение. В противном случае он вернет ложное значение.

В синтаксисе параметр boolean_expression указывает логическое выражение. Параметр true_value указывает значение, которое должно быть возвращено, если выражение boolean_expression равно true, а параметр false_value указывает значение, которое должно быть возвращено, если выражение boolean_expression равно false.

```
SELECT BusinessEntityID, SalesYTD,
       IIF(SalesYTD > 200000, 'Bonus', 'No Bonus') AS 'Bonus?'
FROM Sales.SalesPerson
GO
```

BusinessEntityID	SalesYTD	Бонус?
274	559697.5639	бонус
275	3763178.1787	бонус
285	172524.4512	Нет бонусов

В этом примере вы используете функцию IIF для возврата одного из двух значений. Если продажи года продажи продавцом превышают 200 000, это лицо будет иметь право на получение бонуса. Значения ниже 200 000 означают, что сотрудники не имеют права на получение бонусов.

SQL включает в себя несколько математических функций, которые можно использовать для выполнения вычислений на входных значениях и возврата числовых

результатов.

Одним из примеров является функция `SIGN` , которая возвращает значение, указывающее знак выражения. Значение -1 указывает отрицательное выражение, значение +1 указывает на положительное выражение, а 0 указывает на нуль.

```
SELECT SIGN(-20) AS 'Sign'
```

Знак
-1

В этом примере вход представляет собой отрицательное число, поэтому на панели «Результаты» отображается результат -1.

Еще одна математическая функция - функция `POWER` . Эта функция обеспечивает значение выражения, поднятого до указанной мощности.

В синтаксисе параметр float_expression указывает выражение, а параметр y указывает мощность, к которой вы хотите поднять выражение.

```
SELECT POWER(50, 3) AS Result
```

Результат
125000

Прочитайте Функции (Scalar / Single Row) онлайн: https://riptutorial.com/ru/sql/topic/6898/функции--scalar---single-row-

глава 61: Функции (Агрегат)

Синтаксис

- Функция (выражение [*DISTINCT*]) -DISTINCT является необязательным параметром
- AVG (выражение [ALL | DISTINCT])
- COUNT ({[ALL | DISTINCT] выражение] | *})
- GROUPING (<column_expression>)
- MAX (выражение [ALL | DISTINCT])
- MIN (выражение [ALL | DISTINCT])
- SUM (выражение [ALL | DISTINCT])
- VAR (выражение [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)
- VARP (выражение [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause
- STDEV (выражение [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)
- STDEVP (выражение [ALL | DISTINCT])
  OVER ([partition_by_clause] order_by_clause)

замечания

В управлении базой данных агрегированная функция представляет собой функцию, в которой значения нескольких строк группируются вместе как входные данные по определенным критериям для формирования единственного значения более значимого значения или измерения, такого как набор, сумка или список.

```
MIN         returns the smallest value in a given column
MAX         returns the largest value in a given column
SUM         returns the sum of the numeric values in a given column
AVG         returns the average value of a given column
COUNT       returns the total number of values in a given column
COUNT(*)    returns the number of rows in a table
GROUPING    Is a column or an expression that contains a column in a GROUP BY clause.
STDEV       returns the statistical standard deviation of all values in the specified
expression.
STDEVP      returns the statistical standard deviation for the population for all values in the
specified expression.
VAR         returns the statistical variance of all values in the specified expression. may be
followed by the OVER clause.
VARP        returns the statistical variance for the population for all values in the specified
expression.
```

> Агрегатные функции используются для вычисления «возвращаемого столбца числовых данных» из `SELECT` . Они в основном суммируют результаты конкретного столбца выбранных данных. - SQLCourse2.com

Все агрегатные функции игнорируют значения NULL.

Examples

SUM

`Sum` функции просуммировать значения всех строк в группе. Если предложение group by опущено, то суммирует все строки.

```
select sum(salary) TotalSalary
from employees;
```

TotalSalary
2500

```
select DepartmentId, sum(salary) TotalSalary
from employees
group by DepartmentId;
```

DepartmentID	TotalSalary
1	2000
2	500

Условная агрегация

Таблица платежей

Покупатель	Способ оплаты	Количество
Питер	кредит	100
Питер	кредит	300
Джон	кредит	1000
Джон	Дебет	500

```
select customer,
       sum(case when payment_type = 'credit' then amount else 0 end) as credit,
       sum(case when payment_type = 'debit' then amount else 0 end) as debit
from payments
group by customer
```

Результат:

Покупатель	кредит	Дебет
Питер	400	0
Джон	1000	500

```
select customer,
       sum(case when payment_type = 'credit' then 1 else 0 end) as credit_transaction_count,
       sum(case when payment_type = 'debit' then 1 else 0 end) as debit_transaction_count
from payments
group by customer
```

Результат:

Покупатель	credit_transaction_count	debit_transaction_count
Питер	2	0
Джон	1	1

AVG ()

Агрегатная функция AVG () возвращает среднее значение данного выражения, обычно числовые значения в столбце. Предположим, у нас есть таблица, содержащая ежегодный расчет населения в городах по всему миру. Записи для Нью-Йорка выглядят примерно так:

ПРИМЕР ТАБЛИЦЫ

название города	Население	год
Нью-Йорк	8550405	2015
Нью-Йорк	...	...
Нью-Йорк	8000906	2005

Чтобы выбрать среднее население Нью-Йорка, США из таблицы, содержащей названия городов, измерения численности населения и годы измерения за последние десять лет:

QUERY

```
select city_name, AVG(population) avg_population
from city_population
where city_name = 'NEW YORK CITY';
```

Обратите внимание, что год измерения отсутствует в запросе, поскольку население

усредняется с течением времени.

РЕЗУЛЬТАТЫ

название города	avg_population
Нью-Йорк	8250754

> Примечание. Функция AVG () преобразует значения в числовые типы. Это особенно важно иметь в виду при работе с датами.

Конкатенация списка

Частичный кредит на этот ответ.

Список Concatenation объединяет столбец или выражение, объединяя значения в одну строку для каждой группы. Можно указать строку для разграничения каждого значения (либо пустую, либо запятую, когда она опущена), и порядок значений в результате. Хотя он не является частью стандарта SQL, каждый крупный поставщик реляционных баз данных поддерживает его по-своему.

MySQL

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB ORDER BY ColumnB SEPARATOR ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

Oracle и DB2

```
SELECT ColumnA
     , LISTAGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

PostgreSQL

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',' ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
```

```
 ORDER BY ColumnA;
```

SQL Server

SQL Server 2016 и ранее

(CTE включен для поощрения принципа DRY)

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName)
SELECT t0.ColumnA
     , STUFF((
       SELECT ',' + t1.ColumnB
         FROM CTE_TableName t1
        WHERE t1.ColumnA = t0.ColumnA
        ORDER BY t1.ColumnB
          FOR XML PATH('')), 1, 1, '') AS ColumnBs
  FROM CTE_TableName t0
 GROUP BY t0.ColumnA
 ORDER BY ColumnA;
```

SQL Server 2017 и SQL Azure

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

SQLite

без заказа:

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

для заказа требуется подзапрос или CTE:

```
  WITH CTE_TableName AS (
       SELECT ColumnA, ColumnB
         FROM TableName
        ORDER BY ColumnA, ColumnB)
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
```

```
 FROM CTE_TableName
GROUP BY ColumnA
ORDER BY ColumnA;
```

подсчитывать

Вы можете подсчитать количество строк:

```
SELECT count(*) TotalRows
FROM employees;
```

TotalRows
4

Или подсчитать сотрудников на отдел:

```
SELECT DepartmentId, count(*) NumEmployees
FROM employees
GROUP BY DepartmentId;
```

DepartmentID	NumEmployees
1	3
2	1

Вы можете рассчитывать на столбец / выражение с эффектом, который не будет считать значения `NULL` :

```
SELECT count(ManagerId) mgr
FROM EMPLOYEES;
```

прил
3

(Существует один столбец Manager с нулевым значением)

Вы также можете использовать **DISTINCT** внутри другой функции, такой как **COUNT,** чтобы найти только члены **DISTINCT** набора для выполнения операции.

Например:

```
SELECT COUNT(ContinentCode) AllCount
,      COUNT(DISTINCT ContinentCode) SingleCount
FROM Countries;
```

Вернет разные значения. *SingleCount* будет *рассчитывать* только отдельные континенты один раз, а *AllCount* будет включать в себя дубликаты.

ContinentCode
OC
Евросоюз
KAK
Не Доступно
Не Доступно
AF
AF

AllCount: 7 SingleCount: 5

Максимум

Найти максимальное значение столбца:

```
select max(age) from employee;
```

Приведенный выше пример будет возвращать наибольшее значение для столбца `age` из `employee` таблицы.

Синтаксис:

```
SELECT MAX(column_name) FROM table_name;
```

Min

Найдите наименьшее значение столбца:

```
 select min(age) from employee;
```

Приведенный выше пример будет возвращать наименьшее значение для столбца `age` из `employee` таблицы.

Синтаксис:

```
 SELECT MIN(column_name) FROM table_name;
```

Прочитайте Функции (Агрегат) онлайн: https://riptutorial.com/ru/sql/topic/1002/функции--агрегат-

глава 62: Функции (аналитические)

Вступление

Вы используете аналитические функции для определения значений, основанных на группах значений. Например, вы можете использовать этот тип функции для определения текущих итогов, процентов или верхнего результата в группе.

Синтаксис

1. FIRST_VALUE (скалярное выражение) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
2. LAST_VALUE (скалярное выражение) OVER ([partition_by_clause] order_by_clause [rows_range_clause])
3. LAG (scalar_expression [, offset] [, default]) OVER ([partition_by_clause] order_by_clause)
4. LEAD (scalar_expression [, offset], [default]) OVER ([partition_by_clause] order_by_clause)
5. PERCENT_RANK () OVER ([partition_by_clause] order_by_clause)
6. CUME_DIST () OVER ([partition_by_clause] order_by_clause)
7. PERCENTILE_DISC (numeric_literal) WITHIN GROUP (ORDER BY order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])
8. PERCENTILE_CONT (numeric_literal) WITHIN GROUP (ORDER BY order_by_expression [ASC | DESC]) OVER ([<partition_by_clause>])

Examples

FIRST_VALUE

Вы используете функцию `FIRST_VALUE` для определения первого значения в упорядоченном наборе результатов, который вы определяете с помощью скалярного выражения.

```
SELECT StateProvinceID, Name, TaxRate,
       FIRST_VALUE(StateProvinceID)
        OVER(ORDER BY TaxRate ASC) AS FirstValue
FROM SalesTaxRate;
```

В этом примере функция `FIRST_VALUE` используется для возврата `ID` штата или провинции с самой низкой ставкой налога. Предложение `OVER` используется для заказа ставок налога для получения самой низкой ставки.

StateProvinceID	название	Ставка налога	FirstValue
74	Налог с продаж штата Юта	5,00	74

StateProvinceID	название	Ставка налога	FirstValue
36	Государственный налог с продаж в Миннесоте	6,75	74
30	Государственный налог с продаж в штате Массачусетс	7,00	74
1	Канадский GST	7,00	74
57	Канадский GST	7,00	74
63	Канадский GST	7,00	74

LAST_VALUE

Функция `LAST_VALUE` предоставляет последнее значение в упорядоченном наборе результатов, который вы определяете с помощью скалярного выражения.

```
SELECT TerritoryID, StartDate, BusinessentityID,
       LAST_VALUE(BusinessentityID)
        OVER(ORDER BY TerritoryID) AS LastValue
FROM SalesTerritoryHistory;
```

В этом примере функция `LAST_VALUE` возвращает последнее значение для каждого набора строк в упорядоченных значениях.

TerritoryID	Дата начала	BusinessEntityID	LastValue
1	2005-07-01 00.00.00.000	280	283
1	2006-11-01 00.00.00.000	284	283
1	2005-07-01 00.00.00.000	283	283
2	2007-01-01 00.00.00.000	277	275
2	2005-07-01 00.00.00.000	275	275
3	2007-01-01 00.00.00.000	275	277

LAG и LEAD

Функция `LAG` предоставляет данные о строках перед текущей строкой в том же наборе результатов. Например, в `SELECT` вы можете сравнивать значения в текущей строке со значениями в предыдущей строке.

Вы используете скалярное выражение для указания значений, которые следует сравнивать. Параметр offset - это количество строк перед текущей строкой, которые будут использоваться при сравнении. Если вы не укажете количество строк, используется значение по умолчанию для одной строки.

Параметр по умолчанию указывает значение, которое должно быть возвращено, когда выражение со смещением имеет значение `NULL` . Если вы не укажете значение, возвращается значение `NULL` .

Функция `LEAD` предоставляет данные о строках после текущей строки в наборе строк. Например, в `SELECT` вы можете сравнивать значения в текущей строке со значениями в следующей строке.

Вы указываете значения, которые следует сравнивать с помощью скалярного выражения. Параметр offset - это количество строк после текущей строки, которая будет использоваться при сравнении.

Вы указываете значение, которое должно быть возвращено, когда выражение со смещением имеет значение `NULL` используя параметр по умолчанию. Если вы не укажете эти параметры, используется значение по умолчанию для одной строки и возвращается значение `NULL` .

```
SELECT BusinessEntityID, SalesYTD,
       LEAD(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lead value",
       LAG(SalesYTD, 1, 0) OVER(ORDER BY BusinessEntityID) AS "Lag value"
FROM SalesPerson;
```

В этом примере используются функции LEAD и LAG для сравнения значений продаж для каждого сотрудника на сегодняшний день с показателями сотрудников, перечисленных выше и ниже, с записями, упорядоченными на основе столбца BusinessEntityID.

BusinessEntityID	SalesYTD	Значение свинца	Значение запаса
274	559697.5639	3763178.1787	0,0000
275	3763178.1787	4251368.5497	559697.5639
276	4251368.5497	3189418.3662	3763178.1787
277	3189418.3662	1453719.4653	4251368.5497
278	1453719.4653	2315185.6110	3189418.3662
279	2315185.6110	1352577.1325	1453719.4653

PERCENT_RANK и CUME_DIST

Функция `PERCENT_RANK` вычисляет ранжирование строки относительно набора строк. Процент основан на количестве строк в группе, которые имеют меньшее значение, чем текущая строка.

Первое значение в наборе результатов всегда имеет процентный ранг нуля. Значение для наивысшего ранжированного или последнего значения в наборе всегда одно.

Функция `CUME_DIST` вычисляет относительное положение заданного значения в группе значений, определяя процент значений, меньших или равных этому значению. Это называется кумулятивным распределением.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
PERCENT_RANK() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Percent Rank",
CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours DESC)
       AS "Cumulative Distribution"
FROM Employee;
```

В этом примере вы используете предложение `ORDER` для разделения или группировки - строки, полученные `SELECT` на основе названий рабочих мест сотрудников, при этом результаты в каждой группе сортируются в зависимости от количества часов отпуска по болезни, которые использовались сотрудниками.

BusinessEntityID	Должность	SickLeaveHours	Процент Ранга	Кумулятивное распределение
267	Специалист по применению	57	0	0,25
268	Специалист по применению	56	+0,333333333333333	0,75
269	Специалист по применению	56	+0,333333333333333	0,75
272	Специалист по применению	55	1	1
262	Помощник финансового директора Cheif	48	0	1
239	Специалист по преимуществам	45	0	1

BusinessEntityID	Должность	SickLeaveHours	Процент Ранга	Кумулятивное распределение
252	Покупатель	50	0	+0,1111111111111
251	Покупатель	49	0,125	+0,3333333333333
256	Покупатель	49	0,125	+0,3333333333333
253	Покупатель	48	0,375	+0,5555555555555
254	Покупатель	48	0,375	+0,5555555555555

Функция `PERCENT_RANK` оценивает записи внутри каждой группы. Для каждой записи возвращается процент записей в той же группе, которые имеют более низкие значения.

Функция `CUME_DIST` аналогична, за исключением того, что она возвращает процент значений, меньших или равных текущему значению.

PERCENTILE_DISC и PERCENTILE_CONT

Функция `PERCENTILE_DISC` отображает значение первой записи, где кумулятивное распределение выше, чем процентиль, который вы предоставляете, используя параметр `numeric_literal` .

Значения сгруппированы по набору строк или разделов, как указано в предложении `WITHIN GROUP` .

Функция `PERCENTILE_CONT` аналогична функции `PERCENTILE_DISC` , но возвращает среднее значение суммы первой соответствующей записи и следующей записи.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet"
FROM Employee;
```

Чтобы найти точное значение из строки, которая соответствует или превышает 0,5 процентиля, вы передаете процентиль в виде числового литерала в функции `PERCENTILE_DISC` . Столбец Percentile Discreet в результирующем наборе перечисляет значение строки, в которой кумулятивное распределение выше указанного процентиля.

BusinessEntityID	Должность	SickLeaveHours	Кумулятивное распределение	Percentile Discreet
272	Специалист	55	0,25	**56**

BusinessEntityID	Должность	SickLeaveHours	Кумулятивное распределение	Percentile Discreet
	по применению			
268	Специалист по применению	56	0,75	**56**
269	Специалист по применению	56	0,75	**56**
267	Специалист по применению	57	1	**56**

Чтобы основывать вычисления на наборе значений, вы используете функцию `PERCENTILE_CONT` . Столбец «Percentile Continuous» в результатах отображает среднее значение суммы значения результата и следующего максимального значения соответствия.

```
SELECT BusinessEntityID, JobTitle, SickLeaveHours,
       CUME_DIST() OVER(PARTITION BY JobTitle ORDER BY SickLeaveHours ASC)
       AS "Cumulative Distribution",
       PERCENTILE_DISC(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Discreet",
       PERCENTILE_CONT(0.5) WITHIN GROUP(ORDER BY SickLeaveHours)
          OVER(PARTITION BY JobTitle) AS "Percentile Continuous"
FROM Employee;
```

BusinessEntityID	Должность	SickLeaveHours	Кумулятивное распределение	Percentile Discreet	Процент непрерывноо
272	Специалист по применению	55	0,25	56	**56**
268	Специалист по применению	56	0,75	56	**56**
269	Специалист по применению	56	0,75	56	**56**
267	Специалист	57	1	56	**56**

BusinessEntityID	Должность	SickLeaveHours	Кумулятивное распределение	Percentile Discreet	Процент непрерывноо
	по применению				

Прочитайте Функции (аналитические) онлайн: https://riptutorial.com/ru/sql/topic/8811/функции--аналитические-

глава 63: Функции окна

Examples

Добавление всех строк, выбранных для каждой строки

```
SELECT your_columns, COUNT(*) OVER() as Ttl_Rows FROM your_data_set
```

Я бы	название	Ttl_Rows
1	пример	5
2	Foo	5
3	бар	5
4	Baz	5
5	quux	5

Вместо того, чтобы использовать два запроса, чтобы получить счет, а затем линию, вы можете использовать агрегат как функцию окна и использовать полный набор результатов в качестве окна.
Это можно использовать в качестве основы для дальнейших вычислений без сложностей дополнительных самостоятельных подключений.

Настройка флага, если другие строки имеют общее свойство

Предположим, у меня есть эти данные:

Элементы таблицы

Я бы	название	тег
1	пример	unique_tag
2	Foo	просто
42	бар	просто
3	Baz	Привет
51	quux	Мир

Я хотел бы получить все эти строки и узнать, используется ли тег другими строками

```
SELECT id, name, tag, COUNT(*) OVER (PARTITION BY tag) > 1 AS flag FROM items
```

Результатом будет:

Я бы	название	тег	флаг
1	пример	unique_tag	ложный
2	Foo	просто	правда
42	бар	просто	правда
3	Baz	Привет	ложный
51	quux	Мир	ложный

Если ваша база данных не имеет OVER и PARTITION, вы можете использовать ее для получения того же результата:

```
SELECT id, name, tag, (SELECT COUNT(tag) FROM items B WHERE tag = A.tag) > 1 AS flag FROM
items A
```

Получение общей суммы

Учитывая эти данные:

Дата	количество
2016-03-12	200
2016-03-11	-50
2016-03-14	100
2016-03-15	100
2016-03-10	-250

```
SELECT date, amount, SUM(amount) OVER (ORDER BY date ASC) AS running
FROM operations
ORDER BY date ASC
```

дам тебе

Дата	количество	Бег
2016-03-10	-250	-250
2016-03-11	-50	-300
2016-03-12	200	-100
2016-03-14	100	0
2016-03-15	100	-100

Получение N последних строк по нескольким группировкам

Учитывая эти данные

Идентификатор пользователя	COMPLETION_DATE
1	2016-07-20
1	2016-07-21
2	2016-07-20
2	2016-07-21
2	2016-07-22

```
;with CTE as
(SELECT *,
        ROW_NUMBER() OVER (PARTITION BY User_ID
                           ORDER BY Completion_Date DESC) Row_Num
FROM    Data)
SELECT * FORM CTE WHERE Row_Num <= n
```

Используя n = 1, вы получите самую последнюю строку на `user_id` :

Идентификатор пользователя	COMPLETION_DATE	ROW_NUM
1	2016-07-21	1
2	2016-07-22	1

Поиск записей «вне очереди» с использованием функции LAG ()

Учитывая данные выборки:

Я БЫ	СТАТУС	STATUS_TIME	STATUS_BY
1	ОДИН	2016-09-28-19.47.52.501398	user_1
3	ОДИН	2016-09-28-19.47.52.501511	user_2
1	ТРИ	2016-09-28-19.47.52.501517	USER_3
3	ДВА	2016-09-28-19.47.52.501521	user_2
3	ТРИ	2016-09-28-19.47.52.501524	USER_4

Элементы, идентифицированные значениями `ID` должны перемещаться из `STATUS` 'ONE' в ' TWO' в 'THREE' последовательно, без пропуска статуса. Проблема заключается в том, чтобы найти пользователей (`STATUS_BY`), которые нарушают правило и переходят с «ONE» на «THREE».

Аналитическая функция `LAG()` помогает решить проблему, возвращая для каждой строки значение в предыдущей строке:

```
SELECT * FROM (
 SELECT
  t.*,
  LAG(status) OVER (PARTITION BY id ORDER BY status_time) AS prev_status
  FROM test t
) t1 WHERE status = 'THREE' AND prev_status != 'TWO'
```

Если ваша база данных не имеет LAG (), вы можете использовать ее для получения того же результата:

```
SELECT A.id, A.status, B.status as prev_status, A.status_time, B.status_time as
prev_status_time
FROM Data A, Data B
WHERE A.id = B.id
AND   B.status_time = (SELECT MAX(status_time) FROM Data where status_time < A.status_time and
id = A.id)
AND   A.status = 'THREE' AND NOT B.status = 'TWO'
```

Прочитайте Функции окна онлайн: https://riptutorial.com/ru/sql/topic/647/функции-окна

глава 64: Хранимые процедуры

замечания

Хранимые процедуры - это операторы SQL, хранящиеся в базе данных, которые могут выполняться или вызываться в запросах. Использование хранимой процедуры позволяет инкапсулировать сложную или часто используемую логику и улучшает производительность запросов за счет использования кэшированных планов запросов. Они могут возвращать любые значения, возвращаемые стандартным запросом.

Другие преимущества по динамическим выражениям SQL перечислены в Wikipeida .

Examples

Создание и вызов хранимой процедуры

Хранимые процедуры могут быть созданы с помощью графического интерфейса управления базами данных (пример SQL Server) или с помощью инструкции SQL следующим образом:

```
-- Define a name and parameters
CREATE PROCEDURE Northwind.getEmployee
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS

-- Define the query to be run
SELECT FirstName, LastName, Department
FROM Northwind.vEmployeeDepartment
WHERE FirstName = @FirstName AND LastName = @LastName
AND EndDate IS NULL;
```

Вызов процедуры:

```
EXECUTE Northwind.getEmployee N'Ackerman', N'Pilar';

-- Or
EXEC Northwind.getEmployee @LastName = N'Ackerman', @FirstName = N'Pilar';
GO

-- Or
EXECUTE Northwind.getEmployee @FirstName = N'Pilar', @LastName = N'Ackerman';
GO
```

Прочитайте Хранимые процедуры онлайн: https://riptutorial.com/ru/sql/topic/1701/хранимые-процедуры

кредиты

S. No	Главы	Contributors
1	Начало работы с SQL	Arjan Einbu, brichins, Burkhard, cale_b, CL., Community, Devmati Wadikar, Epodax, geeksal, H. Pauwelyn, Hari, Joey, JohnLBevan, Jon Ericson, Lankymart, Laurel, Mureinik, Nathan , omini data, PeterRing, Phrancis, Prateek, RamenChef, Ray, Simone Carletti, SZenC, t1gor, ypercube
2	ALTER TABLE	Aidan, blackbishop, bluefeet, CL., Florin Ghita, Francis Lord, guiguiblitz, Joe W, KIRAN KUMAR MATAM, Lexi, mithra chintha, Ozair Kafray, Simon Foster, Siva Rama Krishna
3	CREATE Database	Emil Rowland
4	DROP или DELETE Database	Abhilash R Vankayala, John Odom
5	GRANT и REVOKE	RamenChef, user2314737
6	IN	CL., juergen d, walid, Zaga
7	MERGE	Abhilash R Vankayala, CL., Kyle Hale, SQLFox, Zoyd
8	SKIP TAKE (Pagination)	CL., Karl Blacquiere, Matas Vaitkevicius, RamenChef
9	SQL CURSOR	Stefan Steiger
10	SQL Group By vs Distinct	carlosb
11	SQL-инъекция	120196, CL., Clomp, Community, Epodax, Knickerless-Noggins, Stefan Steiger
12	TRUNCATE	Abhilash R Vankayala, CL., Cristian Abelleira, DaImTo, Hynek Bernard, inquisitive_mind, KIRAN KUMAR MATAM, Paul Bambury, ss005
13	XML	Steven
14	Блоки выполнения	Phrancis
15	ВСТАВИТЬ	Ameya Deshpande, CL., Daniel Langemann, Dipesh Poudel, inquisitive_mind, KIRAN KUMAR MATAM, rajarshig, Tot Zam,

		zplizzi
16	ВЫБРАТЬ	Abhilash R Vankayala, aholmes, Alok Singh, Amnon, Andrii Abramov, apomene, Arpit Solanki, Arulkumar, AstraSerg, Brent Oliver, Charlie West, Chris, Christian Sagmüller, Christos, CL., controller, dariru, Daryl, David Pine, David Spillett, day_dreamer, Dean Parker, DeepSpace, Dipesh Poudel, Dror, Durgpal Singh, Epodax, Eric VB, FH-Inway, Florin Ghita, FlyingPiMonster, Franck Dernoncourt, geeksal, George Bailey, Hari K M, HoangHieu, iliketocode, Imran Ali Khan, Inca, Jared Hooper, Jaydles, John Odom, John Slegers, Jojodmo, JonH, Kapep, KartikKannapur, Lankymart, Mark Iannucci, Mark Perera, Mark Wojciechowicz, Matas Vaitkevicius, Matt, Matt S, Mattew Whitt, Matthew Moisen, MegaTom, Mihai-Daniel Virna, Mureinik, mustaccio, mxmissile, Oded, Ojen, onedaywhen, Paul Bambury, penderi, Peter Gordon, Prateek, Praveen Tiwari , Přemysl Šťastný, Preuk, Racil Hilan, Robert Columbia, Ronnie Wang, Ryan, Saroj Sasmal, Shiva, SommerEngineering, sqluser, stark, sunkuet02, ThisIsImpossible, Timothy, user1336087, user1605665, waqasahmed, wintersolider, WMios, xQbert, Yury Fedorov, Zahiro Mor, zedfoxus
17	ГРУППА ПО	3N1GM4, Abe Miessler, Bostjan, Devmati Wadikar, Filipe Manuel, Frank, Gidil, Jaydles, juergen d, Nathaniel Ford, Peter Gordon, Simone - Ali One, WesleyJohnson, Zahiro Mor, Zoyd
18	ДЕЛО	ɐlǝx, Christos, CL., Dariusz, Fenton, Infinity, Jaydles, Matt, MotKohn, Mureinik, Peter Lang, Stanislovas Kalašnikovas
19	Дизайн стола	Darren Bartrup-Cook
20	Идентификатор	Andreas, CL.
21	Индексы	a1ex07, Almir Vuk, carlosb, CL., David Manheim, FlyingPiMonster, forsvarir, Franck Dernoncourt, Horaciux, Jenism, KIRAN KUMAR MATAM, mauris, Parado, Paulo Freitas, Ryan
22	Иностранные ключи	CL., Harjot, Yehuda Shapira
23	Информационная схема	Hack-R
24	Каскадное удаление	Stefan Steiger
25	Комментарии	CL., Phrancis

26	крест применяется, наружный применяется	Karthikeyan, RamenChef
27	КРОМЕ	LCIII
28	Материализованные виды	dmfay
29	НОЛЬ	Bart Schuijt, CL., dd4711, Devmati Wadikar, Phrancis, Saroj Sasmal, StanislavL, walid, ypercube
30	Номер строки	CL., Phrancis, user1221533
31	ОБНОВИТЬ	Akshay Anand, CL., Daniel Vérité, Dariusz, Dipesh Poudel, FlyingPiMonster, Gidil, H. Pauwelyn, Jon Chan, KIRAN KUMAR MATAM, Matas Vaitkevicius, Matt, Phrancis, Sanjay Bharwani, sunkuet02, Tot Zam, TriskalJM, vmaroli, WesleyJohnson
32	Общие выражения таблицы	CL., Daniel, dd4711, fuzzy_logic, Gidil, Luis Lema, ninesided, Peter K, Phrancis, Sibeesh Venu
33	ОБЪЯСНЕНИЕ И ОПИСАНИЕ	Simulant
34	Оператор LIKE	Abhilash R Vankayala, Aidan, ashja99, Bart Schuijt, CL., Cristian Abelleira, guiguiblitz, Harish Gyanani, hellyale, Jenism, Lohitha Palagiri, Mark Perera, Mr. Developer, Ojen, Phrancis, RamenChef, Redithion, Stefan Steiger, Tot Zam, Vikrant, vmaroli
35	Операторы AND & OR	guiguiblitz
36	операции	Amir Pourmand, CL., Daryl, John Odom
37	Очистить код в SQL	CL., Stivan
38	Первичные ключи	Andrea Montanari, CL., FlyingPiMonster, KjetilNordin
39	подзапросов	CL., dasblinkenlight, KIRAN KUMAR MATAM, Nunie123, Phrancis, RamenChef, tinlyx
40	Поиск дубликатов в подмножестве столбцов с деталями	Darrel Lee, mnoronha

41	ПОПРОБУЙ ПОЙМАТЬ	Uberzen1
42	Порядок исполнения	a1ex07, Gallus, Ryan Rockey, ypercube
43	Последовательность	John Smith
44	Примеры баз данных и таблиц	Abhilash R Vankayala, Arulkumar, Athafoud, bignose, Bostjan, Brad Larson, Christian, CL., Dariusz, Dr. J. Testington, enrico.bacis, Florin Ghita, FlyingPiMonster, forsvarir, Franck Dernoncourt, hairboat, JavaHopper, Jaydles, Jon Ericson, Magisch, Matt, Mureinik, Mzzzzzz, Prateek, rdans, Shiva, tinlyx , Tot Zam, WesleyJohnson
45	ПРИСОЕДИНИТЬСЯ	A_Arnold, Akshay Anand, Andy G, bignose, Branko Dimitrijevic , Casper Spruit, CL., Daniel Langemann, Darren Bartrup-Cook, Dipesh Poudel, enrico.bacis, Florin Ghita, forsvarir, Franck Dernoncourt, hairboat, Hari K M, HK1, HLGEM, inquisitive_mind, John C, John Odom, John Slegers, Mark Iannucci, Marvin, Mureinik, Phrancis, raholling, Raidri, Saroj Sasmal, Stefan Steiger, sunkuet02, Tot Zam, xenodevil, ypercube, Рахул Маквана
46	Просмотры	Amir978, CL., Florin Ghita
47	Реляционная алгебра	CL., Darren Bartrup-Cook, Martin Smith
48	Синонимы	Daryl
49	СОЗДАТЬ ТАБЛИЦУ	Aidan, alex9311, Almir Vuk, Ares, CL., drunken_monkey, Dylan Vander Berg, Franck Dernoncourt, H. Pauwelyn, Jojodmo, KIRAN KUMAR MATAM, Matas Vaitkevicius, Prateek
50	СОЗДАТЬ ФУНКЦИЮ	John Odom, Ricardo Pontual
51	СОРТИРОВАТЬ ПО	Andi Mohr, CL., Cristian Abelleira, Jaydles, mithra chintha, nazark, Özgür Öztürk, Parado, Phrancis, Wolfgang
52	СОЮЗ / СОЮЗ ВСЕ	Andrea, Athafoud, Daniel Langemann, Jason W, Jim, Joe Taras, KIRAN KUMAR MATAM, Lankymart, Mihai-Daniel Virna , sunkuet02
53	Строковые функции	ɐləx, Allan S. Hansen, Arthur D, Arulkumar, Batsu, Chris, CL., Damon Smithies, Franck Dernoncourt, Golden Gate, hatchet, Imran Ali Khan, IncrediApp, Jaydip Jadhav, Jones Joseph, Kewin Björk Nielsen, Leigh Riffel, Matas Vaitkevicius, Mateusz

		Piotrowski, Neria Nachum, Phrancis, RamenChef, Robert Columbia, vmaroli, ypercube
54	СУЩЕСТВУЕТ СЛОЖНОСТЬ	Blag, Özgür Öztürk
55	Таблица DROP	CL., Joel, KIRAN KUMAR MATAM, Stu
56	Типы данных	bluefeet, Jared Hooper, John Odom, Jon Chan, JonMark Perry , Phrancis
57	Триггеры	Daryl, IncrediApp
58	УДАЛЯТЬ	Batsu, Chip, CL., Dylan Vander Berg, fredden, Joel, KIRAN KUMAR MATAM, Phrancis, Umesh, xenodevil, Zoyd
59	Фильтровать результаты, используя WHERE и HAVING	Arulkumar, Bostjan, CL., Community, Franck Dernoncourt, H. Pauwelyn, Jon Chan, Jon Ericson, juergen d, Matas Vaitkevicius, Mureinik, Phrancis, Tot Zam
60	Функции (Scalar / Single Row)	CL., Kewin Björk Nielsen, Mark Stewart
61	Функции (Агрегат)	ashja99, CL., Florin Ghita, Ian Kenney, Imran Ali Khan, Jon Chan, juergen d, KIRAN KUMAR MATAM, Mark Stewart, Maverick, Nathan, omini data, Peter K, Reboot, Tot Zam, William Ledbetter, winseybash, Алексей Неудачин
62	Функции (аналитические)	CL., omini data
63	Функции окна	Arkh, beercohol, bhs, Gidil, Jerry Jeremiah, Mureinik, mustaccio
64	Хранимые процедуры	brichins, John Odom, Lamak, Ryan