JSPIDER_JUNT_STUDENTS

Getting started with SQL

Version Short Name	Standard	Release Date	
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

Overview

Structured Query Language (SQL) is a special-purpose programming language designed for managing data held in a Relational Database Management System (RDBMS). SQL-like languages can also be used in Relational Data Stream Management Systems (RDSMS), or in "not-only SQL" (NoSQL) databases.

SQL comprises of 3 major sub-languages:

1. Data Definition Language (DDL): to create and modify the structure of the database;
2. Data Manipulation Language (DML): to perform Read, Insert, Update and Delete operations on the data of the database;
3. Data Control Language (DCL): to control the access of the data stored in the database.

SQL article on Wikipedia

The core DML operations are Create, Read, Update and Delete (CRUD for short) which are performed by the statements INSERT, SELECT, UPDATE and DELETE.
There is also a (recently added) MERGE statement which can perform all 3 write operations (INSERT, UPDATE, DELETE).

CRUD article on Wikipedia

Many SQL databases are implemented as client/server systems; the term "SQL server" describes such a database.
At the same time, Microsoft makes a database that is named "SQL Server". While that database speaks a dialect of SQL, information specific to that database is not on topic in this tag but belongs into the SQL Server documentation.

Identifier

This topic is about identifiers, i.e. syntax rules for names of tables, columns, and other database objects.

Where appropriate, the examples should cover variations used by different SQL implementations, or identify the SQL implementation of the example.

Unquoted identifiers

Unquoted identifiers can use letters (a-z), digits (0-9), and underscore (_), and must start with a letter.

Depending on SQL implementation, and/or database settings, other characters may be allowed, some even as the first character, E.g.

- MS SQL: @, $, #, and other Unicode letters *(source)*
- MySQL: $ *(source)*
- Oracle: $, #, and other letters from database character set *(source)* PostgreSQL: $, and
- other Unicode letters *(source)*

Unquoted identifiers are case-insensitive. How this is handled depends greatly on SQL implementation:

- MS SQL: Case-preserving, sensitivity defined by database character set, so can be case-sensitive.
- MySQL: Case-preserving, sensitivity depends on database setting and underlying file system.
- Oracle: Converted to uppercase, then handled like quoted identifier.
- PostgreSQL: Converted to lowercase, then handled like quoted identifier.
- SQLite: Case-preserving; case insensitivity only for ASCII characters.

Data Types

DECIMAL and NUMERIC

Fixed precision and scale decimal numbers. DECIMAL and NUMERIC are functionally equivalent.

Syntax:

```
DECIMAL ( precision [ , scale] )
NUMERIC ( precision [ , scale] )
```

Examples:

```
SELECT CAST(123 AS DECIMAL(5,2)) --returns 123.00
SELECT CAST(12345.12 AS NUMERIC(10,5)) --returns 12345.12000
```

FLOAT and REAL

Approximate-number data types for use with floating point numeric data.

```
SELECT CAST( PI() AS FLOAT) --returns 3.14159265358979
SELECT CAST( PI() AS REAL) --returns 3.141593
```

Integers

Exact-number data types that use integer data.

Data type	**Range**	**Storage**
bigint	-2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807)	8 Bytes
int	-2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)	4 Bytes
smallint	-2^15 (-32,768) to 2^15-1 (32,767)	2 Bytes
tinyint	0 to 255	1 Byte

MONEY and SMALLMONEY

Data types that represent monetary or currency values.

Data type **Range** **Storage**
money -922,337,203,685,477.5808 to 922,337,203,685,477.5807 8 bytes smallmoney -214,748.3648 to 214,748.3647 4 bytes

BINARY and VARBINARY

Binary data types of either fixed length or variable length.

Syntax:

```
BINARY [ ( n_bytes ) ]
VARBINARY [ ( n_bytes | max ) ]
```

n_bytes can be any number from 1 to 8000 bytes. max indicates that the maximum storage space is 2^31-1.
Examples:

```
SELECT CAST(12345 AS BINARY(10)) -- 0x00000000000000003039
SELECT CAST(12345 AS VARBINARY(10)) -- 0x00003039
```

CHAR and VARCHAR

String data types of either fixed length or variable length.

Syntax:

```
CHAR [ ( n_chars ) ]
VARCHAR [ ( n_chars ) ]
```

Examples:

```
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 and NVARCHAR

UNICODE string data types of either fixed length or variable length.

Syntax:

```
NCHAR [ ( n_chars ) ]
NVARCHAR [ ( n_chars | MAX ) ]
```

Use MAX for very long strings that may exceed 8000 characters.

UNIQUEIDENTIFIER

A 16-byte 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'
```

NULL

NULL in SQL, as well as programming in general, means literally "nothing". In SQL, it is easier to understand as "the absence of any value".

It is important to distinguish it from seemingly empty values, such as the empty string '' or the number 0, neither of which are actually NULL.

It is also important to be careful not to enclose NULL in quotes, like 'NULL', which is allowed in columns that accept text, but is not NULL and can cause errors and incorrect data sets.

Filtering for NULL in queries

The syntax for filtering for NULL (i.e. the absence of a value) in WHERE blocks is slightly different than filtering for specific values.

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

Note that because NULL is not equal to anything, not even to itself, using equality operators = (or) will NULL or <> NULL != NULL always yield the truth value of UNKNOWN which will be rejected by WHERE.

WHERE filters all rows that the condition is FALSE or UKNOWN and keeps only rows that the condition is TRUE.

Nullable columns in tables

When creating tables it is possible to declare a column as nullable or non-nullable.

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

By default every column (except those in primary key constraint) is nullable unless we explicitly set constraint. NOT NULL

Attempting to assign NULL to a non-nullable column will result in an error.

```
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.
```

Updating fields to NULL

Setting a field to NULL works exactly like with any other value:

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

Inserting rows with NULL fields

For example inserting an employee with no phone number and no manager into the Employees example table:

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

Example Databases and Tables

Auto Shop Database

In the following example - Database for an auto shop business, we have a list of departments, employees, customers and customer cars. We are using foreign keys to create relationships between the various tables. Live example: SQL fiddle

Relationships between tables

- Each Department may have 0 or more Employees
- Each Employee may have 0 or 1 Manager
- Each Customer may have 0 or more Cars

Departments

Id	**Name**
1	HR
2	Sales
3	Tech

SQL statements to create the table:

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

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

Employees

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
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

SQL statements to create the table:

```
CREATE TABLE Employees (
      IdINT NOT NULL AUTO_INCREMENT,
     FName VARCHAR(35) NOT NULL,
     LName VARCHAR(35) NOT NULL,
     PhoneNumber VARCHAR(11),
     ManagerId INT,
     DepartmentIdINT 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')
;
```

Customers

Id	FName	LName	Email	PhoneNumber	PreferredContact
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

SQL statements to create the table:

```
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')
;
```

Cars

Id	CustomerId	EmployeeId	Model	Status	Total Cost
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				

```
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')
;
```

Library Database

In this example database for a library, we have *Authors*, *Books* and *BooksAuthors* tables.

Live example: SQL fiddle

Authors and *Books* are known as **base tables**, since they contain column definition and data for the actual entities in the relational model. *BooksAuthors* is known as the **relationship table**, since this table defines the relationship between the *Books* and *Authors* table.

Relationships between tables

- Each author can have 1 or more books
- Each book can have 1 or more authors

Authors

(*view table*)

Id	Name	Country
1	J.D. Salinger	USA
2	F. Scott. Fitzgerald	USA
3	Jane Austen	UK
4	Scott Hanselman	USA
5	Jason N. Gaylord	USA
6	Pranav Rastogi	India
7	Todd Miranda	USA
8	Christian Wenz	USA

SQL to create the table:

```
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')
;
```

Books

(*view table*)

Id	**Title**
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 SQL to

create the table:

```
CREATE TABLE Books (
      IdINT NOT NULL AUTO_INCREMENT,
      TitleVARCHAR(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

(*view table*)

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 to create the table:

```
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)
;
```

Examples

View all authors (view live example):

```
SELECT * FROM Authors;
```

View all book titles (view live example):

```
SELECT * FROM Books;
```

View all books and their authors (view live example):

```
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
;
```

Countries Table

In this example, we have a **Countries** table. A table for countries has many uses, especially in Financial applications involving currencies and exchange rates.

Live example: SQL fiddle

Some Market data software applications like Bloomberg and Reuters require you to give their API either a 2 or 3 character country code along with the currency code. Hence this example table has both the 2-character ISO code column and the 3 character ISO3 code columns.

Countries

(*view table*)

Id	ISO	ISO3	ISONumeric	CountryName	Capital	ContinentCode	CurrencyCode
1	AU	AUS	36	Australia	Canberra	OC	AUD
2	DE	DEU	276	Germany	Berlin	EU	EUR
2	IN	IND	356	India	New Delhi	AS	INR
3	LA	LAO	418	Laos	Vientiane	AS	LAK
4	US	USA	840	United States	Washington	NA	USD
5	ZW	ZWE	716	Zimbabwe	Harare	AF	ZWL

SQL to create the table:

```
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')
;
```

SELECT

The SELECT statement is at the heart of most SQL queries. It defines what result set should be returned by the query, and is almost always used in conjunction with the FROM clause, which defines what part(s) of the database should be queried.

Using the wildcard character to select all columns in a query

Consider a database with the following two tables.

Employees table:

Id FName LName DeptId 1
James Smith 3

2 John Johnson 4

Departments table:

Id	**Name**
1	Sales
2	Marketing
3	Finance
4	IT

Simple select statement

* is the **wildcard character** used to select all available columns in a table.

When used as a substitute for explicit column names, it returns all columns in all tables that a query is selecting FROM. This effect applies to **all tables** the query accesses through its JOIN clauses.

Consider the following query:

```
SELECT * FROM Employees
```

It will return all fields of all rows of the Employees table:

Id FName LName DeptId 1
James Smith 3

2 John Johnson 4

Dot notation

To select all values from a specific table, the wildcard character can be applied to the table with *dot notation*.

Consider the following query:

```
SELECT
        Employees.*,
        Departments.Name
FROM
        Employees
JOIN
```

```
  Departments
 ON Departments.Id = Employees.DeptId
```

This will return a data set with all fields on the Employee table, followed by just the Name field in the Departments table:

Id	FName	LName	DeptId	Name
1	James	Smith	3	Finance
2	John	Johnson	4	IT

Warnings Against Use

It is generally advised that using * is avoided in production code where possible, as it can cause a number of potential problems including:

1. Excess IO, network load, memory use, and so on, due to the database engine reading data that is not needed and transmitting it to the front-end code. This is particularly a concern where there might be large fields such as those used to store long notes or attached files.

2. Further excess IO load if the database needs to spool internal results to disk as part of the processing for a query more complex than >. SELECT <columns > FROM <table
3. Extra processing (and/or even more IO) if some of the unneeded columns are: computed columns in databases that support
   - them in the case of selecting from a view, columns from a table/view that the query optimiser could otherwise
   - optimise out

4. The potential for unexpected errors if columns are added to tables and views later that results ambiguous column names. For example SELECT * FROM orders JOIN people ON people . id = orders . personid ORDER
- if a BY displayname column column called displayname is added to the orders table to allow users to give their orders meaningful names for future reference then the column name will appear twice in the output so the clause will be ORDER BY ambiguous which may cause errors ("ambiguous column name" in recent MS SQL Server versions), and if not in this example your application code might start displaying the order name where the person name is intended because the new column is the first of that name returned, and so on.

When Can You Use *, Bearing The Above Warning In Mind?

While best avoided in production code, using * is fine as a shorthand when performing manual queries against the database for investigation or prototype work.

Sometimes design decisions in your application make it unavoidable (in such circumstances, prefer tablealias.* over just * where possible).

SELECT A . col1 , A . Col2 FROM A WHERE EXISTS (SELECT * FROM B where A . ID
A_ID When using EXISTS, such as =
B.), we are not returning any data from B. Thus a join is unnecessary, and the engine knows no values from B are to be returned, thus no performance hit for using *. Similarly (*) is fine as it also COUNT doesn't actually return any of the columns, so only needs to read and process those that are used for filtering purposes.

SELECT Using Column Aliases

Column aliases are used mainly to shorten code and make column names more readable.

Code becomes shorter as long table names and unnecessary identification of columns *(e.g., there may be 2 IDs in the table, but only one is used in the statement)* can be avoided. Along with table aliases this allows you to use longer descriptive names in your database structure while keeping queries upon that structure concise.

Furthermore they are sometimes *required*, for instance in views, in order to name computed outputs.

All versions of SQL

Aliases can be created in all versions of SQL using double quotes (").

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

Different Versions of SQL

You can use single quotes ('), double quotes (") and square brackets ([]) to create an alias in Microsoft SQL Server.

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

Both will result in:

First Name	Middle Name	Last Name
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

This statement will return FName and LName columns with a given name (an alias). This is achieved using the AS operator followed by the alias, or simply writing alias directly after the column name. This means that the following query has the same outcome as the above.

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

First Name	Middle Name	Last Name
James	John	Smith
John	James	Johnson
Michael	Marcus	Williams

However, the explicit version (i.e., using the AS operator) is more readable.

If the alias has a single word that is not a reserved word, we can write it without single quotes, double quotes or brackets:

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

FirstName	LastName
James	Smith
John	Johnson
Michael	Williams

A further variation available in MS SQL Server amongst others is **<alias> = <column-or-calculation>**, for instance:

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

which is equivalent to:

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

Both will result in:

FullName	Addr1	Addr2
James Smith	123 AnyStreet	TownVille
John Johnson	668 MyRoad	Anytown
Michael Williams	999 High End Dr	Williamsburgh

Some find using = instead of As easier to read, though many recommend against this format, mainly because it is not standard so not widely supported by all databases. It may cause confusion with other uses of the = character.

All Versions of SQL

Also, if you *need* to use reserved words, you can use brackets or quotes to escape:

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

Different Versions of SQL

Likewise, you can escape keywords in MSSQL with all different approaches:

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

SELECT	FROM	WHERE
James	John	Smith
John	James	Johnson

Also, a column alias may be used any of the final clauses of the same query, such as an : ORDER BY

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

However, you may *not* use

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

To create an alias from these reserved words (SELECT and FROM).

This will cause numerous errors on execution.

Select Individual Columns

```
SELECT
    PhoneNumber,
    Email,
    PreferredContact
FROM Customers
```

This statement will return the columns PhoneNumber, Email, and PreferredContact from all rows of the Customers table. Also the columns will be returned in the sequence in which they appear in the SELECT clause.

The result will be:

PhoneNumber	Email	PreferredContact
3347927472	william.jones@example.com	PHONE
2137921892	dmiller@example.net	EMAIL
NULL	richard0123@example.com	EMAIL

If multiple tables are joined together, you can select columns from specific tables by specifying the table name before the column name: [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` * means that the `Id` field of `Orders` table will be returned as a column named `OrderId`. See selecting with column alias for further information.

To avoid using long table names, you can use table aliases. This mitigates the pain of writing long table names for each field that you select in the joins. If you are performing a self join (a join between two instances of the *same* table), then you must use table aliases to distinguish your tables. We can write a table alias like `Customers c` or `Customers AS c`. Here `c` works as an alias for `Customers` and we can select let's say `Email` like this: `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
```

Selecting specified number of records

The SQL 2008 standard defines the `FETCH FIRST` clause to limit the number of records returned.

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

This standard is only supported in recent versions of some RDMSs. Vendor-specific non-standard syntax is provided in other systems. Progress OpenEdge 11.x also supports the `FETCH FIRST <n> ROWS ONLY` syntax.

Additionally, `OFFSET <m> ROWS` before `FETCH FIRST <n> ROWS ONLY` allows skipping rows before fetching rows.

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

The following query is supported in SQL Server and MS Access:

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

To do the same in MySQL or PostgreSQL the LIMIT keyword must be used:

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

In Oracle the same can be done withROWNUM:

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

Results: **10** records.

```
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
```

Vendor Nuances:

It is important to note that the TOP in Microsoft SQL operates after the WHERE clause and will return the specified number of results if they exist anywhere in the table, while ROWNUM works as part of the WHERE clause so if other conditions do not exist in the specified number of rows at the beginning of the table, you will get zero results when there could be others to be found.

Selecting with Condition

The basic syntax of SELECT with WHERE clause is:

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

The *[condition]* can be any SQL expression, specified using comparison or logical operators like >, <, =, <>, >=, <=, LIKE, NOT, IN, BETWEEN etc.

The following statement returns all columns from the table 'Cars' where the status column is 'READY':

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

See WHERE and HAVING for more examples.

Selecting with CASE

When results need to have some logic applied 'on the fly' one can use CASE statement to implement it.

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

also can be chained

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

one also can have CASE inside another CASE statement

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

Select columns which are named after reserved keywords

When a column name matches a reserved keyword, standard SQL requires that you enclose it in double quotation marks:

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

Note that it makes the column name case-sensitive.

Some DBMSes have proprietary ways of quoting names. For example, SQL Server uses square brackets for this purpose:

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

while MySQL (and MariaDB) by default use backticks:

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

Selecting with table alias

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

The Employees table is given the alias 'e' directly after the table name. This helps remove ambiguity in scenarios where multiple tables have the same field name and you need to be specific as to which table you want to return data from.

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

Note that once you define an alias, you can't use the canonical table name anymore. i.e.,

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

would throw an error.

It is worth noting table aliases -- more formally 'range variables' -- were introduced into the SQL language to solve the problem of duplicate columns caused by . The 1992 SQL INNER JOIN standard corrected this earlier design flaw by introducing (implemented in NATURAL JOIN mySQL, PostgreSQL and Oracle but not yet in SQL Server), the result of which never has duplicate column names. The above example is interesting in that the tables are joined on columns with different names (Id and ManagerId) but are not supposed to be joined on the columns with the same name (LName, FName), requiring the renaming of the columns to be performed *before* the join:

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

Note that although an alias/range variable must be declared for the dervied table (otherwise SQL will throw an error), it never makes sense to actually use it in the query.

Selecting with more than 1 condition

The AND keyword is used to add more conditions to the query.

Name	Age	Gender
Sam	18	M
John	21	M
Bob	22	M
Mary	23	F

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

This will return:

Name
John Bob

using OR keyword

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

This will return:

name Sam
John Bob

These keywords can be combined to allow for more complex criteria combinations:

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

This will return:

name Sam

Mary

Selecting without Locking the table

Sometimes when tables are used mostly (or only) for reads, indexing does not help anymore and every little bit counts, one might use selects without LOCK to improve performance.

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;
```

Oracle

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

DB2

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

where UR stands for "uncommitted read".

If used on table that has record modifications going on might have unpredictable results.

Selecting with Aggregate functions

Average

The AVG() aggregate function will return the average of values selected.

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

Aggregate functions can also be combined with the where clause.

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

Aggregate functions can also be combined with group by clause.

If employee is categorized with multiple department and we want to find avg salary for every department then we can use following query.

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

Minimum

The MIN() aggregate function will return the minimum of values selected.

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

Maximum

The MAX() aggregate function will return the maximum of values selected.

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

Count

The COUNT() aggregate function will return the count of values selected.

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

It can also be combined with where conditions to get the count of rows that satisfy specific conditions.

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

Specific columns can also be specified to get the number of values in the column. Note that NULL values are not counted.

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

Count can also be combined with the distinct keyword for a distinct count.

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

Sum

The SUM() aggregate function returns the sum of the values selected for all rows.

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

Select with condition of multiple values from column

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

This is semantically equivalent to

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

i.e. value IN (<value list>) is a shorthand for disjunction (logical OR).

Get aggregated result for row groups

Counting rows based on a specific column value:

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

Getting average income by department:

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

GROUP BY

The important thing is to select only columns specified in the clause or used with aggregate functions.

GROUP BY

There WHERE clause can also be used with , but WHERE filters out records *before* any grouping is done:

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

If you need to filter the results after the grouping has been done, e.g, to see only departments whose average income is larger than 1000, you need to use the HAVING clause:

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

Selection with sorted Results

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

This statement will return all the columns from the table Employees.

Id FName LName PhoneNumber 2 John
Johnson 2468101214

1 James Smith 1234567890

3 Michael Williams 1357911131

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

Or

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

This statement changes the sorting direction.

One may also specify multiple sorting columns. For example:

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

This example will sort the results first by LName and then, for records that have the same LName, sort by FName. This will give you a result similar to what you would find in a telephone book.

In order to save retyping the column name in the ORDER BY clause, it is possible to use instead the column's number. Note that column numbers start from 1.

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

You may also embed a CASE statement in the ORDER BY clause.

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

This will sort your results to have all records with the LName of "Jones" at the top.

Selecting with null

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

Selection with nulls take a different syntax. Don't use =, use IS NULL or IS NOT NULL instead.

Select distinct (unique values only)

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

This query will return all DISTINCT (unique, different) values from ContinentCode column from Countries table

ContinentCode OC

EU

AS

NA AF

SQLFiddle Demo

Select rows from multiple tables

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

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

This is called cross product in SQL it is same as cross product in sets

These statements return the selected columns from multiple tables in one query.

There is no specific relationship between the columns returned from each table.

GROUP BY

Results of a SELECT query can be grouped by one or more columns using the GROUP BY statement: all results with the same value in the grouped columns are aggregated together. This generates a table of partial results, instead of one result. GROUP BY can be used in conjunction with aggregation functions using the HAVING statement to define how non-grouped columns are aggregated.

Basic GROUP BY example

It might be easier if you think of GROUP BY as "for each" for the sake of explanation. The query below:

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

is saying:

"Give me the sum of MonthlySalary's **for each** EmpID" So if your

table looked like this:

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

Result:

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

Sum wouldn't appear to do anything because the sum of one number is that number. On the other hand if it looked like this:

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

Result:

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

Then it would because there are two EmpID 1's to sum together.

Filter GROUP BY results using a HAVING clause

A HAVING clause filters the results of a GROUP BY expression. Note: The following examples are using the Library example database.

Examples:

Return all authors that wrote more than one book (live example).

```
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
;
```

Return all books that have more than three authors (live example).

```
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
;
```

USE GROUP BY to COUNT the number of rows for each unique entry in a given column

Let's say you want to generate counts or subtotals for a given value in a column.

Given this table, "Westerosians":

Name	**GreatHouseAllegience**
Arya	Stark
Cercei	Lannister
Myrcella	Lannister
Yara	Greyjoy
Catelyn	Stark
Sansa	Stark

Without GROUP BY, COUNT will simply return a total number of rows:

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

returns...

Number_of_Westerosians
6

But by adding GROUP BY, we can COUNT the users for each value in a given column, to return the number of people in a given Great House, say:

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

returns...

House	Number_of_Westerosians
Stark	3
Greyjoy	1
Lannister	2

It's common to combine GROUP BY with ORDER BY to sort results by largest or smallest category:

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

returns...

House	Number_of_Westerosians
Stark	3
Lannister	2
Greyjoy	1

ROLAP aggregation (Data Mining)

Description

The SQL standard provides two additional aggregate operators. These use the polymorphic value "ALL" to denote the set of all values that an attribute can take. The two operators are:

- with data cube
- with roll up that it provides all possible combinations than the argument attributes of the clause. that it provides the aggregates obtained by considering the attributes in order from left to right compared how they are listed in the argument of the clause.

SQL standard versions that support these features: 1999,2003,2006,2008,2011.

Examples

Consider this table:

Food Brand Total_amount

Pasta Brand1 100

Pasta Brand2 250

Pizza Brand2 300

With cube

```
select Food,Brand,Total_amount
from Table
```

group by Food,Brand,Total_amount with cube

Food Brand Total_amount

Pasta Brand1 100

Pasta Brand2 250

Pasta ALL 350

Pizza Brand2 300

Pizza ALL 300

ALL Brand1 100

ALL Brand2 550

ALL ALL 650

With roll up

```
select Food,Brand,Total_amount
from Table
```

group by Food,Brand,Total_amount with roll up

Food Brand Total_amount

Pasta Brand1 100

Pasta Brand2 250

Pizza Brand2 300

Pasta ALL 350

Pizza ALL 300

ALL ALL 650

ORDER BY

Sorting by column number (instead of name)

You can use a column's number (where the leftmost column is '1') to indicate which column to base the sort on, instead of describing the column by its name.

Pro: If you think it's likely you might change column names later, doing so won't break this code.

Con: This will generally reduce readability of the query (It's instantly clear what 'ORDER BY Reputation' means, while 'ORDER BY 14' requires some counting, probably with a finger on the screen.)

This query sorts result by the info in relative column position 3 from select statement instead of column name `Reputation`.

SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY 3

DisplayName	JoinDate	Reputation
Community	2008-09-15	**1**

Jarrod Dixon	2008-10-03	**11739**
Geoff Dalgas	2008-10-03	**12567**
Joel Spolsky	2008-09-16	**25784**
Jeff Atwood	2008-09-16	**37628**

Use ORDER BY with TOP to return the top x rows based on a column's value

In this example, we can use GROUP BY not only determined the *sort* of the rows returned, but also what rows *are* returned, since we're using TOP to limit the result set.

Let's say we want to return the top 5 highest reputation users from an unnamed popular Q&A site.

Without ORDER BY

This query returns the Top 5 rows ordered by the default, which in this case is "Id", the first column in the table (even though it's not a column shown in the results).

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
```

returns...

DisplayName	Reputation
Community	1
Geoff Dalgas	12567
Jarrod Dixon	11739
Jeff Atwood	37628
Joel Spolsky	25784

With ORDER BY

```
SELECT TOP 5 DisplayName, Reputation
FROM Users
ORDER BY Reputation desc
```

returns...

DisplayName	Reputation
JonSkeet	**865023**
Darin Dimitrov	**661741**
BalusC	**650237**
Hans Passant	**625870**
Marc Gravell	**601636**

Remarks

Some versions of SQL (such as MySQL) use a LIMIT clause at the end of a SELECT, instead of TOP at the beginning, for example:

```
SELECT DisplayName, Reputation
FROM Users
ORDER BY Reputation DESC
LIMIT 5
```

Customizeed sorting order

To sort this table Employee by department, you would use . ORDER BY Department However, if you want a different sort order that is not alphabetical, you have to map the Department values into different values that sort correctly; this can be done with a CASE expression:

Name Department Hasan IT

Yusuf HR

Hillary HR

Joe IT

Merry HR

```
SELECT *
FROM Employee





         END;
```

Ken Accountant

```
ORDER BY CASE Department
     WHEN 'HR'        THEN 1
     WHEN 'Accountant' THEN 2
     ELSE             3
```

Name Department Yusuf **HR**

Hillary **HR**

Merry **HR**

Ken **Accountant**

Hasan **IT**

Joe **IT**

Order by Alias

Due to logical query processing order, alias can be used in order by.

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY jd, rep
```

And can use relative order of the columns in the select statement .Consider the same example as above and instead of using alias use the relative order like for display name it is 1 , for Jd it is 2 and so on

```
SELECT DisplayName, JoinDate as jd, Reputation as rep
FROM Users
ORDER BY 2, 3
```

Sorting by multiple columns

SELECT DisplayName, JoinDate, Reputation FROM Users ORDER BY JoinDate, Reputation

DisplayName	JoinDate	Reputation
Community	**2008-09-15**	**1**
Jeff Atwood	**2008-09-16**	**25784**
Joel Spolsky	2008-09-16	**37628**
Jarrod Dixon	**2008-10-03**	**11739**
Geoff Dalgas	2008-10-03	**12567**

AND & OR Operators

AND OR Example

Have a table

Name	Age	City
Bob	10	Paris
Mat	20	Berlin
Mary	24	Prague

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

Gives

Name
Mary

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

Gives

Name
Bob
Mary

CASE

The CASE expression is used to implement if-then logic.

Use CASE to COUNT the number of rows in a column match a condition

Use Case

CASE can be used in conjunction with SUM to return a count of only those items matching a pre-defined condition. (This is similar to COUNTIF in Excel.)

The trick is to return binary results indicating matches, so the "1"s returned for matching entries can be summed for a count of the total number of matches.

Given this table ItemSales, let's say you want to learn the total number of items that have been categorized as "Expensive":

Id	ItemId	Price	PriceRating
1	100	34.5	EXPENSIVE
2	145	2.3	CHEAP
3	100	34.5	EXPENSIVE
4	100	34.5	EXPENSIVE
5	145	10	AFFORDABLE

Query

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

Results:

ItemsCount	ExpensiveItemsCount
5	3

Alternative:

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

Searched CASE in SELECT (Matches a boolean expression)

The *searched* CASE returns results when a *boolean* expression is TRUE.

(This differs from the simple case, which can only check for equivalency with an input.)

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

```
Id ItemId Price  PriceRating
1  100    34.5   EXPENSIVE
2  145    2.3    CHEAP
3 100     34.5 EXPENSIVE
4 100     34.5 EXPENSIVE
5 145     10       AFFORDABLE
```

CASE in a clause ORDER BY

We can use 1,2,3.. to determine the type of order:

```sql
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
```

```
ID    REGION        CITY          DEPARTMENT     EMPLOYEES_NUMBER
12 New England Boston             MARKETING          9
15 West             San Francisco MARKETING          12
9    Midwest        Chicago       SALES              8
14 Mid-Atlantic     New York      SALES              12
5    West           Los Angeles   RESEARCH           11
10 Mid-Atlantic     Philadelphia  RESEARCH           13
4    Midwest        Chicago       INNOVATION         11
2    Midwest        Detroit            HUMAN RESOURCES 9
```

Shorthand CASE in SELECT

CASE's shorthand variant evaluates an expression (usually a column) against a series of values. This variant is a bit shorter, and saves repeating the evaluated expression over and over again. The ELSE clause can still be used, though:

```sql
SELECT  Id,  ItemId,  Price,
   CASE  Price  WHEN  5   THEN  'CHEAP'
                WHEN  15  THEN  'AFFORDABLE'

                ELSE          'EXPENSIVE'
   END  as  PriceRating
FROM  ItemSales
```

A word of caution. It's important to realize that when using the short variant the entire statement is evaluated at each WHEN. Therefore the following statement:

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

may produce a NULL result. That is because at each WHEN () is NEWID being called again with a new result. Equivalent to:

```
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
```

Therefore it can miss all the WHEN cases and result as NULL.

Using CASE in UPDATE

sample on price increases:

```
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
```

in this way '0' representing the known values are ranked first, '1' representing the NULL values are sorted by the last:

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

ID	REGION	CITY	DEPARTMENT	EMPLOYEES_NUMBER
10	Mid-Atlantic	Philadelphia	RESEARCH	13
14	Mid-Atlantic	New York	SALES	12
9	Midwest	Chicago	SALES	8
12	New England	Boston	MARKETING	9
5	West	Los Angeles	RESEARCH	11
15	NULL	San Francisco	MARKETING	12
4	NULL	Chicago	INNOVATION	11
2	NULL	Detroit	HUMAN RESOURCES 9	

CASE in ORDER BY clause to sort records by lowest value of 2 columns

Imagine that you need sort records by lowest value of either one of two columns. Some databases could use a non-aggregated MIN() or LEAST() function for this (... ORDER BY MIN(Date1, Date2)), but in standard SQL, you have to use a CASE expression.

The CASE expression in the query below looks at the Date1 and Date2 columns, checks which column has the lower value, and sorts the records depending on this value.

Sample data

Id	Date1	Date2
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

```
SELECT Id, Date1, Date2


          WHEN COALESCE(Date1, '1753-01-01') < COALESCE(Date2, '1753-01-01') THEN Date1
          ELSE Date2
        END
```

6 2017-01-04 2017-01-31 **Query**

```
FROM YourTable
ORDER BY CASE
```

Results		
Id	**Date1**	**Date2**
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	

Explanation

As you see row with = Id 1 is first, that because Date1 have lowest record from entire table , row 2017 - 01 - 01 where = 3 is Id

2017 - 01 - 02

second that because Date2 equals to that is second lowest value from table and so on.

2017 - 01 - 01 to 2017 - 01 - 06

So we have sorted records from ascending and no care on which one column Date1 or Date2 are those values.

LIKE operator

Match open-ended pattern

The % wildcard appended to the beginning or end (or both) of a string will allow 0 or more of any character before the beginning or after the end of the pattern to match.

Using '%' in the middle will allow 0 or more characters between the two parts of the pattern to match.

We are going to use this Employees Table:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
1	John	Johnson	2468101214	1	1	400	23-03-2005
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
3	Ronny	Smith	2462544026	2	1	600	06-08-2015
4	Jon	Sanchez	2454124602	1	1	400	23-03-2005
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

Following statement matches for all records having FName **containing** string 'on' from Employees Table.

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

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
3	R**on**ny	Smith	2462544026	2	1	600	06-08-2015
4	**Jon**	Sanchez	2454124602	1	1	400	23-03-2005

Following statement matches all records having PhoneNumber **starting with** string '246' from Employees.

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

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
1	John	Johnson	**246**8101214	1	1	400	23-03-2005
3	Ronny	Smith	**246**2544026	2	1	600	06-08-2015
5	Hilde	Knag	**246**8021911	2	1	800	01-01-2000

Following statement matches all records having PhoneNumber **ending with** string '11' from Employees.

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

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
2	Sophie	Amudsen	2479100211	1	1	400	11-01-2010
5	Hilde	Knag	2468021911	2	1	800	01-01-2000

All records where Fname **3rd character** is 'n' from Employees.

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

(two underscores are used before 'n' to skip first 2 characters)

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	Hire_date
3	Ronny	Smith	2462544026	2	1	600	06-08-2015
4	To broaden the Jon	Sanchez	2454124602	1	1	400	23-03-2005

Single character match

selections of a structured query language (SQL-SELECT) statement, wildcard characters, the percent sign (%) and the underscore (_), can be used.

The _ (underscore) character can be used as a wildcard for any single character in a pattern match.

Find all employees whose Fname start with 'j' and end with 'n' and has exactly 3 characters in Fname.

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

_ (underscore) character can also be used more than once as a wild card to match patterns.

For example, this pattern would match "jon", "jan", "jen", etc.

These names will not be shown "jn","john","jordan", "justin", "jason", "julian", "jillian", "joann" because in our query one underscore is used and it can skip exactly one character, so result must be of 3 character Fname.

For example, this pattern would match "LaSt", "LoSt", "HaLt", etc.

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

ESCAPE statement in the LIKE-query

If you implement a text-search as LIKE-query, you usually do it like this:

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

However, (apart from the fact that you shouldn't necessarely use LIKE when you can use fulltext-search) this creates a problem when somebody inputs text like "50%" or "a_b".

So (instead of switching to fulltext-search), you can solve that problem using the LIKE-escape statement:

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

That means \ will now be treated as ESCAPE character. This means, you can now just prepend \ to every character in the string you search, and the results will start to be correct, even when the user enters a special character like % or _.e.g.

```
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);
```

Note: The above algorithm is for demonstration purposes only. It will not work in cases where 1 `string stringToSearch = "Les Mise\u0301rables"` grapheme consists out of several characters (utf-8). e.g. ; You'll need to do this for each grapheme, not for each character. You should not use the above algorithm if you're dealing with Asian/East-Asian/South-Asian languages. Or rather, if you want correct code to begin with, you should just do that for each graphemeCluster.

See also ReverseString, a C# interview-question

Search for a range of characters

Following statement matches all records having FName that starts with a letter from A to F from Employees Table.

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

abcdef

Match by range or set

Match any single character within the specified range (e.g.: [a-f]) or set (e.g.: []).

This range pattern would match "gary" but not "mary":

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

This set pattern would match "mary" but not "gary":

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

The range or set can also be negated by appending the ^ caret before the range or set:

This range pattern would *not* match "gary" but will match "mary":

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

This set pattern would *not* match "mary" but will match"gary":

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

Wildcard characters

wildcard characters are used with the SQL LIKE operator. SQL wildcards are used to search for data within a table. Wildcards in SQL

are:%, _, [charlist], [^charlist]

% - A substitute for zero or more characters

```
    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%';
```

_ - A substitute for a single character

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

WHERE City LIKE '_erlin';
```

[charlist] - Sets and ranges of characters to match

```
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] - Matches only a character NOT specified within the brackets

```
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 '_%';
```

IN clause

Simple IN clause

To get records having **any** of the given ids

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

The query above is equal to

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

Using IN clause with a subquery

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

The above will give you all the customers that have orders in the system.

Filter results using WHERE and HAVING

Use BETWEEN to Filter Results

The following examples use the Item Sales and Customers sample databases.

> Note: The BETWEEN operator *is* inclusive.

Using the BETWEEN operator with Numbers:

```
SELECT * From ItemSales
WHERE Quantity BETWEEN 10 AND 17
```

This query will return all ItemSales records that have a quantity that is greater or equal to 10 and less than or equal to 17. The results will look like:

Id	SaleDate	ItemId	Quantity	Price
1	2013-07-01	100	10	34.5
4	2013-07-23	100	15	34.5
5	2013-07-24	145	10	34.5

Using the BETWEEN operator with Date Values:

```
SELECT * From ItemSales
WHERE SaleDate BETWEEN '2013-07-11' AND '2013-05-24'
```

This query will return all ItemSales records with a SaleDate that is greater than or equal to July 11, 2013 and less than or equal to May 24, 2013.

Id SaleDate ItemId Quantity Price

3 2013-07-11 20 34.5
100

4 2013-07-23 100 15 34.5

5 2013-07-24 145 10 34.5

When comparing datetime values instead of dates, you may need to convert the datetime values into a date values, or add or subtract 24 hours to get the correct results.

Using the BETWEEN operator with Text Values:

```
SELECT Id, FName, LName FROM Customers
WHERE LName BETWEEN 'D' AND 'L';
```

Live example: SQL fiddle

This query will return all customers whose name alphabetically falls between the letters 'D' and 'L'. In this case, Customer #1 and #3 will be returned. Customer #2, whose name begins with a 'M' will not be included.

Id FName LName

1 William Jones 3

Richard Davis

Use HAVING with Aggregate Functions

Unlike the WHERE clause, HAVING can be used with aggregate functions.

An aggregate function is a function where the values of multiple rows are grouped together as input on certain criteria to

COUNT(), SUM(), MIN(), and MAX

form a single value of more significant meaning or measurement (Wikipedia).

Common aggregate functions include ().

This example uses the Car Table from the Example Databases.

```
SELECT CustomerId, COUNT(Id) AS [Number of Cars]
FROM Cars
GROUP BY CustomerId
HAVING COUNT(Id) > 1
```

This query will return the CustomerId and count Number of Cars of any customer who has more than one car. In this case, the only customer who has more than one car is Customer #1.

The results will look like:

CustomerId	Number of Cars
1	2

WHERE clause with NULL/NOT NULL values

```
SELECT *
FROM Employees
WHERE ManagerId IS NULL
```

This statement will return all Employee records where the value of the ManagerId column is NULL.

The result will be:

```
Id     FName     LName     PhoneNumber     ManagerId     DepartmentId
1      James     Smith     1234567890      NULL          1
```

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

This statement will return all Employee records where the value of the ManagerId is *not* NULL.

The result will be:

```
Id     FName        LName      PhoneNumber     ManagerId     DepartmentId
2      John         Johnson    2468101214      1             1
3      Michael      Williams   1357911131      1             2
4      Johnathon    Smith      1212121212      2             1
```

Note: The same query will not return results if you change the WHERE clause to WHERE ManagerId = NULL or WHERE ManagerId <> NULL.

Equality

```
SELECT * FROM Employees
```

This statement will return all the rows from the table 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
```

Using a WHERE at the end of your SELECT statement allows you to limit the returned rows to a condition. In this case, where there is an exact match using the = sign:

```
SELECT * FROM Employees WHERE DepartmentId = 1
```

Will only return the rows where the DepartmentId is equal to 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
```

The WHERE clause only returns rows that match its criteria

Steam has a games under $10 section of their store page. Somewhere deep in the heart of their systems, there's probably a query that looks something like:

```
SELECT *
FROM Items
WHERE Price < 10
```

AND and OR

You can also combine several operators together to create more complex WHERE conditions. The following examples use the Employees table:

```
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
```

AND

```
SELECT * FROM Employees WHERE DepartmentId = 1 AND ManagerId = 1
```

Will return:

```
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
```

OR

```
SELECT * FROM Employees WHERE DepartmentId = 2 OR ManagerId = 2
```

Will return:

```
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
```

Use IN to return rows with a value contained in a list

This example uses the Car Table from the Example Databases.

```
SELECT *
FROM Cars
WHERE TotalCost IN (100, 200, 300)
```

This query will return Car #2 which costs 200 and Car #3 which costs 100. Note that this is equivalent to using multiple clauses with OR, e.g.:

```
SELECT *
FROM Cars
WHERE TotalCost = 100 OR TotalCost = 200 OR TotalCost = 300
```

Use LIKE to find matching strings and substrings

See full documentation on LIKE operator.
This example uses the Employees Table from the Example Databases.

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

This query will only return Employee #1 whose first name matches 'John' exactly.

```
SELECT *
FROM Employees
WHERE FName like 'John%'
```

Adding % allows you to search for a substring:

- John
- John
- John % - will return any Employee whose name begins with 'John', followed by any amount of characters

% - will return any Employee whose name ends with 'John', proceeded by any amount of characters

%% - will return any Employee whose name contains 'John' anywhere within the value

In this case, the query will return Employee #2 whose name is 'John' as well as Employee #4 whose name is 'Johnathon'.

Where EXISTS

Will select records in TableName that have records matching in TableName1.

```
SELECT * FROM TableName t WHERE EXISTS (
    SELECT 1 FROM TableName1 t1 where t.Id = t1.Id)
```

Use HAVING to check for multiple conditions in a group

Orders Table

CustomerId	ProductId	Quantity	Price
1	2	5	100
1	3	2	200
1	4	1	500
2	1	4	50
3	5	6	700

To check for customers who have ordered both - ProductID 2 and 3, HAVING can be used

```
select customerId
from orders
where productID in (2,3)
group by customerId
having count(distinct productID) = 2
```

Return value:

customerId
1

The query selects only records with the productIDs in questions and with the HAVING clause checks for groups having 2 productIds

and not just one. Another possibility would be

```
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
```

This query selects only groups having at least one record with productID 2 and at least one with productID 3.

SKIP TAKE (Pagination)

Limiting amount of results

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]
```

Skipping then taking some results (Pagination)

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;
```

Skipping some rows from result

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;
```

EXCEPT

Select dataset except where values are in this other dataset

```
--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
```

EXPLAIN and DESCRIBE

EXPLAIN Select query

An Explain infront of a select query shows you how the query will be executed. This way you to see if the query uses an index or if you could optimize your query by adding an index.

Example query:

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

Example result:

```
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)
```

on type you see if an index was used. In the column possible_keys you see if the execution plan can choose from different indexes of if none exists. key tells you the acutal used index. key_len shows you the size in bytes for one index item. The lower this value is the more index items fit into the same memory size an they can be faster processed. rows shows you the expected number of rows the query needs to scan, the lower the better.

DESCRIBE tablename;

DESCRIBE and EXPLAIN are synonyms. DESCRIBE on a tablename returns the definition of the columns.

```
DESCRIBE tablename;
```

Exmple Result:

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

Here you see the column names, followed by the columns type. It shows if null is allowed in the column and if the column uses an Index. the default value is also displayed and if the table contains any special behavior like an auto_increment.

CLAUSE

Customer Table

Id	FirstName	LastName
1	OzgurOzturk	
2	Youssef	Medi
3	Henry	Tai

Order Table

Id	CustomerId	Amount
1	2	123.50
2	3	14.80

Get all customers with a least one order

```
    SELECT * FROM Order WHERE Order.CustomerId=Customer.Id
)
```
SELECT * FROM Customer WHERE EXISTS (

Result

Id	FirstName	LastName
2	Youssef	Medi
3	Henry	Tai

Get all customers with no order

```
SELECT * FROM Customer WHERE NOT EXISTS (
    SELECT * FROM Order WHERE Order.CustomerId = Customer.Id
)
```

Result

Id	FirstName	LastName
1	Ozgur	Ozturk

Purpose

EXISTS, IN and JOIN could sometime be used for the same result, however, they are not equals :

- EXISTS should be used to check if a value exist in another table
- IN should be used for static list
- JOIN should be used to retrieve data from other(s) table(s)

JOIN

JOIN is a method of combining (joining) information from two tables. The result is a stitched set of columns from both tables, defined by the join type (INNER/OUTER/CROSS and LEFT/RIGHT/FULL, explained below) and join criteria (how rows from both tables relate).

A table may be joined to itself or to any other table. If information from more than two tables needs to be accessed, multiple joins can be specified in a FROM clause.

Self Join

A table may be joined to itself, with different rows matching each other by some condition. In this use case, aliases must be used in order to distinguish the two occurrences of the table.

In the below example, for each Employee in the example database Employees table, a record is returned containing the employee's first name together with the corresponding first name of the employee's manager. Since managers are also employees, the table is joined with itself:

```
SELECT
    e.FName AS "Employee",
    m.FName AS "Manager"
FROM
    Employees e
JOIN
    Employees m
    ON e.ManagerId = m.Id
```

This query will return the following data:

Employee	Manager
John	James
Michael	James
Johnathon	John

So how does this work?

The original table contains these records:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
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

The first action is to create a *Cartesian* product of all records in the tables used in the **FROM** clause. In this case it's the Employees table twice, so the intermediate table will look like this (I've removed any fields not used in this example):

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
1	James	NULL	1	James	NULL
1	Jame s	NULL	2	John	1
1	James	NULL	3	Michael	1
1	James	NULL	4	Johnathon	2
2	John	1	1	James	NULL
2	John	1	2	John	1
2	John	1	3	Michael	1
2	John	1	4	Johnathon	2
3	Michael	1	1	James	NULL
3	Michael	1	2	John	1
3	Michael	1	3	Michael	1
3	Michael	1	4	Johnathon	2
4	Johnathon	2	1	James	NULL
4	Johnathon	2	2	John	1
4	Johnathon	2	3	Michael	1
4	Johnathon	2	4	Johnathon	2

The next action is to only keep the records that meet the **JOIN** criteria, so any records where the aliased `e` table `ManagerId` equals the aliased `m` table `Id`:

e.Id	e.FName	e.ManagerId	m.Id	m.FName	m.ManagerId
2	John	1	1	James	NULL
3	Michael	1	1	James	NULL
4	Johnathon	2	2	John	1

Then, each expression used within the **SELECT** clause is evaluated to return this table:

e.FName	m.FName
John	James
Michael	James
Johnathon	John

`FName` and m. `FName`

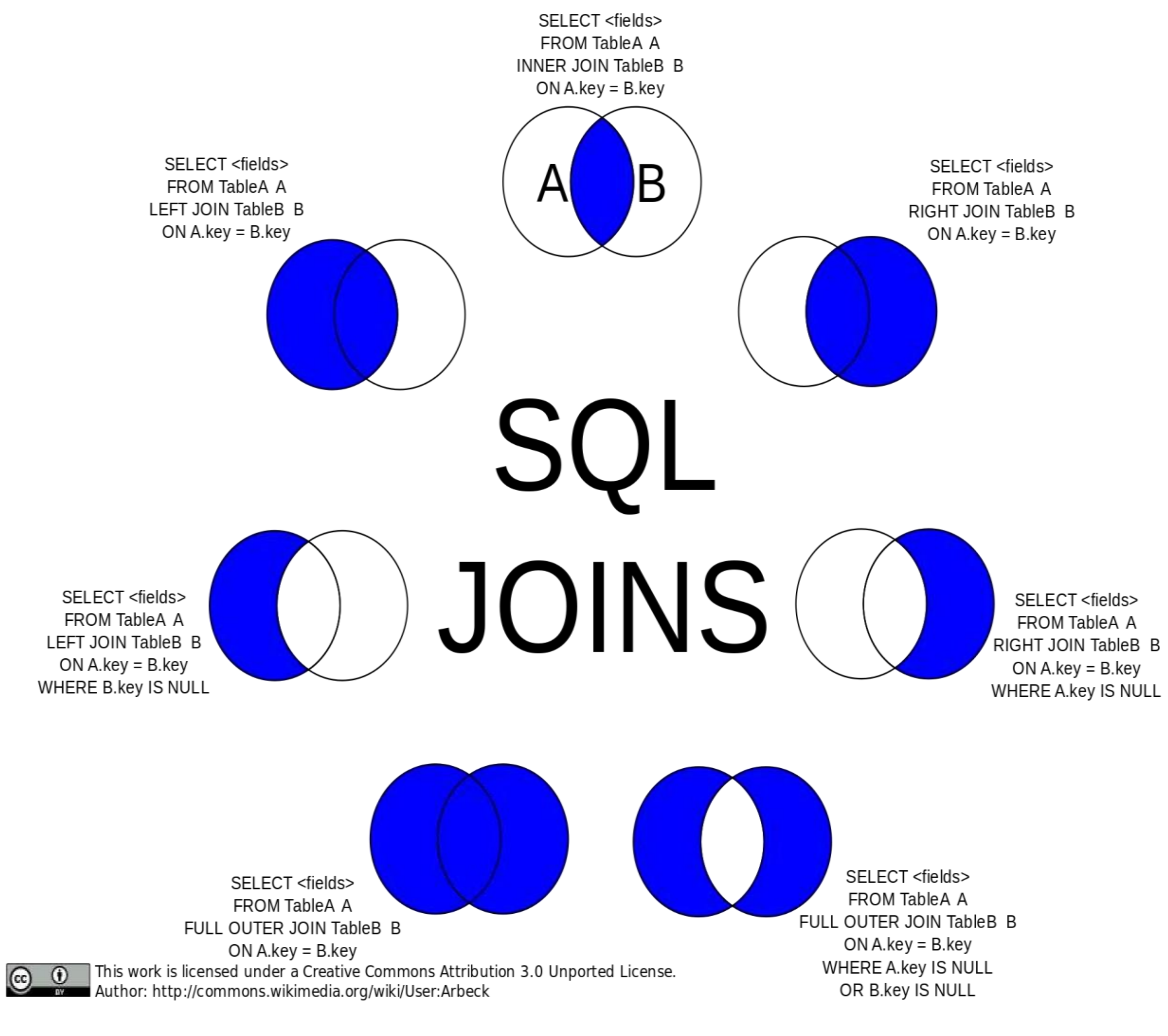
Finally, column names e. are replaced by their alias column names, assigned with the AS operator:

Employee	**Manager**
John	James
Michael	James
Johnathon	John

INNER JOIN

Di erences between inner/outer joins

SQL has various join types to specify whether (non-)matching rows are included in the result: , LEFT OUTER JOIN , RIGHT OUTER JOIN , and FULL OUTER JOIN (the INNER and OUTER keywords are optional). The figure below underlines the differences between these types of joins: the blue area represents the results returned by the join, and the white area represents the results that the join will not return.



```
SELECT <fields>
FROM TableA A
INNER JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
RIGHT 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
```

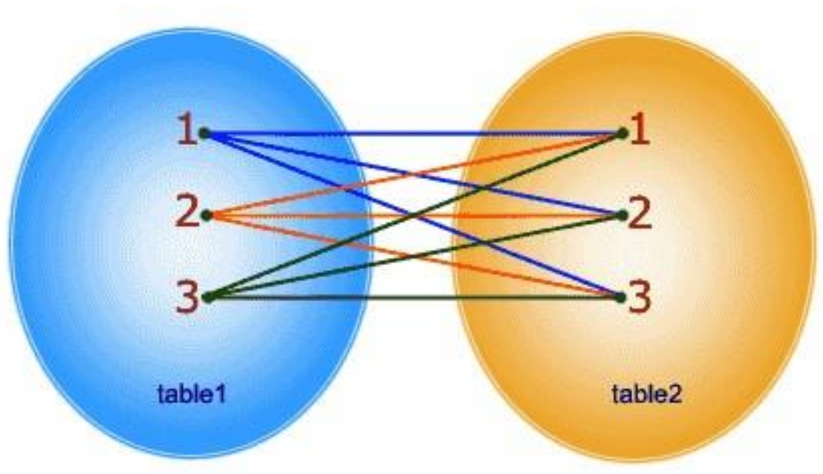
```
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
```

```
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

Cross Join SQL Pictorial Presentation (reference) :

SELECT * FROM table1 CROSS JOIN table2;



In CROSS JOIN, each row from 1st table joins with all the rows of another table.
If 1st table contain x rows and y rows in 2nd one the result set will be x * y rows.

Below are examples from this answer.

For instance there are two tables as below :

```
A    B
-    -
1    3
2    4
3    5
4    6
```

Note that (1,2) are unique to A, (3,4) are common, and (5,6) are unique to B.

Inner Join

An inner join using either of the equivalent queries gives the intersection of the two tables, i.e. the two rows they have in common:

```
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
```

Left outer join

A left outer join will give all rows in A, plus any common rows in B:

```
select * from a LEFT OUTER JOIN b on a.a = b.b;
```

```

a |  b
--+-----
1 | null
2 | null
3 |    3
4 |    4
```

Right outer join

Similarly, a right outer join will give all rows in B, plus any common rows in A:

```
select * from a RIGHT OUTER JOIN b on a.a = b.b;
```

```

a    |  b
-----+----
3    |  3
4    |  4
null |  5
null |  6
```

Full outer join

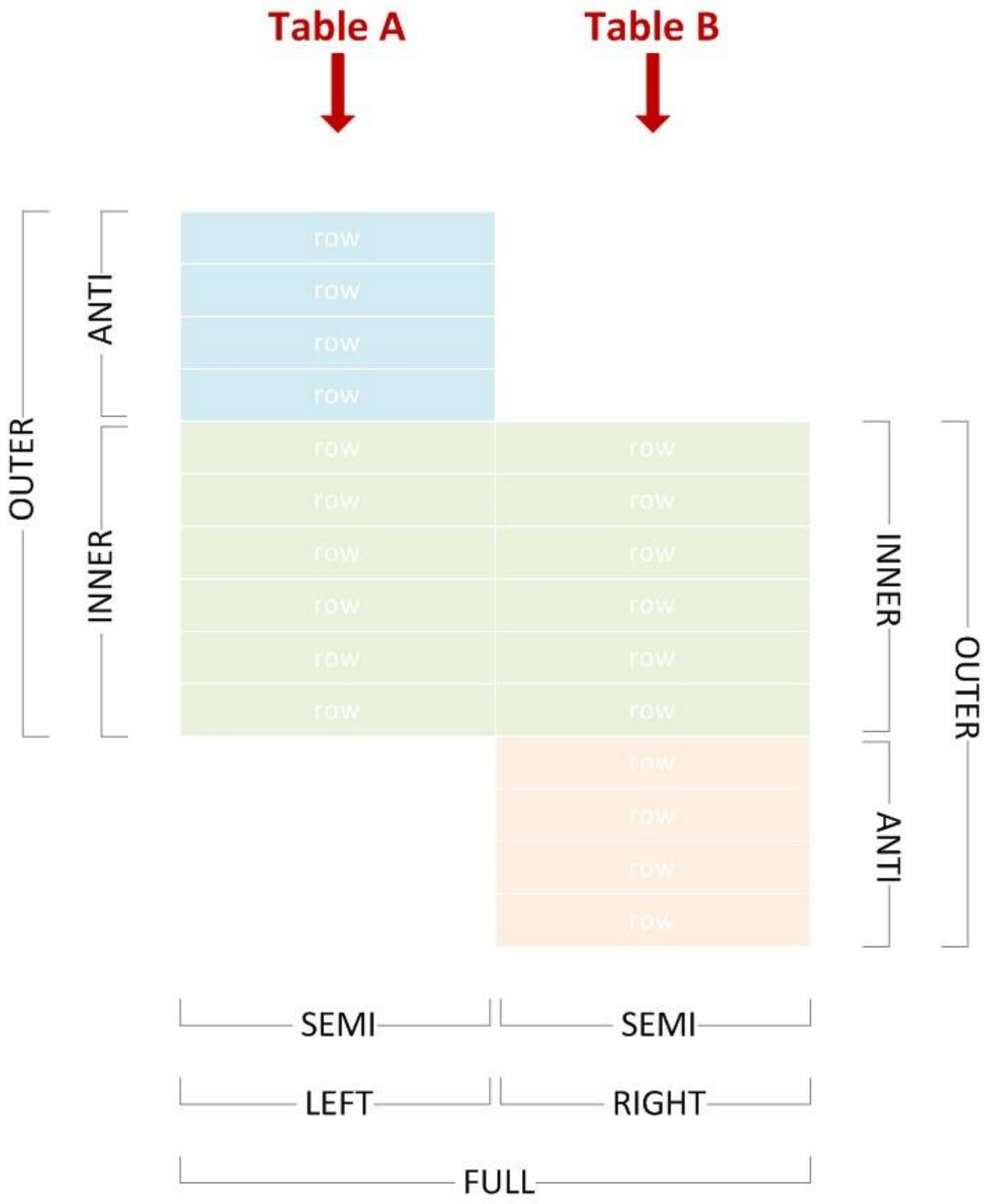
A full outer join will give you the union of A and B, i.e., all the rows in A and all the rows in B. If something in A doesn't have a corresponding datum in B, then the B portion is null, and vice versa.

```
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 Terminology: Inner, Outer, Semi, Anti..

Let's say we have two tables (A and B) and some of their rows match (relative to the given JOIN condition, whatever it may be in the particular case):



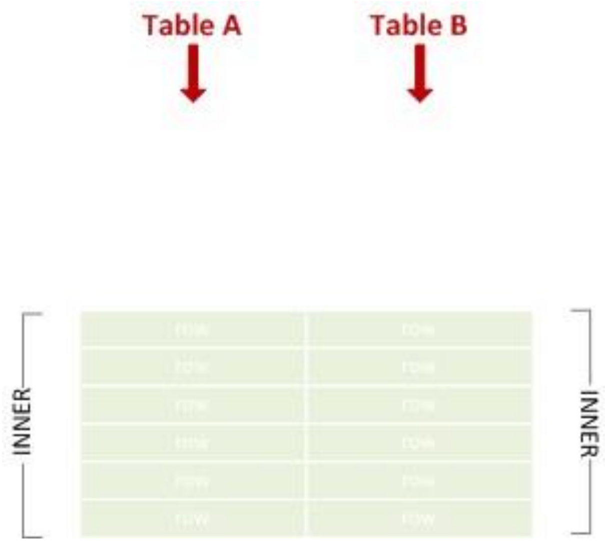
We can use various join types to include or exclude matching or non-matching rows from either side, and correctly name the join by picking the corresponding terms from the diagram above.

The examples below use the following test data:

```
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');
```

Inner Join

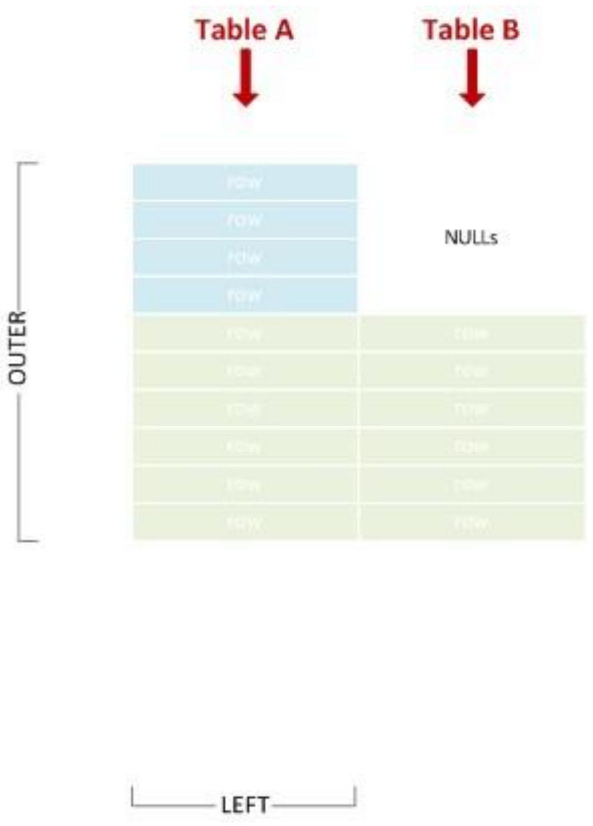
Combines left and right rows that match.



```
SELECT * FROM A JOIN B ON X = Y;

X        Y
------  -----
Lisa     Lisa
Marco   Marco
Phil     Phil
```

Left Outer Join

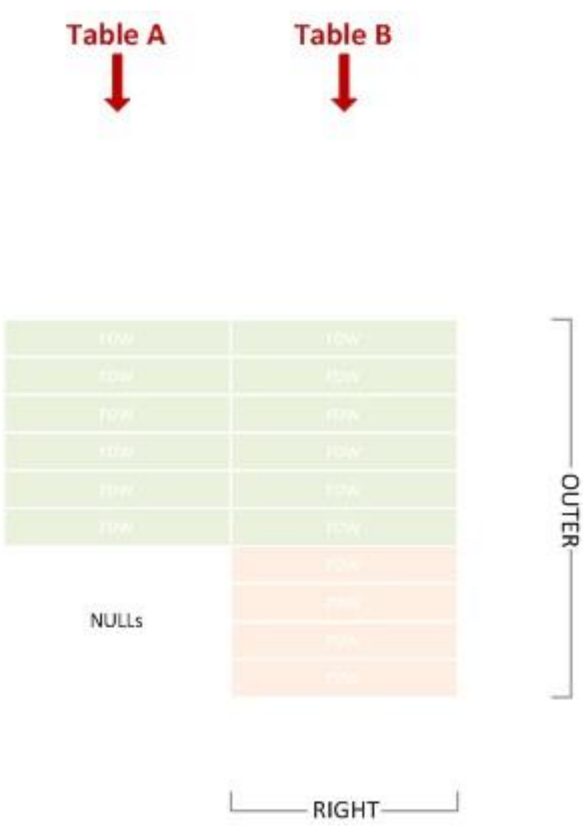
Sometimes abbreviated to "left join". Combines left and right rows that match, and includes non-matching left rows.



```
SELECT * FROM A LEFT JOIN B ON X = Y;

X        Y
-----    -----
Amy      NULL
John     NULL
Lisa     Lisa
Marco    Marco
Phil     Phil
```

Right Outer Join

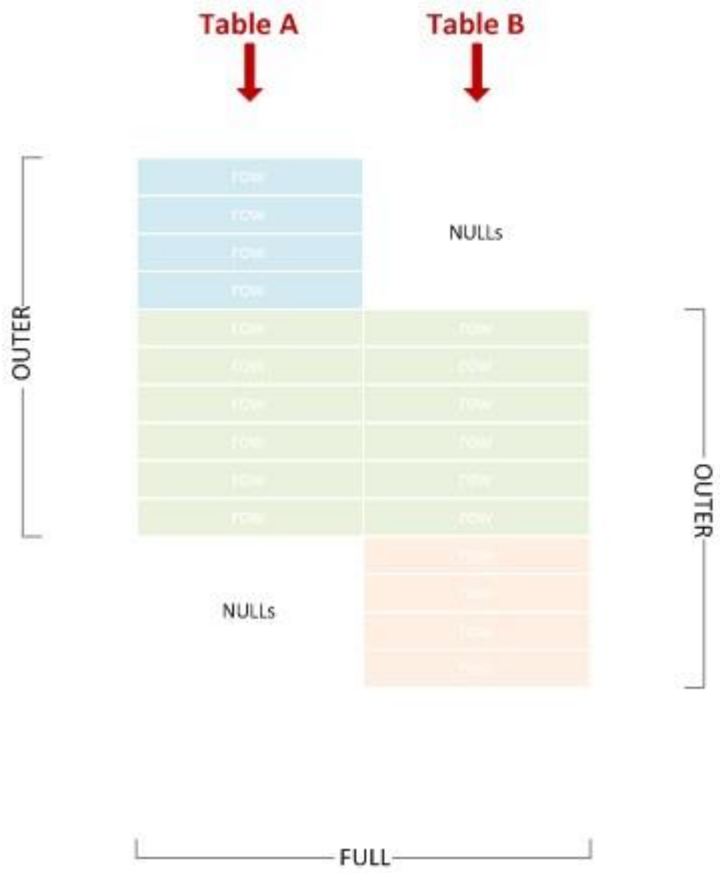
Sometimes abbreviated to "right join". Combines left and right rows that match, and includes non-matching right rows.



```
SELECT * FROM A RIGHT JOIN B ON X = Y;

X          Y
-----      -------
Lisa       Lisa
Marco      Marco
Phil       Phil
NULL       Tim
NULL       Vincent
```

Full Outer Join

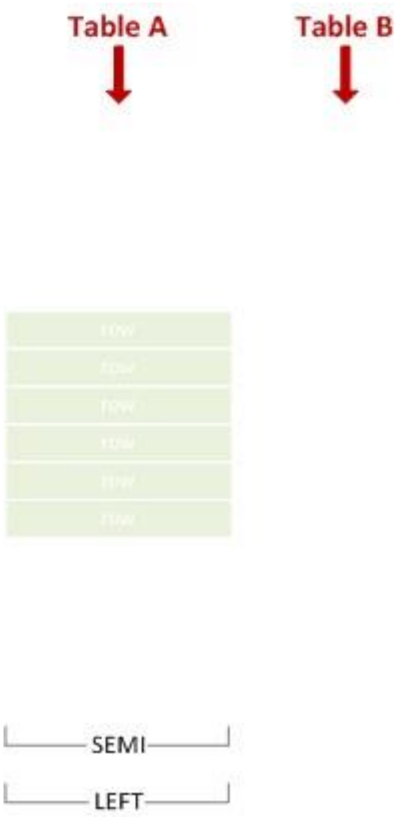
Sometimes abbreviated to "full join". Union of left and right outer join.



```
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
```

Left Semi Join

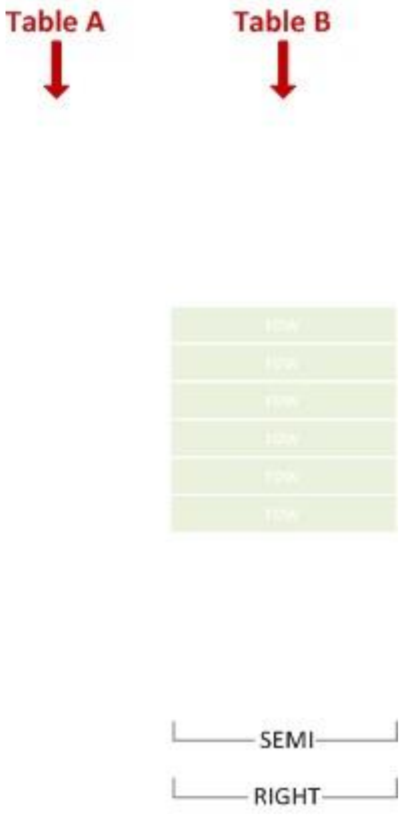
Includes left rows that match right rows.



```
SELECT * FROM A WHERE X IN (SELECT Y FROM B);

X
-----
Lisa
Marco
Phil
```

Right Semi Join

Includes right rows that match left rows.

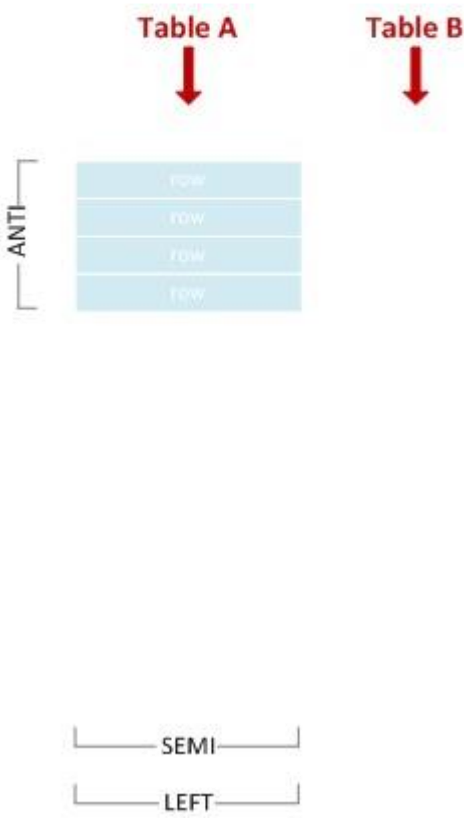


```
SELECT * FROM B WHERE Y IN (SELECT X FROM A);

Y
-----
Lisa
Marco
Phil
```

As you can see, there is no dedicated IN syntax for left vs. right semi join - we achieve the effect simply by switching the table positions within SQL text.

Left Anti Semi Join

Includes left rows that do **not** match right rows.

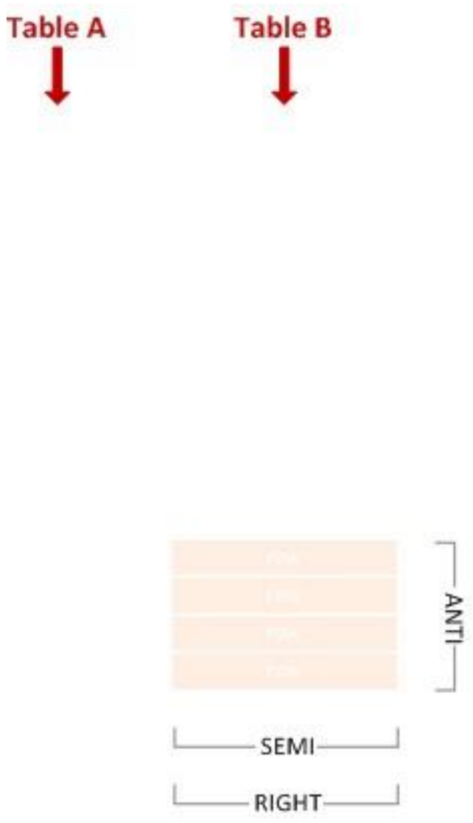


```
SELECT * FROM A WHERE X NOT IN (SELECT Y FROM B);

X
----
Amy
John
```

WARNING: Be careful if you happen to be using NOT IN on a NULL-able column! More details here.

Right Anti Semi Join

Includes right rows that do **not** match left rows.



```
SELECT * FROM B WHERE Y NOT IN (SELECT X FROM A);

Y
-------
Tim
Vincent
```

As you can see, there is no dedicated NOT IN syntax for left vs. right anti semi join - we achieve the effect simply by switching the table positions within SQL text.

Cross Join

A Cartesian product of all left with all right rows.

```
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
```

Cross join is equivalent to an inner join with join condition which always matches, so the following query would have returned the same result:

```
SELECT * FROM A JOIN B ON 1 = 1;
```

Self-Join

This simply denotes a table joining with itself. A self-join can be any of the join types discussed above. For example, this is a an inner self-join:

```
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
```

Left Outer Join

A Left Outer Join (also known as a Left Join or Outer Join) is a Join that ensures all rows from the left table are represented; if no matching row from the right table exists, its corresponding fields are NULL.

The following example will select all departments and the first name of employees that work in that department. Departments with no employees are still returned in the results, but will have NULL for the employee name:

```
SELECT          Departments.Name,  Employees.FName
FROM            Departments
LEFT OUTER JOIN Employees
ON              Departments.Id = Employees.DepartmentId
```

This would return the following from the example database:

Departments.Name	Employees.FName
HR	James
HR	John
HR	Johnathon
Sales	Michael
Tech	NULL

So how does this work?

There are two tables in the FROM clause:

Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
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

and

Id	Name
1	HR
2	Sales
3	Tech

First a *Cartesian* product is created from the two tables giving an intermediate table.
The records that meet the join criteria (*Departments.Id = Employees.DepartmentId*) are highlighted in bold; these are passed to the next stage of the query.

As this is a LEFT OUTER JOIN all records are returned from the LEFT side of the join (Departments), while any records on the RIGHT side are given a NULL marker if they do not match the join criteria. In the table below this will return **Tech** with NULL

Id	Name	Id	FName	LName	PhoneNumber	ManagerId	DepartmentId	Salary	HireDate
1	**HR**	**1**	**James**	**Smith**	**1234567890**	**NULL**	**1**	**1000**	**01-01-2002**
1	**HR**	**2**	**John**	**Johnson**	**2468101214**	**1**	**1**	**400**	**23-03-2005**
1	HR	3	Michael	Williams	1357911131	1	2	600	12-05-2009
1	**HR**	**4**	**Johnathon**	**Smith**	**1212121212**	**2**	**1**	**500**	**24-07-2016**
2	Sales	1	James	Smith	1234567890	NULL	1	1000	01-01-2002
2	Sales	2	John	Johnson	2468101214	1	1	400	23-03-2005
2	**Sales**	**3**	**Michael**	**Williams**	**1357911131**	**1**	**2**	**600**	**12-05-2009**
2	Sales	4	Johnathon	Smith	1212121212	2	1	500	24-07-2016
3	Tech	1	James	Smith	1234567890	NULL	1	1000	01-01-2002
3	Tech	2	John	Johnson	2468101214	1	1	400	23-03-2005
3	Tech	3	Michael	Williams	1357911131	1	2	600	12-05-2009

3 Tech 4 Johnathon Smith 1212121212 2 1 500 24-07-2016

Finally each expression used within the **SELECT** clause is evaluated to return our final table:

Departments.Name	**Employees.FName**
HR	James
HR	John
Sales	Richard
Tech	NULL

Implicit Join

Joins can also be performed by having several tables in the from clause, separated with commas , and defining the relationship between them in the where clause. This technique is called an Implicit Join (since it doesn't actually contain a join clause).
All RDBMSs support it, but the syntax is usually advised against. The reasons why it is a bad idea to use this syntax are:

- It is possible to get accidental cross joins which then return incorrect results, especially if you have a lot of joins in the query.
- If you intended a cross join, then it is not clear from the syntax (write out CROSS JOIN instead), and someone is likely to change it during maintenance.

The following example will select employee's first names and the name of the departments they work for:

```
SELECT  e.FName, d.Name
FROM    Employee e, Departments d
WHERE   e.DeptartmentId = d.Id
```

This would return the following from the example database:

e.FName	**d.Name**
James	HR
John	HR
Richard	Sales

CROSS JOIN

Cross join does a Cartesian product of the two members, A Cartesian product means each row of one table is combined with each row of the second table in the join. For example, if TABLEA has 20 rows and TABLEB has 20 rows, the result would be output rows. 20 * 20 = 400

Using example database

```
SELECT  d.Name, e.FName
FROM    Departments d
CROSS JOIN Employees e;
```

Which returns:

d.Name	**e.FName**
HR	James
HR	John
HR	Michael
HR	Johnathon

Sales	James
Sales	John
Sales	Michael
Sales	Johnathon
Tech	James
Tech	John
Tech	Michael
Tech	Johnathon

It is recommended to write an explicit CROSS JOIN if you want to do a cartesian join, to highlight that this is what you want.

CROSS APPLY & LATERAL JOIN

A very interesting type of JOIN is the LATERAL JOIN (new in PostgreSQL 9.3+), which is also known as CROSS APPLY/OUTER APPLY in SQL-Server & Oracle.

The basic idea is that a table-valued function (or inline subquery) gets applied for every row you join.

This makes it possible to, for example, only join the first matching entry in another table.
The difference between a normal and a lateral join lies in the fact that you can use a column that you previously joined **in the subquery** that you "CROSS APPLY".

Syntax:

PostgreSQL 9.3+

```
left | right | inner JOIN LATERAL
```

SQL-Server:

```
CROSS | OUTER APPLY
```

LEFT JOIN LATERAL is the same as OUTER APPLY
INNER JOIN LATERAL is the same as CROSS APPLY
and

Example usage (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
```

And for 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
```

FULL JOIN

One type of JOIN that is less known, is the FULL JOIN.
(Note: FULL JOIN is not supported by MySQL as per 2016)

A FULL OUTER JOIN returns all rows from the left table, and all rows from the right table.

If there are rows in the left table that do not have matches in the right table, or if there are rows in right table that do not have matches in the left table, then those rows will be listed, too.

Example 1 :

```
SELECT * FROM Table1

FULL JOIN Table2
     ON 1 = 2
```

Example 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
```

Note that if you're using soft-deletes, you'll have to check the soft-delete status again in the WHERE-clause (because FULL JOIN behaves kind-of like a UNION);
It's easy to overlook this little fact, since you put AP_SoftDeleteStatus = 1 in the join clause.

Also, if you are doing a FULL JOIN, you'll usually have to allow NULL in the WHERE-clause; forgetting to allow NULL on a value will have the same effects as an INNER join, which is something you don't want if you're doing a FULL JOIN.

Example:

```
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)
```

Recursive JOINs

Recursive joins are often used to obtain parent-child data. In SQL, they are implemented with recursive common table expressions, for example:

```
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;
```

Basic explicit inner join

A basic join (also called "inner join") queries data from two tables, with their relationship defined in a `join` clause.

The following example will select employees' first names (FName) from the Employees table and the name of the department they work for (Name) from the Departments table:

```
 ON Employees.DepartmentId = Departments.Id
```

This would return the following from the example database:

Employees.FName Departments.Name

```
SELECT Employees.FName, Departments.Name
FROM   Employees
JOIN   Departments
```

James	HR
John	HR
Richard	Sales

Joining on a Subquery

Joining a subquery is often used when you want to get aggregate data from a child/details table and display that along with records from the parent/header table. For example, you might want to get a count of child records, an average of some numeric column in child records, or the top or bottom row based on a date or numeric field. This example uses aliases, which arguable makes queries easier to read when you have multiple tables involved. Here's what a fairly typical subquery join looks like. In this case we are retrieving all rows from the parent table Purchase Orders and retrieving only the first row for each parent record of the child table PurchaseOrderLineItems.

```
SELECT po.Id, po.PODate, po.VendorName, po.Status, item.ItemNo,
  item.Description, item.Cost, item.Price
FROM PurchaseOrders po
LEFT JOIN
     (
       SELECT  .PurchaseOrderId,  .ItemNo,  .Description,  .Cost,  .Price, Min(l.id) as Id
       FROM PurchaseOrderLineItems l
       GROUP BY  .PurchaseOrderId,  .ItemNo,  .Description,  .Cost,  .Price
     ) AS item ON item.PurchaseOrderId = po.Id
```

UPDATE with data from another table

The examples below fill in a PhoneNumber for any Employee who is also a Customer and currently does not have a phone number set in the Employees Table.

(These examples use the Employees and Customers tables from the Example Databases.) **Standard SQL**

Update using a correlated subquery:

```
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

Update using 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

Update using : 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
```

Modifying existing values

This example uses the Cars Table from the Example Databases.

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

Update operations can include current values in the updated row. In this simple example the TotalCost is incremented by 100 for two rows:

-
- The TotalCost of Car #3 is increased from 100 to 200
The TotalCost of Car #4 is increased from 1254 to 1354

A column's new value may be derived from its previous value or from any other column's value in the same table or a joined table.

Updating Specified Rows

This example uses the Cars Table from the Example Databases.

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

This statement will set the status of the row of 'Cars' with id 4 to "READY".

WHERE clause contains a logical expression which is evaluated for each row. If a row fulfills the criteria, its value is updated. Otherwise, a row remains unchanged.

Updating All Rows

This example uses the Cars Table from the Example Databases.

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

This statement will set the 'status' column of all rows of the 'Cars' table to "READY" because it does not have a WHERE clause to filter the set of rows.

Capturing Updated records

Sometimes one wants to capture the records that have just been updated.

```
CREATE TABLE #TempUpdated(ID INT)

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

CREATE Database

CREATE Database

A database is created with the following SQL command:

```
CREATE  DATABASE  myDatabase;
```

This would create an empty database named myDatabase where you can create tables.

CREATE TABLE

Parameter **Details** tableName The name of the table

Contains an 'enumeration' of all the columns that the table have. See **Create a New Table** for more columns details.

The CREATE TABLE statement is used create a new table in the database. A table definition consists of a list of columns, their types, and any integrity constraints.

Create Table From Select

You may want to create a duplicate of a table:

```
CREATE TABLE ClonedEmployees AS SELECT * FROM Employees;
```

You can use any of the other features of a SELECT statement to modify the data before passing it to the new table. The columns of the new table are automatically created according to the selected rows.

```
CREATE TABLE ModifiedEmployees AS
SELECT Id, CONCAT(FName," ",LName) AS FullName FROM Employees
WHERE Id > 10;
```

Create a New Table

A basic Employees table, containing an ID, and the employee's first and last name along with their phone number can be created using

```
CREATE TABLE Employees(
     Idint identity(1,1) primary key not null,
    FName varchar(20) not null,
    LName varchar(20) not null,
    PhoneNumber varchar(10) not null
);
```

This example is specific to Transact-SQL

CREATE TABLE creates a new table in the database, followed by the table name, Employees

This is then followed by the list of column names and their properties, such as the ID

```
Id int identity(1,1) not null
```

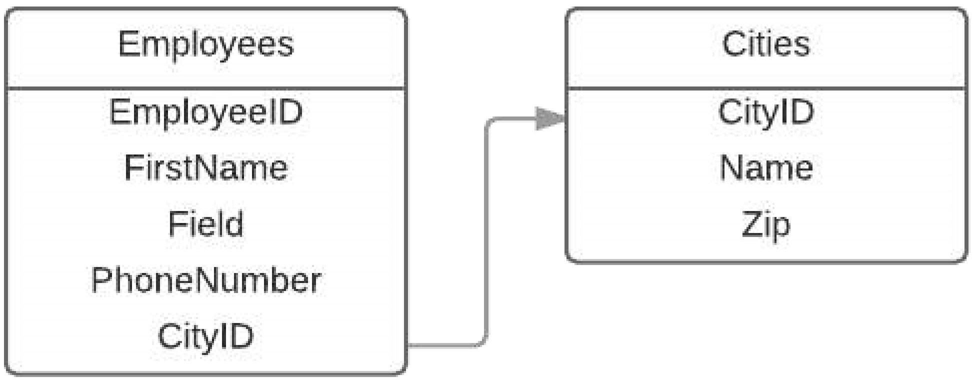
Value	Meaning
Id	the column's name. int is the data type.
identity	states that column will have auto generated values starting at 1 and incrementing by 1 for each (1,1) new row.
primary key	states that all values in this column will have unique values
not null	states that this column cannot have null values

CREATE TABLE With FOREIGN KEY

Below you could find the table Employees with a reference to the table 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)
);
```

Here could you find a database diagram.



The column CityID of table Employees will reference to the column CityID of table Cities. Below you could find the syntax to make this.

```
CityID INT FOREIGN KEY REFERENCES Cities(CityID)
```

Value	Meaning
CityID	Name of the column
int	type of the column
FOREIGN KEY	Makes the foreign key *(optional)*
REFERENCES	Makes the reference
Cities(CityID)	to the table Cities column CityID

Important: You couldn't make a reference to a table that not exists in the database. Be source to make first the table Cities and second the table Employees. If you do it vise versa, it will throw an error.

Duplicate a table

To duplicate a table, simply do the following:

```
CREATE TABLE newtable LIKE oldtable;
INSERT newtable SELECT * FROM oldtable;
```

Create a Temporary or In-Memory Table

PostgreSQL and SQLite

To create a temporary table local to the session:

```
CREATE TEMP TABLE MyTable(...);
```

SQL Server

To create a temporary table local to the session:

```
CREATE TABLE #TempPhysical(...);
```

To create a temporary table visible to everyone:

```
CREATE TABLE ##TempPhysicalVisibleToEveryone(...);
```

To create an in-memory table:

```
DECLARE @TempMemory TABLE(...);
```

CREATE FUNCTION

Argument **Description**

function_name the name of function list_of_paramenters parameters

that function accepts return_data_type type that function returs. Some

SQL data type function_body the code of function

scalar_expression scalar value returned by function

Create a new Function

```
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
```

This example creates a function named **FirstWord**, that accepts a varchar parameter and returns another varchar value.

TRY/CATCH

Transaction In a TRY/CATCH

This will rollback both inserts due to an invalid 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
```

This will commit both inserts:

```
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
```

UNION / UNION ALL

UNION keyword in SQL is used to combine to ***SELECT*** statement results with out any duplicate. In order to use UNION and combine results both SELECT statement should have same number of column with same data type in same order, but the length of column can be different.

Basic UNION ALL query

```
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)
);
```

Let's say we want to extract the names of all the managers from our departments.

Using a UNION we can get all the employees from both HR and Finance departments, which hold the position of a manager

```
SELECT
      FirstName, LastName
FROM
      HR_EMPLOYEES
WHERE
      Position = 'manager'
UNION ALL
SELECT
      FirstName, LastName
FROM
      FINANCE_EMPLOYEES
WHERE
      Position = 'manager'
```

The UNION statement removes duplicate rows from the query results. Since it is possible to have people having the same Name and position in both departments we are using , in order not to remove UNION ALL duplicates.

If you want to use an alias for each output column, you can just put them in the first select statement, as follows:

```
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'
```

Simple explanation and Example

In simple terms:

- UNION joins 2 result sets while removing duplicates from the result set joins 2
- UNION ALL result sets without attempting to remove duplicates

> One mistake many people make is to use a UNION when they do not need to have the duplicates removed. The additional performance cost against large results sets can be very significant.

When you might need UNION

Suppose you need to filter a table against 2 different attributes, and you have created separate non-clustered indexes for each column. A UNION enables you to leverage both indexes while still preventing duplicates.

```
SELECT C1, C2, C3 FROM Table1 WHERE C1 = @Param1
UNION
SELECT C1, C2, C3 FROM Table1 WHERE C2 = @Param2
```

This simplifies your performance tuning since only simple indexes are needed to perform these queries optimally. You may even be able to get by with quite a bit fewer non-clustered indexes improving overall write performance against the source table as well.

When you might need UNION ALL

Suppose you still need to filter a table against 2 attributes, but you do not need to filter duplicate records (either because it doesn't matter or your data wouldn't produce any duplicates during the union due to your data model design).

```
SELECT C1 FROM Table1
UNION ALL
SELECT C1 FROM Table2
```

This is especially useful when creating Views that join data that is designed to be physically partitioned across multiple tables (maybe for performance reasons, but still wants to roll-up records). Since the data is already split, having the database engine remove duplicates adds no value and just adds additional processing time to the queries.

ALTER TABLE

ALTER command in SQL is used to modify column/constraint in a table

Add Column(s)

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

The above statement would add columns named StartingDate which cannot be NULL with default value as current date and DateOfBirth which can be NULL in Employees table.

Drop Column

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

This will not only delete information from that column, but will drop the column salary from table employees(the column will no more exist).

Add Primary Key

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

This will add a Primary key to the table Employees on the field ID. Including more than one column name in the parentheses along with ID will create a Composite Primary Key. When adding more than one column, the column names must be separated by commas.

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

Alter Column

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

This query will alter the column datatype of StartingDate and change it from simple date to datetime and set default to current date.

Drop Constraint

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

This Drops a constraint called DefaultSalary from the employees table definition.

Note: Ensure that constraints of the column are dropped before dropping a column.

INSERT

INSERT data from another table using SELECT

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

This example will insert all Employees into the Customers table. Since the two tables have different fields and you don't want to move all the fields over, you need to set which fields to insert into and which fields to select. The correlating field names don't need to be called the same thing, but then need to be the same data type. This example is assuming that the Id field has an Identity Specification set and will auto increment.

If you have two tables that have exactly the same field names and just want to move all the records over you can use:

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

Insert New Row

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

This statement will insert a new row into the Customers table. Note that a value was not specified for the Id column, as it will be added automatically. However, all other column values must be specified.

Insert Only Specified Columns

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

This statement will insert a new row into the Customers table. Data will only be inserted into the columns specified note that no value was provided for the PhoneNumber column. Note, however, that all columns marked as must be included. not null

Insert multiple rows at once

Multiple rows can be inserted with a single insert command:

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

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

For inserting large quantities of data (bulk insert) at the same time, DBMS-specific features and recommendations exist.

MySQL - LOAD DATA INFILE

MSSQL - BULK INSERT

MERGE

MERGE (often also called UPSERT for "update or insert") allows to insert new rows or, if a row already exists, to update the existing row. The point is to perform the whole set of operations atomically (to guarantee that the data remain consistent), and to prevent communication overhead for multiple SQL statements in a client/server system.

MERGE to make 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
    ;
```

Note: The AND NOT EXISTS portion prevents updating records that haven't changed. Using the INTERSECT construct allows nullable columns to be compared without special handling.

MySQL: counting users by name

Suppose we want to know how many users have the same name. Let us create table users as follows:

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

Now, we just discovered a new user named Joe and would like to take him into account. To achieve that, we need to determine whether there is an existing row with his name, and if so, update it to increment count; on the other hand, if there is no existing row, we should create it.

MySQL uses the following syntax : insert … on duplicate key update …. In this case:

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

PostgreSQL: counting users by name

Suppose we want to know how many users have the same name. Let us create table users as follows:

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

Now, we just discovered a new user named Joe and would like to take him into account. To achieve that, we need to determine whether there is an existing row with his name, and if so, update it to increment count; on the other hand, if there is no existing row, we should create it.

PostgreSQL uses the following syntax : insert … on conflict … do update …. In this case:

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

cross apply, outer apply

CROSS APPLY and OUTER APPLY basics

Apply will be used when when table valued function in the right expression.

create a Department table to hold information about departments. Then create an Employee table which hold information about the employees. Please note, each employee belongs to a department, hence the Employee table has referential integrity with the Department table.

First query selects data from Department table and uses CROSS APPLY to evaluate the Employee table for each record of the Department table. Second query simply joins the Department table with the Employee table and all the matching records are produced.

```
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
```

If you look at the results they produced, it is the exact same result-set; How does it differ from a JOIN and how does it help in writing more efficient queries.

The first query in Script #2 selects data from Department table and uses OUTER APPLY to evaluate the Employee table for each record of the Department table. For those rows for which there is not a match in Employee table, those rows contains NULL values as you can see in case of row 5 and 6. The second query simply uses a LEFT OUTER JOIN between the Department table and the Employee table. As expected the query returns all rows from Department table; even for those rows for which there is no match in the Employee table.

```
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
```

Even though the above two queries return the same information, the execution plan will be bit different. But cost wise there will be not much difference.

Now comes the time to see where the APPLY operator is really required. In Script #3, I am creating a table-valued function which accepts DepartmentID as its parameter and returns all the employees who belong to this department. The next query selects data from Department table and uses CROSS APPLY to join with the function we created. It passes the DepartmentID for each row from the outer table expression (in our case Department table) and evaluates the function for each row similar to a correlated subquery.

The next query uses the OUTER APPLY in place of CROSS APPLY and hence unlike CROSS APPLY which returned only correlated data, the OUTER APPLY returns non-correlated data as well, placing NULLs into the missing columns.

```
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
```

So now if you are wondering, can we use a simple join in place of the above queries? Then the answer is NO, if you replace CROSS/OUTER APPLY in the above queries with INNER JOIN/LEFT OUTER JOIN, specify ON clause (something as 1=1) and run the query, you will get "The multi-part identifier "D.DepartmentID" could not be bound." error. This is because with JOINs the execution context of outer query is different from the execution context of the function (or a derived table), and you can not bind a value/variable from the outer query to the function as a parameter. Hence the APPLY operator is required for such queries.

DELETE

The DELETE statement is used to delete records from a table.

DELETE all rows

Omitting a WHERE clause will delete all rows from a table.

```
DELETE FROM Employees
```

See TRUNCATE documentation for details on how TRUNCATE performance can be better because it ignores triggers and indexes and logs to just delete the data.

DELETE certain rows with WHERE

This will delete all rows that match the WHERE criteria.

```
DELETE FROM Employees
WHERE FName = 'John'
```

TRUNCATE clause

Use this to reset the table to the condition at which it was created. This deletes all rows and resets values such as auto-increment. It also doesn't log each individual row deletion.

```
TRUNCATE TABLE Employees
```

DELETE certain rows based upon comparisons with other tables

It is possible to DELETE data from a table if it matches (or mismatches) certain data in other tables.

Let's assume we want to DELETEdata from Source once its loaded into Target.

```
DELETE FROM Source
WHERE  EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
               FROM Target
               Where Source.ID = Target.ID )
```

Most common RDBMS implementations (e.g. MySQL, Oracle, PostgresSQL, Teradata) allow tables to be joined during DELETE allowing more complex comparison in a compact syntax.

Adding complexity to original scenario, let's assume Aggregate is built from Target once a day and does not contain the same ID but contains the same date. Let us also assume that we want to delete data from Source *only* after the aggregate is populated for the day.

On MySQL, Oracle and Teradata this can be done using:

```
DELETE FROM Source
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.Date = AggregateSchema.Aggregate.Date
```

In PostgreSQL use:

```
DELETE FROM Source
USING  TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
       AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

This essentially results in INNER JOINs between Source, Target and Aggregate. The deletion is performed on Source when the same IDs exist in Target AND date present in Target for those IDs also exists in Aggregate.

Same query may also be written (on MySQL, Oracle, Teradata) as:

```
DELETE Source
FROM  Source, TargetSchema.Target, AggregateSchema.Aggregate
WHERE  Source.ID = TargetSchema.Target.ID
    AND TargetSchema.Target.DataDate = AggregateSchema.Aggregate.AggDate
```

Explicit joins may be mentioned in Delete statements on some RDBMS implementations (e.g. Oracle, MySQL) but not supported on all platforms (e.g. Teradata does not support them)

Comparisons can be designed to check mismatch scenarios instead of matching ones with all syntax styles (observe NOT EXISTS below)

```
DELETE FROM Source
WHERE NOT EXISTS ( SELECT 1 -- specific value in SELECT doesn't matter
               FROM Target
               Where Source.ID = Target.ID )
```

TRUNCATE

The TRUNCATE statement deletes all data from a table. This is similar to DELETE with no filter, but, depending on the database software, has certain restrictions and optimizations.

Removing all rows from the Employee table

```sql
TRUNCATE TABLE Employee;
```

Using truncate table is often better then using DELETE TABLE as it ignores all the indexes and triggers and just removes everything.

Delete table is a row based operation this means that each row is deleted. Truncate table is a data page operation the entire data page is reallocated. If you have a table with a million rows it will be much faster to truncate the table than it would be to use a delete table statement.

Though we can delete specific Rows with DELETE, we cannot TRUNCATE specific rows, we can only TRUNCATE all the records at once. Deleting All rows and then inserting a new record will continue to add the Auto incremented Primary key value from the previously inserted value, where as in Truncate, the Auto Incremental primary key value will also get reset and starts from 1.

Note that when truncating table, **no foreign keys must be present**, otherwise you will get an error.

DROP Table

Check for existence before dropping

MySQL `Version ≥ 3.19`

```
DROP TABLE IF EXISTS MyTable;
```

PostgreSQL `Version ≥ 8.x`

```
DROP TABLE IF EXISTS MyTable;
```

SQL Server `Version ≥ 2005`

```
If Exists(Select * From Information_Schema.Tables
          Where Table_Schema = 'dbo'
            And Table_Name = 'MyTable')
  Drop Table dbo.MyTable
```

SQLite `Version ≥ 3.0`

```
DROP TABLE IF EXISTS MyTable;
```

Simple drop

```
Drop Table MyTable;
```

DROP or DELETE Database

DROP Database

Dropping the database is a simple one-liner statement. Drop database will delete the database, hence always ensure to have a backup of the database if required. Below is the command to drop Employees Database

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

Cascading Delete

ON DELETE CASCADE

Assume you have a application that administers rooms.
Assume further that your application operates on a per client basis (tenant).
You have several clients.
So your database will contain one table for clients, and one for rooms.

Now, every client has N rooms.

This should mean that you have a foreign key on your room table, referencing the client table.

```
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
```

Assuming a client moves on to some other software, you'll have to delete his data in your software. But if you do

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

Then you'll get a foreign key violation, because you can't delete the client when he still has rooms.

Now you'd have write code in your application that deletes the client's rooms before it deletes the client. Assume further that in the future, many more foreign key dependencies will be added in your database, because your application's functionality expands. Horrible. For every modification in your database, you'll have to adapt your application's code in N places. Possibly you'll have to adapt code in other applications as well (e.g. interfaces to other systems).

There is a better solution than doing it in your code.
You can just add `ON DELETE CASCADE` to your foreign key.

```
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 Now
```

you can say

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

and the rooms are automagically deleted when the client is deleted. Problem solved - with no application code changes.

One word of caution: In Microsoft SQL-Server, this won't work if you have a table that references itself. So if you try to define a delete cascade on a recursive tree structure, like this:

```
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
```

it won't work, because Microsoft-SQL-server doesn't allow you to set a foreign key with `ON DELETE CASCADE` on a recursive tree structure. One reason for this is, that the tree is possibly cyclic, and that would possibly lead to a deadlock.

PostgreSQL on the other hand can do this; the
requirement is that the tree is non-cyclic.
If the tree is cyclic, you'll get a runtime error.
In that case, you'll just have to implement the delete function yourselfs.

A word of caution:
This means you can't simply delete and re-insert the client table anymore, because if you do this, it will delete all entries in "T_Room"... (no non-delta updates anymore)

GRANT and REVOKE

Grant/revoke privileges

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

Grant User1 and User2 permission to perform SELECT and UPDATE operations on table Employees.

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

Revoke from User1 and User2 the permission to perform SELECT and UPDATE operations on table Employees.

XML

Query from XML Data Type

```
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)
```

Results

```
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
```

Primary Keys

Creating a Primary Key

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

This will create the Employees table with 'Id' as its primary key. The primary key can be used to uniquely identify the rows of a table. Only one primary key is allowed per table.

A key can also be composed by one or more fields, so called composite key, with the following syntax:

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

Using Auto Increment

Many databases allow to make the primary key value automatically increment when a new key is added. This ensures that every key is different.

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
);
```

Indexes

Indexes are a data structure that contains pointers to the contents of a table arranged in a specific order, to help the database optimize queries. They are similar to the index of book, where the pages (rows of the table) are indexed by their page number.

Several types of indexes exist, and can be created on a table. When an index exists on the columns used in a query's WHERE clause, JOIN clause, or ORDER BY clause, it can substantially improve query performance.

Sorted Index

If you use an index that is sorted the way you would retrieve it, the SELECT statement would not do additional sorting when in retrieval.

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

When you execute the query

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

The database system would not do additional sorting, since it can do an index-lookup in that order.

Partial or Filtered Index

SQL Server and SQLite allow to create indexes that contain not only a subset of columns, but also a subset of rows.

Consider a constant growing amount of orders with order_state_id equal to finished (2), and a stable amount of orders with order_state_id equal to started (1).

If your business make use of queries like this:

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

Partial indexing allows you to limit the index, including only the unfinished orders:

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

This index will be smaller than an unfiltered index, which saves space and reduces the cost of updating the index.

Creating an Index

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

This will create an index for the column *EmployeeId* in the table *Cars*. This index will improve the speed of queries asking the server to sort or select by values in *EmployeeId*, such as the following:

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

The index can contain more than 1 column, as in the following;

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

In this case, the index would be useful for queries asking to sort or select by all included columns, if the set of conditions is ordered in the same way. That means that when retrieving the data, it can find the rows to retrieve using the index, instead of looking through the full table.

For example, the following case would utilize the second index;

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

If the order differs, however, the index does not have the same advantages, as in the following;

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

The index is not as helpful because the database must retrieve the entire index, across all values of EmployeeId and CarID, in order to find which items have . OwnerId = 17

(The index may still be used; it may be the case that the query optimizer finds that retrieving the index and filtering on the OwnerId, then retrieving only the needed rows is faster than retrieving the full table, especially if the table is large.)

Dropping an Index, or Disabling and Rebuilding it

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

We can use command DROP to delete our index. In this example we will DROP the index called *ix_cars_employee_id* on the table *Cars*.

This deletes the index entirely, and if the index is clustered, will remove any clustering. It cannot be rebuilt without recreating the index, which can be slow and computationally expensive. As an alternative, the index can be disabled:

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

This allows the table to retain the structure, along with the metadata about the index.

Critically, this retains the index statistics, so that it is possible to easily evaluate the change. If warranted, the index can then later be rebuilt, instead of being recreated completely;

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

Clustered, Unique, and Sorted Indexes

Indexes can have several characteristics that can be set either at creation, or by altering existing indexes.

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

The above SQL statement creates a new clustered index on Employees. Clustered indexes are indexes that dictate the actual structure of the table; the table itself is sorted to match the structure of the index. That means there can be at most one clustered index on a table. If a clustered index already exists on the table, the above statement will fail. (Tables with no clustered indexes are also called heaps.)

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

This will create an unique index for the column *Email* in the table *Customers*. This index, along with speeding up queries like a normal index, will also force every email address in that column to be unique. If a row is inserted or updated with a non-unique *Email* value, the insertion or update will, by default, fail.

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

This creates an index on Customers which also creates a table constraint that the EmployeeID must be unique.
(This will fail if the column is not currently unique - in this case, if there are employees who share an ID.)

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

This creates an index that is sorted in descending order. By default, indexes (in MSSQL server, at least) are ascending, but that can be changed.

Rebuild index

Over the course of time B-Tree indexes may become fragmented because of updating/deleting/inserting data. In SQLServer terminology we can have internal (index page which is half empty) and external (logical page order doesn't correspond physical order). Rebuilding index is very similar to dropping and re-creating it.

We can re-build an index with

```
ALTER INDEX index_name REBUILD;
```

By default rebuilding index is offline operation which locks the table and prevents DML against it , but many RDBMS allow online rebuilding. Also, some DB vendors offer alternatives to index rebuilding such as REORGANIZE (SQLServer) or SHRINK SPACE COALESCE/(Oracle).

Inserting with a Unique Index

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

This will fail if an unique index is set on the *Email* column of *Customers*. However, alternate behavior can be defined for this case:

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

Row number

Delete All But Last Record (1 to Many Table)

```
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;
```

Row numbers without partitions

Include a row number according to the order specified.

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

Row numbers with partitions

Uses a partition criteria to group the row numbering according to it.

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

SQL Group By vs Distinct

GROUP BY

Di�erence between GROUP BY and DISTINCT

is used in combination with aggregation functions. Consider the following table:

orderId	userId	storeName	orderValue	orderDate
1	43	Store A	25	20-03-2016
2	57	Store B	50	22-03-2016

3	43	Store A	30	25-03-2016
4	82	Store C	10	26-03-2016
5	21	Store A	45	29-03-2016

GROUP BY

The query below uses to perform aggregated calculations.

```
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;
```

and will return the following information

storeName	total_nr_orders	nr_unique_customers	average_order_value	first_order	lastOrder
Store A	3	2	33.3	20-03-2016	29-03-2016
Store B	1	1	50	22-03-2016	22-03-2016
Store C	1	1	10	26-03-2016	26-03-2016

While DISTINCT is used to list a unique combination of distinct values for the specified columns.

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

storeName	userId
Store A	43
Store B	57
Store C	82
Store A	21

Finding Duplicates on a Column Subset with Detail

Students with same name and date of birth

```
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
```

This example uses a Common Table Expression and a Window Function to show all duplicate rows (on a subset of columns) side by side.

String Functions

String functions perform operations on string values and return either numeric or string values.

Using string functions, you can, for example, combine data, extract a substring, compare strings, or convert a string to all uppercase or lowercase characters.

Concatenate

In (standard ANSI/ISO) SQL, the operator for string concatenation is `||`. This syntax is supported by all major databases except SQL Server:

```
SELECT 'Hello' || 'World' || '!'; --returns HelloWorld!
```

Many databases support a `CONCAT` function to join strings:

```
SELECT CONCAT('Hello', 'World'); --returns 'HelloWorld'
```

Some databases support using `CONCAT` to join more than two strings (Oracle does not):

```
SELECT CONCAT('Hello', 'World', '!'); --returns 'HelloWorld!'
```

In some databases, non-string types must be cast or converted:

```
SELECT CONCAT('Foo', CAST(42 AS VARCHAR(5)), 'Bar'); --returns 'Foo42Bar'
```

Some databases (e.g., Oracle) perform implicit lossless conversions. For example, a `CONCAT` on a `CLOB` and `NCLOB` yields a `NCLOB`. A `CONCAT` on a number and a `varchar2` results in a `varchar2`, etc.:

```
SELECT CONCAT(CONCAT('Foo', 42), 'Bar') FROM dual; --returns Foo42Bar
```

Some databases can use the non-standard `+` operator (but in most, `+` works only for numbers):

```
SELECT 'Foo' + CAST(42 AS VARCHAR(5)) + 'Bar';
```

On SQL Server < 2012, where `CONCAT` is not supported, `+` is the only way to join strings.

Length

SQL Server

The LEN doesn't count the trailing space.

```
SELECT LEN('Hello') -- returns 5

SELECT LEN('Hello '); -- returns 5
```

The DATALENGTH counts the trailing space.

```
SELECT DATALENGTH('Hello') -- returns 5

SELECT DATALENGTH('Hello '); -- returns 6
```

It should be noted though, that DATALENGTH returns the length of the underlying byte representation of the string, which depends, i.a., on the charset used to store the string.

```
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
```

Oracle

Syntax: Length (char) Examples:

```
SELECT Length('Bible') FROM dual; --Returns 5
SELECT Length('righteousness') FROM dual; --Returns 13
SELECT Length(NULL) FROM dual; --Returns NULL
```

See Also: LengthB, LengthC, Length2, Length4

Trim empty spaces

Trim is used to remove write-   space at the beginning or end of selection

In MSSQL there is no single ()   TRIM

```
SELECT LTRIM('   Hello   ') --returns 'Hello   '
SELECT RTRIM('   Hello   ') --returns '   Hello'
SELECT LTRIM(RTRIM('   Hello   ')) --returns 'Hello'
```

MySql and Oracle

```
SELECT TRIM('   Hello   ') --returns 'Hello'
```

Upper & lower case

```
SELECT UPPER('HelloWorld') --returns 'HELLOWORLD'
SELECT LOWER('HelloWorld') --returns 'helloworld'
```

STRING_SPLIT

Split

Splits a string expression using a character separator. Note that () is a table-valued function.

```
SELECT value FROM STRING_SPLIT('Lorem ipsum dolor sit amet.', ' '); Result:
```

```
value
-----
Lorem
ipsum
dolor
sit
amet.
```

Section 41.6: Replace

Syntax:

REPLACE (String to search , String to search for and replace , String to place into the original string)

Example:

```
SELECT REPLACE( 'Peter Steve Tom', 'Steve', 'Billy' ) --Return Values: Peter Billy Tom
```

REGEXP

MySQL `Version ≥ 3.19`

Checks if a string matches a regular expression (defined by another string).

```
SELECT 'bedded' REGEXP '[a-f]' -- returns True

SELECT 'beam' REGEXP '[a-f]' -- returns False
```

SUBSTRING (string_expression , start , length

Substring

Syntax is:). Note that SQL strings are 1-indexed.

```
SELECT SUBSTRING('Hello', 1, 2) --returns 'He'
SELECT SUBSTRING('Hello', 3, 3) --returns 'llo'
```

This is often used in conjunction with the () LEN function to get the last n characters of a string of unknown length.

```
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'
```

Stu

Stuff a string into another, replacing 0 or more characters at a certain position.

Note: start position is 1-indexed (you start indexing at 1, not 0).

Syntax:

STUFF (character_expression , start , length , replaceWith_expression) Example:

```
SELECT STUFF('FooBarBaz', 4, 3, 'Hello') --returns 'FooHelloBaz'
```

LEFT - RIGHT

Syntax is:
LEFT (string-expression , integer)
RIGHT (string-expression , integer)

```
SELECT LEFT('Hello',2)  --return He
SELECT RIGHT('Hello',2) --return lo
```

Oracle SQL doesn't have LEFT and RIGHT functions. They can be emulated with SUBSTR and LENGTH.
SUBSTR (string-expression, 1, integer)
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
```

REVERSE

Syntax is: REVERSE (string-expression)

```
SELECT REVERSE('Hello') --returns olleH
```

REPLICATE

The REPLICATE function concatenates a string with itself a specified number of times.

Syntax is: REPLICATE (string-expression , integer)

```
SELECT REPLICATE ('Hello',4) --returns 'HelloHelloHelloHello'
```

Replace function in sql Select and Update query

The Replace function in SQL is used to update the content of a string. The function call is REPLACE( ) for MySQL, Oracle, and SQL Server.

The syntax of the Replace function is:

```
REPLACE (str, find, repl)
```

The following example replaces occurrences of South with Southern in Employees table:

FirstName	Address
James	South New York
John	South Boston Michael
	South San Diego

Select Statement :

If we apply the following Replace function:

```
SELECT
    FirstName,
    REPLACE (Address, 'South', 'Southern') Address
FROM Employees
ORDER BY FirstName
```

Result:

FirstName	Address
James	Southern New York
John	Southern Boston Michael
	Southern San Diego

Update Statement :

We can use a replace function to make permanent changes in our table through following approach.

```
Update Employees
Set city = (Address, 'South', 'Southern');
```

A more common approach is to use this in conjunction with a WHERE clause like this:

```
Update Employees
Set Address = (Address, 'South', 'Southern')
Where Address LIKE 'South%';
```

INSTR

Return the index of the first occurrence of a substring (zero if not found)

Syntax: INSTR (string, substring)

```
SELECT INSTR('FooBarBar', 'Bar') -- return 4
SELECT INSTR('FooBarBar', 'Xar') -- return 0
```

PARSENAME

DATABASE : SQL Server

PARSENAME function returns the specific part of given string(object name). object name may contains string like object name,owner name, database name and server name.

More details MSDN:PARSENAME

Syntax

```
PARSENAME('NameOfStringToParse',PartIndex)
```

Example

To get object name use part index 1

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',1)  // returns `ObjectName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',1)     // returns `Student`
```

To get schema name use part index 2

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',2)  // returns `SchemaName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',2)     // returns `school`
```

To get database name use part index 3

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',3) // returns `DatabaseName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',3)    // returns `SchoolDatabase`
```

To get server name use part index 4

```
SELECT PARSENAME('ServerName.DatabaseName.SchemaName.ObjectName',4)  // returns `ServerName`
SELECT PARSENAME('[1012-1111].SchoolDatabase.school.Student',4)     // returns `[1012-1111]`
```

PARSENAME will returns null is specified part is not present in given object name string

Functions (Aggregate)

Conditional aggregation

Payments Table

Customer	Payment_type	Amount
Peter	Credit	100
Peter	Credit	300

John Credit 1000 John Debit 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
```

Result:

Customer Credit Debit
Peter 400 0 John 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
```

Result:

Customer	credit_transaction_count	debit_transaction_count
Peter	2	0
John	1	1

List Concatenation

Partial credit to this SO answer.

List Concatenation aggregates a column or expression by combining the values into a single string for each group. A string to delimit each value (either blank or a comma when omitted) and the order of the values in the result can be specified. While it is not part of the SQL standard, every major relational database vendor supports it in their own way.

MySQL

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB ORDER BY ColumnB SEPARATOR ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

Oracle & 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 and earlier

(CTE included to encourage the DRY principle)

```
  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 and SQL Azure

```
SELECT ColumnA
     , STRING_AGG(ColumnB, ',') WITHIN GROUP (ORDER BY ColumnB) AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

SQLite without

ordering:

```
SELECT ColumnA
     , GROUP_CONCAT(ColumnB, ',') AS ColumnBs
  FROM TableName
 GROUP BY ColumnA
 ORDER BY ColumnA;
```

ordering requires a subquery or 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;
```

SUM

Sum function sum the value of all the rows in the group. If the group by clause is omitted then sums all the rows.

```
select sum(salary) TotalSalary
from employees;
```

TotalSalary

2500

```
select DepartmentId, sum(salary) TotalSalary
from employees
group by DepartmentId;
```

DepartmentId	TotalSalary
1	2000
2	500

AVG()

The aggregate function AVG() returns the average of a given expression, usually numeric values in a column. Assume we have a table containing the yearly calculation of population in cities across the world. The records for New York City look similar to the ones below:

EXAMPLE TABLE city_name

population year

New York City 8,550,405 2015 New

York City ... ...

New York City 8,000,906 2005

To select the average population of the New York City, USA from a table containing city names, population measurements, and measurement years for last ten years:

QUERY

```
select city_name, AVG(population) avg_population
from city_population
where city_name = 'NEW YORK CITY';
```

Notice how measurement year is absent from the query since population is being averaged over time.

RESULTS city_name

avg_population

New York City 8,250,754

> Note: The AVG() function will convert values to numeric types. This is especially important to keep in mind when working with dates.

Count

You can count the number of rows:

```
SELECT count(*) TotalRows
FROM employees;
```

TotalRows
4

Or count the employees per department:

```
SELECT DepartmentId, count(*) NumEmployees
FROM employees
GROUP BY DepartmentId;
```

DepartmentId	NumEmployees
1	3
2	1

You can count over a column/expression with the effect that will not count the `NULL` values:

```
SELECT count(ManagerId) mgr
FROM EMPLOYEES;
```

mgr
3

(There is one null value managerID column)

You can also use **DISTINCT** inside of another function such as **COUNT** to only find the **DISTINCT** members of the set to perform the operation on.

For example:

```
 SELECT COUNT(ContinentCode) AllCount
 ,      COUNT(DISTINCT ContinentCode) SingleCount
 FROM Countries;
```

Will return different values. The *SingleCount* will only Count individual Continents once, while the *AllCount* will include duplicates.

ContinentCode
OC
EU
AS
NA
NA
AF
AF

AllCount: 7 SingleCount: 5

Min

Find the smallest value of column:

```
select min(age) from employee;
```

Above example will return smallest value for column age of employee table.

Syntax:

```
SELECT MIN(column_name) FROM table_name;
```

Max

Find the maximum value of column:

```
select max(age) from employee;
```

Above example will return largest value for column age of employee table.

Syntax:

```
SELECT MAX(column_name) FROM table_name;
```

Functions (Scalar/Single Row)

SQL provides several built-in scalar functions. Each scalar function takes one value as input and returns one value as output for each row in a result set.

You use scalar functions wherever an expression is allowed within a T-SQL statement.

Date And Time

In SQL, you use date and time data types to store calendar information. These data types include the time, date, smalldatetime, datetime, datetime2, and datetimeoffset. Each data type has a specific format.

Data type **Format**

time hh:mm:ss[.nnnnnnn] date YYYY-MM-

DD

smalldatetime YYYY-MM-DD hh:mm:ss datetime YYYY-MM-DD

hh:mm:ss[.nnn] datetime2 YYYY-MM-DD hh:mm:ss[.nnnnnnn]

datetimeoffset YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+/-]hh:mm

The DATENAME function returns the name or value of a specific part of the date.

```sql
SELECT DATENAME (weekday,'2017-01-14') as Datename
```

Datename
Saturday

You use the GETDATE function to determine the current date and time of the computer running the current SQL instance. This function doesn't include the time zone difference.

```sql
SELECT GETDATE() as Systemdate
```

Systemdate
2017-01-14 11:11:47.7230728

The DATEDIFF function returns the difference between two dates.

In the syntax, datepart is the parameter that specifies which part of the date you want to use to calculate difference. The datepart can be year, month, week, day, hour, minute, second, or millisecond. You then specify the start date in the startdate parameter and the end date in the enddate parameter for which you want to find the difference.

```sql
SELECT SalesOrderID, DATEDIFF(day, OrderDate, ShipDate)
AS 'Processing time'
FROM Sales.SalesOrderHeader
```

SalesOrderID	Processing time
43659	7
43660	7
43661	7
43662	7

The DATEADD function enables you to add an interval to part of a specific date.

```sql
SELECT DATEADD (day, 20, '2017-01-14') AS Added20MoreDays
```

Added20MoreDays

2017-02-03 00:00:00.000

Character modifications

Character modifying functions include converting characters to upper or lower case characters, converting numbers to formatted numbers, performing character manipulation, etc.

The lower (char) function converts the given character parameter to be lower-cased characters.

```
SELECT customer_id, lower(customer_last_name) FROM customer;
```

would return the customer's last name changed from "SMITH" to "smith".

Configuration and Conversion Function

An example of a configuration function in SQL is the @@SERVERNAME function. This function provides the name of the local server that's running SQL.

```
SELECT @@SERVERNAME AS 'Server'
```

Server
SQL064

In SQL, most data conversions occur implicitly, without any user intervention.

To perform any conversions that can't be completed implicitly, you can use the CAST or CONVERT functions.

The CAST function syntax is simpler than the CONVERT function syntax, but is limited in what it can do.

In here, we use both the CAST and CONVERT functions to convert the datetime data type to the varchar data type.

The CAST function always uses the default style setting. For example, it will represent dates and times using the format YYYY-MM-DD.

The CONVERT function uses the date and time style you specify. In this case, 3 specifies the date format 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
```

```
                    Cast                                  Convert
David Hamilton was hired on 2003-02-04 David Hamiltion was hired on 04/02/03
```

Another example of a conversion function is the PARSE function. This function converts a string to a specified data type.

In the syntax for the function, you specify the string that must be converted, the AS keyword, and then the required data type. Optionally, you can also specify the culture in which the string value should be formatted. If you don't specify this, the language for the session is used.

If the string value can't be converted to a numeric, date, or time format, it will result in an error. You'll then need to use CAST or CONVERT for the conversion.

```
SELECT PARSE('Monday, 13 August 2012' AS datetime2 USING 'en-US') AS 'Date in English'
```

```
          Date in English
2012-08-13 00:00:00.0000000
```

Logical and Mathmetical Function

SQL has two logical functions – CHOOSE and IIF.

The CHOOSE function returns an item from a list of values, based on its position in the list. This position is specified by the index.

In the syntax, the index parameter specifies the item and is a whole number, or integer. The val_1 … val_n parameter identifies the list of values.

```
SELECT CHOOSE(2, 'Human Resources', 'Sales', 'Admin', 'Marketing' ) AS Result;
```

Result Sales

In this example, you use the CHOOSE function to return the second entry in a list of departments.

The IIF function returns one of two values, based on a particular condition. If the condition is true, it will return true value. Otherwise it will return a false value.

In the syntax, the boolean_expression parameter specifies the Boolean expression. The true_value parameter specifies the value that should be returned if the boolean_expression evaluates to true and the false_value parameter specifies the value that should be returned if the boolean_expression evaluates to false.

```
SELECT BusinessEntityID, SalesYTD,
    IIF(SalesYTD > 200000, 'Bonus', 'No  Bonus') AS 'Bonus?'
FROM Sales.SalesPerson
GO
```

```
BusinessEntityID        SalesYTD        Bonus?
274                   559697.5639  Bonus
275                   3763178.1787 Bonus
285                    172524.4512       No Bonus
```

In this example, you use the IIF function to return one of two values. If a sales person's year-to-date sales are above 200,000, this person will be eligible for a bonus. Values below 200,000 mean that employees don't qualify for bonuses.

SQL includes several mathematical functions that you can use to perform calculations on input values and return numeric results.

One example is the SIGN function, which returns a value indicating the sign of an expression. The value of -1 indicates a negative expression, the value of +1 indicates a positive expression, and 0 indicates zero.

```
SELECT SIGN(-20) AS 'Sign'
```

Sign -1

In the example, the input is a negative number, so the Results pane lists the result -1.

Another mathematical function is the POWER function. This function provides the value of an expression raised to a specified power.

In the syntax, the float_expression parameter specifies the expression, and the y parameter specifies the power to which you want to raise the expression.

```
SELECT POWER(50, 3) AS Result
```

Result

125000

Functions (Analytic)

You use analytic functions to determine values based on groups of values. For example, you can use this type of function to determine running totals, percentages, or the top result within a group.

LAG and LEAD

The LAG function provides data on rows before the current row in the same result set. For example, in a SELECT statement, you can compare values in the current row with values in a previous row.

You use a scalar expression to specify the values that should be compared. The offset parameter is the number of rows before the current row that will be used in the comparison. If you don't specify the number of rows, the default value of one row is used.

The default parameter specifies the value that should be returned when the expression at offset has a NULL value. If you don't specify a value, a value of NULL is returned.

The LEAD function provides data on rows after the current row in the row set. For example, in a SELECT statement, you can compare values in the current row with values in the following row.

You specify the values that should be compared using a scalar expression. The offset parameter is the number of rows after the current row that will be used in the comparison.

You specify the value that should be returned when the expression at offset has a NULL value using the default parameter. If you don't specify these parameters, the default of one row is used and a value of NULL is returned.

```
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;
```

This example uses the LEAD and LAG functions to compare the sales values for each employee to date with those of the employees listed above and below, with records ordered based on the BusinessEntityID column.

BusinessEntityID	SalesYTD	Lead value	Lag value
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

PERCENTILE_DISC and PERCENTILE_CONT

The PERCENTILE_DISC function lists the value of the first entry where the cumulative distribution is higher than the percentile that you provide using the numeric_literal parameter.

The values are grouped by rowset or partition, as specified by the clause.   WITHIN GROUP

The PERCENTILE_CONT function is similar to the PERCENTILE_DISC function, but returns the average of the sum of the first matching entry and the next entry.

```
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;
```

To find the exact value from the row that matches or exceeds the 0.5 percentile, you pass the percentile as the numeric literal in the PERCENTILE_DISC function. The Percentile Discreet column in a result set lists the value of the row at which the cumulative distribution is higher than the specified percentile.

BusinessEntityID	JobTitle	SickLeaveHours	Cumulative Distribution	Percentile Discreet
272	Application Specialist	55	0.25	**56**
268	Application Specialist	56	0.75	**56**
269	Application Specialist	56	0.75	**56**
267	Application Specialist	57	1	**56**

To base the calculation on a set of values, you use the PERCENTILE_CONT function. The "Percentile Continuous" column in the results lists the average value of the sum of the result value and the next highest matching value.

```
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	JobTitle	SickLeaveHours	Cumulative Distribution	Percentile Discreet	Percentile Continuous
272	Application Specialist	55	0.25	56	**56**
268	Application Specialist	56	0.75	56	**56**
269	Application Specialist	56	0.75	56	**56**
267	Application Specialist	57	1	56	**56**

FIRST_VALUE

You use the FIRST_VALUE function to determine the first value in an ordered result set, which you identify using a scalar expression.

```
SELECT StateProvinceID, Name, TaxRate,
    FIRST_VALUE(StateProvinceID)
     OVER(ORDER BY TaxRate ASC) AS FirstValue
FROM SalesTaxRate;
```

In this example, the FIRST_VALUE function is used to return the ID of the state or province with the lowest tax rate. The OVER clause is used to order the tax rates to obtain the lowest rate.

StateProvinceID	Name	TaxRate	FirstValue
74	Utah State Sales Tax	5.00	74
36	Minnesota State Sales Tax	6.75	74
30	Massachusetts State Sales Tax	7.00	74
1	Canadian GST	7.00	74
57	Canadian GST	7.00	74
63	Canadian GST	7.00	74

LAST_VALUE

The LAST_VALUE function provides the last value in an ordered result set, which you specify using a scalar expression.

```
SELECT  TerritoryID, StartDate, BusinessentityID,
     LAST_VALUE(BusinessentityID)
      OVER(ORDER  BY  TerritoryID) AS  LastValue
FROM  SalesTerritoryHistory;
```

This example uses the LAST_VALUE function to return the last value for each rowset in the ordered values.

TerritoryID	StartDate	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

PERCENT_RANK and CUME_DIST

The PERCENT_RANK function calculates the ranking of a row relative to the row set. The percentage is based on the number of rows in the group that have a lower value than the current row.

The first value in the result set always has a percent rank of zero. The value for the highest-ranked – or last – value in the set is always one.

The CUME_DIST function calculates the relative position of a specified value in a group of values, by determining the percentage of values less than or equal to that value. This is called the cumulative distribution.

```
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;
```

In this example, you use an ORDER clause to partition – or group – the rows retrieved by the SELECT statement based on employees' job titles, with the results in each group sorted based on the numbers of sick leave hours that employees have used.

BusinessEntityID	JobTitle	SickLeaveHours	Percent Rank	Cumulative Distribution
267	Application Specialist	57	0	0.25
268	Application Specialist	56	0.333333333333333	0.75
269	Application Specialist	56	0.333333333333333	0.75
272	Application Specialist	55	1	1
262	Assitant to the Cheif Financial Officer	48	0	1
239	Benefits Specialist	45	0	1
252	Buyer	50	0	0.111111111111111
251	Buyer	49	0.125	0.333333333333333
256	Buyer	49	0.125	0.333333333333333
253	Buyer	48	0.375	0.555555555555555
254	Buyer	48	0.375	0.555555555555555

The PERCENT_RANK function ranks the entries within each group. For each entry, it returns the percentage of entries in the same group that have lower values.

The CUME_DIST function is similar, except that it returns the percentage of values less than or equal to the current value.

Window Functions

Setting up a flag if other rows have a common property

Let's say I have this data:

Table items

id	name	tag
1	example	unique_tag
2	foo	simple
42	bar	simple
3	baz	hello
51	quux	world

I'd like to get all those lines and know if a tag is used by other lines

```
SELECT id, name, tag, COUNT(*) OVER (PARTITION BY tag) > 1 AS flag FROM items
```

The result will be:

id	name	tag	flag
1	example	unique_tag	false
2	foo	simple	true
42	bar	simple	true
3	baz	hello	false
51	quux	world	false

In case your database doesn't have OVER and PARTITION you can use this to produce the same result:

```
SELECT id, name, tag, (SELECT COUNT(tag) FROM items B WHERE tag = A.tag) > 1 AS flag FROM items A
```

Finding "out-of-sequence" records using the LAG() function

Given these sample data:

ID	STATUS	STATUS_TIME	STATUS_BY
1	ONE	2016-09-28-19.47.52.501398	USER_1
3	ONE	2016-09-28-19.47.52.501511	USER_2
1	THREE	2016-09-28-19.47.52.501517	USER_3
3	TWO	2016-09-28-19.47.52.501521	USER_2
3	THREE	2016-09-28-19.47.52.501524	USER_4

Items identified by ID values must move from STATUS 'ONE' to 'TWO' to 'THREE' in sequence, without skipping statuses. The problem is to find users (STATUS_BY) values who violate the rule and move from 'ONE' immediately to 'THREE'.

The LAG() analytical function helps to solve the problem by returning for each row the value in the preceding row:

```
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'
```

In case your database doesn't have LAG() you can use this to produce the same result:

```
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'
```

Getting a running total

Given this data:

date	amount
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
```

will give you

date	amount	running
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

Adding the total rows selected to every row

```
SELECT your_columns, COUNT(*) OVER() as Ttl_Rows FROM your_data_set
```

id	name	Ttl_Rows
1	example	5
2	foo	5
3	bar	5
4	baz	5
5	quux	5

Instead of using two queries to get a count then the line, you can use an aggregate as a window function and use the full result set as the window.
This can be used as a base for further calculation without the complexity of extra self joins.

Getting the N most recent rows over multiple grouping

Given this data

User_ID	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
```

Using n=1, you'll get the one most recent row per user_id:

User_ID	Completion_Date	Row_Num
1	2016-07-21	1
2	2016-07-22	1

Common Table Expressions

generating values

Most databases do not have a native way of generating a series of numbers for ad-hoc use; however, common table expressions can be used with recursion to emulate that type of function.

The following example generates a common table expression called Numbers with a column i which has a row for numbers 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;
```

```
i
1
2
3
4
5
```

This method can be used with any number interval, as well as other types of data.

recursively enumerating a subtree

```
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;
```

Level	ID	FName	LName
1	1	James	Smith
2	2	John	Johnson
3	4	Johnathon	Smith
2	3	Michael	Williams

Temporary query

These behave in the same manner as nested subqueries but with a different syntax.

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

ID	Model	TotalCost
1	Ford F-150	200
2	Ford F-150	230

Equivalent subquery syntax

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

recursively going up in a tree

```
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;
```

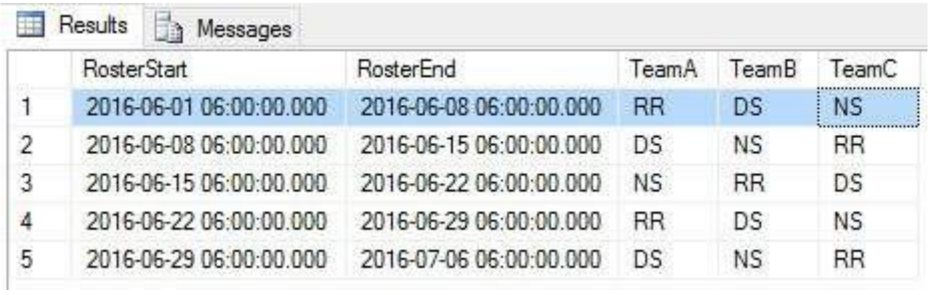
Id	FName	LName	PhoneNumber	ManagerId	DepartmentId
4	Johnathon	Smith	1212121212	**2**	1
2	John	Johnson	2468101214	**1**	1
1	James	Smith	1234567890	NULL	1

Recursively generate dates, extended to include team rostering as example

```
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
```

Result

I.e. For Week 1 TeamA is on R&R, TeamB is on Day Shift and TeamC is on Night Shift.



	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

Oracle CONNECT BY functionality with recursive CTEs

Oracle's CONNECT BY functionality provides many useful and nontrivial features that are not built-in when using SQL standard recursive CTEs. This example replicates these features (with a few additions for sake of completeness), using SQL Server syntax. It is most useful for Oracle developers finding many features missing in their hierarchical queries on other databases, but it also serves to showcase what can be done with a hierarchical query in general.

```
  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 features demonstrated above, with explanations:

- Clauses
    - CONNECT BY: Specifies the relationship that defines the hierarchy.
    - START WITH: Specifies the root nodes.
    - ORDER SIBLINGS BY: Orders results properly.
- Parameters
    - NOCYCLE: Stops processing a branch when a loop is detected. Valid hierarchies are Directed Acyclic Graphs, and circular references violate this construct.
- Operators
    - PRIOR: Obtains data from the node's parent.
    - CONNECT_BY_ROOT: Obtains data from the node's root.
- Pseudocolumns
    - LEVEL: Indicates the node's distance from its root.
    - CONNECT_BY_ISLEAF: Indicates a node without children.
    - CONNECT_BY_ISCYCLE: Indicates a node with a circular reference.
- Functions
    - SYS_CONNECT_BY_PATH: Returns a flattened/concatenated representation of the path to the node from its root.

Views

Simple views

A view can filter some rows from the base table or project only some columns from it:

```
CREATE VIEW new_employees_details AS
SELECT E.id, Fname, Salary, Hire_date
FROM Employees E
WHERE hire_date > date '2015-01-01';
```

If you select form the view:

```
select * from new_employees_details
```

Id	FName	Salary	Hire_date
4	Johnathon	500	24-07-2016

Complex views

A view can be a really complex query(aggregations, joins, subqueries, etc). Just be sure you add column names for everything you select:

```
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;
```

Now you can select from it as from any table:

```
SELECT *
```

FROM dept_income;

DepartmentName	TotalSalary
HR	1900
Sales	600

Materialized Views

A materialized view is a view whose results are physically stored and must be periodically refreshed in order to remain current. They are therefore useful for storing the results of complex, long-running queries when realtime results are not required. Materialized views can be created in Oracle and PostgreSQL. Other database systems offer similar functionality, such as SQL Server's indexed views or DB2's materialized query tables.

PostgreSQL example

```
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)
```

Comments

Single-line comments

Single line comments are preceded by --, and go until the end of the line:

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

Multi-line comments

Multi-line code comments are wrapped in /* ... */:

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

It is also possible to insert such a comment into the middle of a line:

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

Foreign Keys

Foreign Keys explained

Foreign Keys constraints ensure data integrity, by enforcing that values in one table must match values in another table.

An example of where a foreign key is required is: In a university, a course must belong to a department. Code for the this scenario is:

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

Insert values with the following statement:

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

The following table will contain the information of the subjects offered by the Computer science branch:

```
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)
);
```

(The data type of the Foreign Key must match the datatype of the referenced key.)

The Foreign Key constraint on the column Dept_Code allows values only if they already exist in the referenced table, Department. This means that if you try to insert the following values:

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

the database will raise a Foreign Key violation error, because CS300 does not exist in the Department table. But when you try a key value that exists:

```
INSERT INTO Programming_Courses VALUES ('CS205', 'FDB-DB001', 'Database Systems');
INSERT INTO Programming_Courses VALUES ('CS205', 'DB2-DB002', 'Database Systems II');
```
then the database allows these values.

A few tips for using Foreign Keys

- A Foreign Key must reference a UNIQUE (or PRIMARY) key in the parent table.
- Entering a NULL value in a Foreign Key column does not raise an error.
- Foreign Key constraints can reference tables within the same database.
- Foreign Key constraints can refer to another column in the same table (self-reference).

Creating a table with a foreign key

In this example we have an existing table, SuperHeros.
This table contains a primary key ID.

We will add a new table in order to store the powers of each super hero:

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

The column HeroId is a **foreign key** to the table SuperHeros.

Sequence

Create Sequence

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

Creates a sequence with a starting value of 1000 which is incremented by 1.

Using Sequences

a reference to *seq_name*.NEXTVAL is used to get the next value in a sequence. A single statement can only generate a single sequence value. If there are multiple references to NEXTVAL in a statement, they use will use the same generated number.

NEXTVAL can be used for INSERTS

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

It can be used for UPDATES

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

It can also be used for SELECTS

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

Subqueries

Subquery in FROM clause

A subquery in a FROM clause acts similarly to a temporary table that is generated during the execution of a query and lost afterwards.

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

Subquery in SELECT clause

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

Subquery in WHERE clause

Use a subquery to filter the result set. For example this will return all employees with a salary equal to the highest paid employee.

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

Correlated Subqueries

Correlated (also known as Synchronized or Coordinated) Subqueries are nested queries that make references to the current row of their outer query:

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

Subquery `SELECT AVG(Salary) ...` is *correlated* because it refers to `Employee` row `eOuter` from its outer query.

Filter query results using query on di erent table

This query selects all employees not on the Supervisors table.

```
SELECT *

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

The same results can be achieved using a LEFT JOIN.

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

Subqueries in FROM clause

You can use subqueries to define a temporary table and use it in the FROM clause of an "outer" query.

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

The above finds cities from the weather table whose daily temperature variation is greater than 20. The result is:

city	temp_var
ST LOUIS	21
LOS ANGELES	31
LOS ANGELES	23
LOS ANGELES	31
LOS ANGELES	27
LOS ANGELES	28
LOS ANGELES	28
LOS ANGELES	32

. ## Subqueries in WHERE clause

The following example finds cities (from the cities example) whose population is below the average temperature (obtained via a sub-qquery):

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

Here: the subquery (SELECT avg(pop2000) FROM cities) is used to specify conditions in the WHERE clause. The result is:

name	pop2000
San Francisco	776733
ST LOUIS	348189
Kansas City	146866

Execution blocks

Using BEGIN ... END

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

Stored Procedures

Create and call a stored procedure

Stored procedures can be created through a database management GUI (SQL Server example), or through a SQL statement as follows:

```
-- 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;
```

Calling the procedure:

```
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
```

Triggers

CREATE TRIGGER

This example creates a trigger that inserts a record to a second table (MyAudit) after a record is inserted into the table the trigger is defined on (MyTable). Here the "inserted" table is a special table used by Microsoft SQL Server to store affected rows during INSERT and UPDATE statements; there is also a special "deleted" table that performs the same function for DELETE statements.

```
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
```

Use Trigger to manage a "Recycle Bin" for deleted items

```
CREATE TRIGGER BooksDeleteTrigger
    ON MyBooksDB.Books
    AFTER DELETE
AS
  INSERT INTO BooksRecycleBin
    SELECT *
    FROM deleted;
GO
```

Transactions

Simple Transaction

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

Rollback Transaction

When something fails in your transaction code and you want to undo it, you can rollback your transaction:

```
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
```

Table Design

Properties of a well designed table

A true relational database must go beyond throwing data into a few tables and writing some SQL statements to pull that data out. At best a badly designed table structure will slow the execution of queries and could make it impossible for the database to function as intended.

A database table should not be considered as just another table; it has to follow a set of rules to be considered truly relational. Academically it is referred to as a 'relation' to make the distinction.

The five rules of a relational table are:

1. Each value is *atomic*; the value in each field in each row must be a single value.
2. Each field contains values that are of the same data type.
3. Each field heading has a unique name.
4. Each row in the table must have at least one value that makes it unique amongst the other records in the table.
5. The order of the rows and columns has no significance.

A table conforming to the five rules:

Id	Name	DOB	Manager
1	Fred	11/02/1971	3
2	Fred	11/02/1971	3
3	Sue	08/07/1975	2

- Rule 1: Each value is atomic. Id, Name, DOB and Manager only contain a single value.
- Rule 2: Id contains only integers, Name contains text (we could add that it's text of four characters or less), DOB contains dates of a valid type and Manager contains integers (we could add that corresponds to a Primary Key field in a managers table).
- Rule 3: Id, Name, DOB and Manager are unique heading names within the table.
- Rule 4: The inclusion of the Id field ensures that each record is distinct from any other record within the table.

A badly designed table:

Id	Name	DOB	Name
1	Fred	11/02/1971	3
1	Fred	11/02/1971	3
3	Sue	Friday the 18th July 1975	2, 1

- Rule 1: The second name field contains two values - 2 and 1.
- Rule 2: The DOB field contains dates and text.
- Rule 3: There's two fields called 'name'.
- Rule 4: The first and second record are exactly the same. Rule 5: This
- rule isn't broken.

Synonyms

Create Synonym

```
CREATE SYNONYM EmployeeData
FOR MyDatabase.dbo.Employees
```

Information Schema

Basic Information Schema Search

One of the most useful queries for end users of large RDBMS's is a search of an information schema.

Such a query allows users to rapidly find database tables containing columns of interest, such as when attempting to relate data from 2 tables indirectly through a third table, without existing knowledge of which tables may contain keys or other useful columns in common with the target tables.

Using T-SQL for this example, a database's information schema may be searched as follows:

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

The result contains a list of matching columns, their tables' names, and other useful information.

Order of Execution

Logical Order of Query Processing in 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
```

The order in which a query is processed and description of each section.

VT stands for 'Virtual Table' and shows how various data is produced as the query is processed

1. FROM: A Cartesian product (cross join) is performed between the first two tables in the FROM clause, and as a result, virtual table VT1 is generated.
2. ON: The ON filter is applied to VT1. Only rows for which the is TRUE are inserted to VT2.
3. OUTER (join): If an OUTER JOIN is specified (as opposed to a CROSS JOIN or an INNER JOIN), rows from the preserved table or tables for which a match was not found are added to the rows from VT2 as outer rows, generating VT3. If more than two tables appear in the FROM clause, steps 1 through 3 are applied repeatedly between the result of the last join and the next table in the FROM clause until all tables are processed.
4. WHERE: The WHERE filter is applied to VT3. Only rows for which the is TRUE are inserted to VT4.
5. GROUP BY: The rows from VT4 are arranged in groups based on the column list specified in the GROUP BY clause. VT5 is generated.
6. CUBE | ROLLUP: Supergroups (groups of groups) are added to the rows from VT5, generating VT6.
7. HAVING: The HAVING filter is applied to VT6. Only groups for which the is TRUE are inserted to VT7.
8. SELECT: The SELECT list is processed, generating VT8.
9. DISTINCT: Duplicate rows are removed from VT8. VT9 is generated.
10. ORDER BY: The rows from VT9 are sorted according to the column list specified in the ORDER BY clause. A cursor is generated (VC10).
11. TOP: The specified number or percentage of rows is selected from the beginning of VC10. Table VT11 is generated and returned to the caller. LIMIT has the same functionality as TOP in some SQL dialects such as Postgres and Netezza.

Clean Code in SQL

How to write good, readable SQL queries, and example of good practices.

Formatting and Spelling of Keywords and Names

Table/Column Names

Two common ways of formatting table/column names are CamelCase and snake_case:

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

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

Names should describe what is stored in their object. This implies that column names usually should be singular. Whether table names should use singular or plural is a heavily discussed question, but in practice, it is more common to use plural table names.

Adding prefixes or suffixes like `tbl` or `col` reduces readability, so avoid them. However, they are sometimes used to avoid conflicts with SQL keywords, and often used with triggers and indexes (whose names are usually not mentioned in queries).

Keywords

SQL keywords are not case sensitive. However, it is common practice to write them in upper case.

Indenting

There is no widely accepted standard. What everyone agrees on is that squeezing everything into a single line is bad:

```
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;
```

At the minimum, put every clause into a new line, and split lines if they would become too long otherwise:

```
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;
```

Sometimes, everything after the SQL keyword introducing a clause is indented to the same column:

```
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;
```

(This can also be done while aligning the SQL keywords right.)

Another common style is to put important keywords on their own lines:

```
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;
```

Vertically aligning multiple similar expressions improves readability:

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

Using multiple lines makes it harder to embed SQL commands into other programming languages. However, many languages have a @"... " in C#, """...""" in Python, or R" (... mechanism for multi-line strings, e.g.,)" in C++.

Section

SELECT

SELECT*

* returns all columns in the same order as they are defined in the table.

When using SELECT *, the data returned by a query can change whenever the table definition changes. This increases the risk that different versions of your application or your database are incompatible with each other.

Furthermore, reading more columns than necessary can increase the amount of disk and network I/O.

So you should always explicitly specify the column(s) you actually want to retrieve:

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

(When doing interactive queries, these considerations do not apply.)

However, SELECT * does not hurt in the subquery of an EXISTS operator, because EXISTS ignores the actual data anyway (it checks only if at least one row has been found). For the same reason, it is not meaningful to list any specific column(s) for EXISTS, so * actually makes more sense: SELECT

```
-- 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');
```

Joins

Explicit joins should always be used; implicit joins have several problems:

- The join condition is somewhere in the WHERE clause, mixed up with any other filter conditions. This makes it harder to see which tables are joined, and how.
- Due to the above, there is a higher risk of mistakes, and it is more likely that they are found later.
- In standard SQL, explicit joins are the only way to use outer joins:

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

- Explicit joins allow using the USING clause:

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

(This requires that both tables use the same column name.
USING automatically removes the duplicate column from the result, e.g., the join in this query returns a single RecipeID column.)

SQL Injection

SQL injection is an attempt to access a website's database tables by injecting SQL into a form field. If a web server does not protect against SQL injection attacks, a hacker can trick the database into running the additional SQL code. By executing their own SQL code, hackers can upgrade their account access, view someone else's private information, or make any other modifications to the database.

SQL injection sample

Assuming the call to your web application's login handler looks like this:

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

Now in login.ashx, you read these values:

strUserName = getHttpsRequestParameterString("username"); strPassword = getHttpsRequestParameterString("password"); and query your database to determine whether a user with that password exists.

So you construct an SQL query string:

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

This will work if the username and password do not contain a quote.

However, if one of the parameters does contain a quote, the SQL that gets sent to the database will look like this:

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

This will result in a syntax error, because the quote after the d in d ends 'Alambert the SQL string.

You could correct this by escaping quotes in username and password, e.g.:

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

However, it's more appropriate to use parameters:

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

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

If you do not use parameters, and forget to replace quote in even one of the values, then a malicious user (aka hacker) can use this to execute SQL commands on your database. For example, if an attacker is evil, he/she will set the password to

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

and then the SQL will look like this:

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

Unfortunately for you, this is valid SQL, and the DB will execute this!

This type of exploit is called an SQL injection.

There are many other things a malicious user could do, such as stealing every user's email address, steal everyone's password, steal credit card numbers, steal any amount of data in your database, etc.

This is why you always need to escape your strings.
And the fact that you'll invariably forget to do so sooner or later is exactly why you should use parameters. Because if you use parameters, then your programming language framework will do any necessary escaping for you.

simple injection sample

If the SQL statement is constructed like this:

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

Then a hacker could retrieve your data by giving a password like 1; the pw' or ' 1'=' resulting SQL statement will be:

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

This one will pass the password check for all rows in the Users table because is always '1' ='1' true.

To prevent this, use SQL parameters:

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

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content, more changes c