2019

SQL Database

Youssouf Sougueh

Professeur d'informatique

Contents

INTRODUCTION

SQL is a database computer language designed for the retrieval and management of data in a relational database. SQL stands for Structured Query Language.

This tutorial will give you a quick start to **SQL**. It covers most of the topics required for a basic understanding of SQL and to get a feel of how it works.

What is Data? In simple words data can be facts related to any object inconsideration. For example your name, age height, weight, etc. are some data related to you.

Why to Learn SQL

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database.

SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle, Sybase, Informix, Postgres and SQL Server use SQL as their standard database language.

Also, they are using different dialects, such as −

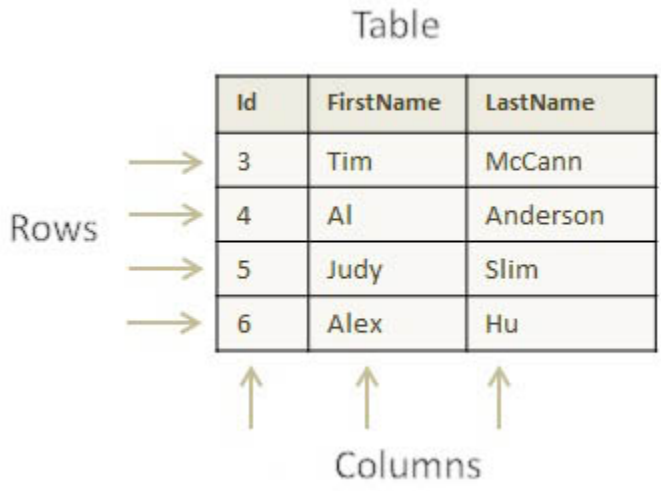
- MS SQL Server using T-SQL,
- Oracle using PL/SQL,
- MS Access version of SQL is called JET SQL (native format) etc.

What is a Database?

- A database is a place to store data.
- A relational database system (RDMS) stores data in tables.

Relational Database Tables

- A relational database stores data in tables. Each table has a number of rows and columns.
- The table below has 4 rows and 3 columns.

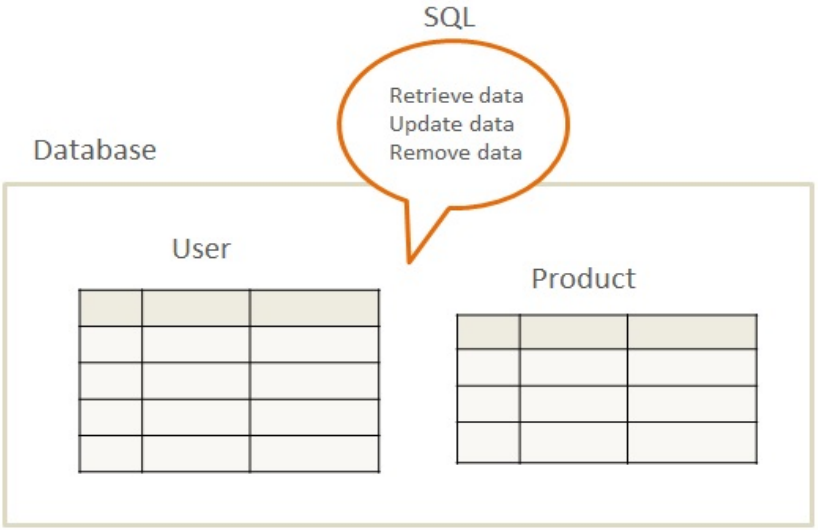


SQL and Relational Databases

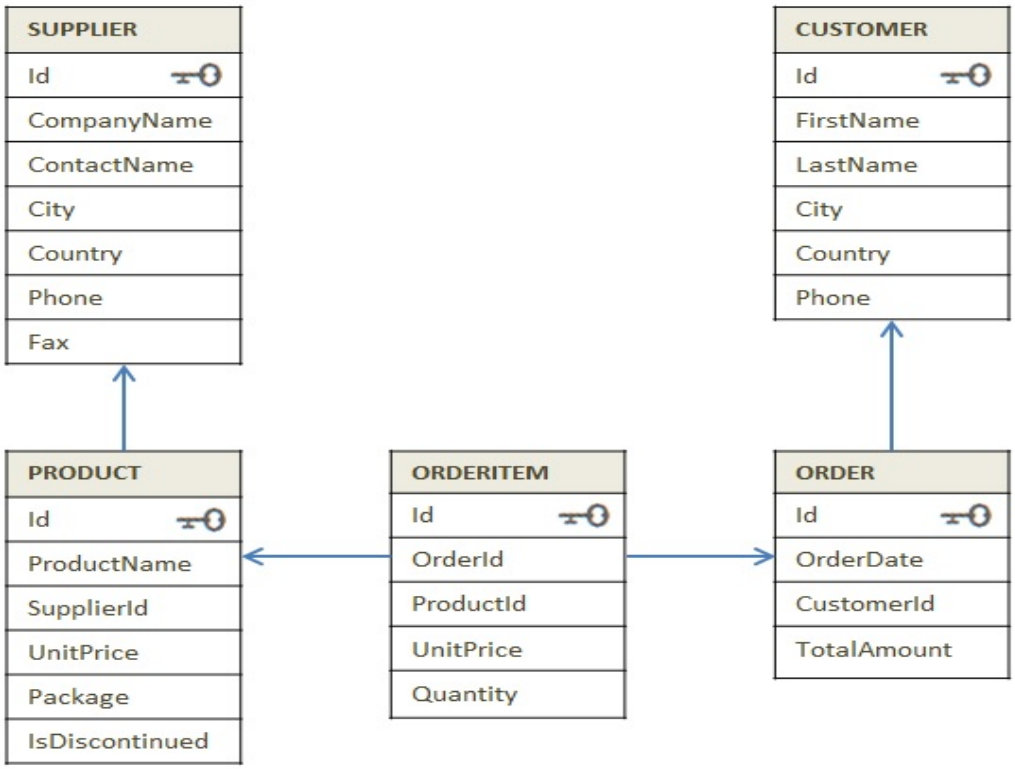
A relational database contains tables which store data that is related in some way.

SQL is the language that allows retrieval and manipulation of table data in a relational database.

The database below has 2 tables: one with data on Users and another with data on Products.



Example database: This tutorial uses a database which is a modernized version. Below is an Entity Relationship Diagram (**ERD**) which shows the tables and their relationships.



Chapter 1: 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

Section 1.1: 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.

Chapter 2: Data Types

Section 2.1: 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
```

Section 2.2: 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
```

Section 2.2: 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

Section 2.3: 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

Section 2.4: 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
```

Section 2.5: 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)
```

Section 2.6: 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.

Chapter 3: 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.

Section 3.1: 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 `= NULL` or `<> NULL` (or `!= NULL`) will 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`.

Section 3.2: 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 `NOT NULL` constraint.

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.
```

Section 3.3: Updating fields to NULL

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

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

Chapter 4: Example Databases and Tables

Section 4.1: 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.

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 (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(25) NOT NULL,
    PRIMARY KEY(Id)
);

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

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 (
    Id INT NOT NULL AUTO_INCREMENT,
    FName VARCHAR(35) NOT NULL,
    LName VARCHAR(35) NOT NULL,
    PhoneNumber VARCHAR(11),
    ManagerId INT,
    DepartmentId INT NOT NULL,
```

```
    Salary INT NOT NULL,
    HireDate DATETIME NOT NULL,
    PRIMARY KEY(Id),
    FOREIGN KEY (ManagerId) REFERENCES Employees(Id),
    FOREIGN KEY (DepartmentId) REFERENCES Departments(Id)
);

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

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

SQL statements to create the table:

```
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')
;
```

Section 4.2: Library Database

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

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 (
    Id INT NOT NULL AUTO_INCREMENT,
    Title VARCHAR(50) NOT NULL,
    PRIMARY KEY(Id)
);

INSERT INTO Books
    (Id, Title)
VALUES
    (1, 'The Catcher in the Rye'),
    (2, 'Nine Stories'),
    (3, 'Franny and Zooey'),
    (4, 'The Great Gatsby'),
    (5, 'Tender id the Night'),
    (6, 'Pride and Prejudice'),
    (7, 'Professional ASP.NET 4.5 in C# and VB')
;
```

BooksAuthors

(*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:

```sql
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)
;
```

```sql
SELECT * FROM Authors;
```

```sql
SELECT * FROM Books;
```

```sql
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
;
```

Section 4.3: 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.

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

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')
;
```

Chapter 5: 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.

Section 5.1: 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 BY displayname` - if a 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 `ORDER BY` clause will be 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).

When using `EXISTS`, such as `SELECT A.col1, A.Col2 FROM A WHERE EXISTS (SELECT * FROM B where A.ID = B.A_ID)`, 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 `COUNT(*)` is fine as it also doesn't actually return any of the columns, so only needs to read and process those that are used for filtering purposes.

Section 5.2: 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
Michael	Marcus	Williams

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.

Section 5.3: 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
```

Section 5.4: 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 with `ROWNUM`:

```
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.

Section 5.5: 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.

Section 5.6: 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
```

Section 5.7: 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
```

Section 5.8: 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 `INNER JOIN`. The 1992 SQL standard corrected this earlier design flaw by introducing `NATURAL JOIN` (implemented in 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.

Section 5.9: 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

Section 5.10: 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.

Section 5.11: 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.

Section 5.12: 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

Chapter 6: 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.

Section 6.1: 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.

Section 6.2: 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

```
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

```
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
;
```

Section 6.3: 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

Section 6.4: 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` that it provides all possible combinations than the argument attributes of the clause.
- `with roll up` 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

Chapter 7: ORDER BY

Section 7.1: 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**

Section 7.2: 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
```

Section 7.3: 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
Ken	Accountant

```
SELECT *
FROM Employee
ORDER BY CASE Department
         WHEN 'HR'          THEN 1
         WHEN 'Accountant' THEN 2
         ELSE                    3
         END;
```

Name	Department
Yusuf	**HR**
Hillary	**HR**
Merry	**HR**
Ken	**Accountant**
Hasan	**IT**
Joe	**IT**

Section 7.4: 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
```

Section 7.5: 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**

Chapter 8: AND & OR Operators

Section 8.1: 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

Chapter 9: LIKE operator

Section 9.1: 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	J**on**	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	2479100**11**	1	1	400	11-01-2010
5	Hilde	Knag	2468021**11**	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	Jon	Sanchez	2454124602	1	1	400	23-03-2005

Section 9.2: Single character match

To broaden the 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'
```

Section 9.3: 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 grapheme consists out of several characters (utf-8). e.g. `string stringToSearch = "Les Mise\u0301rables";` 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.

Section 9.4: 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]%'
```

Section 9.5: Match by range or set

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

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'
```

Section 9.6: 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 '_%';
```

Chapter 10: IN clause

Section 10.1: 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
```

Section 10.2: 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.

Chapter 11: Filter results using WHERE and HAVING

Section 11.1: 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	100	20	34.5
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';
```

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

Section 11.2: 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 form a single value of more significant meaning or measurement (Wikipedia).

Common aggregate functions include `COUNT()`, `SUM()`, `MIN()`, and `MAX()`.

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 `Number of Cars` count 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

Section 11.3: 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`.

Section 11.4: 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
```

Section 11.5: 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
```

Section 11.6: 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
```

Section 11.7: 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
```

Section 11.8: 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`%` - will return any Employee whose name begins with 'John', followed by any amount of characters
- `%`John - will return any Employee whose name ends with 'John', proceeded by any amount of characters
- `%`John`%` - 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'.

Section 11.9: 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)
```

Section 11.10: 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.

Chapter 12: 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.

Section 12.1: 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	James	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:

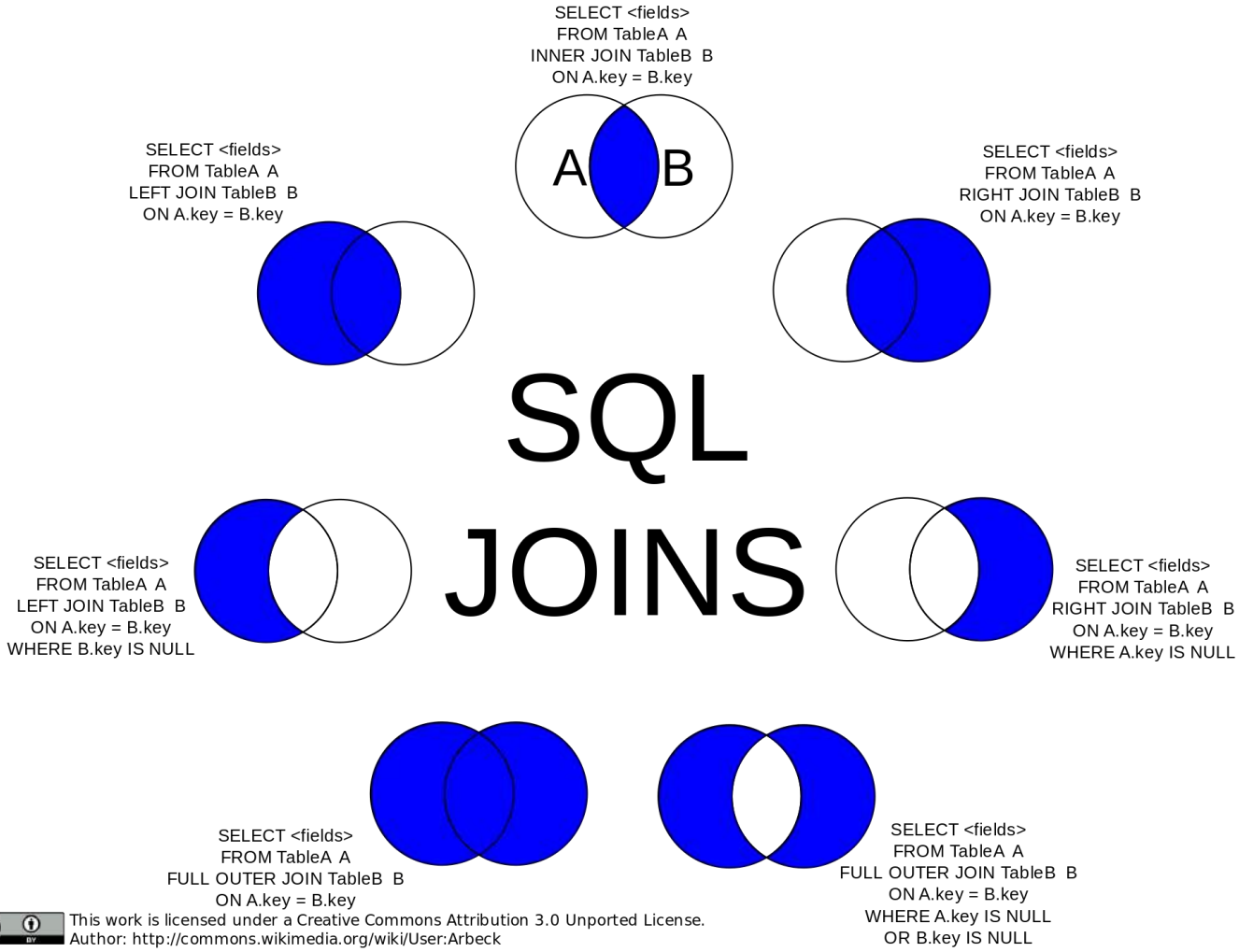
e.FName	m.FName
John	James
Michael	James
Johnathon	John

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

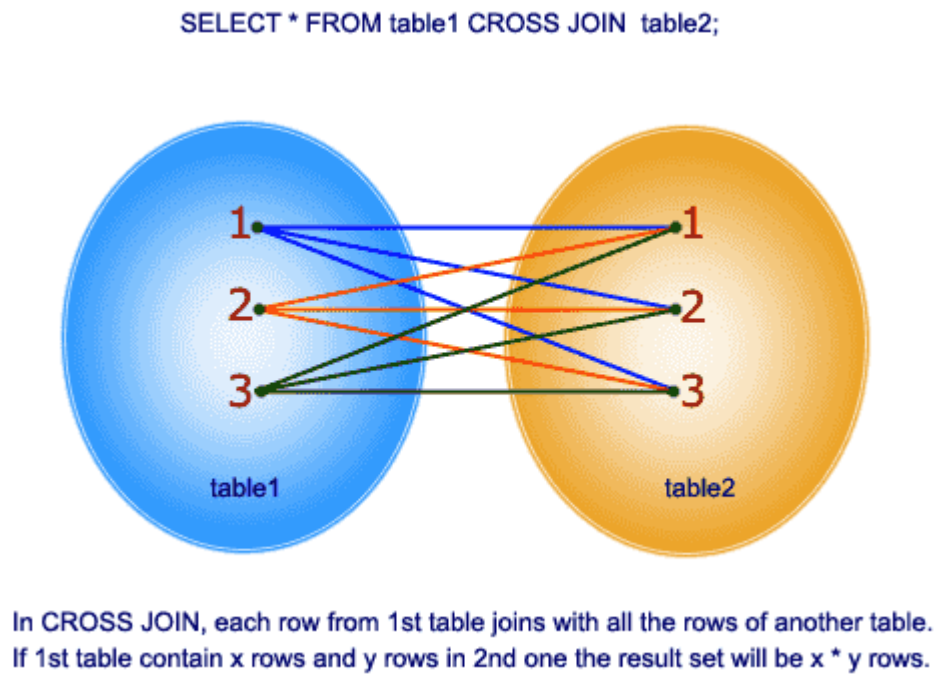
Employee	Manager
John	James
Michael	James
Johnathon	John

Section 12.2: Differences between inner/outer joins

SQL has various join types to specify whether (non-)matching rows are included in the result: `INNER JOIN`, `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.



Cross Join SQL Pictorial Presentation (reference) :



```
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
-----+-----
```

Chapter 13: UPDATE

Section 13.1: 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
```

Section 13.2: 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.

Section 13.3: 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.

Section 13.4: 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.

Section 13.5: 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
```

Chapter 14: CREATE Database

Section 14.1: 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.

Chapter 15: CREATE TABLE

Parameter	Details
tableName	The name of the table
columns	Contains an 'enumeration' of all the columns that the table have. See **Create a New Table** for more 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.

Section 15.1: 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;
```

Section 15.2: 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(
    Id int 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(1,1)`	states that column will have auto generated values starting at 1 and incrementing by 1 for each 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

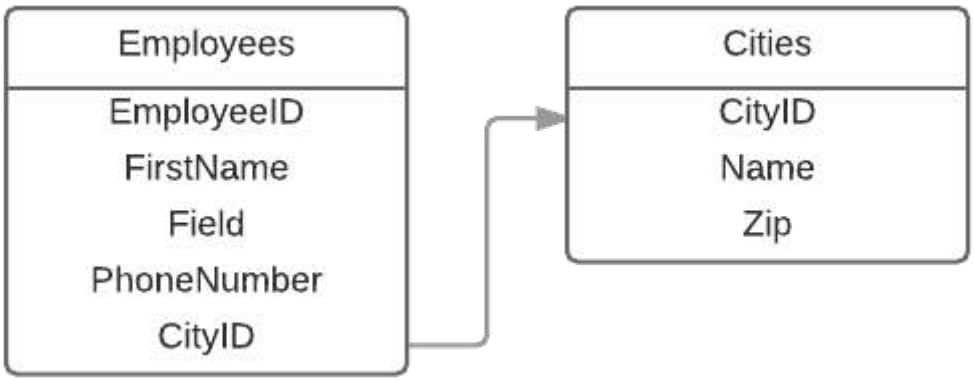
Section 153: CREATE TABLE With FOREIGN KEY

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 Cities(CityID)`	Makes the reference 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.

Section 15.4: Duplicate a table

To duplicate a table, simply do the following:

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

Section 15.5: 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(...);
```

Chapter 16: Primary Keys

Section 16.1: Creating a Primary Key

```
CREATE TABLE Employees (
    Id int 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)
)
```

Section 16.2: 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
);
```

Chapter 17: Foreign Keys

Section 50.1: 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).

Section 17.2: 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`.