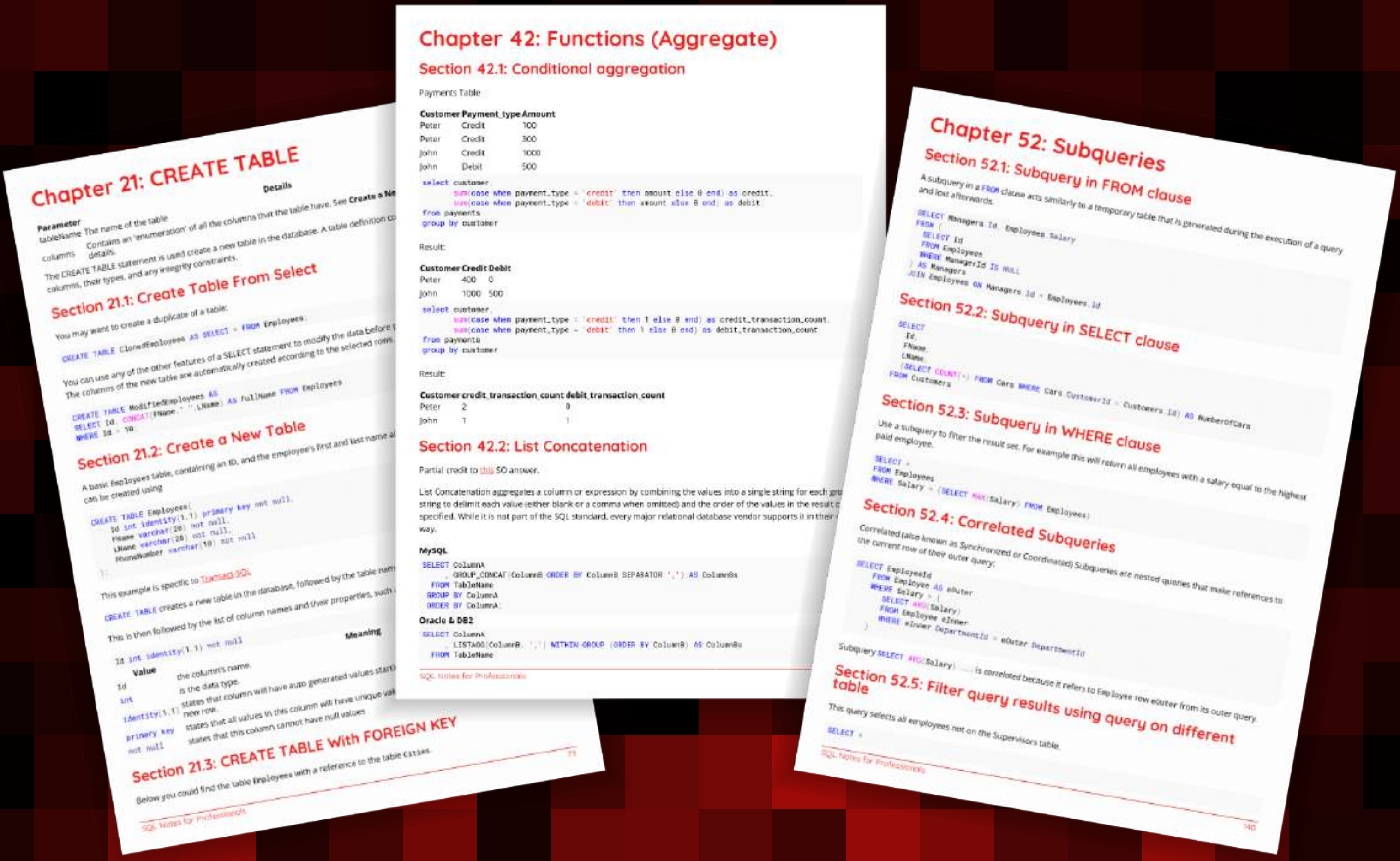
SQL
Notes for Professionals



Chapter 42: Functions (Aggregate)

Section 42.1: 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

Section 42.2: 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
```

Chapter 21: CREATE TABLE

Section 21.1: Create Table From Select

Section 21.2: Create a New Table

Section 21.3: CREATE TABLE With FOREIGN KEY

Chapter 52: Subqueries

Section 52.1: Subquery in FROM clause

Section 52.2: Subquery in SELECT clause

Section 52.3: Subquery in WHERE clause

Section 52.4: Correlated Subqueries

Section 52.5: Filter query results using query on different table

100+ pages

of professional hints and tricks

GoalKicker.com
Free Programming Books

Disclaimer
This is an unofficial free book created for educational purposes and is not affiliated with official SQL group(s) or company(s).
All trademarks and registered trademarks are the property of their respective owners

Contents

About

Please feel free to share this PDF with anyone for free, latest version of this book can be downloaded from: https://goalkicker.com/SQLBook

This *SQL Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official SQL group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

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: 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.

Section 2.1: 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.

Chapter 3: Data Types

Section 3.1: 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 3.2: 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 3.3: 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 3.4: 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 3.5: 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 3.6: 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 3.7: 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.

Section 3.8: 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'
```

Chapter 4: 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 4.1: 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 4.2: 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 4.3: Updating fields to NULL

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

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

Section 4.4: 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') ;
```

Chapter 5: Example Databases and Tables

Section 5.1: 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.

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 (
    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 5.2: 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 (
    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:

```
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
;
```

Section 5.3: 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.

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')
;
```

Chapter 6: 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 6.1: 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 6.2: 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 6.3: 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 6.4: 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 6.5: 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 6.6: 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 6.7: 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 6.8: 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 6.9: 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 6.10: 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.

Section 6.11: 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.