SQL AND RDBMS

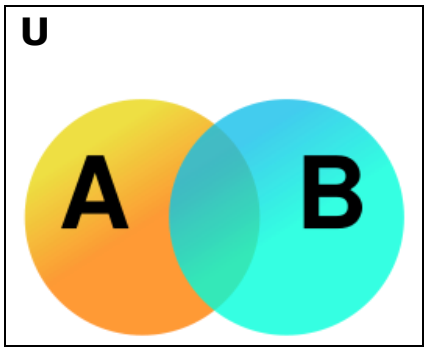
SQL (**S**tructured **Q**uery **L**anguage) is an ANSI (**A**merican **N**ational **S**tandards **I**nstitute) programming language for creating, updating, and retrieving information that resides in a database. Databases are made up of tables and each table contains records (rows) and the records contain columns (fields) where data resides. There are often multiple tables in a RDBMS (**R**elational **D**ata**b**ase **M**anagement **S**ystem) and each table has a unique name and each column in the table is also unique.

With SQL you can turn ordinary questions ("Where do the clients that we serve reside?") into a SQL statement that the database understands: `select City, State from ClientAddresses`

This class will utilize the DBMS (Database Management System) – Microsoft SQL Server and SQL Management Studio as a tool to create and execute SQL statements.

RELATIONAL MODEL

The relational model represents data in the form of tables. Each table may represent some real-world person, place, thing, or event about which information is collected. In the following example the rectangle (U) represents the database and the circles (A and B) inside represent tables. The relative position and overlap of the circles indicate relationships between the tables.



TABLES

A table is the database structure that holds data and is a two-dimensional grid that has columns and rows.

An excerpt of the dbo.Clients table, which is in the eV database, is displayed below:

id	ClientCode	BirthDate	SSN	PreferredName
20210512052654461C90724DE8C22487B983	JSMITH1293	06/26/1990	875454555	John

COLUMNS

Each column represents a specific attribute of the client. In the dbo.Clients table, ClientCode shows the unique Client Code of the client within the same row. JSMITH1293 is John Smith's Client Code.

ROWS

Each row describes a fact about the client which is a unique instance of that client. Each row contains a value or null for each of the table's columns. No two rows in a table can be identical. Each row in a table is identified uniquely by its primary key: ID.

PRIMARY KEYS

Primary Keys are required within each table of a database and are unique within the row.

Each time a client is created within echoVantage a dbo.Clients record is created and assigned a unique ID value.

FOREIGN KEYS

A column (or group of columns) in a table whose values relate to, or reference, values in some other table to ensure that rows in one table have corresponding rows in another table.

Client is an example of a **Foreign Key** or value that relates to the ID (primary key) within the dbo.Clients table. Client is found in many tables within the echoVantage database.

RELATIONSHIPS

A relationship is an association established between common columns in two tables. A relationship can be: One-to-One, One-to-Many or Many-to-Many.

ONE-TO-ONE

In a one-to-one relationship, each row in the first table can have at most one matching row in the second table, and each row in the second table can have at most one matching row in the first table. A one-to-one relationship is established when the primary key of one table (dbo.Clients.id) also is a foreign key referencing the primary key of another table (table.ClientID). *Staff and User is one to one

dbo.Staff	
Id	**StaffCode**
485BBBF09B194BDCB2	EVSTAF394

dbo.AppUsers	
Staff	**UserName**
485BBBF09B194BDCB2	evStaff

ONE-TO-MANY

In a one-to-many relationship, each row in the dbo.Clients table can have many matching rows in the dbo.Services table, but each row in the dbo.Services table can have only one matching row in the dbo.Clients table.

dbo.Clients	
Id	**ClientCode**
485BBBF09B194BDCB2	JSMITH1293

dbo.Services	
Client	**StartDate**
485BBBF09B194BDCB2	01/01/2021
485BBBF09B194BDCB2	01/23/2021
485BBBF09B194BDCB2	03/20/2021

The dbo.Clients.Client is the foreign key and it relates to dbo.Clients.Id which allows you to determine the Services associated to each client.

SQL - BASICS

A SQL script is a sequence of SQL statements executed in order. To write a script, you must know the rules that govern SQL syntax.

COMMENTS

A comment is optional text that explains your SQL. Comments describe what a script does and how or why code was changed. Comments are for humans –the computer ignores them. A comment is represented by either two dashes (--) preceding the comment or you may comment multiple lines with /* at the beginning and */ at the end of the comment. Comments should be used extensively throughout scripts to ease future maintenance.

```
--This is a comment.

/* This is a multi line
comment. */
```

SQL STATEMENT

A SQL statement is a valid combination of tokens introduced by a keyword. Tokens are the basic indivisible particles of the SQL language; they can't be reduced grammatically. Tokens include keywords, identifiers, operators, literals (constants), and punctuation symbols.

CLAUSES

An SQL statement has one or more clauses. In general, a clause is a fragment of an SQL statement that's introduced by a keyword, is required or optional, and must be given in a particular order. SELECT, FROM, WHERE, and ORDER BY introduces four clauses in this example.

KEYWORDS

Keywords are words that SQL reserves because they have special meaning in the language. Using a keyword outside its specific context (as an identifier, for example) causes an error.

IDENTIFIERS

Identifiers are words that are used to name database objects such as tables, columns, aliases, indexes, and views. `ClientCode, DeclinedEthnicity`, dbo.Clients are the identifiers in the example below.

```
SELECT ClientCode, DeclinedEthnicity --select [column_name(s)]
FROM dbo.Clients                     -- from a table
WHERE DeclinedEthnicity = 'N'        --conditional on a [column_name]
ORDER BY ClientCode                  -- order by a [column_name]
```

```
SELECT ClientCode, DeclinedEthnicity
FROM dbo.Clients
WHERE DeclinedEthnicity = 'N'
ORDER BY ClientCode
```

ClientCode	DeclinedEthnicity	
JSMITH1293	N	

DATA TYPES

Each column in a table has a single data type. The data type determines a column's allowable values and the operations it supports. An integer data type, for example, can represent any whole number and supports the usual arithmetic operations: addition, subtraction, multiplication, and division (among others). But an integer can't represent a nonnumeric value such as 'john' and doesn't support character operations such as capitalization and concatenation.

The data type affects the column's sort order. The integers 1, 2, and 10 are sorted numerically, yielding 1, 2, 10. The character strings '1', '2', and '10' are sorted alphabetically, yielding '1', '10', '2'. Alphabetical ordering sorts strings by examining the values of their characters individually. Here, '10' comes before '2' because '1' (the first character of '10') is less than '2' alphabetically.

The following data types are used within the eV database; datetime, date, character (char), variable character (varchar), decimal (amount), and integer (whole number).

USING QUOTES WHEN RETRIEVING DATA

Single quotes are used for Character, Variable Character, Datetime and Date values whereas Numeric and Integer values should not be enclosed in single quotes.

For text values:

```
--Correct usage:
SELECT * FROM dbo.VRaceCodes WHERE ConceptCode =
'2076-8'

--Incorrect Usage:
SELECT * FROM dbo.VRaceCodes WHERE ConceptCode =
2076-8
```

For numeric values:

```
--Correct usage:
SELECT * FROM dbo.Services WHERE ClientDuration
> 50

--Incorrect Usage:
SELECT * FROM dbo.Services WHERE ClientDuration
> '50'
```

RETRIEVING DATA FROM A TABLE USING THE SELECT STATEMENT

The SELECT statement is used to select data from a table. The tabular result is stored in a result table (called the result set).

```
SELECT column_name(s) FROM table_name
```

SELECT COLUMNS

To select the columns "FirstName" and "LastName", use a SELECT statement like this:

```
SELECT FirstName, LastName FROM ClientNames
```

dbo.ClientNames table

FirstName	MiddleName	LastName
John	A	Smith
Jane	A	Smith
John	A	Doe
Jane	A	Doe

THE RESULT SET

The result from a SQL query is stored in a result-set. Most database software systems allow navigation of the result set with programming functions, like: Move-To-First-Record, Get-Record-Content, Move-To-Next-Record, etc.

FirstName	LastName
John	Smith
Jane	Smith
John	Doe
Jane	Doe

CREATING COLUMN ALIASES WITH AS

In the query results so far we've allowed the DBMS to use default values for column headings. (A column's default heading in a result is the source column's name in the table definition.) You can use the AS clause to create a column alias. A *column alias* is an alternative name (identifier) that you specify to control how column headings are displayed within the result set. Use column aliases if column names are cryptic, hard to type, too long, or too short.

A column alias immediately follows a column name in the SELECT clause of a SELECT statement. Enclose the alias in single or double quotes if it's a reserved keyword or if it contains spaces, punctuation, or special characters. You can omit quotes if the alias is a single non-reserved word that contains only letters, digits or underscores.

```
SELECT column AS column_alias FROM table
```

Using a column alias:

EXAMPLE: USING A COLUMN ALIAS

This table (Clients):

```
SELECT ClientCode AS 'Client Code' FROM dbo.Clients
```

Returns this result:

Client Code
John

TABLE NAME ALIAS

An alias may also be used when selecting a table which is discussed in more depth within Joins.

```
SELECT column FROM table AS table_alias
```

SELECT DISTINCT STATEMENT

The DISTINCT keyword is used to return only distinct (different) values. With SQL, add the DISTINCT keyword at the beginning of the SELECT statement:

```
SELECT DISTINCT column_name(s) FROM table_name
```

USING THE DISTINCT KEYWORD

To select ALL values from the column "Name" we use a SELECT statement:

```
SELECT FirstName, LastName FROM dbo.ClientNames
```

Result Set – Client names listed from the dbo.Clients table.

FirstName	LastName
John	Smith
John	Doe

EXAMPLE:

```
SELECT DISTINCT FirstName FROM
dbo.ClientNames
```

Result Set – John is listed once each because it is a distinct values:

FirstName
John

ORDER BY - SORTING THE ROWS

Rows in a result set are unordered, so you should view the order in which rows appears as being arbitrary. This situation arises because order is irrelevant for table operations. You can use the ORDER BY clause to sort rows by a specified column or columns in ascending (lowest to highest) or descending (highest to lowest) order. The ORDER By clause always is the last clause in a SELECT statement.

dbo.ClientNames Table:

Id	FirstName	MiddleName	LastName
485BBBF09B194BDCB2	John	A	Smith
4222CE7C621946DBB6	Jane	A	Smith
3BDA009CFE83449FA4	John	A	Doe

EXAMPLE

To display the clients in alphabetical order:

```
SELECT FirstName, LastName
FROM dbo.ClientNames
ORDER BY FirstName
```

RESULT SET

FirstName	LastName
Jane	Smith
John	Smith
John	Doe

EXAMPLE

To display the clients in reverse alphabetical order (descending):

```
SELECT FirstName, LastName
FROM dbo.ClientNames
ORDER BY FirstName DESC
```

RESULT SET

FirstName	LastName
John	Smith
John	Doe
Jane	Smith

FILTERING ROWS WITH WHERE

The result of each SELECT statement so far has included every row in the table (for specified columns). You can use the WHERE clause to filter unwanted rows from the result set. This filtering capability gives SELECT statement its real power. In a WHERE clause, you specify a search condition that has one or more conditions that need to be satisfied by the rows of a table. A condition is a logical expression that evaluates to true, false, or unknown. (The unknown result arises from NULLs; discussed later.) Rows for which the condition is true are included in the result set; rows which the condition is false, or unknown are excluded. SQL provides operators that express different types of conditions. Operators are symbols or keywords that specify actions to perform on values or other elements.

Use a WHERE clause in the SELECT statement to conditionally select data from a table or to filter the data that is selected.

```
SELECT column FROM table
WHERE column operator value
```

TYPES OF CONDITIONS

Condition	SQL Operators
Comparison	=, <>, <, <=, >, >=, !=
Pattern Matching	LIKE
Range Filtering	BETWEEN
List Filtering	IN
Null Testing	IS NULL OR ISNULL() Function

COMPARISON OPERATORS

Operator	Description
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

USING THE WHERE CLAUSE

To select only the clients where their race is declined, we add a WHERE clause to the SELECT statement:

```
select ClientCode,
DeclinedEthnicity
from Clients
where DeclinedEthnicity = 'Y'
```

Result Set:

ClientCode	DeclinedEthnicity
000001	Y
000002	Y
000003	Y

CONDITIONS WITH AND, OR, AND NOT

You can specify multiple conditions in a single WHERE clause to, say, retrieve rows based on the values in multiple columns. You can use the AND and OR operators to combine two or more conditions into a compound condition. AND, OR, and a third operator, NOT, are logical operators. Logical operators are designed to work with truth values: true, false, and unknown. SQL uses three-value logic, the result of a logical expression is true, false, or unknown. If the result of a compound condition is false or unknown, the row is excluded from the result.

AND

AND connects two conditions and returns true only if both conditions are true. Any number of conditions can be connected with ANDs. All the conditions must be true for the row to be included in the result. AND is independent of order: WHERE condition1 and condition2 is equivalent to WHERE condition2 and condition1. You can enclose one or both of the conditions in parentheses. Some compound conditions need parentheses to force the order in which conditions are evaluated.

EXAMPLE

Use AND to display each person named John and where the race is W:

```
select ClientCode,
DeclinedEthnicity
from Clients
where DeclinedEthnicity
= 'Y'
```

"dbo.Clients" table

Id	FirstName	MiddleName	LastName	Race
485BBBF09B194BDCB2	John	A	Smith	W
4222CE7C621946DBB6	Jane	A	Smith	W
3BDA009CFE83449FA4	John	A	Doe	A

Result Set:

Id	FirstName	MiddleName	LastName	Race
485BBBF09B194BDCB2	John	A	Smith	W
4222CE7C621946DBB6	Jane	A	Smith	W

OR

OR connects two conditions and returns true if *either* condition is true or if *both* conditions are true. Any number of conditions can be connected with ORs. OR will retrieve rows that match any condition or all the conditions. Like AND, OR's order does not matter. You can enclose one or both of the conditions in parentheses.

EXAMPLE

Use OR to display each client named John or where the last name is Doe.

```
SELECT * FROM dbo.ClientNames
WHERE FirstName = 'John'  or LastName = 'Doe'
```

"dbo.Clients" table

Id	FirstName	MiddleName	LastName
485BBBF09B194BDCB2	John	A	Smith

4222CE7C621946DBB6	Jane	A	Doe
3BDA009CFE83449FA4	John	A	Doe

Result Set:

Id	FirstName	MiddleName	LastName
485BBBF09B194BDCB2	John	A	Smith
4222CE7C621946DBB6	Jane	A	Doe
3BDA009CFE83449FA4	John	A	Doe

NOT

Unlike AND and OR, NOT doesn't connect two conditions. Instead, it negates (reverses) a single condition. In comparisons, place NOT before the column name or expression

```
SELECT * FROM dbo.ClientNames
WHERE NOT FirstName = 'John'
```

and not before the operator (even though it sounds better when read):

```
SELECT * FROM dbo. ClientNames
WHERE FirstName NOT = 'John'
```

Not acts on one condition. To negate two or more conditions, repeat the NOT for each condition. In comparisons, using NOT often is a matter of style. The following two clauses are equivalent:

```
SELECT * FROM dbo. ClientNames
WHERE NOT FirstName = 'John'
```

and

```
SELECT * FROM dbo. ClientNames
WHERE FirstName <> 'John'
```

You can enclose the NOT condition in parentheses.

USING AND, OR, AND NOT TOGETHER

You can combine the three logical operators in a compound condition. NOT is evaluated first, then AND, and finally OR. You can override this order with parentheses: everything in parentheses is evaluated first. When parenthesized conditions are nested, the innermost condition is evaluated first. Under the default precedence rules, the condition **x AND NOT y OR z** is equivalent to **((x and (NOT y)) or z)**. It's wise to use parentheses, rather than rely on the default evaluation order, to make the evaluation order clear.

MATCHING PATTERNS WITH THE LIKE CONDITION

The preceding examples retrieved rows based on the exact value of a column or columns. You can use LIKE to retrieve rows based on partial information. LIKE is useful if you don't know an exact value (the client's last name is Smi-something) or you want to retrieve rows with similar values (the client's first name starts with a J). The LIKE condition works with only character strings, not numbers or datetimes. LIKE uses a pattern that values are matched against. A *pattern* is a quoted string that contains the literal characters to match any combination of wildcards. *Wildcards* are special characters used to match parts of a value.

Operator	**Matches**
%	A percent sign matches any string of zero or more characters.
_	An underscore matches any one character

Examples of % and _ Patterns

Operator	**Matches**
A%	Matches a string of lengths >= 1 that begins with *a*, including the single letter A. Matches 'A', 'Attire' and 'AC/DC'.
%s	Matches a string of lengths >=1 that ends with *s*, including the single letter s. A string with trailing blanks (after the s) won't match. Matches 's', 'Falls', and 'DBMSes'
%in%	Matches a string of length >= 2 that contains *in* anywhere. Matches 'in', 'inch', 'Pine' and 'linchpin'.

'____'	Matches any four-character string. Matches 'ABCD', 'I AM', and 'Jack'.
'Qua__'	Matches any five-character string that begins with *Qua.* Matches 'Quack', 'Quaff', and 'Quake'.
'_re%'	Matches a string of length >=3 that begins with any character and has *re* as its second and third characters. Matches 'Tree', 'area', 'fret' and 'are'.

EXAMPLE

The following SQL statement will return clients with last names that begin with a 'D':

```
SELECT LastName
FROM dbo.ClientNames
WHERE LastName like 'S%'
```

dbo.Clients table

Id	FirstName	MiddleName	LastName	Race
485BBBF09B194BDCB2	John	A	Doe	W
4222CE7C621946DBB6	Jane	A	Smith	W
3BDA009CFE83449FA4	John	A	Doe	A

Result Set:

LastName
Doe
Doe

RANGE FILTERING WITH THE BETWEEN OPERATOR

Use BETWEEN to determine whether a given value falls within a specified range. The BETWEEN operator works with character strings, numbers and datetimes. The between range contains a low value and a high value, separated by AND. The low value must be less than or equal to the high value.

BETWEEN is a convenient, shorthand clause that you can replicate by using AND.

```
WHERE testcolumn BETWEEN low_value AND high_value
```

is equivalent to:

```
WHERE (testcolumn >= low_value) AND (testcolumn <= high_value)
```

BETWEEN specifies an *inclusive range*, in which the high value and low value are included in the search. To specify an *exclusive range*, which excludes end points, use > and < comparisons instead of BETWEEN. You can negate a BETWEEN condition with NOT BETWEEN. You can combine BETWEEN conditions with AND and OR.

EXAMPLE

The following SQL statement will return clients born in the year 1998.

```
SELECT ClientCode, BirthDate
FROM dbo.Clients
WHERE BirthDate BETWEEN '01/01/1998' and '12/31/1998'
```

RESULT SET

ClientCode	BirthDate
000001	05/01/1998
000004	03/01/1998
000006	06/26/1998

LIST FILTERING WITH THE IN CONDITION

Use IN to determine whether a given value matches any value in a specified list. The IN condition works with character strings, numbers, and datetimes. The IN list is a parenthesized listing of one or more comma-separated value. The list items needn't be in any order. IN is a convenient, shorthand clause that you can replicate by using OR.

```
WHERE TestColumn IN (value1, value2, value3)
```

is equivalent to:

```
WHERE TestColumn = value1 or TestColumn = value2 or TestColumn = value3
```

You can negate an IN condition with NOT IN. You can combine IN conditions and other conditions with AND and OR.

EXAMPLE

The following SQL statement will return clients where the last name is Doe or Smith.

```
SELECT FirstName, LastName FROM dbo.ClientNames
WHERE LastName IN ('Doe', 'Smith')
```

DBO.CLIENTNAMES TABLE

Id	FirstName	MiddleName	LastName
485BBBF09B194BDCB2	John	A	Doe
4222CE7C621946DBB6	Jane	A	Smith
3BDA009CFE83449FA4	John	A	Doe
89DB23309D239DDIKH	Daffy	B	Duck

RESULT SET

FirstName	LastName

John	Doe
Jane	Smith
John	Doe

TROUBLESHOOTING WHERE

If the WHERE clause isn't working, you can debug it by displaying the result of each condition individually. To see the result of each comparison, put each comparison expression in the SELECT clause's output column list, along with the values you're comparing:

```
SELECT FirstName,
                FirstName = 'John',
                LastName,
                LastName = 'Doe'
FROM dbo.ClientNames
```

To verify the following:

```
SELECT FirstName, LastName
FROM dbo.ClientNames
WHERE FirstName = 'John'
        or LastName = 'Doe'
```

AGGREGATE FUNCTIONS/GROUP BY /HAVING

Aggregate functions look at a set of values and perform a calculation to return a single value. Count is the only one that does not ignore null values. Aggregate functions are often used with GROUP BY at the end of a SELECT.

Aggregate functions are allowed as expressions only in:

- The select list of a SELECT statement (either a subquery or an outer query).
- A HAVING clause.

AVG(ALL | DISTINCT expression) - Returns the average of the values in a numeric expression

COUNT(ALL | DISTINCT expression) - Number of values in the expression

COUNT(*) - Returns the number selected rows

MAX(expression) - Highest value in the expression

MIN(expression) - Lowest value in the expression

SUM(ALL | DISTINCT expression) - Returns the sum of all the values, or only the DISTINCT values, in the expression

**SUM, AVG, COUNT, MAX, and MIN ignore null values; COUNT(*) does not.

The optional keyword DISTINCT can be used with SUM, AVG, and COUNT to eliminate duplicate values before an aggregate function is applied (the default is ALL).

SUM and AVG are used with numeric values like int, decimal, numeric and money data types. MIN and MAX can be used with any data types. Aggregate functions other than COUNT(*) cannot be used with text and image data types.

GROUPING ROWS WITH GROUP BY

GROUP BY... was added to SQL because aggregate functions (like SUM) return the aggregate of all column values every time they are called, and without the GROUP BY function it was impossible to find the sum for each individual group of column values. The GROUP BY clause comes after the WHERE clause and before the ORDER BY clause. Grouping columns can be column names or derived columns. No columns from the table can appear in an aggregate query's SELECT clause unless they're also included in the GROUP BY clause. When using a case statement or some other complex nonaggregate expression, the GROUP BY expression must match the SELECT expression exactly. The only part of the expression that should not go in the GROUP BY clause is the alias if one is used on the column. If a grouping column contains a NULL that row becomes a group in the result. Use a WHERE clause in a query to eliminate rows before grouping occurs. Order by can be used and should contain the alias given to the aggregate column.

Syntax for the GROUP BY function:

```
SELECT testcolumn, SUM(testolumn2) FROM test_table GROUP BY testcolumn
```

EXAMPLE

Dbo.Services table

Client	Duration
JSMI012345	1.5
JSMI012345	.5
JDOE012347	3
JDOE012347	1

SQL

```
select Client, sum(ClientDuration)  as ClientDuration
from Services
```

```
Msg 8120, Level 16, State 1, Line 172

Column 'Services.Client' is invalid in the select list because it is not contained
in either an aggregate function or the GROUP BY clause.
```

The GROUP BY clause will solve this problem:

```
select Client, sum(ClientDuration)  as ClientDuration
from Services
group by Client
```

Returns the total time per client:

Client	ClientDuration
JSMI012345	2.0
JDOE012347	4.0

FILTERING GROUPS WITH HAVING

The HAVING clause sets conditions on the GROUP BY clause similar to the way that WHERE interacts with SELECT. The HAVING clause comes after the GROUP BY clause and before the ORDER BY clause. The WHERE search condition is applied before grouping occurs and HAVING search condition is applied after. HAVING syntax is similar to the WHERE syntax, except that HAVING can contain aggregate functions. A HAVING clause can reference any of the item s that appear in the SELECT list.

The sequence in which the WHERE, GROUP BY, and HAVING clauses are applied is:

1. The WHERE clause filters the rows that result from the operations specified in the FROM and JOIN clauses.
2. The GROUP BY clause groups the output of the WHERE clause.
3. The HAVING clause filters rows from the grouped result.

The syntax for the HAVING function:

```
select testolumn1, SUM(testolumn2)
from test_table
group by testolumn1
having sum(testolumn2) condition value
```

Dbo.Services table

ClientCode	ClientDuration_n
JSMI012345	1.5
JSMI012345	.5
JDOE012347	3
JDOE012347	1

Using the HAVING clause

```
select client, sum(clientduration)
from dbo.services
group by client
having sum(clientduration) > 30
```

Returns the total time per client where the total duration of time is greater than 30:

ClientCode	ClientDuration
JDOE012347	4.0

JOINS

All the queries so far have retrieved rows from one table. When reporting it's often that we'll need to gather data from multiple tables simultaneously. A join is a table operation that uses related columns to combine rows from two input tables into one result table. You can join as many tables as necessary.

Why do joins matter? The most important database information isn't stored in the rows of individual tables; rather, it's the implied relationships between sets of related rows.

QUALIFING COLUMNS

Qualifying column names becomes necessary when the same column exists within multiple tables that will be joined in the same SQL statement. To identify an otherwise-ambiguous column uniquely in a query that involves multiple tables, use it's qualified name. A qualified name is a table name followed by a dot and the name of the column in the table. Because tables must have different names within a database, a qualified name identifies a single column uniquely within the entire database. table.column is a qualified column.

Because Clientcode exists in both the dbo.Clients and dbo.services table, it must be qualified when joining the two tables.

Example

```
select c.ClientCode
       , s.StartDate
       , cn.FirstName
       , cn.LastName
from dbo.Clients c
join Services s on s.Client = c.id
join dbo.ClientNames cn on cn.Client = c.id
```

You can mix qualified and unqualified names within the same statement. Notice that the only column that needed to be qualified is clientcode. This is because it exists within both tables and would be ambiguous unless qualified. It's recommended to qualify all columns when joining tables to ensure that changes to a table's structure don't introduce ambiguities.

TABLE ALIAS

Table aliases are often used when joining tables. A table alias is required if joining the same table multiple times as each table must be unique. A couple other reasons to use a table alias is that it saves typing and reduces statement clutter.

Example:

```
select *
from dbo.episodestaff as es
inner join dbo.staffroles as ps on ps.primarystaffid = es.staffrole
inner join dbo.staffroles as ss on ss.primarystaffid = es.staffrole
inner join dbo.staffroles as ts on ts.primarystaffid = es.staffrole
```

USING JOINS

As discussed previously, tables in a database are related to each other with keys. A primary key is a column with a unique value for each row. A foreign key links to a primary key in another table. The purpose is to bind data together, across tables, without repeating all of the data in every table.

In the dbo.Clients table below, the id column is the primary key, meaning that **no** two rows can have the same id. The id distinguishes two clients even if they have the same name.

When you look at the example tables below, notice that:

- The id column is the primary key of the dbo.Clients table
- The clientid column is a foreign key within the dbo.services table
- The clientid column in the dbo.services table is used to refer to clients in the dbo.Clients table.

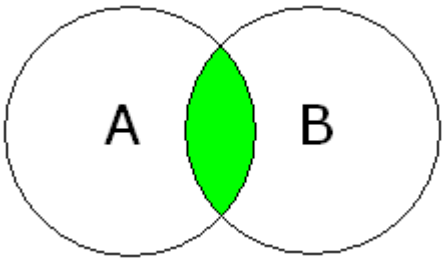
DBO.CLIENTS TABLE

Id	FirstName	MiddleName	LastName	Race
485BBBF09B194BDCB2	John	A	Smith	W
4222CE7C621946DBB6	Jane	A	Smith	AA
3BDA009CFE83449FA4	John	A	Doe	A

DBO.SERVICES TABLE

ClientID	ActivityDate_d	Starttime_t	Endtime_t	Program
485BBBF09B194BDCB2	01/01/2012	8.00	8.30	AOD
4222CE7C621946DBB6	02/01/2012	10.00	11.00	MH
3BDA009CFE83449FA4	03/01/2012	11.00	12.00	MH

INNER JOIN



The A circle represents all records within one table and the B circle represents all records in another table. Notice how there is a bit of overlap between A and B in the middle. The overlapping section is what will pull from the two tables when you perform an INNER JOIN. An inner join only returns those records that have "matches" in both tables. So for every record in A, you will also get the record linked by the foreign key in B. I used green to signify the records that would be returned by the inner join. In SQL think in terms of AND.

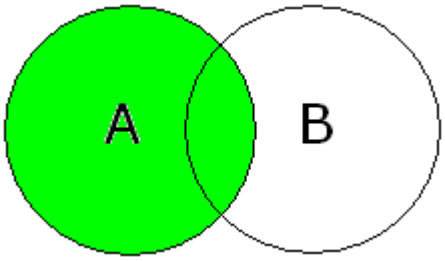
We can select data from two tables with the INNER JOIN keyword, like this:

```
SELECT field1, field2, field3
FROM first_table
INNER JOIN second_table ON first_table.keyfield = second_table.foreign_keyfield
```

The INNER JOIN returns all rows from both tables where there is a match. If there are rows in dbo.Clients that do not have matches in dbo.services, those rows will **not** be listed.

LEFT JOIN

A left join returns all records on the "left" table (A) whether they have matching records in the "right" table (B) or not. If, however, they do have a match in the right table it will return the matching records in that table.




```
SELECT field1, field2, field3
FROM first_table LEFT JOIN second_table ON first_table.keyfield = second_table.foreign_keyfield
```