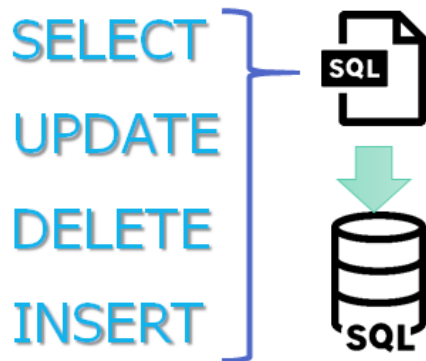




(U4284) Python程式設計 SQL Statement



Speaker : 吳淳硯

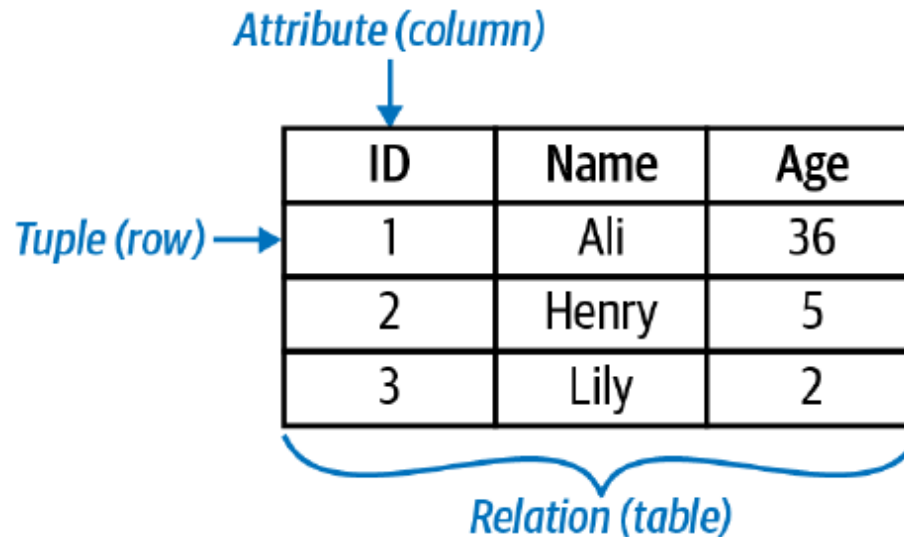
What is Database (DB)

- A database is a place to store data in an organized way. The two categories that database fall into are
 - **SQL.**
 - **NoSQL.**



SQL

- SQL is short for *Structured Query Language*.
- SQL databases are often called *relational database* because they are made up of relation, which are more commonly referred to as tables. Many tables connected to each other make up a database.



- The main thing to note about SQL databases is that they require *predefined schemas*. You can think of a scheme as the way that data in database is *organized or structured*.

NoSQL

- NoSQL stands for *not only SQL*. NoSQL databases are often referred to as *non-relational database*, and they come in all shapes and sizes.
- The main characteristics are that
 - they have *dynamic schemas* (schema doesn't have to be locked in up front)
 - they allow for *horizontal scaling* (data can spread across multiple machines).

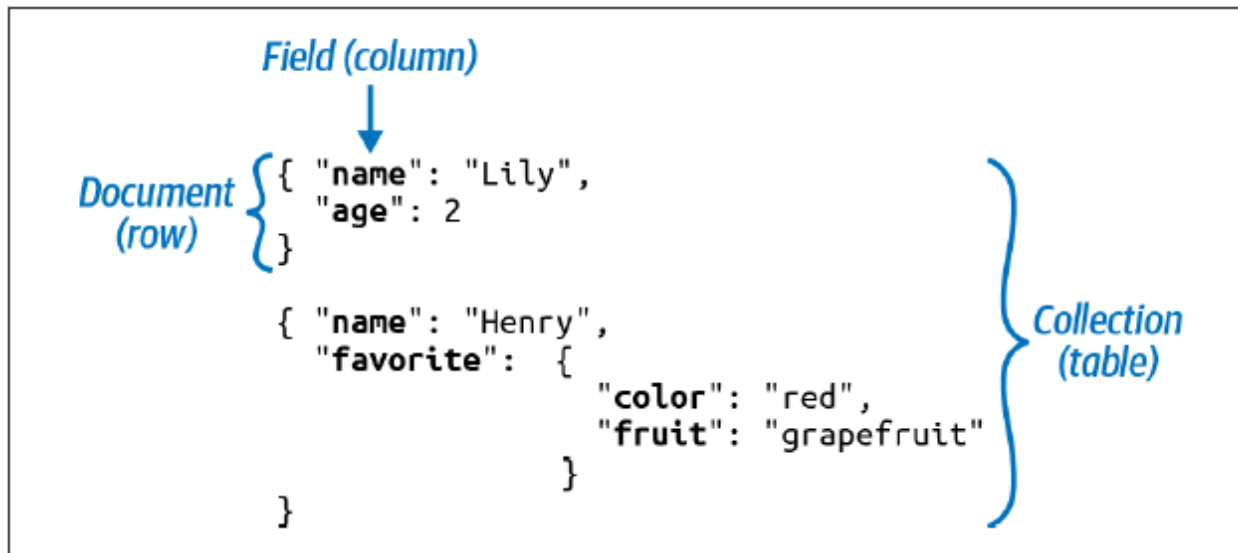


Figure 1-2. A collection (a variant of a table) in *MongoDB*, a NoSQL database

Six Main Clause

- For the most part, the **SELECT** and **FROM** clauses are required and all other clauses are optional.
- The exception is if you are selecting a particular database function, then only the SELECT is required.

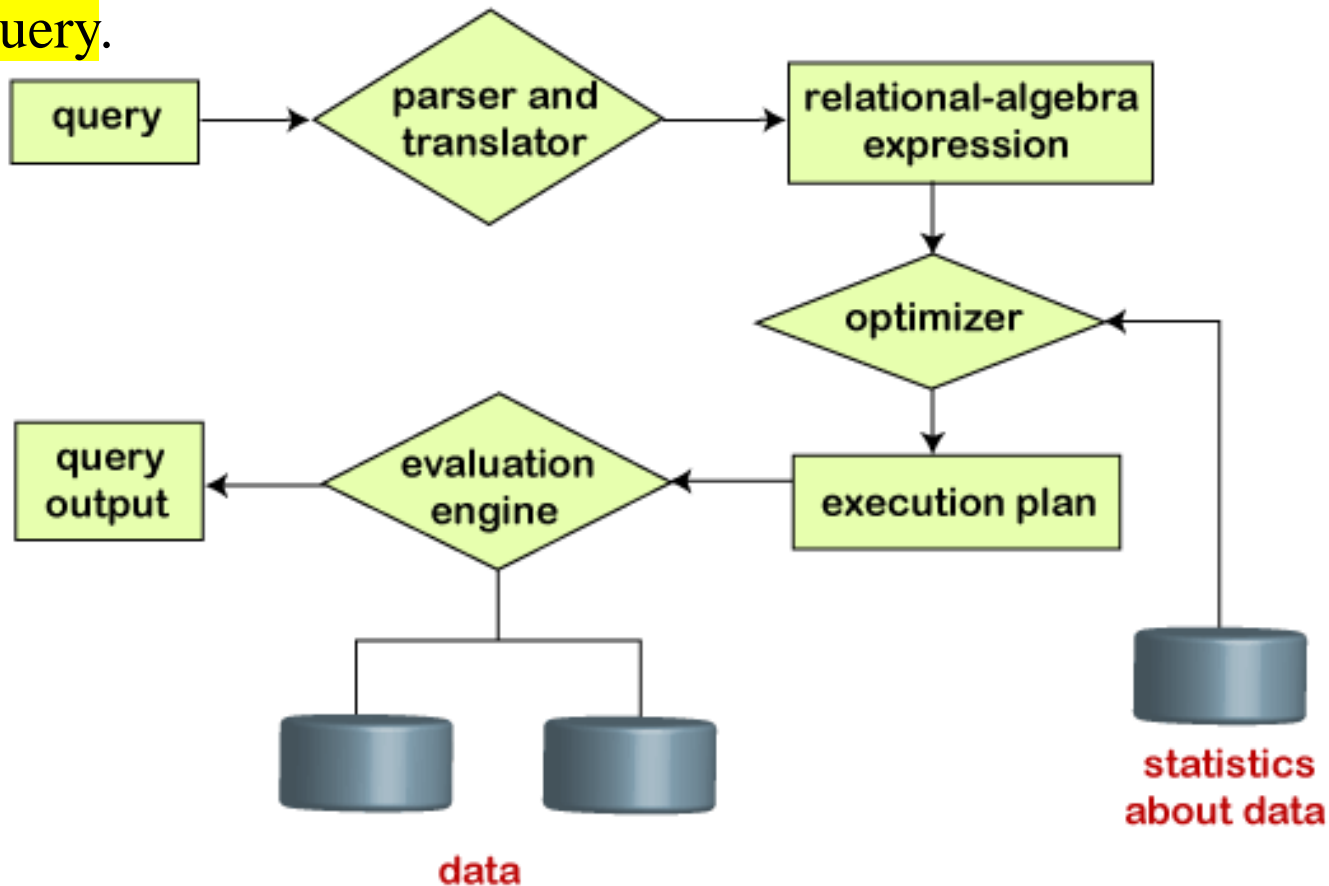
Memorize This Order

All SQL queries will contain some combination of these clauses.
If you remember nothing else, remember this order!

SELECT	-- columns to display
FROM	-- table(s) to pull from
WHERE	-- filter rows
GROUP BY	-- split rows into groups
HAVING	-- filter grouped rows
ORDER BY	-- columns to sort

Why Relation Algebra?

- Consequently, **SQL is the best suitable choice for humans**. But, it is not perfectly suitable for the internal representation of the query to the system. **Relational algebra is well suited for the internal representation of a query.**



Steps in query processing

SQL to Relational Algebra

- A basic SQL query is of the form

SELECT a_{i_1}, \dots, a_{i_L}
FROM R_1, \dots, R_M
WHERE Cd_1, \dots, Cd_K

R_i for $1 \leq i \leq M$ are relation names,
 a_{i_r} for $1 \leq r \leq L$ are attribute names,
 Cd_k for $1 \leq k \leq K$ are atomic constraints.

- Each basic SQL query can be translated to a relational algebra expression involving the project (π), select (σ), and product (\times) operators as follows:

$$\pi_{a_{i_1}, \dots, a_{i_L}} \left(\sigma_{Cd_1} \left(\dots \left(\sigma_{Cd_K} (R_1 \times R_2 \times \dots \times R_M) \right) \right) \right)$$

Search Tips

- When searching for SQL syntax online, always include the RDBMS you are working with in the search.
- Each RDBMS has a different implementation of SQL, meaning that the syntax varies slightly from software to software.



MySQL, PostgreSQL, and SQLite

```
SELECT * FROM birthdays LIMIT 10;
```

Microsoft SQL Server

```
SELECT TOP 10 * FROM birthdays;
```

Oracle Database

```
SELECT * FROM birthdays WHERE ROWNUM <= 10;
```


RDBMS

RDBMS	Owner	Highlights
Microsoft SQL Server	Microsoft	<ul style="list-style-type: none">- Popular proprietary RDBMS- Often used alongside other Microsoft products including Microsoft Azure and the .NET framework- Common on the Windows platform- Also referred to as <i>MSSQL</i> or <i>SQL Server</i>
MySQL	Open Source	<ul style="list-style-type: none">- Popular open source RDBMS- Often used alongside web development languages like HTML/CSS/Javascript- Acquired by Oracle, though still open source
Oracle Database	Oracle	<ul style="list-style-type: none">- Popular proprietary RDBMS- Often used at large corporations given the amount of features, tools, and support available- Also referred to simply as <i>Oracle</i>
PostgreSQL	Open Source	<ul style="list-style-type: none">- Quickly growing in popularity- Often used alongside open source technologies like Docker and Kubernetes- Efficient and great for large datasets
SQLite	Open Source	<ul style="list-style-type: none">- World's most used database engine- Common on iOS and Android platforms- Lightweight and great for a small database

Connect Python to Database

- Three steps to connect Python to database:
 - Install a database driver for Python.
 - Set up a database connection in Python.
 - Write SQL code in Python.

RDBMS	Option	Code
SQLite	n/a	No install necessary (Python 3 comes with <code>sqlite3</code>)
MySQL	pip	<code>pip install mysql-connector-python</code>
	conda	<code>conda install -c conda-forge mysql-connector-python</code>
Oracle	pip	<code>pip install cx_Oracle</code>
	conda	<code>conda install -c conda-forge cx_oracle</code>
PostgreSQL	pip	<code>pip install psycopg2</code>
	conda	<code>conda install -c conda-forge psycopg2</code>
SQL Server	pip	<code>pip install pyodbc</code>
	conda	<code>conda install -c conda-forge pyodbc</code>

Define a database connection

RDBMS	Code
SQLite	<pre>import sqlite3 conn = sqlite3.connect('my_new_db.db')</pre>
MySQL	<pre>import mysql.connector conn = mysql.connector.connect(host='localhost', database='my_new_db', user='alice', password='password')</pre>
Oracle	<pre># Connecting to Oracle Express Edition import cx_Oracle conn = cx_Oracle.connect(dsn='localhost/XE', user='alice', password='password')</pre>
PostgreSQL	<pre>import psycopg2 conn = psycopg2.connect(host='localhost', database='my_new_db', user='alice', password='password')</pre>
SQL Server	<pre># Connecting to SQL Server Express import pyodbc conn = pyodbc.connect(driver='{SQL Server}', host='localhost\\SQLEXPRESS', database='my_new_db', user='alice', password='password')</pre>

SQL code in Python

- Get the query result by **pyodbc**

```
cursor = conn.cursor()
cursor.execute('SELECT * FROM test;')
result = cursor.fetchall()
print(result)
```

[(1, 100),
(2, 200)]

- Get the query result by **pandas**

```
# pandas must already be installed
import pandas as pd

df = pd.read_sql(''SELECT * FROM test;'', conn)
print(df)
print(type(df))
```

	id	num
0	1	100
1	2	200

- Close the connection when you are done using the database

```
cursor.close()
conn.close()
```

Select Clause

- The simplest SELECT clause lists **one or more columns** names from tables in the FROM clause.

```
SELECT id, name
FROM owner;
```

id	name

1	Pictured Rocks
2	Michigan Nature
3	AF LLC
4	MI DNR
5	Horseshoe Falls

- To return **all columns** from a table, you can use a **single asterisk (*)** rather than write out each column name.

```
SELECT *
FROM owner
```

id	name	phone	type

1	Pictured Rocks	906.387.2607	public
2	Michigan Nature	517.655.5655	private
3	AF LLC		private
4	MI DNR	906.228.6561	public
5	Horseshoe Falls	906.387.2635	private

Select Expressions

- To simply listing columns, you can also list more **complex expressions** with the SELECT clause to return as columns in the results.

```
SELECT name, ROUND(population * 0.9, 0)
FROM county;
```

name	ROUND(population * 0.9, 0)
Alger	8876
Baraga	7871
Ontonagon	7036

- Expressions in SELECT list **typically refer to columns in the tables** that you are pulling from, but there are exceptions. Ex. A common function that **doesn't refer to any tables** is the one to return the current data.

```
SELECT CURRENT_DATE;
```

```
CURRENT_DATE
```

```
-----  
2021-12-01
```

Aliasing Columns

- The purpose of a column alias is to give a temporary name to any column or expression listed in the SELECT clause.

```
SELECT id, name,  
       ROUND(population * 0.9, 0)
```

```
FROM county;
```

id	name	ROUND(population * 0.9, 0)
2	Alger	8876
6	Baraga	7871
7	Ontonagon	7036

```
SELECT id AS county_id, name,  
       ROUND(population * 0.90, 0) AS estimated_pop
```

```
FROM county;
```

county_id	name	estimated_pop
2	Alger	8876
6	Baraga	7871
7	Ontonagon	7036

Qualifying tables

- Write a query that pulls data from two tables and they both contain a column called name. If you were to just include name in the SELECT clause, the code wouldn't know which table you were referring to.
- To address the issue, you can qualify a column name by its table name. You can give a column a prefix to specify which table it belongs to using dot notation

```
table_name.column_name.
```

- If you qualify a column name by its table name, you can also qualify that table name by its database or schema name.

```
SELECT sqlbook.owner.id, sqlbook.owner.name  
FROM sqlbook.owner;
```

- The preceding code is lengthy since sqlbook.owner is repeated multiple times. To save on typing, you can provide a table alias.

```
SELECT o.id, o.name  
FROM sqlbook.owner o;
```


Subqueries

- A subquery is a query that is nested inside another query.
- In the following, we want to see the average population of all countries. By including a subquery, we are creating a new column in the results for the average population.

```
SELECT id, name, population,  
       (SELECT AVG(population) FROM county)  
       AS average_pop  
FROM county;
```

id	name	population	average_pop
2	Alger	9862	18298
6	Baraga	8746	18298
7	Ontonagon	7818	18298

Caution Note

- A few things to note here:
 - A subquery must be surrounded by **parentheses**.
 - When writing a subquery within the SELECT clause, it is highly recommended that you specify a **column alias**. That way, the column has a simple name in the result
 - There is only one value in the average_pop column that is **repeated across all rows**. When including a subquery within the SELECT clause, the result of the subquery must return a single column and either zero or one row.

```
SELECT AVG(population) FROM county;
```

```
AVG(population)
```

```
-----
```

```
18298
```

- If the subquery returned zero row, then the new column would be filled with NULL values.

Noncorrelated vs Correlated Subqueries

- **Noncorrelated subquery**: subquery does **not refer to the outer query**. The subquery can be run on its own independent of outer query.
- **Correlated subquery**: subquery does refer to values in the outer query. This often significantly **slows down** processing time.
- The following query return the number of waterfalls for each owner. Note the **o.id = w.owner_id** step in subquery reference the owner table in the outer query.

```
SELECT o.id, o.name,  
       (SELECT COUNT(*) FROM waterfall w  
        WHERE o.id = w.owner_id) AS num_waterfalls  
FROM owner o;
```

- A better approach would be rewrite the query with **JOIN**.

```
SELECT  o.id, o.name,  
        COUNT(w.id) AS num_waterfalls  
FROM    owner o LEFT JOIN waterfalls w  
        ON o.id = w.owner_id  
GROUP BY o.id, o.name
```

Distinct

- When a column is listed in the SELECT clause, by default, all of the rows are returned.

```
SELECT o.type, w.open_to_public  
FROM owner o  
JOIN waterfall w ON o.id = w.owner_id;
```

type	open_to_public
-----	-----
public	y
public	y
public	y
private	y
private	y
private	y
private	y
public	y

- If you want to remove duplicate rows from the results, you can use **DISTINCT** keyword

```
SELECT DISTINCT o.type, w.open_to_public  
FROM owner o  
JOIN waterfall w ON o.id = w.owner_id;
```

type	open_to_public
-----	-----
public	y
private	y

Count and Distinct

- To count the number of unique values within a single column, combine the **COUNT** and **DISTINCT** keywords within the SELECT clause.

```
SELECT COUNT(DISTINCT type) AS unique  
FROM owner;
```

```
unique  
-----  
2
```

- To count the number of unique combinations of multiple columns, you can wrap a **DISTINCT** query up as a subquery, and then do a **COUNT** on the subquery.

```
SELECT COUNT(*) AS num_unique  
FROM (SELECT DISTINCT o.type, w.open_to_public  
      FROM owner o JOIN waterfall w  
      ON o.id = w.owner_id) my_subquery;
```

From Clause

- The FROM clause is used to specify the source of the data you want to retrieve.

```
SELECT name  
FROM waterfall;
```

- You can qualify a table or view with either a **database** or **schema** name using the dot notation.

```
SELECT name  
FROM sqlbook.waterfall;
```

- Instead of retrieving from one table, you'll often want to **pull together data from multiple tables**. The most common way to do this is using JOIN clause within FROM clause.

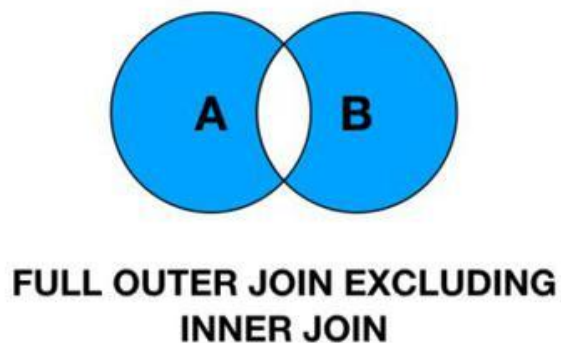
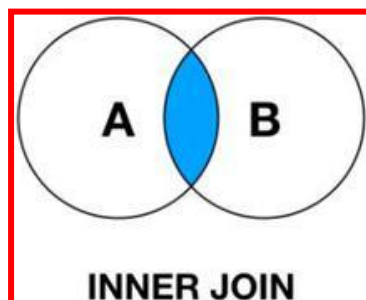
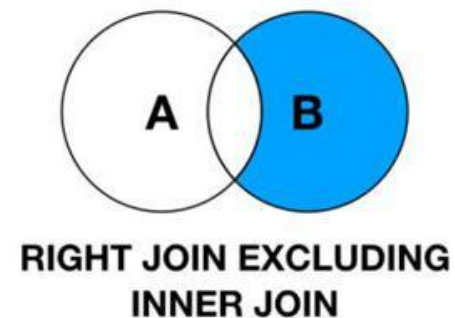
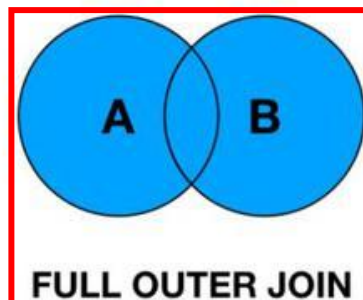
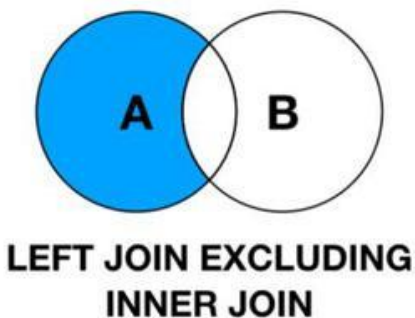
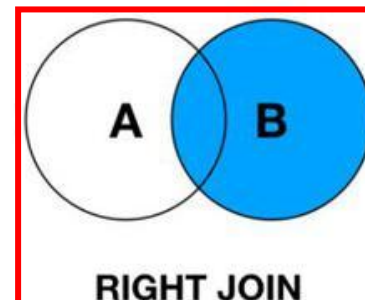
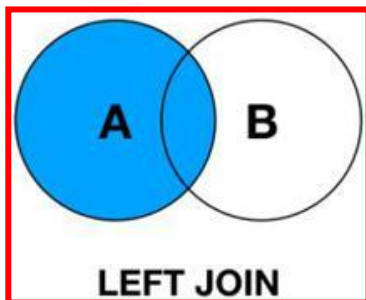
```
SELECT *  
FROM waterfall w JOIN tour t  
ON w.id = t.stop;
```

- These two tables are pulled together with the JOIN keyword. A **JOIN clause is always followed by an ON clause**.

```
JOIN ... ON ...
```

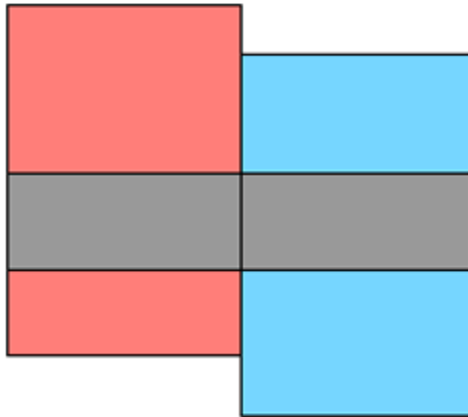
Join Types

SQL JOINS

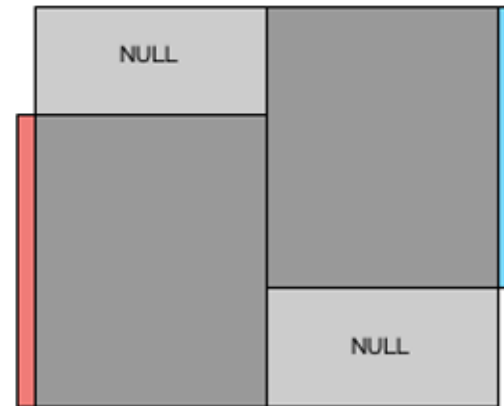


Visual of Join

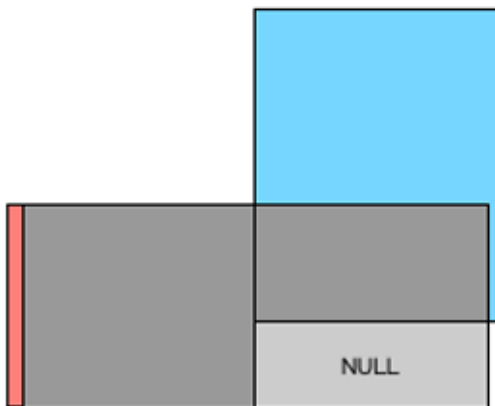
JOIN (INNER JOIN)



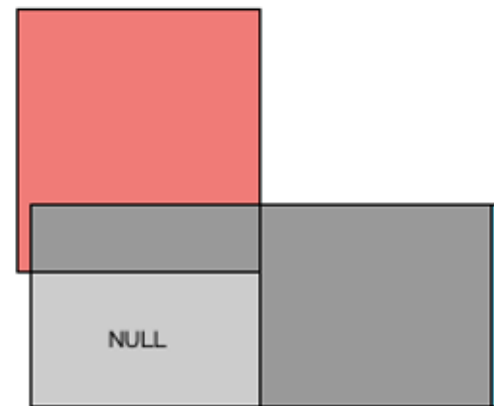
FULL JOIN



LEFT JOIN



RIGHT JOIN



Left Join (A \ B)

```
SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
LEFT JOIN Orders
ON Customers.customer_id = Orders.customer;
```

Table: Customers

customer_id	first_name
1	John
2	Robert
<u>3</u>	David
4	John
<u>5</u>	Betty

Table: Orders

order_id	amount	customer
1	200	10
2	500	<u>3</u>
3	300	6
4	800	<u>5</u>
5	150	8

customer_id	first_name	amount
1	John	
2	Robert	
3	David	500
4	John	
5	Betty	800

Inner Join ($A \cap B$)


```
SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
INNER JOIN Orders
ON Customers.customer_id = Orders.customer;
```

Table: Customers

customer_id	first_name
1	John
2	Robert
<u>3</u>	David
4	John
<u>5</u>	Betty

Table: Orders

order_id	amount	customer
1	200	10
2	500	<u>3</u>
3	300	6
4	800	<u>5</u>
5	150	8



customer_id	first_name	amount
3	David	500
5	Betty	800

Full Outer Join ($A \cup B$)

```
SELECT Customers.customer_id, Customers.first_name, Orders.amount
FROM Customers
FULL OUTER JOIN Orders
ON Customers.customer_id = Orders.customer;
```

Table: Customers

customer_id	first_name
1	John
2	Robert
3	David
4	John
5	Betty

Table: Orders

order_id	amount	customer
1	200	10
2	500	3
3	300	6
4	800	5
5	150	8

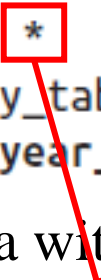
customer_id	first_name	amount
1	John	NULL
2	Robert	NULL
3	David	500
4	John	NULL
5	Betty	800
NULL	NULL	200
NULL	NULL	300
NULL	NULL	150

Where Clause

- The **WHERE clause** is used to restrict query results to only rows of interests, or simply put, it is the place to filter data.

- Filter by a column within the WHERE clause

```
SELECT *  
FROM my_table  
WHERE year_id = 2021;
```



- Show the top few rows of data with Top keyword.

TOP 10 *

- The following query finds all waterfalls that do not contain Falls in the name (**LIKE** keywords).

```
SELECT id, name  
FROM waterfall  
WHERE name NOT LIKE '%Falls%';
```

id	name
7	Little Miners
14	Rapid River Fls

Like Operator

- Use LIKE to match a simple pattern. (%) is a wildcard that means one or more characters.
 - Find all row contain the term you.

```
SELECT *  
FROM my_table  
WHERE txt LIKE '%you%';
```

- Find all rows that start with term you.

```
SELECT *  
FROM my_table  
WHERE txt LIKE 'You%';
```

- In SQL Server, the pattern is case insensitive. Both You and you are captured by '%you%'.
- Use NOT LIKE to return rows that don't contain the characters.
- For more advanced pattern searches, you can use regular expression.

Logical Operators

- Logical operators are used to **modify conditions**, which result in **TRUE, FALSE or NULL**.

```
SELECT *  
FROM employees  
WHERE start_date IS NOT NULL  
      AND (title = 'analyst' OR pay_rate < 25);
```

- When using AND and OR to **combine multiple conditional statements**, it's a good idea to clearly state the order of operations with parentheses ().

Operator	Description
AND	Returns TRUE if both conditions are TRUE. Returns FALSE if either is FALSE. Returns NULL otherwise.
OR	Returns TRUE if either condition is TRUE. Returns FALSE if both are FALSE. Returns NULL otherwise.
NOT	Returns TRUE if the condition is FALSE. Returns FALSE if it is TRUE. Returns NULL otherwise.

Comparison Operators

Table 7-6. Comparison operators (symbols)

Operator	Description
=	Tests for equality
!=, <>	Tests for inequality
<	Tests for less than
<=	Tests for less than or equal to
>	Tests for greater than
>=	Tests for greater than or equal to

Table 7-7. Comparison operators (keywords)

Operator	Description
BETWEEN	Tests whether a value lies within a given range
EXISTS	Tests whether rows exist in a subquery
IN	Tests whether a value is contained in a list of values
IS NULL	Tests whether a value is null or not
LIKE	Tests whether a value matches a simple pattern

Between

- Use BETWEEN to test if a value falls within a range. BETWEEN is a combination of \geq and \leq . The smaller of the two values should always be written 1st, with the AND operator separating the two.
- To find all rows where the ages greater than or equal to 35 and less than or equal to 44.

```
SELECT *  
FROM my_table  
WHERE age BETWEEN 35 AND 44;
```

- To find rows where the ages are less than 35 or greater than 44

```
SELECT *  
FROM my_table  
WHERE age NOT BETWEEN 35 AND 44;
```


Exists

- Use EXISTS to test if a subquery return results or not. Typically the subquery references another table.

```
SELECT e.id, e.name
FROM employees e
WHERE EXISTS (SELECT *
              FROM customers c
              WHERE c.email = e.email);
```

- The EXISTS query could also be written with JOIN.

```
SELECT *
FROM employees e INNER JOIN customers c
ON e.email = c.email;
```

- A JOIN is preferred when you want values from both tables to be returned. An EXISTS is preferred when you want values from a single table to be returned.
- This type of query is sometimes referred to as a semi-join. EXISTS is also useful when the second table has duplicate rows and you're only interested in whether a row exists or not

In

- Use IN to test whether a value falls within a list of values.
 - The following query return value for a few employees.

```
SELECT *  
FROM employees  
WHERE e.id IN (10001, 10032, 10057);
```

- The following query return employees who have not taken a vacation day.

```
SELECT e.id  
FROM employees e  
WHERE e.id NOT IN (SELECT v.emp_id  
FROM vacations v);
```

- When using NOT IN, if there is even a single NULL value in the column in the subquery, the subquery will never be TRUE, meaning no rows will be returned
- If there are potential NULL values in the column in the subquery, it is better to use NOT EXISTS.

IS NULL

- Use **IS NULL** or **IS NOT NULL** to test whether a value is null or not.
- The following query return employees who don't have a manager.

```
SELECT *  
FROM employees  
WHERE manager IS NULL;
```

- The following query return employees who have manager

```
SELECT *  
FROM employees  
WHERE manager IS NOT NULL;
```

Math Operator

- Math operators are math symbols that can be used in SQL.

```
SELECT salary / 52 AS weekly_pay  
FROM my_table;
```

- In SQL Server, dividing an integer by an integer results in a integer.

```
SELECT 15/2;  
  
7
```

- If you want to include decimals, you can either divide by a decimal or use the CAST function.

```
SELECT 15/2.0;  
  
7.5
```

```
-- PostgreSQL and SQL Server  
SELECT CAST(15 AS DECIMAL) /  
       CAST(2 AS DECIMAL);  
  
7.5
```

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)
(MOD in Oracle)	

Math Function

Category	Function	Description	Code	Result
Positive and Negative Values	ABS	Absolute value	SELECT ABS(-5);	5
	SIGN	Returns -1, 0, or 1 depending on if a number is negative, zero, or positive	SELECT SIGN(-5);	-1
Exponents and Logarithms	POWER	x raised to the power of y	SELECT POWER(5,2);	25
	SQRT	Square root	SELECT SQRT(25);	5
	EXP	e (=2.71828) raised to the power of x	SELECT EXP(2);	7.389
	LOG (LOG(y , x) in SQL Server)	Log of y base x	SELECT LOG(2,10); SELECT LOG(10,2);	3.322

Round and Truncate Number

Function	Description	Code	Output
CEIL (CEILING in SQL Server)	Rounds up to the nearest integer	SELECT CEIL(98.7654); SELECT CEILING(98.7654);	99
FLOOR	Rounds down to the nearest integer	SELECT FLOOR(98.7654);	98
ROUND	Rounds to a specific number of decimal places, defaults to 0 decimals	SELECT ROUND(98.7654,2);	98.77
TRUNC (TRUNCATE in MySQL; ROUND(x,y,1) in SQL Server)	Cuts off number at a specific number of decimal places, default to 0 decimals	SELECT TRUNC(98.7654,2); SELECT TRUNCATE(98.7654,2); SELECT ROUND(98.7654,2,1);	98.76

Convert Data type

- The CAST function is used to convert between various data types, and is often used for numeric data.

id	str_col
1	1.33
2	5.5
3	7.8

```
SELECT *
FROM my_table
WHERE str_col > 3;

-- PostgreSQL and SQL Server results
Error
```

- Cast the string column to a decimal column to compare it with a number.

```
SELECT *
FROM my_table
WHERE CAST(str_col AS DECIMAL) > 3;
```

id	str_col
2	5.5
3	7.8

- CAST function does **not permanently change the data type** of the column, it is only for the duration of the query.

String Function

- String function can be applied to columns with **string data types**.
- In SQL Server, use **LEN** instead of **LENGTH**.

In the **SELECT** clause:

```
SELECT LENGTH(name)  
FROM my_table;
```

In the **WHERE** clause:

```
SELECT *  
FROM my_table  
WHERE LENGTH(name) < 10;
```

- Change the Case of a String.

UPPER:

```
SELECT UPPER(type)  
FROM my_table;
```

LOWER:

```
SELECT *  
FROM my_table  
WHERE LOWER(type) = 'public';
```


More on String

- Use CONCAT function.
 - MySQL, PostgreSQL, and SQL Server
 - SELECT CONCAT(id, '_', name) AS id_name
 - FROM my_table;
- There are 2 ways to search for text in a string.
 - LIKE operator.
 - CHARINDEX function.

```
+-----+
| id_name |
+-----+
| 1_Boots |
| 2_Pumpkin |
| 3_Tiger |
+-----+
```

RDBMS	Code Format
MySQL	INSTR(string, substring) LOCATE(substring, string, position)
Oracle	INSTR(string, substring, position, occurrence)
PostgreSQL	POSITION(substring IN string) STRPOS(string, substring)
SQL Server	CHARINDEX(substring, string, position)
SQLite	INSTR(string, substring)

String you are searching in

String you are searching for

Starting position for search

Extract Portion of a String

-- MySQL, PostgreSQL, and SQL Server
 SUBSTRING(*string*, *start*, *length*)

the string you are searching

*the length of the string returned

the starting location of the search

```
+-----+
| my_text |
+-----+
| Here is some text. |
| And some numbers - 1 2 3 4 5 |
| And some punctuation! :) |
+-----+
```

→ SELECT SUBSTR(my_text, 14, 8) AS sub_str
 FROM my_table;

↓

```
+-----+
| sub_str |
+-----+
| text.   |
| ers - 1 |
| tuation! |
+-----+
```

Replace(Delete) Text in a String

```
REPLACE(string, old_string, new_string)
```

```
+-----+  
| my_text |  
+-----+  
| Here is some text. |  
| And some numbers - 1 2 3 4 5 |  
| And some punctuation! :) |  
+-----+
```

Delete Text:
specify an **empty string**
as the replace value.

```
SELECT REPLACE(my_text, 'some', 'the')  
      AS new_text  
FROM my_table;
```

```
+-----+  
| new_text |  
+-----+  
| Here is the text. |  
| And the numbers - 1 2 3 4 5 |  
| And the punctuation! :) |  
+-----+
```

Date or Time

- The following statements return the current data, current time and current date and time.

```
SELECT CAST(CURRENT_TIMESTAMP AS DATE);  
SELECT CAST(CURRENT_TIMESTAMP AS TIME);  
SELECT CURRENT_TIMESTAMP;
```

- There are many other functions equivalent to these including **GETDATE()** in SQL Server
- You can add or subtract various time intervals (years, months, days, hours, minutes, seconds, etc.) from date and time values.
- Return yesterday's date

```
SELECT CAST(CURRENT_TIMESTAMP - 1 AS DATE);  
SELECT DATEADD(DAY, -1, CAST(  
    CURRENT_TIMESTAMP AS DATE));
```

- Return the date and time three hours from now

```
SELECT DATEADD(HOUR, 3, CURRENT_TIMESTAMP);
```

Multiple Predicates

- It is also possible to combine multiple predicates with operators like **AND** or **OR**.

```
SELECT id, name
FROM waterfall
WHERE name NOT LIKE '%Falls'
      AND owner_id IS NULL;
```

- A subquery is a query nested inside another query, and the WHERE clause is a common place to find one.

```
SELECT w.name
FROM   waterfall w
WHERE  w.open_to_public = 'y'
      AND w.county_id IN (
          SELECT c.id FROM county c
          WHERE c.name = 'Alger');
```

- Unlike subqueries within SELECT clause or the FROM clause, subqueries in the WHERE clause do not require an alias.

Working > Optimizing

Working > Optimizing

When writing SQL code, there are often multiple ways to do the same thing.

Your top priority should be to write *working* code. If it takes a long time to run or it's ugly, it doesn't matter...it works!

The next step, if you have time, is to *optimize* the code by improving the performance by perhaps rewriting it with a JOIN, making it more readable with indentations and capitalizations, etc.

Don't stress about writing the most optimized code up front, but rather writing code that works. Writing elegant code comes with experience.


Other ways to Filter Data

- WHERE clause is not the only place within a SELECT statement to filter rows of data
 - FROM clause:
When joining together tables, the ON clause specified how they should be linked together. This is where you can include conditions to restrict rows of data returned by the query.
 - HAVING clause:
If there are aggregations within the SELECT statement, the HAVING clause is where you specify how the aggregations should be filtered.
 - TOP clause:
To display a specific number of rows, you can use the TOP clause.

Group By Clause

- The purpose of the **Group By clause** is to collect rows into groups and summarize the rows within the groups in some way, ultimately returning just one row per group.

```
SELECT    t.name AS tour_name,  
          COUNT(*) AS num_waterfalls  
FROM      waterfall w INNER JOIN tour t  
          ON w.id = t.stop  
GROUP BY t.name;
```



- There are two parts to focus on here
- The collecting of rows**, which is specified within the GROUP by cause.
- The **summarizing of rows**, within groups, which is specified withing the SELECT clause.

Aggregate Function

- An aggregation function performs a calculation on many rows of data and results in a single values.
- Aggregate function apply calculations to non-null values in a column. The only exception is **COUNT(*)**, which counts all rows, including null values.

Function	Description
COUNT()	Counts the number of values
SUM()	Calculates the sum of a column
AVG()	Calculates the average of a column
MIN()	Finds the minimum of a column
MAX()	Finds the maximum of a column

Having Clause

- The **HAVING** clause places restrictions on the row returned from a **group by query**. In other words, it allows you filter on the results after a **GROUP BY** has been applied.
- A **HAVING** clause always immediately follows a **GROUP BY** clause. **Without a GROUP BY clause, there can be no HAVING clause.**
- If we only want to list the tours that have exactly six stops.

```
SELECT    t.name AS tour_name,  
          COUNT(*) AS num_waterfalls  
FROM      waterfall w INNER JOIN tour t  
          ON w.id = t.stop  
GROUP BY t.name  
HAVING   COUNT(*) = 6;
```

Order By Clause

- The **ORDER BY** clause is used to specify how you want the results of a query to be sorted.
- The following query 1st sorted alphabetically by owner and then by waterfall.

```
SELECT COALESCE(o.name, 'Unknown') AS owner,  
       w.name AS waterfall_name  
FROM   waterfall w  
       LEFT JOIN owner o ON w.owner_id = o.id  
ORDER BY owner, waterfall_name;
```

Replace all NULL values in column with a different value.

- The default sort is in ascending order meaning text will go from **A to Z** and numbers will go from **lowest to highest**.
- Use the keywords **ASCENDING** and **DECENDING** (which can be abbreviated as **ASC** and **DESC**)

More on Order Clause

- The following is a modification of the previous sort, but this time, it sorts owner names in **reverse order**.

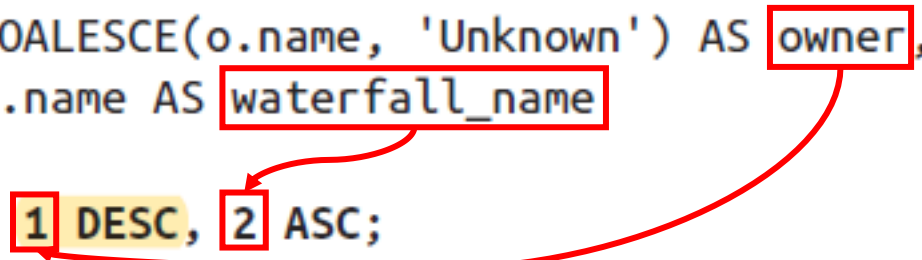
```
SELECT COALESCE(o.name, 'Unknown') AS owner,  
       w.name AS waterfall_name  
...  
ORDER BY owner DESC, waterfall_name ASC;
```

- You can **sort by columns and expression that are not in your SELECT list**.

```
SELECT COALESCE(o.name, 'Unknown') AS owner,  
       w.name AS waterfall_name  
FROM   waterfall w  
       LEFT JOIN owner o ON w.owner_id = o.id  
ORDER BY o.id DESC, w.id;
```

- You can also **sort by numeric column position (implicit)**.

```
SELECT COALESCE(o.name, 'Unknown') AS owner,  
       w.name AS waterfall_name  
...  
ORDER BY 1 DESC, 2 ASC;
```



Caution Note on Order

- Because the rows of a SQL tables are unordered, if you don't include an ORDER BY clause in a query, each time you execute the query, the results could be displayed in different order.

ORDER BY Cannot Be Used in a Subquery

Of the six main clauses, only the ORDER BY clause cannot be used in a subquery. Unfortunately, you can't force the rows of a subquery to be ordered.

To avoid this issue, you would need to rewrite your query with different logic to avoid using an ORDER BY clause within the subquery, and only include an ORDER BY clause in the outer query.

CRUD

- 4 basic database operations
 - Create (C)
 - Read (R)
 - Update (U)
 - Delete (D)
- Display Names of Existing Databases

```
SQL Server  SELECT name FROM master.sys.databases;
```

- In SQL Server, the default database is **master**

```
USE master;  
go  
DROP DATABASE my_new_db;  
go
```

- Display Names of Existing Tables

```
'select name from NTPU_CLASS.sys.tables'
```

Create & Delete Database

- If you delete a database, you will lose all of the data in the database. There is no undo, unless a backup has been created.

RDBMS	Code
MySQL, Oracle, PostgreSQL, SQL Server	CREATE DATABASE my_new_db;
SQLite	> sqlite3 my_new_db.db

- You can't drop a database you are currently in. You would have to 1st switch to another database, before dropping the database.

RDBMS	Code
MySQL, Oracle, PostgreSQL, SQL Server	DROP DATABASE my_new_db;
SQLite	Delete the .db file in the file browser

Creating Table

- Simple Steps

- ▶ Step 1: Create a table

```
CREATE TABLE my_simple_table (  
    id INTEGER,  
    country VARCHAR(2),  
    name VARCHAR(15)  
);
```

- ▶ Step 2: Insert rows

```
INSERT INTO my_simple_table  
    (id, country, name)  
VALUES (2, 'US', 'Selena'),  
    (3, 'CA', 'Shawn'),  
    (4, 'US', 'Sutton');
```

- ▶ After inserting the data

```
SELECT * FROM my_simple_table;
```

id	country	name
1	US	Sam
2	US	Selena
3	CA	Shawn
4	US	Sutton

Null setting & Default

- Not allowing NULL values in a column with **NOT NULL**. By not specifying NULL or NOT NULL, the defaults to NULL.

```
CREATE TABLE my_table (  
    id INTEGER NOT NULL,  
    country VARCHAR(2) NULL,  
    name VARCHAR(15)  
);
```

- Setting default values in a column with **DEFAULT**.

```
CREATE TABLE my_table (  
    id INTEGER,  
    country VARCHAR(2) DEFAULT 'CA',  
    name VARCHAR(15)  
);
```

- Restricting values in a column with **CHECK**.

```
CREATE TABLE my_table (  
    id INTEGER,  
    country VARCHAR(2) CHECK  
        (country IN ('CA', 'US')),  
    name VARCHAR(15)  
);
```

```
CREATE TABLE my_table (  
    id INTEGER,  
    country VARCHAR(2),  
    name VARCHAR(15),  
    CHECK (country IN ('CA', 'US'))  
);
```

Unique

- Requiring unique values in a column with UNIQUE.

```
CREATE TABLE my_table (  
    id INTEGER UNIQUE,  
    country VARCHAR(2),  
    name VARCHAR(15)  
);
```

```
CREATE TABLE my_table (  
    id INTEGER,  
    country VARCHAR(2),  
    name VARCHAR(15),  
    UNIQUE (id)  
);
```

- You can also include logic that forces the combination of multiple columns to be unique.

```
CREATE TABLE my_table (  
    id INTEGER,  
    country VARCHAR(2),  
    name VARCHAR(15),  
    CONSTRAINT uniq_country_name  
    UNIQUE (country, name)  
);
```

Primary Key = Not Null + Unique

- A primary key **uniquely identifies each row of data** in a table. A primary key can be made up of one or more columns in a table.
- Every table should have primary key.

```
CREATE TABLE my_table (  
    id INTEGER PRIMARY KEY,  
    country VARCHAR(2),  
    name VARCHAR(15)  
);
```

```
CREATE TABLE my_table (  
    id INTEGER,  
    country VARCHAR(2),  
    name VARCHAR(15),  
    PRIMARY KEY (id)  
);
```

- To specify a primary key consisting of multiple columns (also known as a **composite key**)

```
CREATE TABLE my_table (  
    id INTEGER NOT NULL,  
    country VARCHAR(2),  
    name VARCHAR(15) NOT NULL,  
    CONSTRAINT pk_id_name  
    PRIMARY KEY (id, name)  
);
```

Insert

- ▶ Insert the Results of a Query into a Table.
 - ▶ Create a new table

```
CREATE TABLE new_table_two_columns (  
    id INTEGER,  
    name VARCHAR(15)  
);
```

id	country	name
1	US	Sam
2	US	Selena
3	CA	Shawn
4	US	Sutton

- ▶ Insert the results from a query in to new table.

```
INSERT INTO new_table_two_columns  
    (id, name)  
SELECT id, name  
FROM my_simple_table  
WHERE id < 3;
```

id	name
1	Sam
2	Selena

Delete & Update

- Use **DELETE FROM** to delete rows of data from table

```
DELETE FROM my_table  
WHERE id = 9;
```

- Omit the WHERE clause to remove all rows from a table

```
DELETE FROM my_table;
```

- Use **UPDATE...SET...WHERE** to update the values in a column of data

id	country	name	awards
2	CA	Celine	5
3	CA	Michael	4
4	US	Stefani	9

```
UPDATE my_table  
SET awards = (SELECT MIN(awards) FROM my_table)  
WHERE country = 'CA';
```

id	country	name	awards
2	CA	Celine	4
3	CA	Michael	4
4	US	Stefani	9

Data Type

- In SQL table, each column can only include values of a single data type.

```
CREATE TABLE my_table (  
    id INT,  
    name VARCHAR(30),  
    dt DATE  
);
```

- When deciding on a data type for a column, it is important to balance storage size and flexibility.
- Below table lists 4 categories of data types, along with common subcategories.

Numeric	String	Datetime	Other
Integer (123)	Character	Date	Boolean
Decimal (1.23)	('hello')	('2021-12-01')	(TRUE)
Floating Point	Unicode	Time ('2:21:00')	Binary (images,
(1.23e10)	('西瓜')	Datetime	documents,
		('2021-12-01	etc.)
		2:21:00')	

Numeric - 0

- Columns with numeric data can be input into numeric function such as **SUM()** and **ROUND()**.
- Numeric values include
 - Integers
 - Decimal numbers
 - Floating point numbers

SQL Server	TINYINT	0 to 255	1 byte
	SMALLINT	−32,768 to 32,767	2 bytes
	INT or INTEGER	−2,147,483,648 to 2,147,483,647	4 bytes
	BIGINT	-2^{63} to $2^{63} - 1$	8 bytes

Numeric - 1

- Decimal numbers are also known as **fixed point** numbers. They include a decimal point and are stored as an exact value.

```
CREATE TABLE my_table (
    my_decimal_column DECIMAL(5,2)
);
INSERT INTO my_table VALUES
    (123.45),
    (-123),
    (12.3);
```

- When defining the type **DECIMAL(5,2)**
 - 5 is maximum of **total digits**. (This is called **precision**)
 - 2 is number of **digits to the right of the decimal point**. (This is called the **scale**)

```
+-----+
| my_decimal_column |
+-----+
|           123.45 |
|        -123.00 |
|           12.30 |
+-----+
```

SQL Server

DECIMAL or
NUMERIC

Total: 38

After decimal point: 38

DECIMAL(18,0)

Numeric - 2

- When a number has many digits, either before or after a decimal point, instead of storing all the digits, floating point number **only store a limited number** of them to save on space.
- There are two floating point data type
 - **Single precision**: number is represented by at least 6 digits, with a full range of around **1E-38 to 1E+38**
 - **Double precision**: number is represented by at least 15 digits, with a full range of around **1E-308 to 1E+308**

```
CREATE TABLE my_table (      INSERT INTO my_table VALUES
  my_float_column FLOAT,      (123.45, 123.45),
  my_double_column DOUBLE    (-12345.6789, -12345.6789),
);                               (1234567.890123456789, 1234567.890123456789);
```

my_float_column	my_double_column
123.45	123.45
-12345.7	-12345.6789
1234570	1234567.8901234567

String - 0

- The most common way to hold string values is to use character data types.
 - **VARCHAR** (*variable character*):
This is most popular string data type. If the data type is VARCHAR(50), then the column will allow up to 50 characters. In other words, the string length is variable.
 - **CHAR** (*character*):
If the data type is CHAR(5), then each value in the column will have exactly 5 characters. In other words, the string length is fixed. Data will be right-padded with spaces to be exactly the length specified.
 - **TEXT**:
Unlike VARCHAR and CHAR, TEXT require no inputs, meaning you do not have to specify a length for the text.

String - 1

SQL Server	CHAR	1 to 8,000 bytes	Input required	Varies
	VARCHAR	1 to 8,000 bytes, or max	Input required	Varies, or up to 2 GB
	TEXT	No inputs	No inputs	2,147,483,647 bytes

- Character data types are typically stored as **ASCII** data, but can also be stored as **Unicode** data if a larger library of characters is needed.
- The following code show the difference between the **VARCHAR** and **NVARCHAR**.

```
CREATE TABLE my_table (
    ascii_text VARCHAR(10),
    unicode_text NVARCHAR(10)
);

INSERT INTO my_table VALUES
('abc', 'abc'),
(N'赵欣婉', N'赵欣婉');
```

```
+-----+-----+
| ascii_text | unicode_text |
+-----+-----+
| abc       | abc         |
| ???       | 赵欣婉      |
+-----+-----+
```

Date & Datetime - 0

- Columns with datetime data can be input into datetime function such as **DATEDIFF()** and **EXTRACT()**.

Data Type	Format	Range
DATE	YYYY-MM-DD	0001-01-01 to 9999-12-31
TIME	hh:mm:ss	00:00:00.0000000 to 23:59:59.9999999
SMALLDATETIME	YYYY-MM-DD hh:mm:ss	<i>Date:</i> 1900-01-01 to 2079-06-06 <i>Time:</i> 0:00:00 through 23:59:59
DATETIME	YYYY-MM-DD hh:mm:ss	<i>Date:</i> 1753-01-01 to 9999-12-31 <i>Time:</i> 00:00:00 to 23:59:59.999
DATETIME2	YYYY-MM-DD hh:mm:ss	<i>Date:</i> 0001-01-01 to 9999-12-31 <i>Time:</i> 00:00:00 to 23:59:59.9999999
DATETIMEOFFSET	YYYY-MM-DD hh:mm:ss +hh:mm	Time zone offset ranges from –12:00 to +14:00

Date & Datetime - 1

- The following create six different datetime columns


```
CREATE TABLE my_table (
    dt DATE,
    tm TIME,
    dttm_sm SMALLDATETIME,
    dttm DATETIME,
    dttm2 DATETIME2,
    dttm_off DATETIMEOFFSET
);
```

```
INSERT INTO my_table VALUES (
    '2021-7-4', '6:30', '2021-12-25 7:00:01',
    '2021-12-25 7:00:01', '2021-12-25 7:00:01',
    '2021-12-25 7:00:01-06:00'
);
```

dt	tm	dttm	dttm2	dttm_sm	dttm_off
2021-07-04	06:30:00.000000000	2021-12-25 07:00:01.000	2021-12-25 07:00:01.00000000	2021-12-25 07:00:00	2021-12-25 07:00:01.00000000 -06:00

Pivoting


- SQL Server support the PIVOT and UNPIVOT operation.
 - PIVOT**: takes a single column and splits it out into multiple columns.
 - UNPIVOT**: takes multiple columns and consolidates them into a single column.



id	name	fruit
1	Henry	strawberries
2	Henry	grapefruit
3	Henry	watermelon
4	Lily	strawberries
5	Lily	watermelon
6	Lily	strawberries
7	Lily	watermelon

```

SELECT *
FROM fruits
PIVOT
(COUNT(id) FOR fruit IN ([strawberries],
                        [grapefruit], [watermelon])
) AS fruits_pivot;
    
```



name	strawberries	grapefruit	watermelon
Henry	1	1	1
Lily	2	0	2

Unpivot

id	name	fruit_one	fruit_two	fruit_thr
1	Anna	apple	banana	
2	Barry	raspberry		
3	Liz	lemon	lime	orange
4	Tom	peach	pear	plum

```

SELECT *
FROM favorite_fruits
UNPIVOT
(fruit FOR rank IN (fruit_one,
                    fruit_two,
                    fruit_thr)
) AS fruit_unpivot
WHERE fruit <> '';

```

id	name	fruit	rank
1	Anna	apple	1
1	Anna	banana	2
2	Barry	raspberry	1
3	Liz	lemon	1
3	Liz	lime	2
3	Liz	orange	3
4	Tom	peach	1
4	Tom	pear	2
4	Tom	plum	3