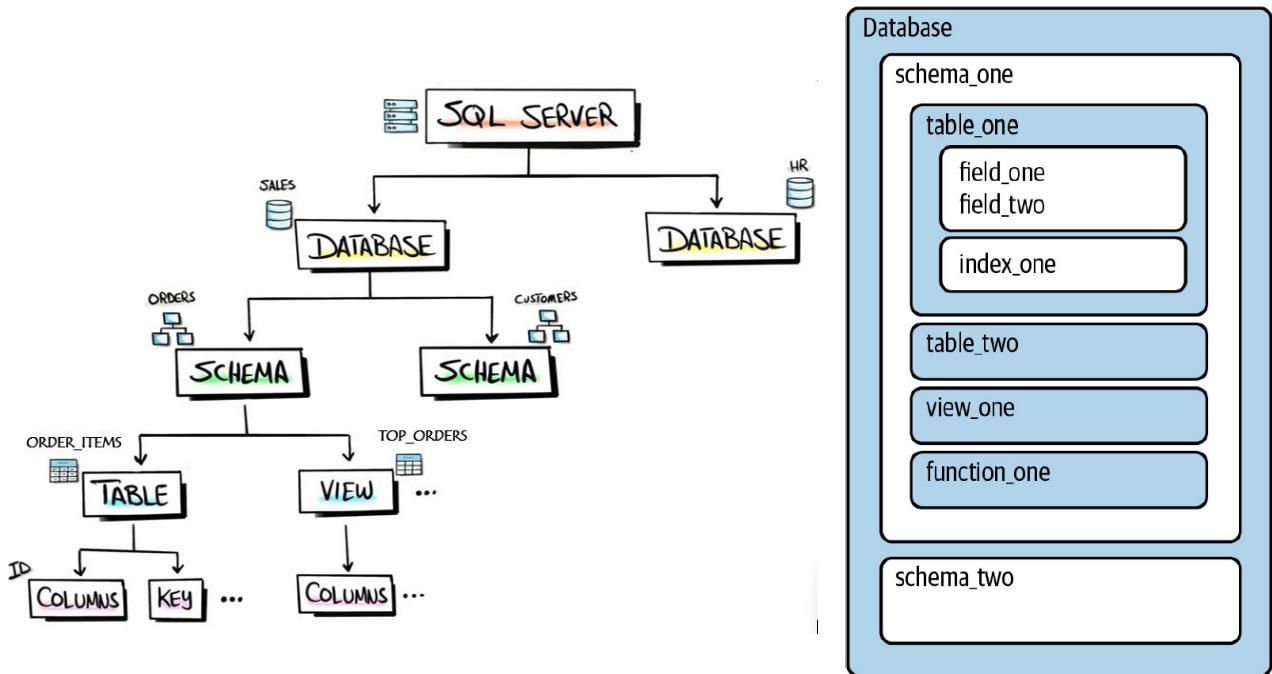


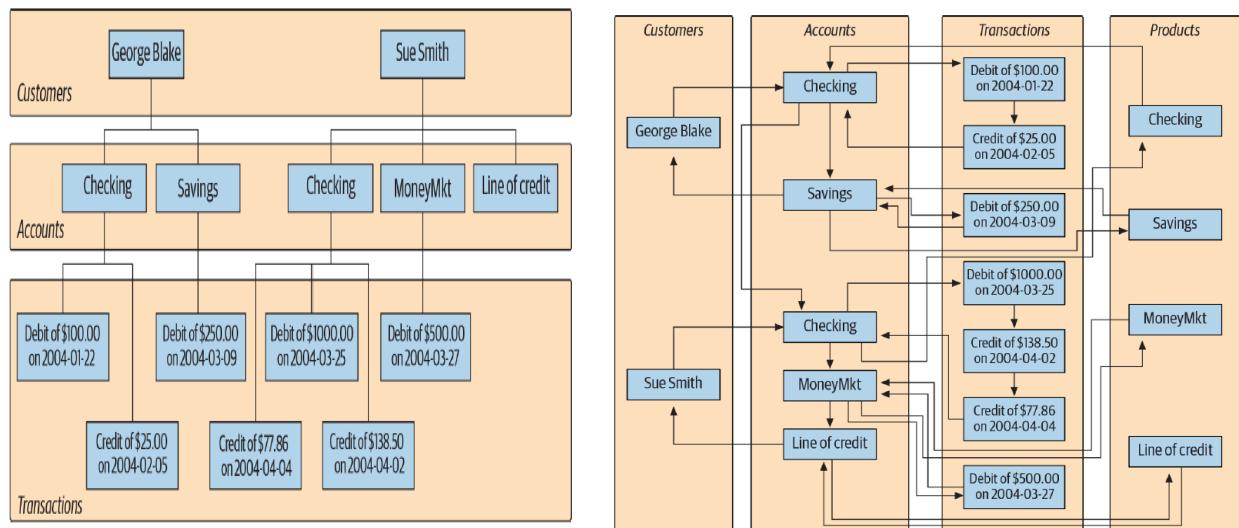
# SQL Database Structure

Database systems, are reliable & systematic computerized data storage and access mechanisms.



- Database schema can be thought as a working template with data, custom functions, and built-in functions- eg. mathematical & ML operations in BigQuery.

Fundamental data management model: file directory (hierarchical) & relational (network)



- There are two main storage formats for storing & managing data: **Row- & Column-Store**  
:: Ideal for transactional data with frequent insertions & updates, or analytical workloads involving large-scale data aggregations.

( Row vs. Col: **optimized for store & retrieval of long rows—ie. samples, but slower for features** )

# COMMON AI USE CASES FOR SQL

- Learn how to use AI as a personal coach, to explain complex SQL code or concepts
- Use AI tools to generate SQL queries from scratch by describing your desired output
- Ask AI tools to help you debug errors in your SQL queries and rewrite the code without errors
- Comment and optimize complex SQL queries with the help of AI tools

#1 Act as a SQL expert and walk me through some of the most common data types used to store numbers in a database. Also, navigate which specific cases they might be used?

#2 Act as a Senior Analyst who is an expert in SQL and explain the following query to me.

#3 Act as a SQL expert and add comments to the following queries to make it easier for other analysts to understand?

---

( SQL statements ... )

#4 Can you help tell me what might be wrong?

The error message is below ...

---

( SQL error response ... )

The full sql query is below ...

---

( SQL statements ... )

#5 I have 2 tables that I can work with. (\*\* Optional: provide data modeling / EER diagram)

The 1<sup>st</sup> table is called Students. It has the following columns ...

student\_id | created\_time | segment | student\_goal

The 2<sup>nd</sup> table is called Revenue. It has the following columns ...

revenue\_id | dollars\_usd | refunded\_dollars\_usd | student\_id | created\_time

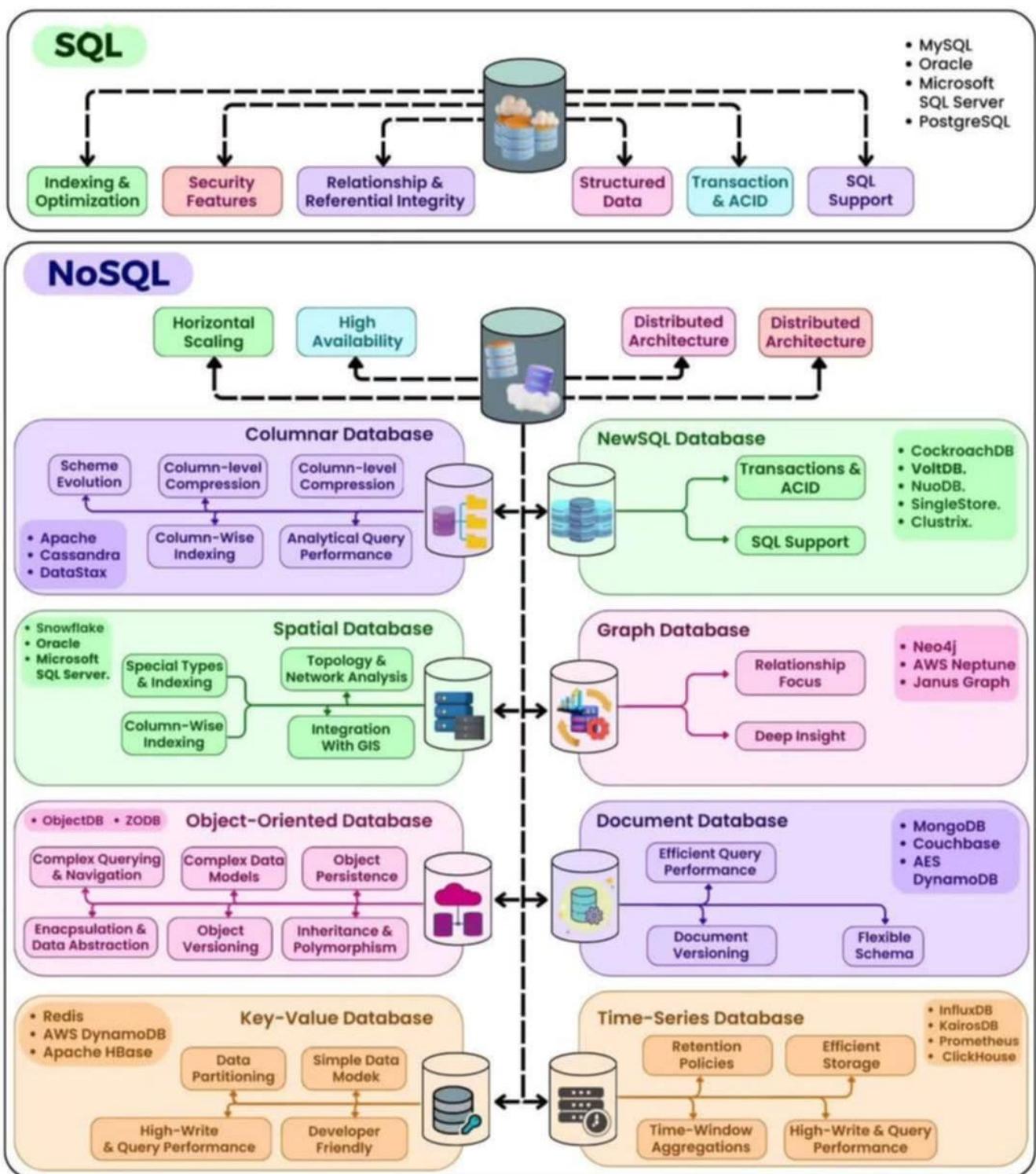
Can you create a query to pull a list of the most valuable students in terms of how much revenue they have generated?

#6 Act as a SQL expert, and help me understand how to optimize my SQL query.

Write the query in the most efficient way, and explain to me the changes you made like I'm someone who is pretty new to SQL ...

---

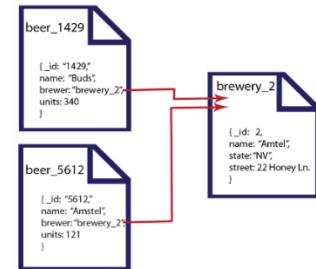
# TYPES OF DATABASES



*Non-relational databases & search-based data models* are typically used for low latency applications and searching of log [unstructured data] files such as—

(1) Key-value store

- ✓ data is serialized as collection of key-value pairs where both can be anything from simple int/string to JSON document.
- ✗ Not fit for:
  - query on specific data value
  - need relationship btw records & multiple unique keys



(2) Document-store

- ✓ serialize each record and its associate data into a single document, enabling flexible indexing, powerful ad hoc queries & analytics over collections of documents.
- ✗ Not fit for:
  - running complex search query
  - performing multi-operation transactions

(Preferred for eCommerce / CRM platform, and medical records storage)

(3) Column-store

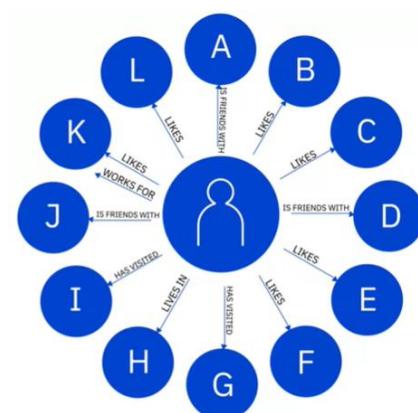
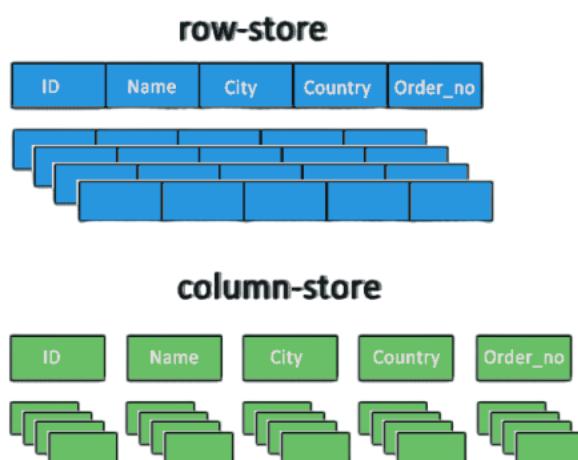
- ✓ serializing data into columns w/o enforcing primary keys & indexes to join tables, and optimized for row-intensive query at few columns, and large data repositories.  
( where it is aligned w/ continuous disk entry, allowing easier : faster access & search )
- ✗ Not fit for:
  - running complex search & changing patterns query frequently
  - frequent updates & deletes, as a record is distributed across multiple columns

(Preferred for heavy write requests, time-series & IoT data)

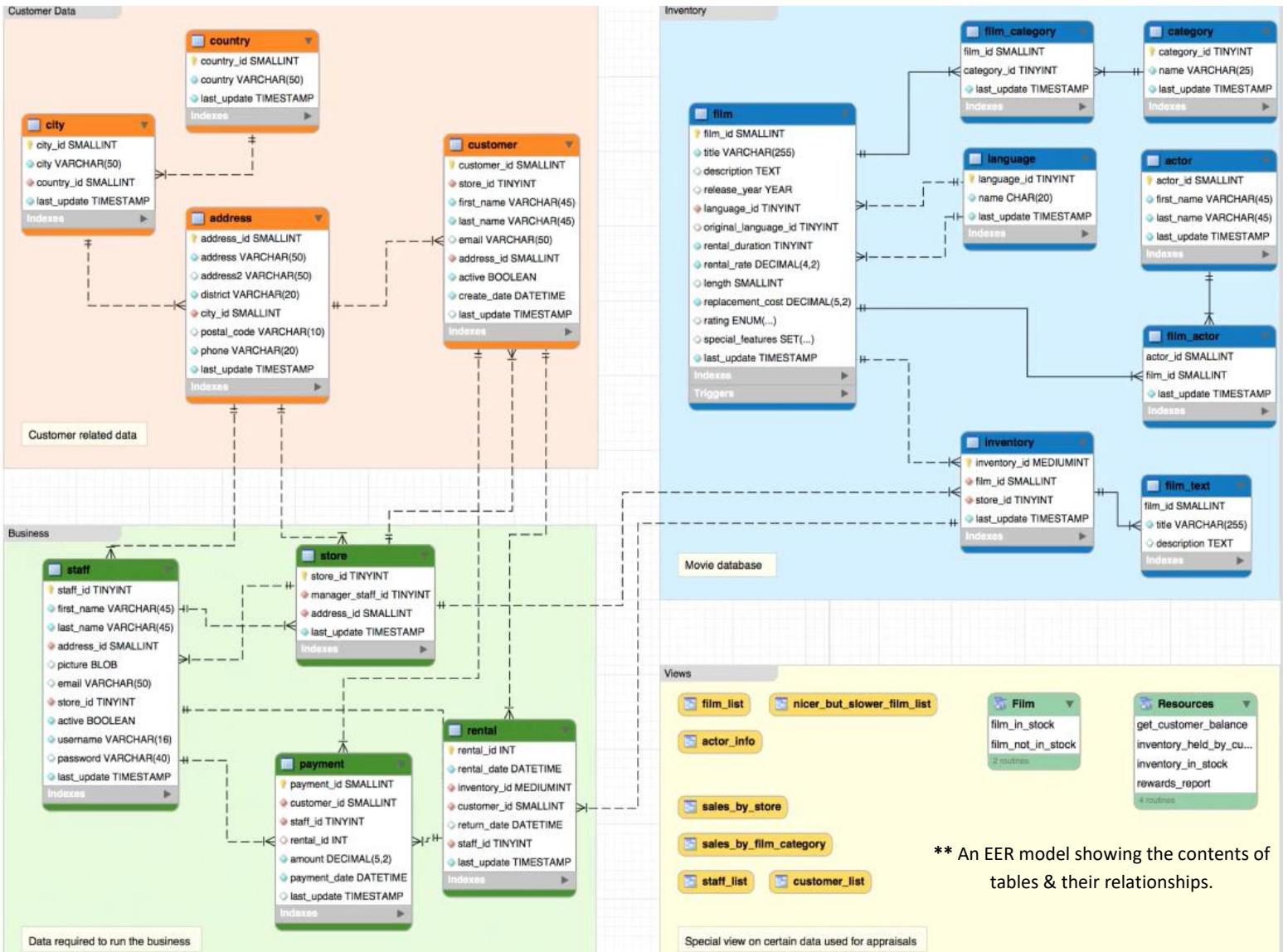
(4) Graph-store

- ✓ uses a graphical model : nodes to represent & serialize data  
( which is useful for visualizing and analyzing connections btw records )
- ✗ Not fit for:
  - processing high volume of transactions

(Preferred for product recommendations, Fraud detection & Access management)



# DATABASE DESIGN & MODELLING



\*\* An EER model showing the contents of tables & their relationships.

When designing / exploring databases, it can be helpful to create **EER diagrams** manually to enhance our understanding on how all the data is related like:

(ie. one of the first thing to do when start working in a new database )

- Which tables are in the database
- Which columns exist in each table
- Data types & constraints of the various columns
- Primary & Foreign keys within tables
- Relationship cardinality (ie. one-to-many) between tables

- 
- Primary key - unique identifiers for each records in the table
  - Foreign key - PK defined in other tables, to establish link btw tables
  - Logical key - labels for effective lookup by logical expr, (eg. WHERE title= '?')



Never try to connect data tables directly to each other; Connect them via shared lookup tables

— By doing so, you're trying to force many-to-many relationships.

**Data Models** generally contain 2 types of tables:

{ fact & dimension }

- ❖ **Data Tables** contain numbers / metrics, typically at the most granular level possible, that are used for summarization.
- ❖ **Lookup Tables** provide descriptive, usually text-based attributes about data fields, that are used for filtering / grouping.

**Database Normalization** is the structuring process of data from a single merged table in a relational database into multiple related tables to

- **Eliminate redundant data** to decrease table sizes & improve processing speed & efficiency.
- **Minimize inconsistencies & anomalies** from repeated insert, update, or delete.
- **Simplify queries** and structure the database for meaningful analysis.

The diagram illustrates the normalization of a single merged table into three related tables:

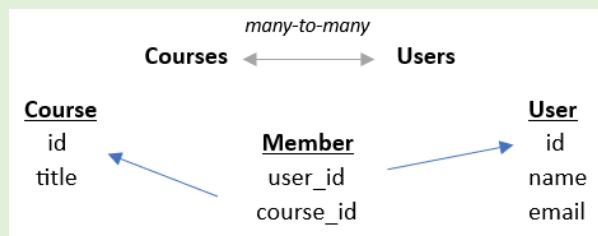
- Transaction Table:** Contains columns for date, product\_id, and quantity.
- Calendar Lookup Table:** Provides additional attributes about each date (month, year, weekday, quarter, etc.).
- Product Lookup Table:** Provides additional attributes about each product (brand, product name, sku, price, etc.).

These columns are matched by foreign (data) & primary (lookup) keys.

(ie. Each table should serve a **distinct & specific** purpose: *product, calendar fields, trxn records, etc.*)

**When building an entity relationship model for an application,**

- #1 Start w/ first table – main description of the application (eg. Music API = a map that manages tracks)
- #2 Brainstorm to see what kind of information is helpful to include.
- #3 Create a connection table w/ foreign keys for many-to-many relationships.



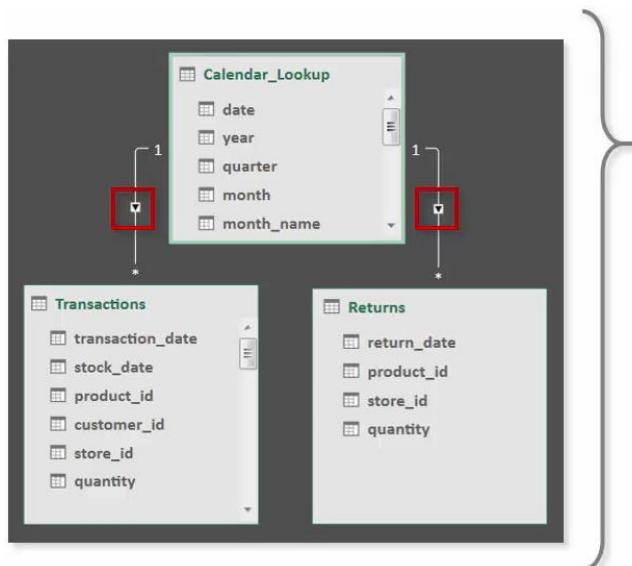
**Note:** Unlike many-to-many, one-to-one relationship is not illegal just inefficient;

- ❖ Often, INT value is used as primary key as it has lesser overhead for quick sorting & scanning of rows.

**As a general rule of thumb**, leave non-normalized dataset as it is unless you're dealing with a very large no. of records or prioritize storage is needed.

—ie. it can increase complexity when it comes to querying the model as more relationships or JOINS are required to consolidate the data.

## (Flow Direction is Important)



The **filter directions** in each relationship will always point from the “one” side to the “many” side.

- Filtering a table will impact any tables “downstream” of it as defined.
- In the same PivotTable, filtering by Calender\_Lookup date will output correctly filtered data from both tables; however by the Transactions date will yield unfiltered / error output.

[# Arrange the lookup tables above data tables to remind that the filters always flow downstream ]

In this model, the only way to filter both **Sales** and **Returns** data by **Territory** is to use the **Territory Key** from the lookup table, which is upstream and related to both fact tables

- Filtering using Territory Key from the **Sales** table yields **incorrect Returns values**, since the filter context can't flow to any other table
- Filtering using Territory Key from the **Returns** table yields **incorrect Sales values**, and is limited to territories that exist in the returns table

TerritoryKey	OrderQuantity	ReturnQuantity
1	12,513	270
2	40	
3	30	
4	17,191	362
5	49	1
6	1,694	38
7	7,862	186
8	7,950	163
9	17,951	404
10	9,694	204
Total	84,174	1,828

TerritoryKey	OrderQuantity	ReturnQuantity
1	12,513	1,828
2	40	1,828
3	30	1,828
4	17,191	1,828
5	49	128
6	1,694	1,828
7	7,862	1,828
8	7,950	1,828
9	17,951	1,828
10	9,694	1,828
Total	84,174	1,828

TerritoryKey	OrderQuantity	ReturnQuantity
1	84,174	270
4	84,174	362
5	84,174	1
6	174	238
7	174	186
8	84,174	163
9	84,174	404
10	84,174	204
Total	84,174	1,828

Filtering by **Territory Lookup[Territory Key]**

Filtering by **Sales Data[Territory Key]**

Filtering by **Returns Data[Territory Key]**

- ✓ Understanding the filter context & flow is critical for ensuring the analysis we're making is correct.

## Creating & Populating a Table

1. CREATE TABLE User (  
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
    name TEXT  
    email TEXT  
);  
  
CREATE TABLE Course (  
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
    title TEXT  
);  
  
CREATE TABLE Member (  
    user\_id INTEGER,  
    course\_id INTEGER,  
    role INTEGER,  
    PRIMARY KEY (user\_id, course\_id) *unique compound key table constraint*  
);
  
2. INSERT INTO User (name, email) VALUES ('JANE','jane@tsugi.org'), ('Ed', 'ed@tsugi.org'),  
INSERT INTO Course (title) VALUES ('Python'), ('SQL')  
INSERT INTO Member (user\_id, course\_id, role) VALUES (1,1,1), (2,1,0), ...  
*# connecting the rows btw tables by integer id representation in foreign keys.*
  
3. SELECT User.name, Member.role, Course.title FROM User JOIN Member JOIN Course  
ON Member.user\_id = User.id AND Member.course\_id = Course.id  
ORDER BY Course.title, Member.role DESC, User.name

### Pro Tip:

- A DCL script can be edited & imported into database to ease the creation of tables esp. for large relational data. (Refer. CreateStatements.sql)
- Likewise, an external data may be loaded from an SQL Script or CSV file using 'Table Data Import Wizard'.

*"The no. of columns for a relational table is usually large enough, the constraint is often due to disk memory size &/ maintainability on a server w.r.t the no. of rows for an application rather than the software capability to process the information. "*

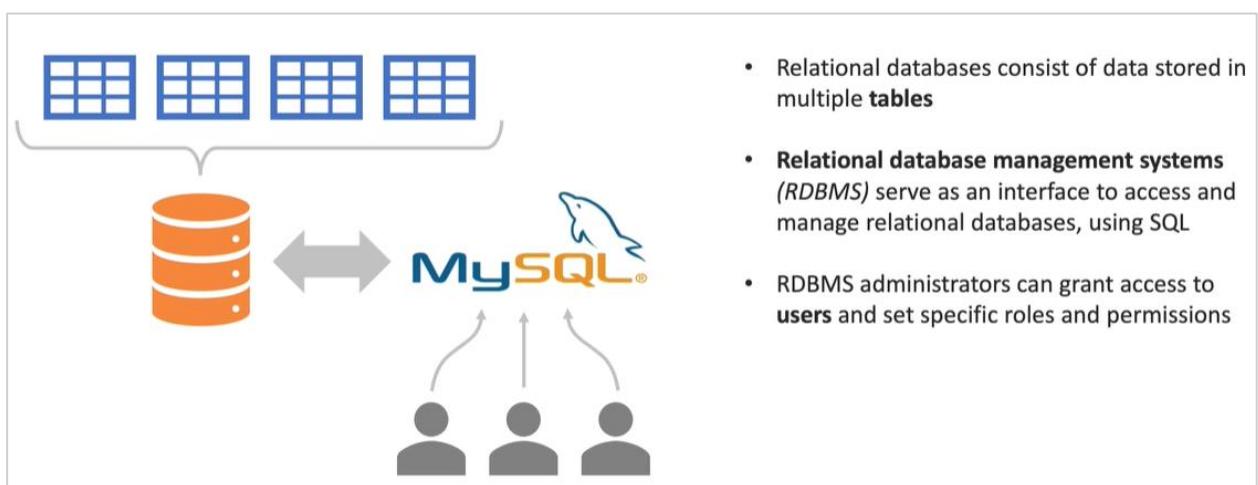
When creating a relation table, it is required to specify the homogenous datatype to a field (sometimes, with expected memory size – eg. int64) for efficient data storage & management.

### Common SQL Datatypes:

Type	Name	Description
String	CHAR(n) / VARCHAR	Holds fixed (n) / flexible length up to maximum (0-255 / n) which differs in the way they are stored & retrieved w±wo trailing spaces. : 'ab' - CHAR(4) ~ 'ab' {4 bytes} ; VARCHAR(4) ~ 'ab' {2 bytes}.
	TEXT /VARBINARY(byte)	Holds long context / binary byte string – (0-65535). (eg. Byte image content or free text entered by survey respondents)
	ENUM(val1, val2, ...) /SET(val1, val2, ...)	- Holds one str from the specified list, or a blank is inserted. - Holds 0 or more than one str from the list.
Numeric	TINY/SMALL/MEDIUM INT/BIGINT(size)	Holds signed / unsigned integers of {1, 2, 3, 4, 8, } bytes : (-128 : 127), (-32,768 : 32,767), (-8,388,608 : 8,388,607), UNSIGNED (-2,147,483,648 : 2,147,483,647), ... (-2 <sup>63</sup> : 2 <sup>63</sup> -1).
	FLOAT/DOUBLE(n,d) /DECIMAL(n,d)	Holds unsigned values of : Decimal (precise to 23 digits), (23 to 53 digits), (to 65 digits) Ie. total no. decimal (n) default: 10-65 ± points (d) 0-30 specified.
Logical	BOOLEAN	Holds only TRUE {non-zero} / False {0} values.
Datetime	DATETIME /TIMESTAMP	Holds YYYY-MM-DD HH:MM:SS / YYYYMMDDHHMMSS format (w±wo the support of time zones depending on database)
	DATE/TIME /INTERVAL	Holds separate date / time / a span of time values only : YYYY-MM-DD; HH:MM:SS; '1 hour 30 minutes'.

**Note:** ① ZEROFILL constraint- INT(4) :: 0005 with a fixed maximum display width (size) of 4.

② Float stores approximate values, but is able to do faster calculations while decimal stores exact values.



- **When we're setting up a database and partitioning out permission as an administrator, we want to restrict access to just the types of activities that a given person would need.**

—eg. An analyst who's never going to manipulate tables, should be granted read only access to database.

## Creating Indexes

### Searching Without an Index:

inventory_id	item_name	number_in_stock
1	fur coat	0
2	moccassins	4
3	velour jumpsuit	12
4	house slippers	6
5	brown leather jacket	3
6	broken keyboard	6
7	ski blanket	1
8	kneeboard	2
9	pro wings sneakers	0
10	wolf skin hat	1
11	fur fox skin	1
12	plaid button up shirt	8
13	flannel zebra jammies	6

### Using an Index:

inventory_id	item_name	number_in_stock
1	fur coat	0
2	moccassins	4
3	velour jumpsuit	12
4	house slippers	6
5	brown leather jacket	3
6	broken keyboard	6
7	ski blanket	1
8	kneeboard	2
9	pro wings sneakers	0
10	wolf skin hat	1
11	fur fox skin	1
12	plaid button up shirt	8
13	flannel zebra jammies	6

**Note:** In MySQL, any **PK & FK** created on a table is set as an index automatically.

# 1 SQL AUTOMATION TOOLS

## 2 Creating Triggers

- 3 MySQL allows us to create **Triggers**, where we can prescribe certain actions on a table to trigger one or more other actions to occur.
- 4 **Triggers** are a common way to make sure related tables remain in sync as they are updated over time, *i.e.* BEFORE/AFTER an INSERT, UPDATE, or a DELETE.

```
CREATE TRIGGER Trigger_Name
(BEFORE | AFTER) (INSERT | UPDATE | DELETE)
ON t (FOR EACH ROW ROW | FOR EACH STATEMENT)
[FOLLOW | PROCEDES Existing_Trigger_Name]
... -- Any SQL Query
[ WHERE col_name = [OLD | NEW].col_name ];
```

Note: MySQL only supports row-level trigger\_type.

- 9 (1) To alter a Trigger in MySQL, you'll need to DROP TRIGGER then recreate them again.
- 10 (2) OLD / NEW must be explicitly defined when referencing columns in the UPDATE statement to specify whether you're interested in the value before or after the change.
- 11 (3) Multiple Triggers can exist on the same table. By default, they'll fire in the order they were created but can be modified using FOLLOWS / PROCEDES.
- 12 (4) To see Triggers in a schema,
- 13 use- SHOW TRIGGERS / SELECT \* FROM INFORMATION\_SCHEMA.TRIGGERS .

17 \*\* You'll need to have the Triggers privilege enabled by DBA in order to CREATE, DROP, or view Triggers.

The screenshot shows two panels side-by-side. The left panel, titled 'Global', displays the 'Administrative Roles' tab for the 'root' user. It lists various system roles like DBA, MaintenanceAdmin, etc., each with a description and a checkbox indicating if they have been granted. A green circle highlights the 'Administrative Roles' tab. The right panel, titled 'Schema-Specific', shows the 'Schema Privileges' tab for the 'thrifshop' schema. It lists 'Privileges' such as CREATE, INSERT, etc., for the 'root' user. A green circle highlights the 'Schema Privileges' tab. A large green box labeled 'Schema-Specific' covers the middle section of the right panel. At the bottom of both panels are buttons for 'Revoke All Privileges', 'Delete Entry', and 'Add Entry'.

```

CREATE TRIGGER purchaseUpdateInventory
AFTER INSERT ON customer_purchases
FOR EACH ROW
    UPDATE inventory
        -- subtracting an item for each purchase
        SET number_in_stock = number_in_stock - 1
    WHERE inventory_id = NEW.inventory_id;
        -- When inventory_id =
        -- new record inventory_id

INSERT INTO customer_purchases VALUES
(13,NULL,3,NULL), -- inventory_id = 3, velour jumpsuit
(14,NULL,4,NULL) -- inventory_id = 4, house slippers
;

```



Result Grid			
	inventory_id	item_name	number_in_stock
▶	1	fur coat	0
	2	moccassins	4
	3	velour jumpsuit	12
	4	house slippers	6
	5	brown leather jacket	3
	6	broken keyboard	6
	8	kneeborder	2
	9	pro wings sneakers	0
	10	wolf skin hat	1
	11	fur fox skin	1
	12	plaid button up shirt	8
	13	flannel zebra jammies	6

Result Grid			
	inventory_id	item_name	number_in_stock
▶	1	fur coat	0
	2	moccassins	4
	3	velour jumpsuit	11
	4	house slippers	5
	5	brown leather jacket	3
	6	broken keyboard	6
	8	kneeborder	2
	9	pro wings sneakers	0
	10	wolf skin hat	1
	11	fur fox skin	1
	12	plaid button up shirt	8
	13	flannel zebra jammies	6

\* A trigger that will update `inventory` table whenever a customer makes a purchase.

## 1 Creating Stored Procedures

2 - store & call frequently used queries on the server for  
3 [ *more efficient query writing* ],  
4 [ *to share complex procedures more easily btw Analysts & DB users* ], and  
5 [ *to centralize one module of code that can be managed & updated while being used* ].

6

7 **DELIMITER //**

8 **CREATE PROCEDURE** ProcedureName([**IN** var dtype, ..., **OUT** var dtype, ...])  
9 **BEGIN**  
10     ...; -- Complex SQL Statements  
11 **END //**  
12 **DELIMITER ;**

13 **CALL** ProcedureName(in\_value, @var); -- specify var to store the output  
14

15 [ Optional Parameters ]  
16     • **IN**: to take in a value from the user or application  
17     • **OUT**: to return a single scalar value or specific result set  
18     • **INOUT**: a combination of IN & OUT parameter modes.

19

20 ( Think of it as some lightweight automation you can build for less SQL literate people  
21     to have dynamic reports that they can manipulate vars for a certain event. )

22

23     **DELIMITER // -- changing the default delimiter to prevent ending midway**

24

25     **CREATE PROCEDURE** customer\_rentals\_w\_total  
26         (**IN** custid **BIGINT**, **OUT** total\_rentals **BIGINT**)  
27     **BEGIN**  
28         **SELECT** COUNT(rental\_id) **INTO** total\_rentals **FROM** rental **WHERE** customer\_id = custid;  
29  
30         **SELECT** \* **FROM** rental **WHERE** customer\_id = custid;  
31     **END //**  
32     -- next, changing the delimiter back  
33     **DELIMITER ;**

34

35     **CALL** customer\_rentals\_w\_total (135,@total\_rentals);

36

37     **SELECT** @total\_rentals; -- show the single var result

38

39 ( Multiple Result Sets can be defined for a Stored Procedure )

40

41 - Programming Languages: Iterate through each using an iterator, eg. Python- for... in.  
42 - MySQL Command-Line: It will display the output of each query sequentially.  
43 - MySQL Workbench: It will show each in a separate tab within the result window.

44

45 \*\* To return a specific result, you need to explicitly specify the OUT parameter as shown.

## 1 Creating Scheduled Events

- 2 - A user defined SQL operation similar to stored procedure that will fire at a certain time,  
3 or at regular recurring intervals.

```
5 CREATE EVENT EventName  
6  
7     ON SCHEDULE (AT | EVERY) 'DATETIME'  
8  
9     DO  
10  
11        INSERT INTO t VALUES (...); -- Any Query / Stored Procedure  
12  
13  
14        DROP EVENT EventName; -- Remove a scheduled event
```

... ( EXAMPLE )

```
CREATE TABLE sillytable (  
    timestamps_via_event DATETIME  
);  
  
CREATE EVENT myfirstevent  
    ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 7 second  
    DO  
        INSERT INTO sillytable VALUES (NOW());  
  
SELECT * FROM sillytable;
```

15      Result Grid    Filter Rows:  Search    Export:

timestamps\_via\_event

▶ 2020-10-25 16:40:31

16

17

18

## WORKING WITH MULTIPLE TABLES

Step-by-step approach when joining multiple tables:

- Start with table with complete keys desired as leftmost table
- Then join one table on some condition
- Check if the output is correct
- Join another table on another condition
- Check the output again & specify the column desired as final output

( You can join any tables as long as you specify the columns that link the tables together )

```
SELECT hs.year, hs.country, hs.happiness_score,
       cs.continent, ir.inflation_rate
  FROM happiness_scores hs
    LEFT JOIN country_stats cs
      ON hs.country = cs.country
    LEFT JOIN inflation_rates ir
      ON hs.year = ir.year AND hs.country = ir.country_name;
```

ANY, ALL, EXISTS: To provide more specific filtering logic

### EXAMPLE

Only return happiness scores for countries that EXIST in the inflation rates table

```
SELECT *
  FROM happiness_scores h
 WHERE EXISTS (
    SELECT i.country_name
      FROM inflation_rates i
     WHERE i.country_name = h.country);
```

```
SELECT *
  FROM happiness_scores h
    INNER JOIN inflation_rates i
      ON h.country = i.country_name
      AND h.year = i.year;
```

year	country	region	happiness_score	year	country_name	inflation_rate
2015	Bangladesh	South Asia	4.694	2015	Bangladesh	6.1
2015	Brazil	Latin America and Caribbean	6.983	2015	Brazil	9.0
2015	China	East Asia	5.140	2015	China	1.4
2015	Ethiopia	Sub-Saharan Africa	4.512	2015	Ethiopia	6.1
2015	France	Western Europe	6.575	2015	France	0.1
2015	Germany	Western Europe	6.750	2015	Germany	0.3

\* Correlated subqueries can be slow in some RDBMSs;

In best practice test it out out, and replace it (as shown) with a JOIN if it's an issue.

A **SELF-JOIN** is used when you want to compare rows within the same table side-by-side.  
(e.g. comparing country's happiness score to every other countries in the same region)

Through:

- Combine table with itself based on a matching column
- Filter on the resulting rows based on some criteria

```
SELECT * FROM employees;
```

employee_id	name	salary
1	Ava	85000
2	Bob	72000
3	Cat	59000
4	Dan	85000

**EXAMPLE (1)** | Finding Related / Sequential Events

-- Employees with the same salary

```
SELECT e1.name, e1.salary,
       e2.name, e2.salary
  FROM employees e1 INNER JOIN employees e2
    ON e1.salary = e2.salary
   WHERE e1.name <> e2.name
  ORDER BY e1.name;
```

name	salary	name	salary
Ava	85000	Dan	85000
Dan	85000	Ava	85000

-- Here, to find *log entries by the same user that occurred within a short time of each other.*

```
SELECT l1.UserID, l1.activity AS activity1, l1.Timestamp AS timestamp1,
       l2.activity AS activity2, l2.Timestamp AS timestamp2,
  FROM LogEntries l1 INNER JOIN LogEntries l2
    ON l1.UserID AND TIMESTAMPDIFF(MINUTE, l2.Timestamp, l1.Timestamp) < 10
 WHERE l1.EntryID <> l2.EntryID AND l1.Timestamp < l2.Timestamp;
```

```
SELECT * FROM employees;
```

employee_id	name	salary
1	Ava	85000
2	Bob	72000
3	Cat	59000
4	Dan	85000

**EXAMPLE (2)** | Comparing values within rows in a table

-- Employees that have a greater salary

```
SELECT e1.name, e1.salary,
       e2.name, e2.salary
  FROM employees e1 INNER JOIN employees e2
    ON e1.salary > e2.salary
  ORDER BY e1.name;
```

name	salary	name	salary
Ava	85000	Bob	72000
Ava	85000	Cat	59000
Bob	72000	Cat	59000
Dan	85000	Bob	72000
Dan	85000	Cat	59000

```
SELECT * FROM managers;
```

employee_id	employee_name	manager_id
1	Ava	NULL
2	Bob	1
3	Cat	1
4	Dan	2

**EXAMPLE (3)** | Hierarchical Data

-- Employees and their managers

```
SELECT m1.employee_id, m1.employee_name,
       m1.manager_id,
       m2.employee_name AS manager_name
  FROM managers m1 LEFT JOIN managers m2
    ON m1.manager_id = m2.employee_id;
```

employee_id	employee_name	manager_id	manager_name
1	Ava	NULL	NULL
2	Bob	1	Ava
3	Cat	1	Ava
4	Dan	2	Bob

## Subqueries & CTEs

- to allow aggregation & logical comparison in WHERE & HAVING clauses.

```
:: SELECT * FROM Employees      ⇒   SELECT * FROM Employees
    WHERE salary > AVG(salary)      WHERE salary >
                                         (SELECT AVG(salary) FROM Employees)
```

- to create an aggregated column that is aligned with each row value for comparison.

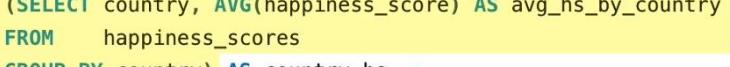
```
:: SELECT id, salary, AVG(salary)  ⇒  SELECT id, salary,
    FROM Employees                      ( SELECT AVG(salary) FROM Employees
                                            WHERE id = outer_table.id )
                                         FROM Employees AS outer_table
```

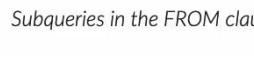
- to allow creation of a virtual table to facilitate sub-selection in FROM clause.

```
:: SELECT COUNT(*) FROM
    (SELECT c1, c2, ... FROM t1) t
```

--Return each country's happiness score for the year alongside its average

```
SELECT hs.year, hs.country, hs.happiness_score,
       country_hs.avg_hs_by_country
  FROM happiness_scores hs LEFT JOIN
       (SELECT country, AVG(happiness_score) AS avg_hs_by_country
        FROM happiness_scores
        GROUP BY country) AS country_hs
      ON hs.country = country_hs.country;
```

 This subquery calculates the average happiness score by country, and is then joined with the main query

 Subqueries in the FROM clause need to have an alias

year	country	happiness_score	avg_hs_by_country
2015	Afghanistan	3.575	2.9907778
2015	Albania	4.959	4.8932222
2015	Algeria	5.605	5.4090000
2016	Afghanistan	3.360	2.9907778
2016	Albania	4.655	4.8932222
2016	Algeria	6.355	5.4090000
2017	Afghanistan	3.794	2.9907778
2017	Albania	4.644	4.8932222
2017	Algeria	5.872	5.4090000

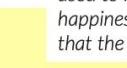


**PRO TIP:** Using subqueries within the JOIN clause is great for speeding up queries, since it allows you to join smaller tables

Queries can contain **multiple subqueries** as long as each one has a different alias

```
SELECT * FROM
```

```
(SELECT hs.year, hs.country, hs.happiness_score,
       country_hs.avg_hs_by_country
  FROM (SELECT year, country, happiness_score FROM happiness_scores
        UNION ALL
        SELECT 2024, country, ladder_score FROM happiness_scores_current) AS hs
 LEFT JOIN
       (SELECT country, AVG(happiness_score) AS avg_hs_by_country
        FROM happiness_scores
        GROUP BY country) AS country_hs
      ON hs.country = country_hs.country) AS hs_country_hs
```

 This is a **nested subquery** now, used to return years where the happiness is a whole point greater than the country's average score

```
WHERE happiness_score > avg_hs_by_country + 1;
```

year	country	happiness_score	avg_hs_by_country
2015	Lesotho	4.898	3.8561429
2015	Venezuela	6.810	5.3042222

For simple queries, sometimes a subquery is readable enough and works just fine.

b-d **READABILITY**

 **MODULARITY**

 **REUSABILITY**

- Complex queries with CTEs are much easier to read.
- CTEs can handle recursive queries.
- CTEs can be referenced multiple times within a query.

**MAIN**  
**QUERY** —

**SELECT \* FROM**

```
(WITH hs23 AS (SELECT *
               FROM happiness_scores
               WHERE year = 2023),
     hs24 AS (SELECT *
               FROM happiness_scores_current)

SELECT hs23.country,
       hs23.happiness_score AS hs_2023,
       hs24.ladder_score AS hs_2024
  FROM hs23 INNER JOIN hs24
    ON hs23.country = hs24.country) AS hs_23_24
```

**WHERE hs\_2023 < hs\_2024;**

country	hs_2023	hs_2024
Albania	5.277	5.304
Algeria	5.329	5.364
Argentina	6.024	6.188
Armenia	5.342	5.455

① **Multiple CTEs** can be used in a query, or even **combine them with subqueries**.

② **Temporary tables & views** are other options for reference in multiple main queries.

- Both require additional permissions, they exist for a session or store in schema until modified.

### (Applications of SQL View)

▪ **To enhance data security & control access.**

▪ **To simplify complex queries.**

— make repeated multi-table JOINS, Subqueries, or intricate WHERE clause to `SELECT \* FROM View`.

▪ **To provide schema independence.**

— if you anticipate future changes to your underlying table structure (alter col, split, merge tables), applications that query views are not affected directly and the view definition can be updated at a later time to reflect changes.

**Recursive CTE:** A query that references itself, which is useful for generating sequences and working with hierarchical data.

```
SELECT *
FROM stock_prices;
```

date	price
2024-11-01	678.27
2024-11-03	688.83
2024-11-04	645.40
2024-11-06	591.01

Notice the missing dates

Step 1: Generate by recursive CTE

```
WITH RECURSIVE my_dates(dt) AS
  (SELECT '2024-11-01'
  UNION ALL
  SELECT dt + INTERVAL 1 DAY
  FROM my_dates
  WHERE dt < '2024-11-06')
```

Step 2: Join with stock price table

```
SELECT md.dt, sp.price
FROM my_dates md
LEFT JOIN stock_prices sp
ON md.dt = sp.date;
```

dt	price
2024-11-01	678.27
2024-11-02	NULL
2024-11-03	688.83
2024-11-04	645.40
2024-11-05	NULL
2024-11-06	591.01

## EXAMPLE

Return the reporting chain for each employee

```
SELECT *
FROM employees;
```

employee_id	employee_name	manager_id
1	Ava	NULL
2	Bob	1
3	Cat	1
4	Dan	2

This stops the recursion when there is nothing left to join

```
WITH RECURSIVE employee_hierarchy AS (
  SELECT employee_id, employee_name, manager_id,
  employee_name AS hierarchy
  FROM employees
  WHERE manager_id IS NULL) → This sets the highest-ranking employee as the anchor
  UNION ALL

  SELECT e.employee_id, e.employee_name, e.manager_id,
  CONCAT(eh.hierarchy, ' > ', e.employee_name) AS hierarchy
  FROM employees e INNER JOIN employee_hierarchy eh
  ON e.manager_id = eh.employee_id)
```

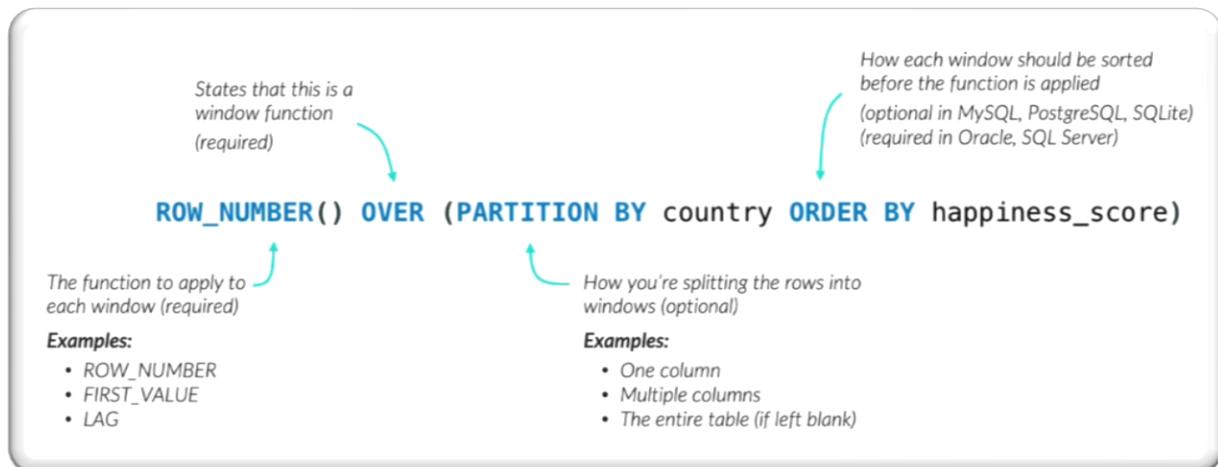
```
SELECT employee_id, employee_name, manager_id, hierarchy
FROM employee_hierarchy
ORDER BY employee_id;
```

employee_id	employee_name	manager_id	hierarchy
1	Ava	NULL	Ava
2	Bob	1	Ava > Bob
3	Cat	1	Ava > Cat
4	Dan	2	Ava > Bob > Dan



**PRO TIP:** Recursive CTEs aren't very common so instead of memorizing syntax, keep in mind the general concepts of generating sequences and returning hierarchies

# WINDOW FUNCTIONS



**Note:** It operates by assessing the rows sequentially within each window. (See example below)

Condition in <code>OVER()</code> Clause	Default Window Frame	Resulting Behavior
<code>PARTITION BY ... ORDER BY ...</code>	<code>RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW</code>	<b>Running Aggregate</b>
<code>PARTITION BY ... (no ORDER BY )</code>	<code>RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING</code>	<b>Aggregate over entire partition</b>
<code>ORDER BY ... (no PARTITION BY )</code>	<code>RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW</code>	<b>Running Aggregate over entire result set</b>
<code>Empty OVER() (no PARTITION BY , no ORDER BY )</code>	<code>RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING</code>	<b>Aggregate over entire result set</b>

Row Numbering	<ul style="list-style-type: none"> <li>• <b>ROW_NUMBER()</b> gives every row a unique number</li> <li>• <b>RANK()</b> accounts for ties</li> <li>• <b>DENSE_RANK()</b> accounts for ties and leaves no missing numbers in between</li> </ul>
Value Within a Window	<ul style="list-style-type: none"> <li>• <b>FIRST_VALUE()</b> extracts the first value in a window, in sequential row order</li> <li>• <b>LAST_VALUE()</b> extracts the last value</li> <li>• <b>NTH_VALUE()</b> extracts the value at a specified position</li> </ul>
Value Relative to a Row	<ul style="list-style-type: none"> <li>• LEAD</li> <li>• LAG</li> </ul>
Aggregate Functions	<ul style="list-style-type: none"> <li>• SUM, AVG, COUNT</li> <li>• MIN, MAX</li> </ul>
Statistical Functions	<ul style="list-style-type: none"> <li>• NTILE</li> <li>• CUME_DIST</li> <li>• PERCENT_RANK</li> </ul>

\*\* LEAD / LAG(c1, offset -> int, default-> NULL) can specify a default if there's no lead/lagging record.

```

SELECT country, year, happiness_score,
       AVG(happiness_score) OVER(PARTITION BY country ORDER BY year) row_num
FROM   happiness_scores
ORDER BY country, year;

```

country	year	happiness_score	row_num
Afghanistan	2015	3.575	3.5750000
Afghanistan	2016	3.360	3.4675000
Afghanistan	2017	3.794	3.5763333
Afghanistan	2018	3.632	3.5902500
Afghanistan	2019	3.203	3.5128000
Afghanistan	2020	2.567	3.3551667
Afghanistan	2021	2.523	3.2362857
Afghanistan	2022	2.404	3.1322500

**NTILE:** divides the rows in a window into  $n$  groups containing equal (or nearly) no.

**EXAMPLE** View the top 25% of happiness scores for each region

```

WITH hs_pct AS (SELECT region, country, happiness_score,
                      NTILE(4) OVER(PARTITION BY region ORDER BY happiness_score DESC) AS hs_percentile
               FROM   happiness_scores
               WHERE   year = 2023)

```

```

SELECT *
FROM   hs_pct
WHERE  hs_percentile = 1
ORDER BY region, happiness_score DESC;

```

Return the percentiles in a CTE

There are many **practical applications** of window functions, incl.: calculating moving averages, running totals, and more.

```

-- Calculate the three year moving average of happiness scores
SELECT country, year, happiness_score,
       AVG(happiness_score) OVER (PARTITION BY country
                                  ORDER BY year ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
       AS three_year_ma
FROM   happiness_scores;

```

Note that this uses an AVG() aggregate function as part of a window function

We've added some additional keywords at the end that specify rows within the window

country	year	happiness_score	three_year_ma
Afghanistan	2015	3.575	3.5750000
Afghanistan	2016	3.360	3.4675000
Afghanistan	2017	3.794	3.5763333
Afghanistan	2018	3.632	3.5953333
Afghanistan	2019	3.203	3.5430000
Afghanistan	2020	2.567	3.1340000
Afghanistan	2021	2.523	2.7643333
Afghanistan	2022	2.404	2.4980000
Afghanistan	2023	1.859	2.2620000



## STRING FUNCTIONS

Category	Function	Description
Length	LENGTH	Returns the number of characters in a string
Update	TRIM	Removes leading or trailing spaces (or other characters)
	REPLACE	Replaces a substring with another
Case	UPPER / LOWER	Converts all characters to uppercase or lowercase
Combine	CONCAT	Combines multiple strings into one
Find Location	INSTR	Returns the position of a substring within a string
	SUBSTR	Extracts part of a string of a specified length, starting from a given position
Find Pattern	LIKE*	Performs a pattern match within a string, typically using wildcards
	REGEXP	Matches a string against a regular expression pattern

```
SELECT event_name,
       CASE WHEN INSTR(event_name, ' ') = 0 THEN event_name
             ELSE SUBSTR(event_name, 1, INSTR(event_name, ' ') - 1)
        END AS first_word
  FROM events;
```

event_name	first_word
New Year's Day	New
Lunar New Year	Lunar
Persian New Year	Persian
Birthday	Birthday
Last Day of School	Last
Vacation	Vacation

```
-- Return words with hyphens in them
SELECT event_desc,
       REGEXP_SUBSTR(event_desc, '[A-Z][a-z]+(-[A-Za-z]+)+') AS hyphen_phrase
  FROM events;
```

event_desc	hyphen_phrase
A much-anticipated break from daily routines, this vacation period allows individuals and fam...	much-anticipated
An exciting and sometimes nerve-wracking day for students, marking the beginning of a new...	nerve-wracking
A festive occasion celebrated with costumes, trick-or-treating, and various spooky activities....	trick-or-treating
A holiday rooted in gratitude and family, Thanksgiving is celebrated with a large feast that ty...	NULL
A major holiday celebrated around the world, Christmas commemorates the birth of Jesus C...	gift-giving

- When dealing with RegExp, instead of memorizing the syntax, think about what you want to do and use AI tools (ChatGPT, Gemini, etc.) to help generate the syntax.
- eg. create a regex that returns sentences that start with a 3 letter word.



## NULL FUNCTIONS

To do a simple IF-ELSE NULL check:

- Use the **IFNULL** function (*NVL in Oracle, not supported in PostgreSQL*)

To do more complex IF-ELSE NULL checks:

- Use the **COALESCE** function (*supported in most modern RDBMS's*)



**PRO TIP:** COALESCE is a more flexible version of the two, and will allow you to do multiple NULL checks, and returns the first non-NUL value

```
-- Return an alternative field after multiple checks
```

```
SELECT name, email, alt_email,
       IFNULL(email, 'no email') AS contact_email_value,
       IFNULL(email, alt_email) AS contact_email_column,
       COALESCE(email, alt_email, 'no email') AS contact_email_coalesce
  FROM contacts;
```

name	email	alt_email	contact_email_value	contact_email_column	contact_email_coalesce
Anna	anna@example.com	NULL	anna@example.com	anna@example.com	anna@example.com
Bob	NULL	bob.alt@example.com	no email	bob.alt@example.com	bob.alt@example.com
Charlie	NULL	NULL	no email	NULL	no email
David	david@example.com	david.alt@example....	david@example.com	david@example.com	david@example.com

# DATETIME MANIPULATION

## I. Time Zones.

( Time zone may be localized or referred to the universal standard: UTC / older GMT )

- Convert time zone from UTC to PST

```
SELECT '2020-09-01 00:00:00 -0' at time zone 'pst';
```

- Some databases offer dedicated functions

```
SELECT CONVERT_TZ('pst', '2020-09-01 00:00:00 -0')
```

\* UTC time may be convenient, but it doesn't reflect the actual moment of the day to an activity.

( Look out for timezone offset / database's documentation for the exact specification )

**Note:** Timestamp values won't necessarily have the time zone embedded, and you may require consulting with the source / vendor to figure out how the data was stored.

## II. Date / Timestamp Format.

- Convert string / Unix epoch with the right datetime format:

```
STR_TO_DATE(c1, '%m/%d/%Y') or, TO_TIMESTAMP(c1, format)
```

- Truncate a date to the desired period:

```
DATE_TRUNC('unit', c1) or, DATE_FORMAT(c1, '%Y-%m-01')  
::2020-10-01 00:00:00
```

- Extract individual date / time part:

```
DATE_PART('unit', c1) or, EXTRACT('YEAR_MONTH' FROM c1)  
::return float dtype :: 202002
```

- Compose date & time from separate columns into a timestamp:

```
SELECT date_col + time_col AS timestamp
```

The standard time unit incl. — microsecond, millisecond, second, minute, hour, day, week, dayofweek, month, quarter, year, decade, century, millennium.

## III. Datetime math

It can involve:

dt1 - dt2 / INTERVAL ie. Python timedelta

—*the numeral system of date & time unit* [# year(s), month(s), day(s), hours, minute(s) & second(s)]

<sup>①</sup> `SELECT c1 FROM t WHERE c1 < current_date - INTERVAL '3 months'`

<sup>②</sup> `SELECT DATEDIFF('unit', dt1, dt2) FROM t` (Refer. [PostgreSQL](#) for complete functions & operators)

## DATA ANALYSIS APPLICATIONS

- Data analysts often retrieve & manipulate sample data on database server apart from applications that rely on user's machine power to optimize the computational resources.

— Often databases keep only the current datasets (ie. collected data of one full / several working days);  
Check data warehouse for the daily snapshots of changing data fields- the history, if at all.

**DO NOT rush into writing query**, instead draft & visualize your idea first.

- Sketch out Tables:** Draw out the structure of each table involved in the query.
- Map Relationships:** When working with joins, visualizing how data flows btw tables can make complex joins easier to handle.
- Plan the Logic:** Outline the steps required to compute the resulting columns to help organize thoughts & spot any logic gaps.

( aggregations / window functions, link to tables, querying the results of query, conditionals, filters )

# The hard part about SQL is not syntax writing, but what questions to ask of the data.

### To answer the question of —

“ How many people were of the official age for secondary education broken down by region of the world in 2015? ”

- Exclude rows with a missing region.
- Use the **SUM(value)** to calculate the total population for a given grain size.
- Sort by highest population region first.

```
SELECT summary.region, SUM(edu.value) secondary_edu_population
FROM
    `bigquery-public-data.world_bank_intl_education.internation_education` AS edu
INNER JOIN
    `bigquery-public-data.world_bank_intl_education.country_summary` AS summary
ON edu.country_code = summary.country_code
    WHERE summary.region IS NOT NULL
        AND edu.indicator_name = 'Population of the official age for secondary
            education, both sexes (number)'
        AND edu.year = 2015
GROUP BY summary.region
ORDER BY secondary_edu_population DESC
```

## I. Profiling: Feature Distributions

### a. Histogram / Frequency distribution:

```
SELECT fruit, COUNT(*) / DISTINCT col_name) AS quantity --(1)
FROM fruit_inventory
GROUP BY 1;      --output that facilitates data visualizations: histogram / bar chart
```

[# Output the feature distributions, or examine how frequently a typical class / value: unusual, sparse, missing etc. appear in the data.]

To return the distribution of orders from a table that needs to be aggregated beforehand, with date, customer identifier, order identifier, & an amount, instead of simple unique identifier counting:

```
SELECT orders, COUNT(*) AS num_customers --(2)
FROM
(
    SELECT customer_id, COUNT(order_id) AS orders
    FROM orders
    GROUP BY 1
) a
GROUP BY 1;
```

[# Count the no. of orders by each customer\_id, then work out how many customers with the same size.]

A simplified logical order often looks like this:

1. `FROM`, `JOIN`s (determine the base dataset)
2. `WHERE` (filter rows)
3. `GROUP BY` (**aggregate rows**)
4. `HAVING` (filter grouped rows)
5. **Window Functions (PARTITION BY is here)**
6. `SELECT` (select columns, apply expressions)
7. `DISTINCT`
8. `ORDER BY` (sort final results)
9. `LIMIT / TOP`

## b. Binning:

```
# Discretize a continuous feature into a defined finite no. of bins / groups.

SELECT
    CASE WHEN order_amount <= 100 THEN 'up to 100'
        WHEN order_amount <= 500 THEN '100 - 500'
        ELSE '500+' END AS amount_bin,
    CASE WHEN order_amount <= 100 THEN 'small'
        WHEN order_amount <= 500 THEN 'medium'
        ELSE 'large' END AS amount_category,
    COUNT(customer_id) AS customer
FROM orders
GROUP BY 1,2
;
```

## c. n-Tiles:

```
# Instead of arbitrary-sized bins, particular shape bins can be achieved by
# using {rounding / logarithms / n-tiles} numeric function.

SELECT ROUND(sales,-1) AS bin          SELECT LOG(10,sales) AS bin #for skewed dist.
,COUNT(customer_id) AS customers      ,COUNT(customer_id) AS customers
FROM table                           FROM table
GROUP BY 1                           GROUP BY 1
;
```

[# Create equal-width bins.]

Decimal places	Formula	Result
2	round(123456.789,2)	123456.79
1	round(123456.789,1)	123456.8
0	round(123456.789,0)	123457
-1	round(123456.789,-1)	123460
-2	round(123456.789,-2)	123500
-3	round(123456.789,-3)	123000

[# Create increase size bins following a useful pattern.]

Formula	Result
log(1)	0
log(10)	1
log(100)	2
log(1000)	3
log(10000)	4

**Note:** Logarithm function will return null / error on  $>=0$  value depending on database.

```
# Create n-quantiles of the data

SELECT ntile,
       MIN(order_amount) AS lower_bound,
       MAX(order_amount) AS upper_bound,
       COUNT(order_id) AS orders
FROM
(
    SELECT customer_id, order_id, order_amount,
           NTILE(10) OVER (ORDER BY order_amount) AS ntile
    FROM orders
) a
GROUP BY 1
;
```

\*\* Both NTILE & PERCENT\_RANK can be expensive to compute over large dataset as it requires sorting of rows.  
(filter to only data needed can help optimize process)

## II. Profiling: Data Quality

- One common consistency check is, comparing data against what is known to be true.  
(eg. Compare the row counts in each system to verify that the data is complete with all rows arrived & received when working with a replica of production database )

### a. Detecting Duplicates:

( Often figuring out why duplicate occurs from its repeated trait is useful for improving work process )  
—eg. accident with a hidden many-to-many JOIN, multiple tracking call in ETL, or manual steps etc.

```
WITH Duplicate_rows AS          OR          SELECT c1, c2, c3, ..., COUNT(*)  
(                                         FROM table  
    SELECT *                                GROUP BY 1,2,3...  
    , ROW_NUMBER() OVER(                      HAVING COUNT(*) > 1  
        PARTITION BY c1, c2, ...) AS row_count ;  
    FROM t                                     [# List out all the fully duplicate rows.]  
)  
  
SELECT *  
FROM Duplicate_rows  
WHERE row_count > 1  
;  
[# List out all the partially duplicate rows in data.]
```

### b. Removing duplications by DISTINCT / GROUP BY:

```
# For nonduplicate records in a single table:  
SELECT DISTINCT * FROM t /   DELETE FROM duplicate_cte WHERE row_num > 1 --(1)  
  
# For nonduplicate records of customers that appear in the other table only:  
SELECT DISTINCT a.customer_id, a.customer_name, a.customer_email --(2)  
FROM customers a --alias  
JOIN transaction b ON a.customer_id = b.customer_id
```

Another useful approach is to perform an aggregation that returns 1-row per entity:

—eg. we can return 1-record per customer with any aggregated info when we have a no. of transactions by the same customers.

```
SELECT customer_id  
, MIN(transaction_date) AS first_transaction_date  
, MAX(transaction_date) AS last_transaction_date  
, COUNT(*) AS total_orders  
FROM table  
GROUP BY 1  
;
```

### c. Return most recent records with Min / Max value Filtering

```
-- Return the most recent sales date for each sales rep
SELECT sales_rep, MAX(date) AS most_recent_date
FROM sales
GROUP BY sales_rep;
```

(1) -- Number of sales on most recent date: Group by + join approach

```
WITH rd AS (SELECT sales_rep, MAX(date) AS most_recent_date
            FROM sales
            GROUP BY sales_rep)
```

```
SELECT rd.sales_rep, rd.most_recent_date, s.sales
FROM rd LEFT JOIN sales s
ON rd.sales_rep = s.sales_rep
AND rd.most_recent_date = s.date;
```

(2) -- Number of sales on most recent date: Window function approach

```
SELECT * FROM
```

```
(SELECT sales_rep, date, sales,
       ROW_NUMBER() OVER (PARTITION BY sales_rep ORDER BY date DESC) AS row_num
FROM sales) AS rn
```

```
WHERE row_num = 1;
```

id	sales_rep	date	sales
1	Emma	2024-08-01	6
2	Emma	2024-08-02	17
3	Jack	2024-08-02	14
4	Emma	2024-08-04	20
5	Jack	2024-08-05	5
6	Emma	2024-08-07	1
NULL	NULL	NULL	NULL



sales_rep	date	sales	row_num
Emma	2024-08-07	1	1
Jack	2024-08-05	5	1

### III. Data Cleaning

#### a. Handling missing value

-- Examine records with NULL / Blank values

```
SELECT *
FROM StagingTable
WHERE c1 IS NULL OR c1 = '';
```

-- Drop rows with missing value

```
DELETE
FROM StagingTable
WHERE c1 IS NULL OR c2 IS NULL;
```

-- Drop columns

```
ALTER TABLE StagingTable
DROP COLUMN c1;
```

-- Validate Outcome & Populate with known data

```
SELECT t1.industry, t2.industry
FROM layoffs_staging2 t1
JOIN layoffs_staging2 t2
ON t1.company = t2.company
WHERE (t1.industry IS NULL OR t1.industry = '')
AND t2.industry IS NOT NULL;
```

```
UPDATE layoffs_staging2 t1
JOIN layoffs_staging2 t2
ON t1.company = t2.company
SET t1.industry = t2.industry
WHERE ...
AND ...
```

-- Sometimes, we can detect missing data by comparing values from two tables

```
SELECT DISTINCT t1.emp_name, t1.emp_dept, t2.location_name
FROM Employee_Data AS t1
RIGHT JOIN Department_Data AS t2 ON t1.emp_dept = t2.department_name
WHERE t1.emp_name IS NULL
;
```

emp_name	emp_dept	location_name
Gaurav	HR	Building 1 => NULL Marketing/Sales Building 3
Anjali	IT	Building 2 ...
NULL	Marketing/Sales	Building 3
...		

**Before making any changes, it is essential to evaluate the missing data first.**

—Missing data is not always an ill-condition that needed to be fixed right away, but can be an important signal to reveal underlying design flaws or biases in the data collection process.



## IMPUTING NULL VALUES BY:

1. Hard coded value (integer)
2. Average of a column (subquery)
3. Prior row's value (window function)
4. Smoothed value (two window functions)

```
#1 - CASE WHEN c1 IS NULL AND c2 = 'name' THEN 600 ELSE c1 END  
- SELECT *, COALESCE(c1, 600) AS updated_by_600  
  
#2. SELECT *, COALESCE(c1, (SELECT AVG(c1) FROM t) AS updated_by_avg  
  
#3. SELECT *, COALESCE(c1, LAG/LEAD(price) OVER(PARTITION BY product  
ORDER BY order_date))  
AS updated_price_avg  
  
#4. SELECT *, COALESCE(c1, (LAG(price) OVER() + LEAD(price) OVER())/2)  
AS updated_price_smooth
```

::Imputing values based on statistical metric/interpolation/ insightful assumption about typical value (eg. price of the same date).

**Note:** <sup>1</sup> Similar to Python, null value is contagious where any operation with it will result in null.

<sup>2</sup> In PostgreSQL & Oracle, NULL is considered larger than any non-NULL value when ordered; Vice-versa for SQLite, MySQL & SQL Server.

Beware of empty strings disguise as NULL, where there is no visible value present in a cell.

( Sometimes, empty string - ' ' is appropriate as value overcoming NOT NULL constraint when there is no explicit / suitable answer to a question and leaving it empty is not an option. )

# Collectively, LENGTH() & TRIM() can be used to check and remove unintended chr / spaces.

### b. Standardize the data

```
-- Trim excessive chr, whitespace, or carriage return
UPDATE t SET c1 = TRIM(TRAILING '.' FROM c1) / REPLACE(c1, '\r', '')

-- Rectify string of a class

CASE WHEN gender = 'F' THEN 'Female'
     WHEN gender = 'female' THEN 'Female'
     WHEN gender = 'femme' THEN 'Female'
     ELSE gender
END AS gender_cleaned

CASE WHEN likelihood IN (0,1,2,3,4,5,6) THEN 'Detractor'
     WHEN likelihood IN (7,8) THEN 'Passive'
     WHEN likelihood IN (9,10) THEN 'Promoter'
     END
```

---

@ for multiple columns consideration,

```
CASE WHEN likelihood <= 6 AND country = 'US' AND high_value = TRUE
      THEN 'US high value detractor'
      WHEN likelihood >= 9 AND (country IN ('CA', 'JP') OR high_value = TRUE)
      THEN 'some other label'
      END
```

**Note:** The CASE statement works well only for a relatively short list of values that isn't expected to change. Otherwise, the utility table would be a better option.

Another use is to create dummy vars for statistical / numerical analysis.

```
SELECT customer_id
,CASE WHEN gender = 'F' THEN 1 ELSE 0 END AS is_female
,CASE WHEN likelihood IN (9,10) THEN 1 ELSE 0 END AS is_promoter
FROM table
;
```

### c. Creating flags

—eg. identify if a customer belongs to a treatment group based on the no. of matching conditions from various transactions in a dataset that has multiple rows per entity: name / id.

```
SELECT customer_id,
       MAX(CASE WHEN fruit = 'apple' AND quantity > 5 THEN 1 ELSE 0
              END) AS loves_apples
       MAX(CASE WHEN fruit = 'orange' AND quantity > 5 THEN 1 ELSE 0
              END) AS loves_oranges
FROM t
GROUP BY 1
;
```

## d. Datatype Conversion

(Type conversion function): `cast(c1 AS VARCHAR)` or, `c1::VARCHAR`

```
# To manipulate a string field for numerical application:  
SELECT REPLACE('$19.99', '$', '')::float or, CAST(REPLACE('$19.99', '$', '') AS float)  
  
# To concatenate strings from different fields: (eg. time-series data)  
SELECT (year || ' ' || month || '-' || day)::date or, CAST(CONCAT(year, '-', month, '-', day) AS date);  
, COUNT(transactions) AS num_trx           MAKE_DATE(2020, 09, 01);  
FROM table  
GROUP BY 1  
;
```

[# Double pipe ||, is a concatenation operator to assemble values from separate columns.]

```
# If categorization of numeric values w/ upper / lower bound is needed:  
CASE WHEN order_items <= 3 THEN order_items::VARCHAR  
ELSE '4+'  
END
```

[# This is especially handy when database does not support type coercion (eg. INT → FLOAT)  
— population / 5.0 dividing integer by a float, and requires data to be explicitly converted.]

<code>to_char(timestamp, text) → text</code>
<code>to_char(timestamp with time zone, text) → text</code> Converts time stamp to string according to the given format. <code>to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS') → 05:31:12</code>
<code>to_char(interval, text) → text</code> Converts interval to string according to the given format. <code>to_char(interval '15h 2m 12s', 'HH24:MI:SS') → 15:02:12</code>
<code>to_char(numeric_type, text) → text</code> Converts number to string according to the given format; available for integer, bigint, numeric, real, double precision. <code>to_char(125, '999') → 125</code> <code>to_char(125.8::real, '999D9') → 125.8</code> <code>to_char(-125.8, '999D99S') → 125.80-</code>
<code>to_date(text, text) → date</code> Converts string to date according to the given format. <code>to_date('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05</code>
<code>to_number(text, text) → numeric</code> Converts string to numeric according to the given format. <code>to_number('12,454.8-', '99G999D9S') → -12454.8</code>
<code>to_timestamp(text, text) → timestamp with time zone</code> Converts string to time stamp according to the given format. (See also <code>to_timestamp(double precision)</code> in <a href="#">Table 9.33</a> .) <code>to_timestamp('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05 00:00:00-05</code>

(For more pointed behavior, the dedicated datatype conversion function may be used.)

## e. Converting datetime field into different levels of granularity

date	day_of_month	day_of_year	day_of_week	day_name	week	month_number	month_name	quarter_number	quarter_name	year	decade
2000-01-01	1	1	6	Saturday	1999-12-27		1 January		1 Q1	2000	2000
2000-01-02	2	2	0	Sunday	1999-12-27		1 January		1 Q1	2000	2000
2000-01-03	3	3	1	Monday	2000-01-03		1 January		1 Q1	2000	2000
2000-01-04	4	4	2	Tuesday	2000-01-03		1 January		1 Q1	2000	2000
2000-01-05	5	5	3	Wednesday	2000-01-03		1 January		1 Q1	2000	2000
2000-01-06	6	6	4	Thursday	2000-01-03		1 January		1 Q1	2000	2000
2000-01-07	7	7	5	Friday	2000-01-03		1 January		1 Q1	2000	2000
2000-01-08	8	8	6	Saturday	2000-01-03		1 January		1 Q1	2000	2000
2000-01-09	9	9	0	Sunday	2000-01-03		1 January		1 Q1	2000	2000
2000-01-10	10	10	1	Monday	2000-01-10		1 January		1 Q1	2000	2000
2000-01-11	11	11	2	Tuesday	2000-01-10		1 January		1 Q1	2000	2000
2000-01-12	12	12	3	Wednesday	2000-01-10		1 January		1 Q1	2000	2000
2000-01-13	13	13	4	Thursday	2000-01-10		1 January		1 Q1	2000	2000
2000-01-14	14	14	5	Friday	2000-01-10		1 January		1 Q1	2000	2000
2000-01-15	15	15	6	Saturday	2000-01-10		1 January		1 Q1	2000	2000
2000-01-16	16	16	0	Sunday	2000-01-10		1 January		1 Q1	2000	2000
2000-01-17	17	17	1	Monday	2000-01-17		1 January		1 Q1	2000	2000
2000-01-18	18	18	2	Tuesday	2000-01-17		1 January		1 Q1	2000	2000
2000-01-19	19	19	3	Wednesday	2000-01-17		1 January		1 Q1	2000	2000
2000-01-20	20	20	4	Thursday	2000-01-17		1 January		1 Q1	2000	2000
2000-01-21	21	21	5	Friday	2000-01-17		1 January		1 Q1	2000	2000
2000-01-22	22	22	6	Saturday	2000-01-17		1 January		1 Q1	2000	2000
2000-01-23	23	23	0	Sunday	2000-01-17		1 January		1 Q1	2000	2000
2000-01-24	24	24	1	Monday	2000-01-24		1 January		1 Q1	2000	2000
2000-01-25	25	25	2	Tuesday	2000-01-24		1 January		1 Q1	2000	2000
2000-01-26	26	26	3	Wednesday	2000-01-24		1 January		1 Q1	2000	2000

- A sample data dimension table with extensible breakdown of date components since a given date.

Through a JOIN to the created or manually generated date dimension table using `generate_series(start, stop, step)` function as followed:

```

SELECT a.generate_series AS order_date, b.customer_id, b.items      ----(1)
FROM
(
    SELECT * FROM GENERATE_SERIES('2020-01-01'::date, '2020-12-31'::date, '1 day')
) a
LEFT JOIN
(
    SELECT customer_id, order_date, COUNT(item_id) AS items
    FROM Orders
    GROUP BY 1,2
) b ON a.generate_series = b.order_date

```

**Note:** With such a table we can ensure that a query returns a result for every date of interest, whether or not there was a record for that date in the underlying dataset.

If a date dimension is not available in database, a subquery can be used to simulate one by `SELECT`-ing the `DISTINCT` dates from any source that has the timeseries needed and `JOIN`.

```

SELECT          ----(2)
    a.date, b.customer_id
    , b.subscription_date, b.annual_amount / 12 AS monthly_subscription
FROM
(
    SELECT DISTINCT sales_month
    FROM RetailSales
    WHERE sales_month BETWEEN '2020-01-01' AND '2020-12-31'
) a
JOIN CustomerSubscriptions b ON a.date BETWEEN b.subscription_date
    AND b.subscription_date + INTERVAL '11 months'

```

---

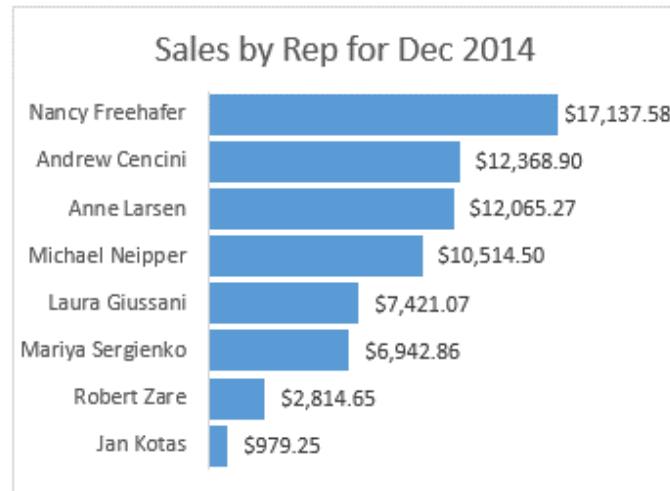
\* For sample code to create the date dimension utility table, refer. [GitHub site](#).

## IV. Pivot Table

- This can be achieved using CASE statements
- PIVOT is available in some RDBMS's like SQL Server and Oracle

### a. Pivoting

Row Labels	Sum of Revenue
Nancy Freehafer	\$17,137.58
Andrew Cencini	\$12,368.90
Anne Larsen	\$12,065.27
Michael Neipper	\$10,514.50
Laura Giussani	\$7,421.07
Mariya Sergienko	\$6,942.86
Robert Zare	\$2,814.65
Jan Kotas	\$979.25
<b>Grand Total</b>	<b>\$70,244.08</b>



- Summarize data by transforming unique values in rows into columns.

It is typically done with conditional aggregations to shape data in a more readable format:

```
SELECT order_date
, SUM(CASE WHEN product = 'shirt' THEN order_amount ELSE 0 END) AS shirts_amount
, SUM(CASE WHEN product = 'shoes' THEN order_amount ELSE 0 END) AS shoes_amount
, SUM(CASE WHEN product = 'hat' THEN order_amount ELSE 0 END) AS hats_amount
FROM orders
GROUP BY 1
;
order_date    shirts_amount    shoes_amount    hats_amount
-----
2020-05-01    5268.56        1211.65        562.25
2020-05-02    5533.84        522.25         325.62
2020-05-03    5986.85        1088.62        858.35
...           ...           ...           ...
```

[# As SQL pivoting employs explicit categorization with CASE statement, it is not suitable for rapid changing / expanding datasets and advisable to compute in other analysis tool.]

**Note:** ELSE statement is included for sum aggregation to avoid NULL, otherwise ignored in COUNT where a substitute value could inflate unnecessary count.

## b. Unpivoting

```
# To move data stored in columns into rows to create tidy data:  
SELECT country,'1980' AS year,year_1980 AS population FROM country_populations  
UNION ALL  
SELECT country,'1990' AS year,year_1990 AS population FROM country_populations  
UNION ALL  
SELECT country,'2000' AS year,year_2000 AS population FROM country_populations  
UNION ALL  
SELECT country,'2010' AS year,year_2010 AS population FROM country_populations  
;
```

```
Postgres -> SELECT country  
    ,UNNEST(array['1980', '1990', '2000', '2010']) AS year  
    ,UNNEST(array[year_1980, year_1990, year_2000, year_2010]) AS pop  
FROM country_populations  
;
```

\* PostgreSQL offers array data structure for minor elements collection as object relational database.

Country	year_1980	year_1990	year_2000	year_2010	country	year	population
Canada	24,593	27,791	31,100	34,207	Canada	1980	24593
Mexico	68,347	84,634	99,775	114,061	Mexico	1980	68347
United States	227,225	249,623	282,162	309,326	United States	1980	227225
					...	...	...

[# Integrate selections from multiple queries w/ compatible dtype using UNION statement.]

Recognizing that the pivot & unpivot are common, some databases have created its dedicated functions that operate in similar manner: (Microsoft SQL Server & Snowflake)

<pre>SELECT ... FROM ... pivot(aggregation(value_c1)       for label_c2 in ('cate1', 'cate2', ...)) GROUP BY ... ;</pre>	<pre>SELECT * FROM country_populations unpivot(population)       for year in (year_1980, year_1990, ...)</pre>
--	--

(Although the syntax is more compact than CASE construction, the desired cols still need to be specified and it doesn't solve the problem for newly arriving or rapidly changing sets of fields.)

## V. Rolling Calculations

### a. Subtotals

order_id	customer_name	order_date	pizza_name	price
1	Jack	2024-12-01	Pepperoni	18.75
2	Jack	2024-12-02	Pepperoni	18.75
3	Jack	2024-12-03	Pepperoni	18.75
4	Jack	2024-12-04	Pepperoni	18.75
5	Jack	2024-12-05	Spicy Italian	22.75
6	Jill	2024-12-01	Five Cheese	18.50
7	Jill	2024-12-03	Margherita	19.50
8	Jill	2024-12-05	Garden Delight	21.50
9	Jill	2024-12-05	Greek	21.50
10	Tom	2024-12-02	Hawaiian	19.50
11	Tom	2024-12-04	Chicken Pesto	20.75
12	Tom	2024-12-05	Spicy Italian	22.75
13	Jerry	2024-12-01	California Chi...	21.75
14	Jerry	2024-12-02	Margherita	19.50
15	Jerry	2024-12-04	Greek	21.50

Subtotals



customer_name	order_date	total_sales
Jack	2024-12-01	18.75
Jack	2024-12-02	18.75
Jack	2024-12-03	18.75
Jack	2024-12-04	18.75
Jack	2024-12-05	22.75
Jack	HULL	97.75
Jerry	2024-12-01	21.75
Jerry	2024-12-02	19.50
Jerry	2024-12-04	21.50
Jerry	HULL	62.75
Jill	2024-12-01	18.50
Jill	2024-12-03	19.50
Jill	2024-12-05	43.00
Jill	HULL	81.00
Tom	2024-12-02	19.50
Tom	2024-12-04	20.75
Tom	2024-12-05	22.75
Tom	HULL	63.00
HULL	HULL	304.50

```
SELECT customer_name, order_date, SUM(price) AS total_sales
FROM PizzaOrders
GROUP BY customer_name, order_date WITH ROLLUP;
```

### b. Cumulative sum

order_id	customer_name	order_date	pizza_name	price
1	Jack	2024-12-01	Pepperoni	18.75
2	Jack	2024-12-02	Pepperoni	18.75
3	Jack	2024-12-03	Pepperoni	18.75
4	Jack	2024-12-04	Pepperoni	18.75
5	Jack	2024-12-05	Spicy Italian	22.75
6	Jill	2024-12-01	Five Cheese	18.50
7	Jill	2024-12-03	Margherita	19.50
8	Jill	2024-12-05	Garden Delight	21.50
9	Jill	2024-12-05	Greek	21.50
10	Tom	2024-12-02	Hawaiian	19.50
11	Tom	2024-12-04	Chicken Pesto	20.75
12	Tom	2024-12-05	Spicy Italian	22.75
13	Jerry	2024-12-01	California Chi...	21.75
14	Jerry	2024-12-02	Margherita	19.50
15	Jerry	2024-12-04	Greek	21.50

Cumulative sum



order_date	cumulative_sum
2024-12-01	59.00
2024-12-02	116.75
2024-12-03	155.00
2024-12-04	216.00
2024-12-05	304.50

```
WITH ts AS (SELECT order_date, SUM(price) AS total_sales
FROM PizzaOrders
GROUP BY order_date)
```

```
SELECT order_date,
SUM(total_sales) OVER(ORDER BY order_date) AS cumulative_sum
FROM ts;
```

### c. Moving Averages



The diagram illustrates the process of calculating moving averages. On the left, there is a table of raw data for two countries: Afghanistan and Albania. The columns are 'country', 'year', and 'happiness\_score'. The data shows the happiness score for each year from 2015 to 2020. On the right, there is a second table showing the 'moving\_average' for each country. This table includes the same three columns as the raw data, plus a fourth column 'moving\_average'. The 'moving\_average' is calculated as the average of the current year's score and the scores of the previous two years. For example, for Afghanistan, the moving average starts at 3.575 in 2015 and decreases to 3.134 in 2020. For Albania, it starts at 4.959 in 2015 and decreases to 4.729 in 2020.

country	year	happiness_score	
Afghanistan	2015	3.575	
Afghanistan	2016	3.360	
Afghanistan	2017	3.794	
Afghanistan	2018	3.632	
Afghanistan	2019	3.203	
Afghanistan	2020	2.567	
Afghanistan	2021	2.523	
Afghanistan	2022	2.404	
Afghanistan	2023	1.859	
Albania	2015	4.959	
Albania	2016	4.655	
Albania	2017	4.644	
Albania	2018	4.586	
Albania	2019	4.719	
Albania	2020	4.883	

country	year	happiness_score	moving_average
Afghanistan	2015	3.575	3.575
Afghanistan	2016	3.360	3.468
Afghanistan	2017	3.794	3.576
Afghanistan	2018	3.632	3.595
Afghanistan	2019	3.203	3.543
Afghanistan	2020	2.567	3.134
Afghanistan	2021	2.523	2.764
Afghanistan	2022	2.404	2.498
Afghanistan	2023	1.859	2.262
Albania	2015	4.959	4.959
Albania	2016	4.655	4.807
Albania	2017	4.644	4.753
Albania	2018	4.586	4.628
Albania	2019	4.719	4.650
Albania	2020	4.883	4.729

- Calculate the 3 year moving average of happiness score for each country.

```

SELECT country, year, happiness_score,
       ROUND(AVG(happiness_score)
OVER(PARTITION BY country ORDER BY year
      ROWS BETWEEN 2 PRECEDING AND CURRENT ROW), 3) AS moving_average
FROM happiness_scores
ORDER BY country, year;
    
```

# TIME SERIES ANALYSIS

**Areas to pay attention** when aligning data from different sources with JOIN / UNION:

- (1) **Consistent formatting & time zone** to serialize the data (eg. UTC), or isolate time zone to a separate field so that timestamp can be converted as needed.
- (2) **Look out for timestamps from different sources that are slightly out of sync** (eg. recorded on client devices instead of common server), and can be fixed by adjusting the time window for complete records with BETWEEN & date math.
- (3) **When dealing with data from mobile apps, pay particular attention to how the timestamps were recorded-** when the action happened on the device or arrived in the database (eg. real-time / store & forward) which datetime can be corrupted.

1

## Time-Based Aggregation

### Monthly Aggregation

```
SELECT  
    DATE_TRUNC('month',  
    order_date) AS month,  
    SUM(sales) AS  
    total_sales  
FROM sales_data  
GROUP BY  
    DATE_TRUNC('month',  
    order_date)  
ORDER BY month;
```

2

## Moving Average

### 3-Period Moving Average

```
SELECT  
    order_date,  
    sales,  
    AVG(sales) OVER (  
        ORDER BY order_date  
        ROWS BETWEEN 2  
        PRECEDING AND CURRENT ROW  
    ) AS moving_avg_3  
FROM sales_data;
```

3

## Year-over-Year

```
SELECT EXTRACT(YEAR FROM  
order_date) AS year,  
SUM(sales),  
LAG(SUM(sales)) OVER  
(ORDER BY year) AS  
prev_year,  
(SUM(sales) - prev_year) *  
100.0 / NULLIF(prev_year,  
0) AS yoy_growth  
FROM sales_data  
GROUP BY year  
ORDER BY year;
```

4

## Cumulative Total

```
SELECT  
    DATE_TRUNC('month',  
    order_date) AS month,  
    SUM(sales) AS  
    total_sales  
FROM sales_data  
GROUP BY  
    DATE_TRUNC('month',  
    order_date)  
ORDER BY month;
```

5

## Same Day, Previous Year Comparison

```
SELECT  
    order_date,  
    sales,  
    LAG(sales, 365) OVER  
(ORDER BY order_date) AS  
sales_last_year  
FROM sales_data;
```

Assumes daily data and no missing dates. For accurate yearly lag, consider using INTERVAL joins instead.

6

## Sales by Day of Week (Seasonality)

```
SELECT  
    TO_CHAR(order_date,  
'Day') AS day_of_week,  
    ROUND(AVG(sales), 2) AS  
avg_sales  
FROM sales_data  
GROUP BY  
    TO_CHAR(order_date, 'Day')  
ORDER BY avg_sales DESC;
```

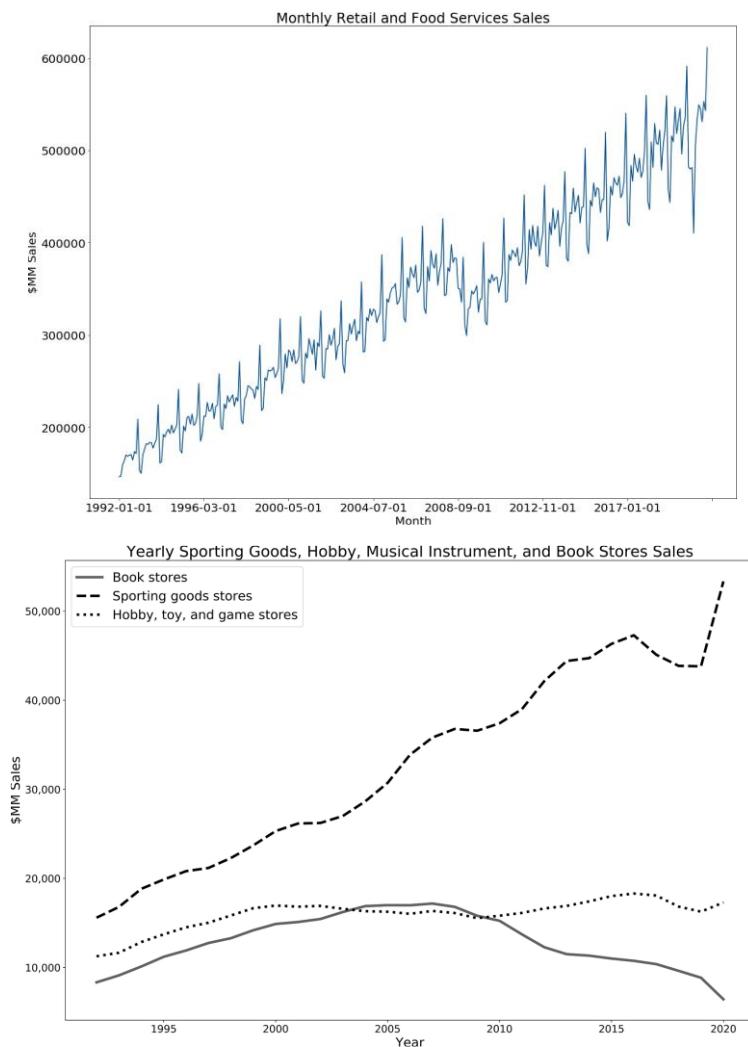
## VI. Exploring trends of time series data

With time series data, we often want to look for trends- <sup>①</sup> to examine for unusual events and/or <sup>②</sup> depict the typical pattern in parts as comparisons or whole component in the data.  
— ( ie. how a measured property / metric changes over time )

```
SELECT DATE_PART('year', sales_month) AS sales_year, ---- Simple trends
       kind_of_business, SUM(sales) AS sales
FROM RetailSales
WHERE kind_of_business
IN ('Book stores', 'Sporting goods stores', 'Hobby, toy, and game stores')
GROUP BY 1,2
;
:: sales_year kind_of_business           sales      -- X: dates / timestamps ( here, yearly )
1992.0     Book stores                 8327      -- Y: numerical values
1992.0     Hobby, toy, and game stores  11251
1992.0     Sporting goods stores      15583
...
...
```

[# Usually, data is prepared in SQL and fed into other charting tool to create visualizations. ]

**! All currency trx / operations are usually adjusted for inflation to reflect their true nature.**



- Create a smoother time series data (trend) by transforming datetime & aggregating at yearly level to help us gain a better understanding of the pattern, separating from the noise.

## Comparing Components

(eg. to quantify the difference over time by: <sup>1</sup> size / <sup>2</sup> ratio – sales amt or, pct of total)

```

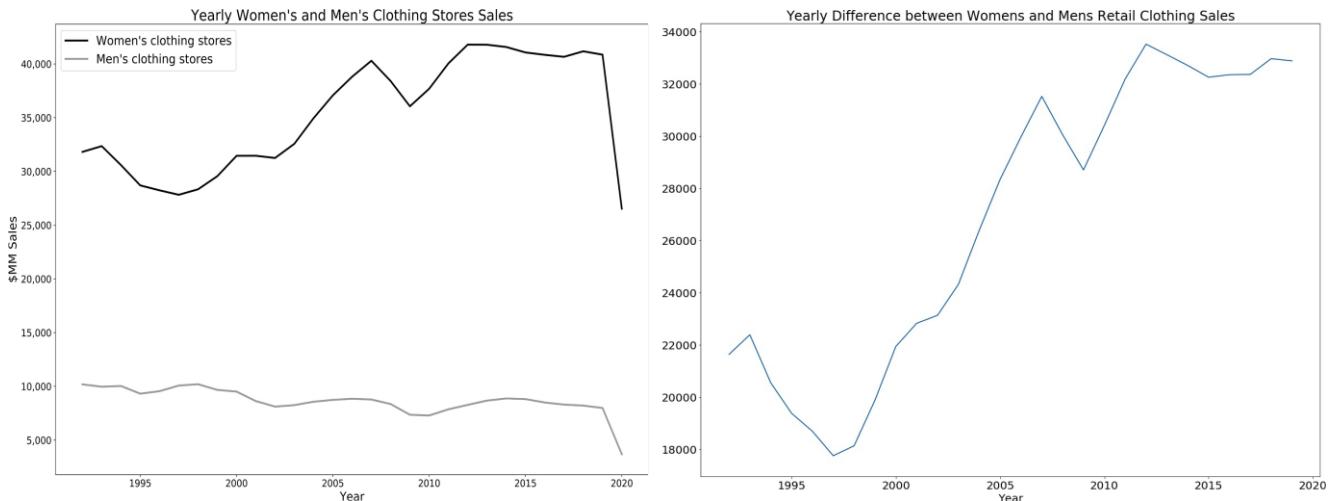
SELECT DATE_PART('year', sales_month) AS sales_year ----(1)
    ,SUM(CASE WHEN kind_of_business = "Women's clothing stores" THEN sales END)
    -
    SUM(CASE WHEN kind_of_business = "Men's clothing stores" THEN sales END)
AS womens_minus_mens
FROM RetailSales
WHERE kind_of_business IN ("Men's clothing stores", "Women's clothing stores")
AND sales_date <= '2019-12-01'
GROUP BY 1
;

sales_year  womens_minus_mens  /  womens_pct_of_mens
1992.0      21636            212.5552608311229000
1993.0      22388            224.7339891588034500
1994.0      20553            204.8744019138756000
...
...
```

```

SELECT sales_year, (womens_sales / mens_sales - 1) * 100 AS womens_pct_of_mens
FROM
-----(2)
(
    SELECT DATE_PART('year', sales_month) AS sales_year
    ,SUM(CASE WHEN kind_of_business = "Womens's clothing stores" THEN sales
              END) AS womens_sales
    ,SUM(CASE WHEN kind_of_business = "mens's clothing stores" THEN sales
              END) AS mens_sales
    FROM RetailSales
    WHERE kind_of_business IN ("Men's clothing stores", "Women's clothing stores")
    AND sales_month <= '2019-12-01'
    GROUP BY 1
) derived_t
;
```



[# It shows that the gap decreased btw 1992 and about 1997, began a long increase through about 2011 (with a brief dip in 2007), and then was moving more or less flat through 2019. ]

---

NOTE: Percentage / Ratio is used to quantify sales under influence of the market on both categories than simple comparison.

## Percentage of segment / category to Total

(eg. men's & women's clothing sales from Total / monthly sales from each year sales)

#1 Computing combined men's & women's sales from total sales:

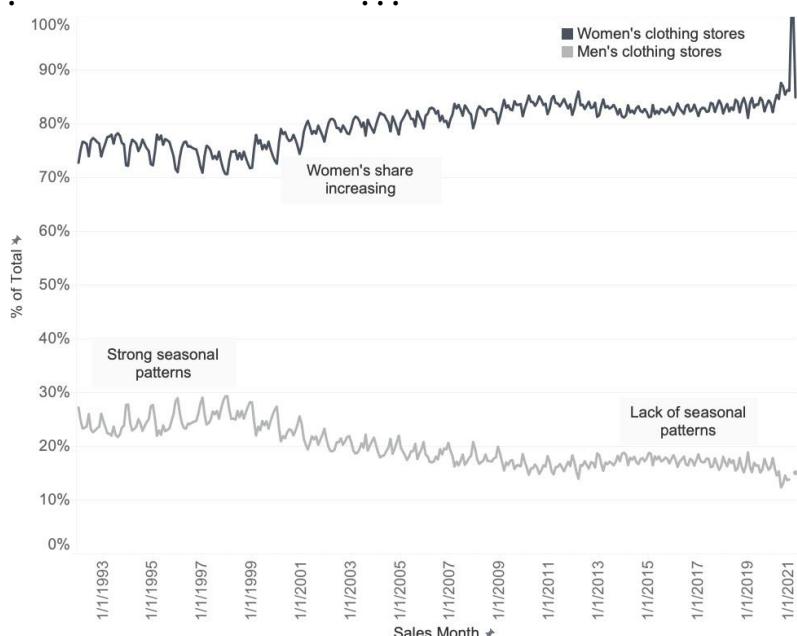
```
SELECT sales_month, kind_of_business, sales, total_sales
, SUM(sales) OVER (PARTITION BY sales_month) AS total_sales
,sales / SUM(sales) * 100 OVER (PARTITION BY sales_month) AS pct_total
FROM RetailSales
WHERE kind_of_business IN ("Men's clothing stores", "Women's clothing stores")
;

-- OR --
```

```
SELECT sales_month, kind_of_business, sales / total_sales * 100 pct_total_sales
FROM
```

```
(  
    SELECT a.sales_month, a.kind_of_business, a.sales, SUM(b.sales) total_sales
    FROM RetailSales t1
    JOIN RetailSales t2 ON --Cartesian JOIN of 2 rows from t2 for each row of t1- total clothing
        t1.sales_month = t2.sales_month
        AND t2.kind_of_business IN ("Men's clothing stores"
            , "Women's clothing stores")
    WHERE t1.kind_of_business IN ("Men's clothing stores"
        , "Women's clothing stores")
    GROUP BY 1,2,3
) subquery
;
```

sales_month	kind_of_business	sales <i>optional</i>	total_sales <i>optional</i>	pct_total_sales
1992-01-01	Men's clothing stores	701	2574	27.23
1992-01-01	Women's clothing stores	1873	2574	72.77
1992-02-01	Men's clothing stores	1991	2649	75.16
...	...	...		



- Women's clothing sales became an increasing pct of the total starting in the late 1990s.
- Seasonal patterns in men's sales spike as pct of total in DEC & JAN, but by the late of 2010s they're damped almost to the point of randomness.

