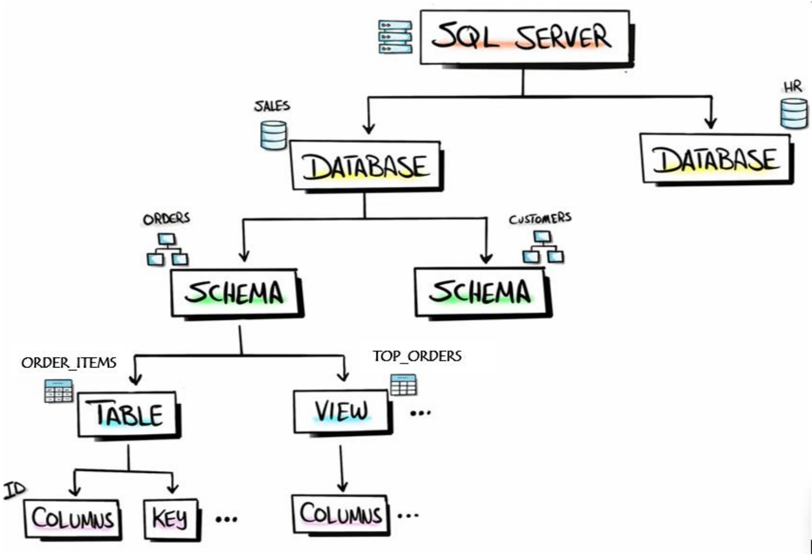
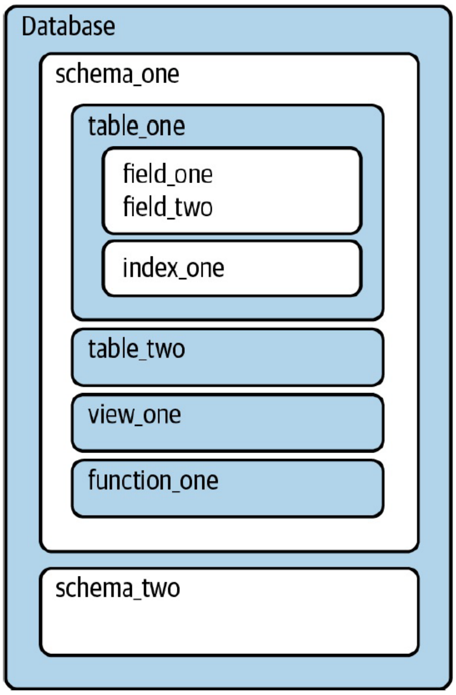
SQL Database Structure

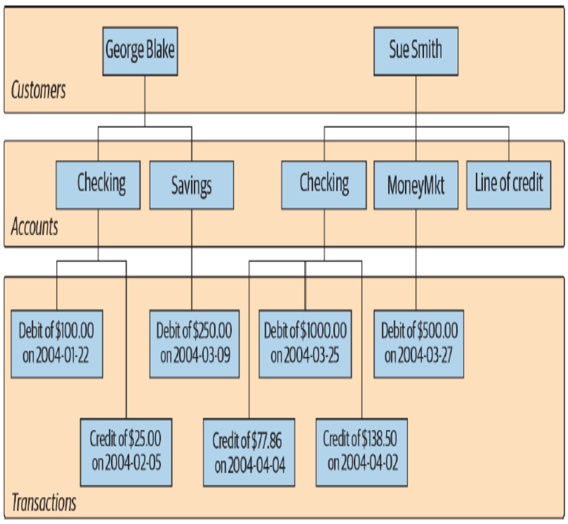
Database systems, are reliable & systematic computerized **data storage** and **access** mechanisms.

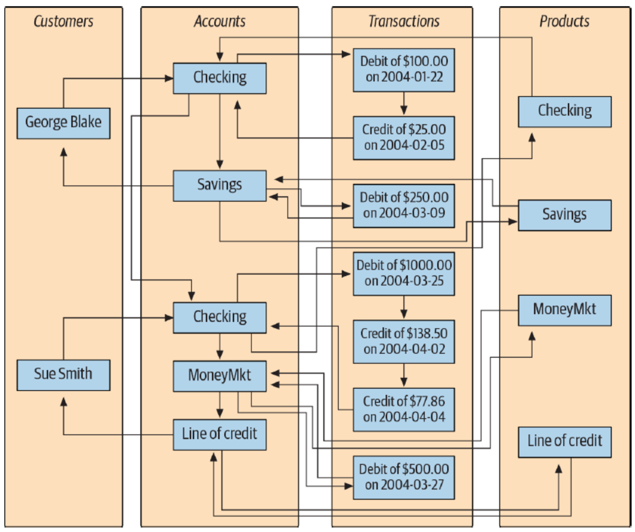




This, database schema {*explicit structure & type categorizations*} can be thought as the working template of storage containing built-in functionalities, model properties, and demonstration of object.

(These two types of data management structures- hierchical & network are fundamental to today's structural system such as: file directory & relational model)





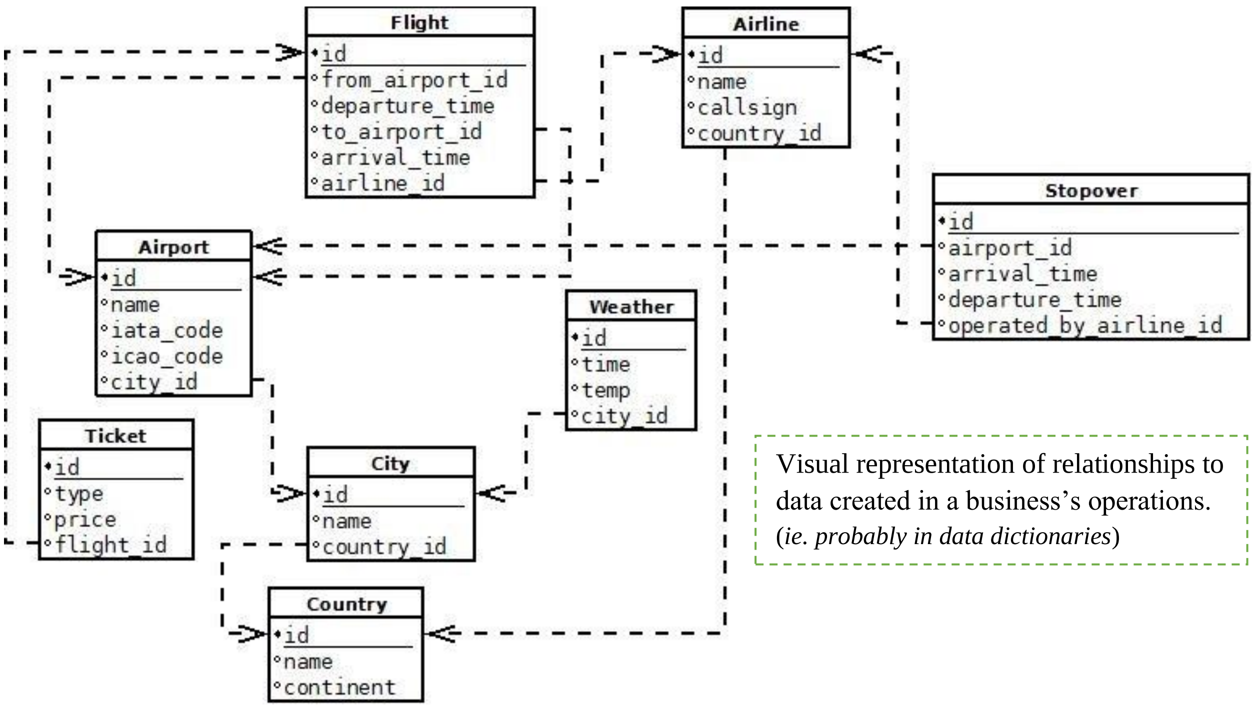
The single parent hierarchy may look clean & easy to follow even at large dataset, but it is difficult for arbitrary retrieval due to its transversing constraint from the root nodes.

There are 2-broad categories intended for store & access of data: row-store & column-store.

RELATIONAL MODEL

- a row-store mapping designed for organizing data elements -how they relate to one another, and efficient at managing transactions in databases.

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



Visual representation of relationships to data created in a business's operations. (*ie. probably in data dictionaries*)

[a design seeks to reduce the width of tables and avoid duplication & inconsistencies, but called for many joins during processing of a complete dataset]

A table might incl.—

(i) Primary key - unique identifier for a record in a table,

(ii) Foreign key - PK defined in other tables, as links btw tables,

(iii) Logical key - indexes that allow effective lookup from outside the program, eg. title

- ❖ Some databases may take a compound type (ie. multiple fields' values) as primary key, which is the combination of natural & surrogate key that enforces uniqueness.

  (the choice should be made carefully, as it is not allowed to be changed once assigned)

Usually the primary key is indexed,

- ❖ when a query takes long time to run, it can be useful to identify fields w/ indexing property and adding it to speed up searches on common joins.

  (but – slow down commit as new values added at each insert)

#Beyond reporting, exploratory w/ complex queries & join patterns are often conducted than bothering with optimizing indexes.

When building a relational model for an application,

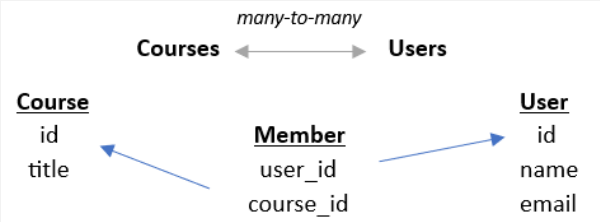
#1 Start w/ first table –"the one {main} description of application" (*eg.* Music API = a map that manages tracks)

#2 Brainstorm to see what kind of information is helpful to include.

#3 Do not put the same string twice, separate main feature to its attrs using relationships.

#4 Create a connection table w/ foreign keys for many-to-many relation / ≥3 JOIN tables.

(*as multiple values are generally not encouraged within a single cell in table – separating PK*)



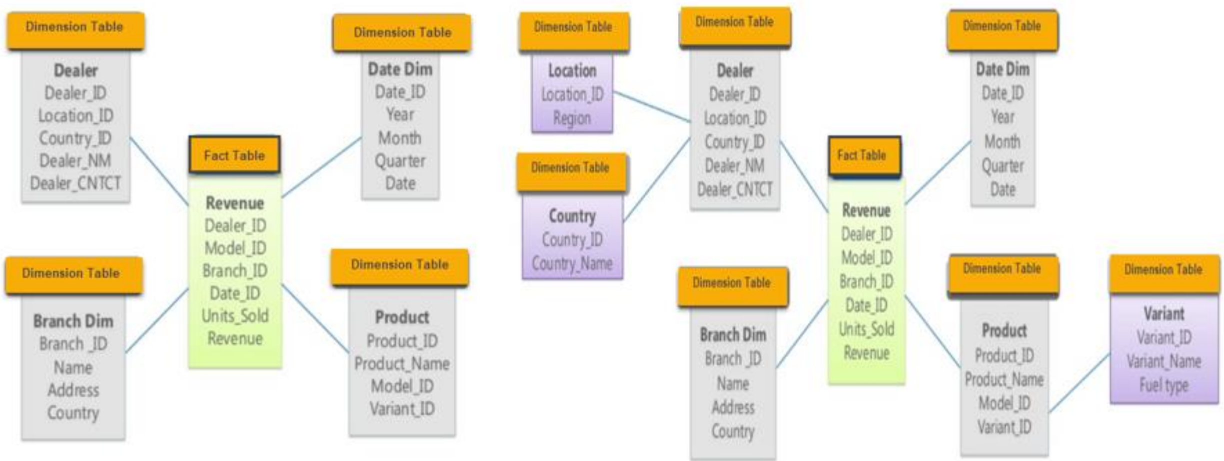
[Through careful tuning & efficient interface to model design, the representation deduces respective the capabilities to application]

ie. efficient utilities supported by ideal navigation mapping & information retrieval.

Note: Sometimes, an entity relationship data model which proposes thinking of database by emphasis of relationships btw collection of entities (tables) is first outlined.

Besides, Star & Snowflake schema are also available as fundamental modeling approach in data warehouses, advocating series of -

fact table: events [eg. retail store transaction] & dimension tables: descriptors [eg. customer & product]



ie. extensible dimensions to make row-store databases more friendly to processing tasks.

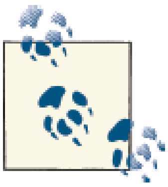
When creating a relation table, it is required to specify the homogenous datatype to a field (sometimes, with expected memory size – *eg*. int64) for efficient data storage & management.

Common SQL Datatypes: (refer. [Columns_and_Data_Types] for more information)

Type	Name	Description
String	`CHAR(n)` `/VARCHAR`	Hold fixed / variable no. of characters up to a maximum length (n) for all values in the field.
	`TEXT(byte)` `/BLOB`	Holds context / binary of a long string that don't fit in a VARCHAR. (*eg*. Byte image content or free text entered by survey respondents)
	`ENUM(`val1, val2, …`)` `/SET(`val1, val2, …`)`	- Hold one str from the specified list, or a blank is inserted. - Hold 0 or more than one str from the list.
Numeric	`INT/SMALLINT` `/BIGINT UNSIGNED`	Holds general integer of size up to 2^{32} or, typically smaller / larger no. of digits with corresponding memory size at signed / unsigned.
	`FLOAT/DOUBLE(n,d)` `/DECIMAL(n,d)`	Holds general floating-point or, double / higher unsigned values of total no. of digits (n) $^{\text{default: 10-65}}$ & decimal points (d) $^{0\text{-}30}$ specified.
Logical	`BOOLEAN`	Holds only TRUE {non-zero} / False {0} values.
Datetime	`DATETIME` `/TIMESTAMP`	Holds date & time in YYYY-MM-DD hh:mi:ss format (with or without support of time zones depending on database)
	`DATE/TIME` `/INTERVAL`	Holds date / time / a span of time values only : YYYY-MM-DD, hh:mi:ss, '1 hour 30 minutes'.

Note: Optional display width: INT(4) :: 0005 may be used for displaying integer within a fixed width if needed.

[ETL - Data engineering process]



a. `Extract`, batch/ stream retrieval of desired data via web scraping, databases, …, or gathered from different formats (XLSX, CSV, JSON, etc.).

b. `Transform`, serialize structure, performs data quality cleaning, or aggregate the data so that it conforms to how data is managed in the program or storage system.

c. `Load`, commit the [drafted / transformed] data to a data warehouse subjecting to if SQL process is deemed to be conducted in the database before it is loaded.

Usually, data engineers would ensure data integrity & privacy within data ecosystem following the process, and make it accessible to data users.

Creating & Populating a Database

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 (        #connection table
   user_id INTEGER,
   course_id INTEGER,
   role INTEGER,
   PRIMARY KEY (user_id, course_id) unique compound key table constraint
);
```

***MySQL** - PRIMARY KEY (c1) / CONSTRAINT alias PRIMARY KEY (c1,c2) :: 1 or 2-cols specification
Table Constraint - FOREIGN KEY (cust_id) REFERENCES Customer(id) / ... (prod_cate, prod_id) ... (cate,id)
(For column constraint, MYSQL takes on *DEFAULT* value, *AUTO_INCREMENT*, *NOT NULL*)

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), …
# establishing the link btw tables by integer id representation in SQLite
```

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
```

Note:

- Alter the example DCL script and import it to the database to ease creation of data tables esp. for large relational data. (refer. Database_CreateTables_script)
- Following up, external data could be imported (loaded) into the database's tables accordingly. (*however, make sure the column specified as primary key is unique in imported data*)

" 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. "

Single chr / integer ('1A' / 1 / 'A') is often used as primary key to distinguish row as lesser scanning & comparative chrs in sorting with fast overhead identification or, reference besides just memory issue.

SQL DATA PREPARATION

- 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.

When working with a new project, we need to fist familiarize with the available data:

1. How the data is arranged in schemas & tables, and
2. Check the fields & sort out how they are related in the organization's process.

—*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 brainstorm & visualize your idea on paper first.

(1) **Sketch out table(s):** Draw out the structure of each table that is involved.
(2) **Map Relationships:** Illustrate how the tables are connected when working with joins, to clarify how data flows and make complex joins easier to handle.
(3) **Plan the Logic:** Outline the main steps— aggregations, conditions, or filters you plan to apply. This allows catching mistakes early & save debugging time.

* This is crucial but often ignored, to determine whether you're on the right track & avoid wasting time circling around, sorting out a confusion.

For example, 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? "

For this query, some additional tasks need to be performed before returning the result:

- 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

```
# To examine how frequently a certain classes(1)/entities(2). appear in the data
SELECT fruit, COUNT(* / DISTINCT col_name) AS quantity  --(1)
FROM fruit_inventory
GROUP BY 1;        --output as histogram / bar chart using visualization tool
```

[# Checking the frequency of values in each field is a good way to start, learning about the feature distributions & its range or examine for typical / unusual values present, and detect sparse 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;
```

CREATE TABLE with roundtrip

A	B	(A-B)
B	A	(A-B)
A	A	(A-A)

LEAST(c1,c2)||'-'||GREATEST(c1,c2) and aggregate by c3.

[# A query w/ subquery which- first count the no. of orders by each customer_id, then work out how many customers with the similar order sizes.]

More complex distribution can be created at, (eg. to evaluate a few monthly sales stats for- " How many of each product is sold at each store? " AND relates to inventory with if needed.)

```
WITH sales_history AS (
  SELECT
    EXTRACT('YEAR' FROM Date) AS YEAR --time grouping
  , EXTRACT('MONTH' FROM Date) AS MONTH --time grouping
  , ProductId --need to know which products are sold
  , StoreID --need to know which stores are selling
  , SUM(quantity) AS UnitsSold --how many (impacts inventory)
  , AVG(UnitPrice) AS UnitPriceProxy --can be interesting
  , COUNT(DISTINCT salesID) AS NumTransactions --unique transactions
  FROM [project_name].sales.sales_info
  GROUP BY YEAR, MONTH, ProductId, StoreID
)
SELECT  --create inventory with monthly sales ref to verify stock level
 inventory.*
 ,( SELECT AVG(UnitsSold) FROM sales_history
    WHERE inventory.ProductID = sales_history.ProductID
    AND inventory.StoreID = sales_history.StoreID) AS avg_quantity_sold_in_a_month
FROM [project_name].sales.inventory AS inventory
```

For each ProductID + StoreID, you'll find:

- The current inventory of every product for each store
- The average monthly sales quantity for each product at every store

Sparse data is problematic for analysis due to its inconsistent & insignificant nature to reliably distinguish from chance variation

A Subquery is incorporated in the event of,

- to overcome the hindrance of aggregates in WHERE clause as row-wise interpretation.

```
:: 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
```

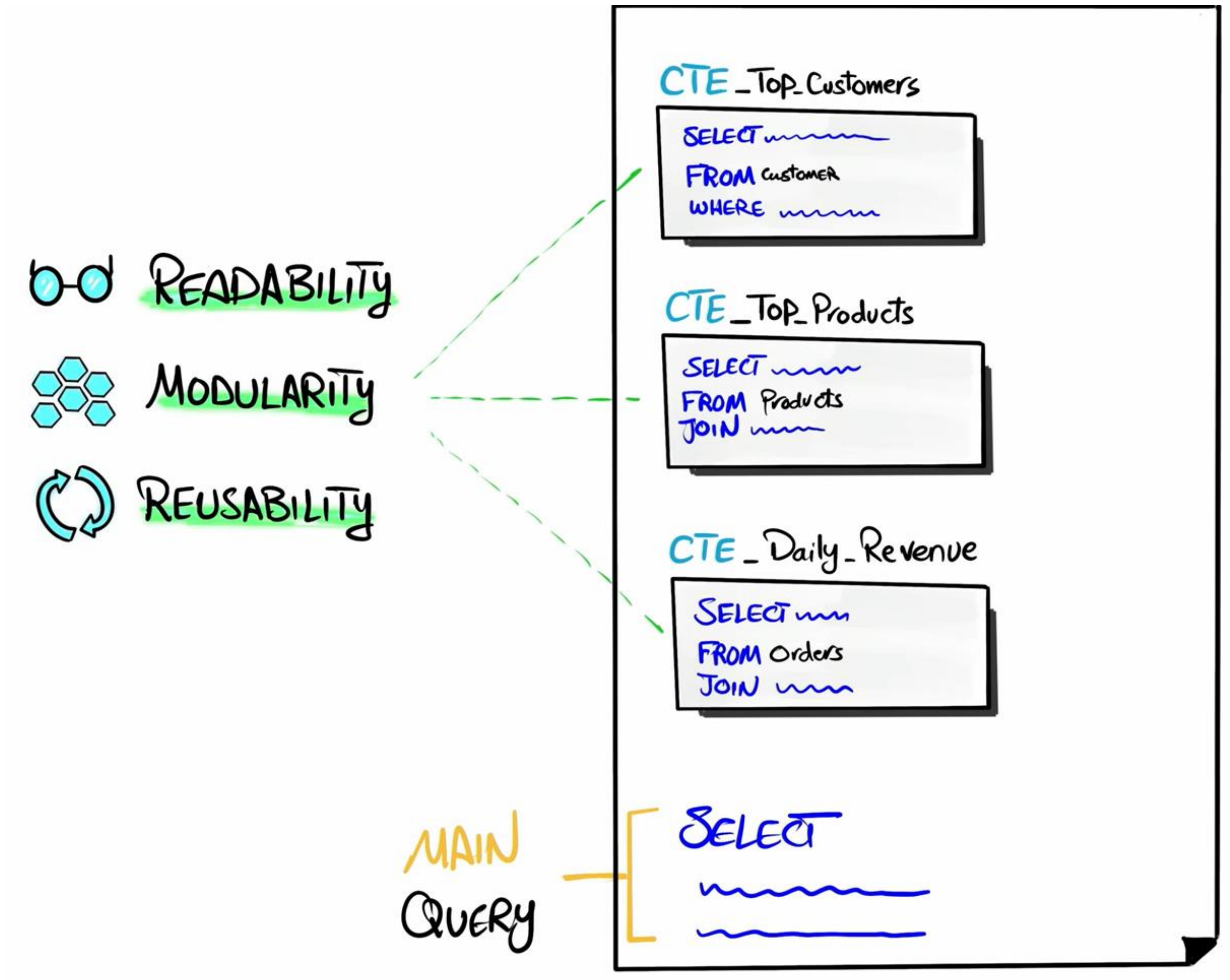
- when working w/ multiple tables, a virtual table may be created for main query to facilitate sub-selection as a separate table via FROM clause

```
:: SELECT COUNT(*) FROM          OR    WITH cte_name AS (
     (SELECT c1, c2, … FROM table)       SELECT … FROM t1), … cte_name2(…)
```

or, to based on matching entities from both tables in WHERE clause.

```
:: SELECT * FROM Employees
   WHERE dep_id IN
   (SELECT dept_id_dep FROM departments);
```

* Common table expression (CTE) is used when a common table is referenced by multiple subqueries.



2. For continuous field w/ distinct but closely related values, binning normalization can be more useful to review the overall distribution at a defined variable size.

```
# To verify how many customer will be affected in discounted shipping offers
# by grouping order_amount into 3 buckets.
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
    ;
```

[# Control the no. of bins, grouping of values, and how bins are named using CASE statement.]

```
# 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
,COUNT(customer_id) AS customers
FROM table
GROP 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

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

[# 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.

```
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. (filtering to only data needed can help optimize process)

[# Create n-tiles bins: n partitioned percentile of the data.]

II. Profiling: Data Quality

- One common consistency check is, comparing data against what is known to be true.

(eg. When working with a replica of production database, one could compare the row counts in each system to verify that the data is complete with all rows arrived & received)

3. To detect if duplicate rows in a table / a piece of data,

(Often figuring out why duplicates occur is useful for improving work process, or problem upstream)

—*eg. accident* with *a hidden many-to-many JOIN, multiple tracking call, or mistakes of manual steps* etc.

```
SELECT COUNT(*) AS records
FROM
(
    SELECT c1, c2, c3, ...
    FROM table
    ORDER BY 1,2,3,...
) a
WHERE records > 1
;
```

```
SELECT c1, c2, c3, ..., COUNT(*)
FROM table
GROUP BY 1,2,3,...
HAVING COUNT(*) > 1
;
```

[# Returns total no. of duplicates, or list out all the fields for more detail.]

```
# Subsequently, duplicates may be removed by joining tables which restricts to
# only common customers btw the 2 and also finding repeated order / trait.
SELECT DISTINCT a.customer_id, a.customer_name, a.customer_email
FROM customers a   --alias
JOIN transaction b ON a.customer_id = b.customer_id
```

III. Data Cleaning

4. For inconsistent / missing data, we can often detect by comparing data in 2-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
...
```

[# Check if non-coincidence employee w/ missing record exists in the Employee_Data table.]

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.

If the data is to be included in a calculation, then the missing values ought to be fixed.

```
# To replace NULL value in a row,
```

i. `CASE WHEN c1 IS NULL AND c2 = 'name' THEN expr1 ELSE c1 END AS col_name`

::imputing values based on statistical metric/likelihood/interpolation

ii. `LAG/LEAD(product_price) OVER (PARTITION BY product ORDER BY order_date)`

::fill-forward / backward w/ insightful assumption about typical value (eg. price of same date).

iii. - `SELECT c1, c2, COALESCE(min_num, c2 * 0.5) AS threshold`

- ```
  SELECT product || '-' || COALESCE(subcategory, category, family, 'no product description')
  AS product_and_subcategory
  FROM stock
  ```

::replace NULLs with the first non-NULL value in the list.

```
product_and_subcategory
pork ribs – pork meat
tomatoes – vegetables
lettuce – leaf vegetables
hamburger – cow meat
hamburger – no product description
```

Note: [1] Similar to Python, null value is contagious and any operation with it will result in NULL.
[2] 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.
(empty string - ' ' is appropriate as value overcoming NOT NULL constraint in an explicit non-existing / no suitable answer than missing)

The lag / lead(col, offset -> int, default->1), can specify a default if there's no lead/lagging record- eg. lag(c1, 1, NULL)
```

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

5. To standardize text / categorical values of a field or encoding categorical data,

```
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
```

Alternatively, 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, an utility table will be a better option.

Another use is to create flags (add 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
;
```

6. For flattening the data in a feature containing multiple values per cell,

```
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 table
GROUP BY 1
;
```

[# The flag will be 1 if any matching value is in the compounded data; else 0.]

7. To fix or override the current datatype of a field,

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

```
# To manipulate string feature or value for numerical application:
SELECT REPLACE('$19.99', '$', '')::float or, CAST(REPLACE('$19.99', '$', '') AS float)
```

```
# To assemble text from different columns: (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), and requires data to be explicitly defined.]

`to_char ( timestamp, text ) → text`
`to_char ( timestamp with time zone, text ) → text`
Converts time stamp to string according to the given format.
`to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS') → 05:31:12`

`to_char ( interval, text ) → text`
Converts interval to string according to the given format.
`to_char(interval '15h 2m 12s', 'HH24:MI:SS') → 15:02:12`

`to_char ( numeric_type, text ) → text`
Converts number to string according to the given format; available for `integer`, `bigint`, `numeric`, `real`, `double precision`.
`to_char(125, '999') → 125`
`to_char(125.8::real, '999D9') → 125.8`
`to_char(-125.8, '999D99S') → 125.80-`

`to_date ( text, text ) → date`
Converts string to date according to the given format.
`to_date('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05`

`to_number ( text, text ) → numeric`
Converts string to numeric according to the given format.
`to_number('12,454.8-', '99G999D9S') → -12454.8`

`to_timestamp ( text, text ) → timestamp with time zone`
Converts string to time stamp according to the given format. (See also `to_timestamp(double precision)` in Table 9.33.)
`to_timestamp('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05 00:00:00-05`

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

8. To convert data into the different levels of granularity needed,

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 pre-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. Shaping Dataset

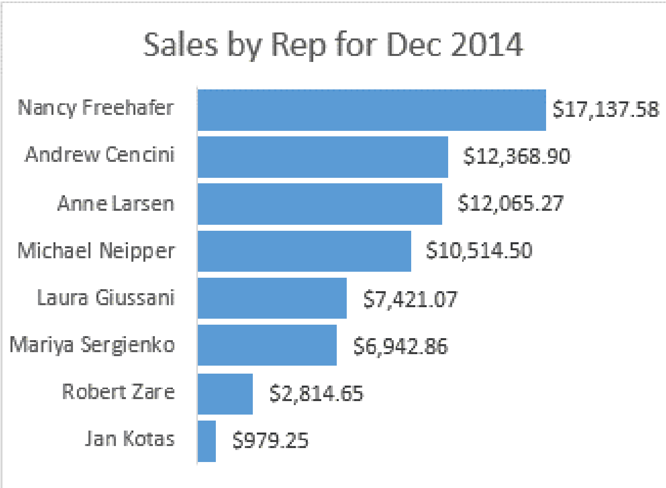
Often the decision on the shape depends on how data will be used downstream

- Smaller, aggregated, and highly specific datasets are preferred for visualizations considering the repetitive & level of aggregation and slices, or timeline the end user will need to filter on.
- Generally, the notion of tidy data will follow as:

  ☑ Each variable forms a column ☑ Each record forms a row ☑ Each entry in a cell

(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
Grand Total	**$70,244.08**



To create a pivot table which aggregates & shapes data into more compact and easily understandable form, each distinctive group of a feature is presented as a new column:

```
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 w/ 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 should be ignored in count / count DISTINCT where a substitute value could inflate unneccesary count.

(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
;
```

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

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

```
country        year  population
-------------  ----  ----------
Canada         1980  24593
Mexico         1980  68347
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 dedication functions that operate in similar manner: (Microsoft SQL Server & Snowflake)

```
SELECT ...
FROM ...
    pivot(aggregation(value_c1)
        for label_c2 in ('cate1', 'cate2', ...))
GROUP BY ...
;
```

```
SELECT *
FROM country_populations
    unpivot(population)
        for year in (year_1980, year_1990, ...)
;
```

(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.)

DATETIME MANIPULATION

- Time-series is a serial of records or measurements documented in datetime order, often at regular intervals to outline changes over time.

I. Time Zones.

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

a. Converting time zone for a timestamp data using 'at time zone statement':

```
SELECT '2020-09-01 00:00:00 -0' at time zone 'pst';    # UTC(offset - 0) to PST
```

b. Some databases offer dedicated functions - CONVERT_TIMEZONE( ) / CONVERT_TZ( )

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

* Using standard UTC time as reference may be convenient, but it does not reflect the actual moment of the day an activity is being carried out.
(Check database's documentation for the exact specification or lookout for timezone offset)

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

II. Date / Timestamp Format.

a. Converting string / Unix epoach to appropriate datetime format:

TO_DATE(source, 'yyyy-mm-dd') or, TO_TIMESTAMP(source, 'yyyy-mm-dd HH:MI:SS')

b. Abstracting the datetime data to desired period:

DATE_TRUNC('period', time) ; DATE_FORMAT(datetime_source, '%Y-%m-01') MySQL
::2020-10-01 00:00:00

c. Extracting individual date / time part:

DATE_PART('component', timestamp) or, EXTRACT('component' FROM timestamp)
::return float dtype
TO_CHAR(timestamp, 'Day'/'Month'/…)
::return text dtype

d. Composing date & time from separate columns into a timestamp:

```
SELECT date_source + time_source AS timestamp
```

The standard time period incl.- microsecond, millisecond, second, minute, hour, day, week, month, quarter, year, decade, century, millennium.

DATETIME math involves: the dates themselves / `INTERVAL` *ie.* Python timedelta - which defines the numeral system of date & time unit. [# year(s), month(s), day(s), hours, minute(s) & second(s)]

```
SELECT event_date FROM t1 WHERE event_date < current_date - INTERVAL '3 months'
SELECT time '05:00' + INTERVAL '3 hours'
```

(Refer. PostgreSQL doc for complete functions & operators)

* Calculation may be included in JOIN during query but it will be slower than equality btw dates.

Some databases don't require INTERVAL {eg. SELECT date '2020-06-01' + 7}, but it's a good practice for cross-database compatibility & code readability.

There are few areas on datetime records needed particular attention when aligning data from different sources with uniform 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 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.

III. Extracting trends of-

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

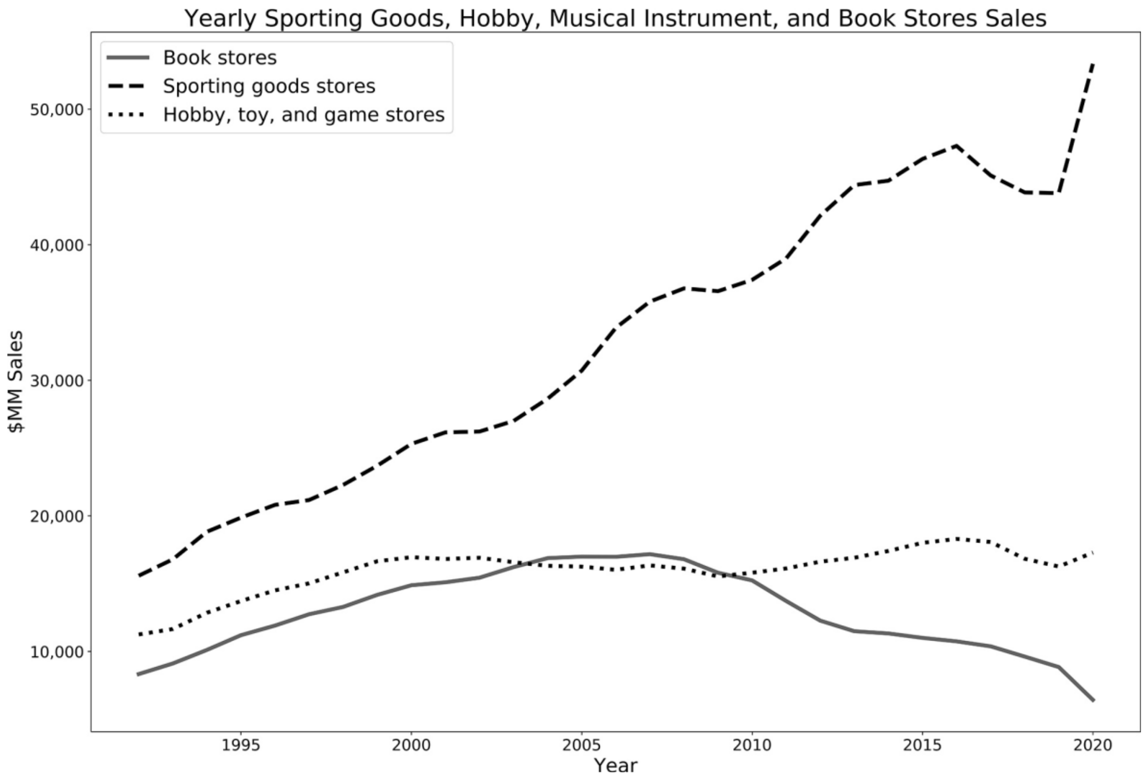
```
SELECT DATE_PART('year', sales_month) AS sales_year ---- Simple trends
,kind_of_business, SUM(sales) AS sales
FROM RetailSales
WHERE kind_of_business
IN ('Book stores','Sporting goods stores', 'Hobby, toy, and game stores')
GROUP BY 1,2
;
```

:: -- X: dates / timestamps (here, yearly) -- Y: numerical values

sales_year	kind_of_business	sales
1992.0	Book stores	8327
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 or program to create visualizations.]

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



Note: Creating trend can be a step in profiling & understanding, or it may be the final result depending on the goals of the analysis.

Often datasets contain not just a single time series and graphing the data at different levels of aggregation is a good way to understand the trends.

Comparing Components

(*eg.* to quantify the gap btw 2 categories by: [1] difference / [2] ratio)

```sql
SELECT DATE_PART('year', sales_month) AS sales_year
 ,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
1992.0        21636
1993.0        22388
1994.0        20553
...           ...
```