07

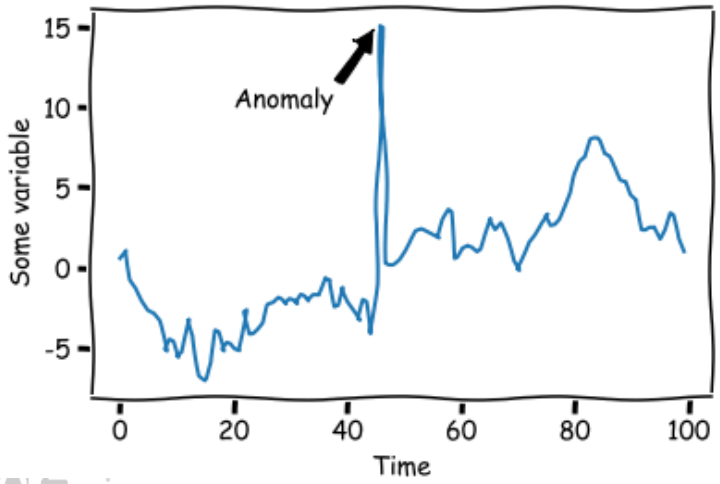
Anomaly Detection with SQL

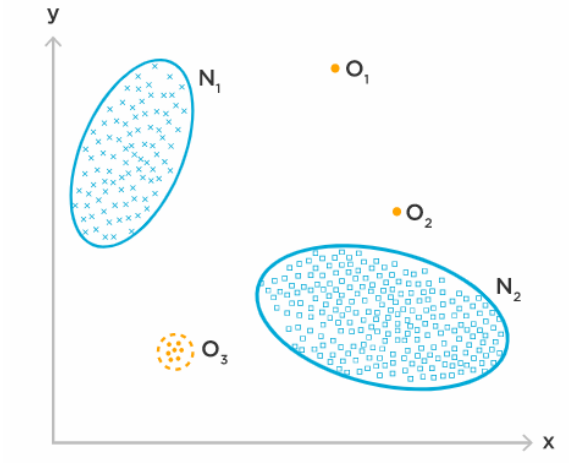
How to detect outliers, noise, deviations
and exception in your data

What is an anomaly?

Definition:

An anomaly is something that is different from other members of the same group. In data, an anomaly is a **record**, an **observation**, or a **value** that differs from the remaining data points in a way that raises concerns or suspicions. Anomalies go by a number of different names, including outliers, novelties, noise, deviations, and exceptions, to name a few.

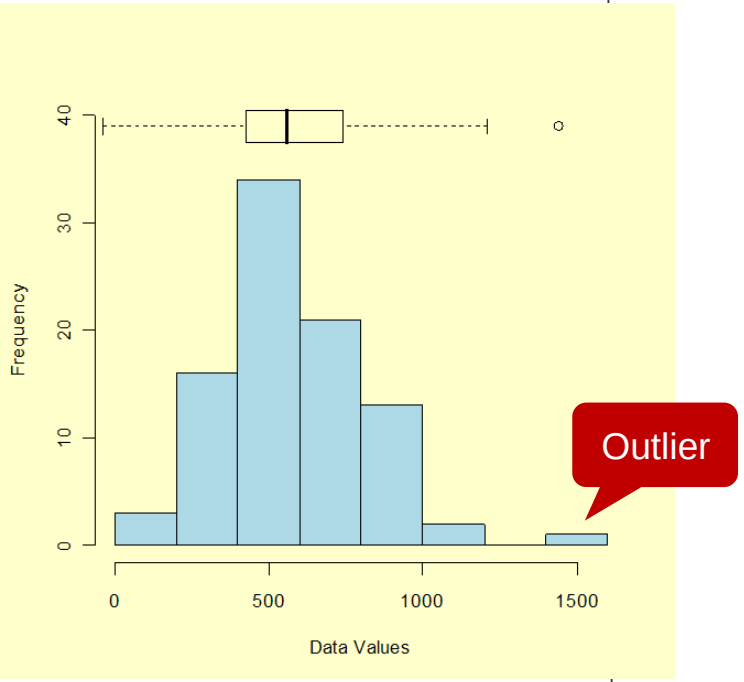




Anomalies are good or bad?

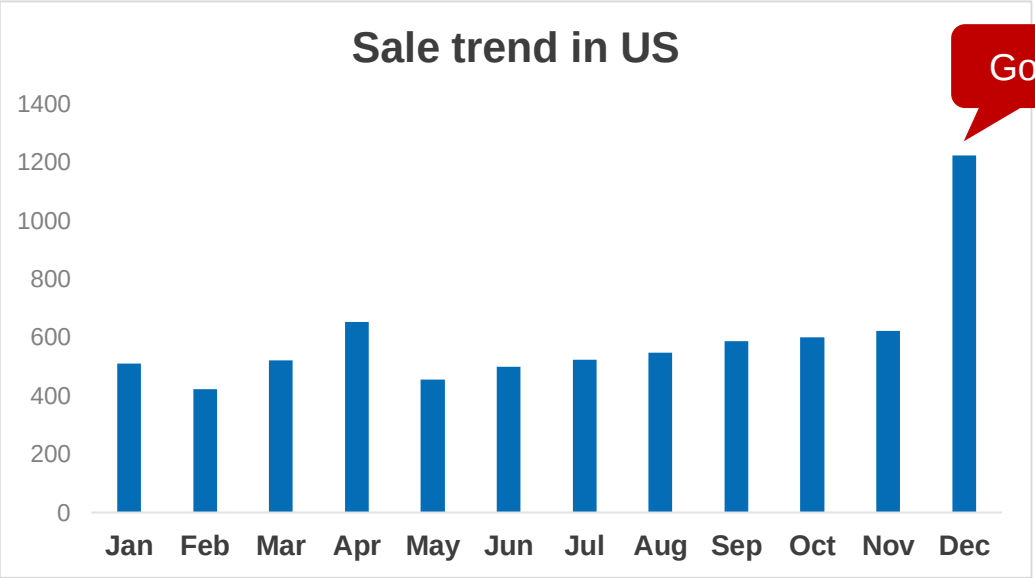
Without Outlier	With Outlier
4, 4, 5, 5, 5, 5, 6, 6, 6, 7, 7	4, 4, 5, 5, 5, 5, 6, 6, 6, 7, 7,300
Mean = 5.45	Mean = 30.00
Median = 5.00	Median = 5.50
Mode = 5.00	Mode = 5.00
Standard Deviation = 1.04	Standard Deviation = 85.03

Bad



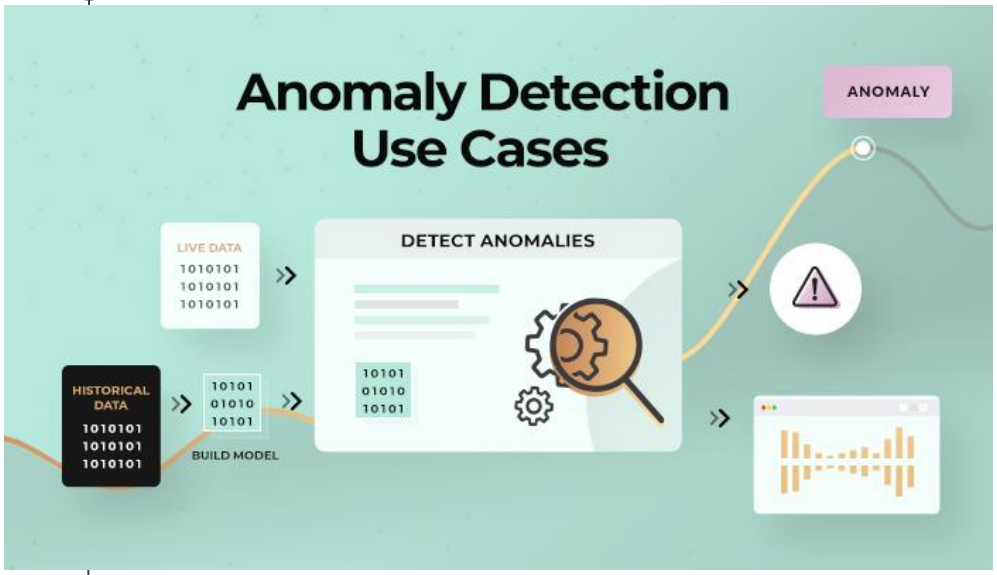
- The outliers may negatively bias the entire result of an analysis
- The behavior of outliers may be precisely what is being sought

Anomalies are good or bad?



Sales increased because of the high demand for shopping during the festival

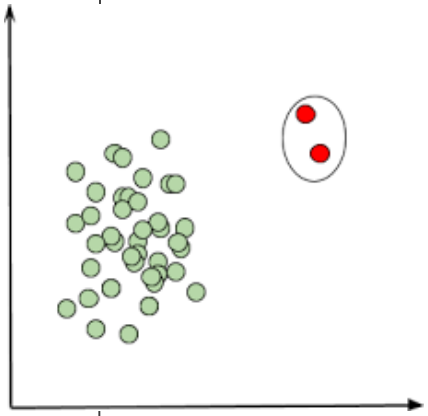
What are the benefits of anomaly detection?

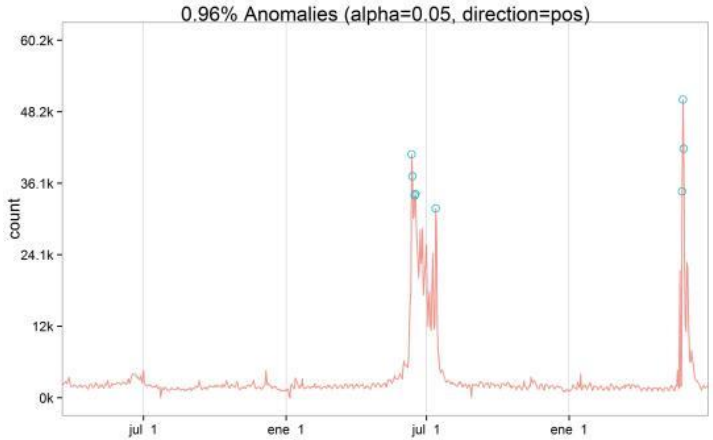


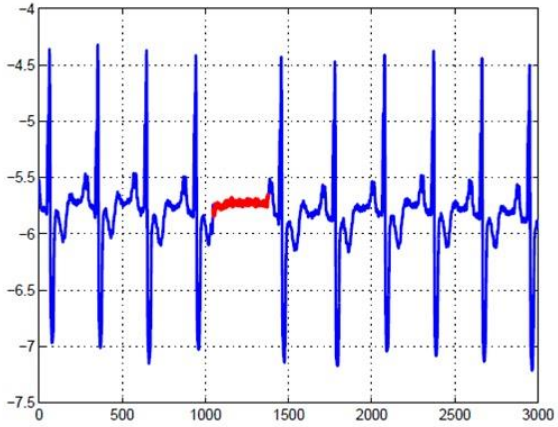
1. Monitor KPI metrics/
2. Monitor system performance
3. Detect fraud

What are the three types of anomalies?

When looking at a time series of data (data that is collected sequentially, over a period of time), there are three main types of anomalies: **global (or point) anomalies, contextual anomalies and collective anomalies**.







How to detect anomaly/outlier in your dataset?

1. Sorting your table to find outliers

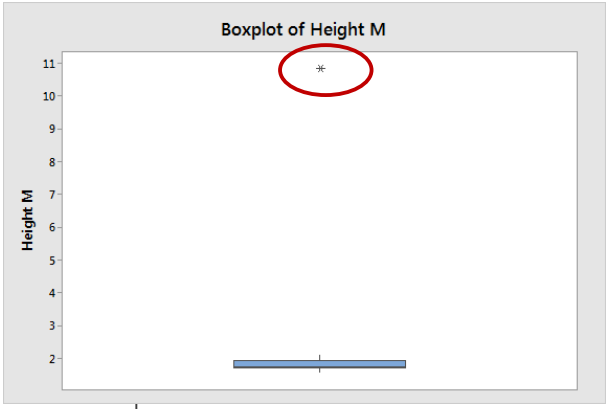
Sorting your datasheet is a simple but effective way to highlight unusual values. Simply sort your data sheet for each variable and then look for unusually high or low values

Height M
1.5895
1.6508
1.7131
1.7136
1.7212
1.7296
1.7343
1.7663
1.8018
1.8394
1.8869
1.9357
1.9482
2.1038
10.8135

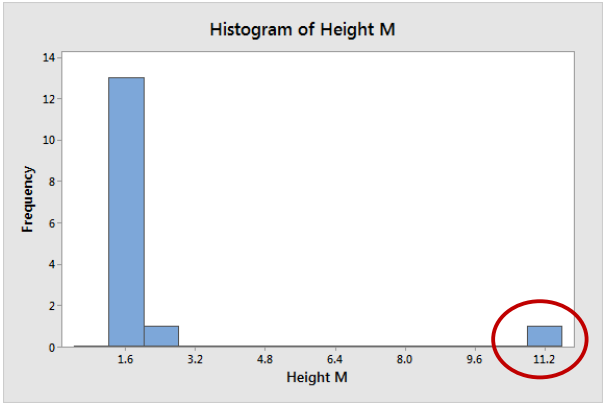
How to detect anomaly/outlier in your dataset?

2. Graphing Your Data to Identify Outliers

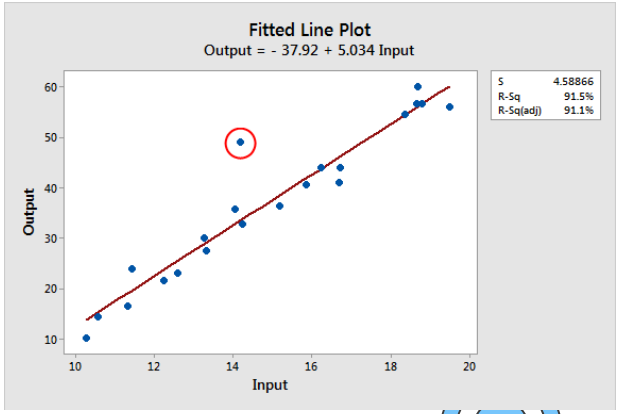
Boxplots



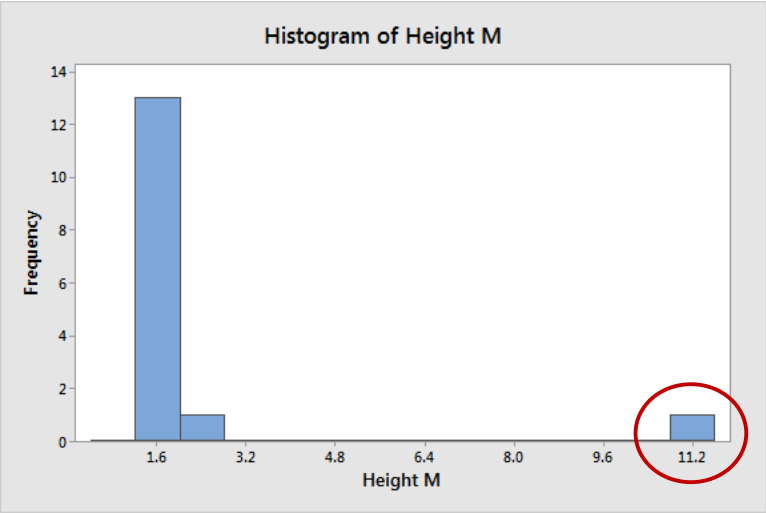
Histograms



Scatterplot



Handling outliers



Remove if the outliers may negatively bias the entire result of an analysis

Handling outliers

Without Outlier	With Outlier
4, 4, 5, 5, 5, 5, 6, 6, 6, 7, 7	4, 4, 5, 5, 5, 5, 6, 6, 6, 7, 7,300
Mean = 5.45	Mean = 30.00
Median = 5.00	Median = 5.50
Mode = 5.00	Mode = 5.00
Standard Deviation = 1.04	Standard Deviation = 85.03

Replace with median value

08

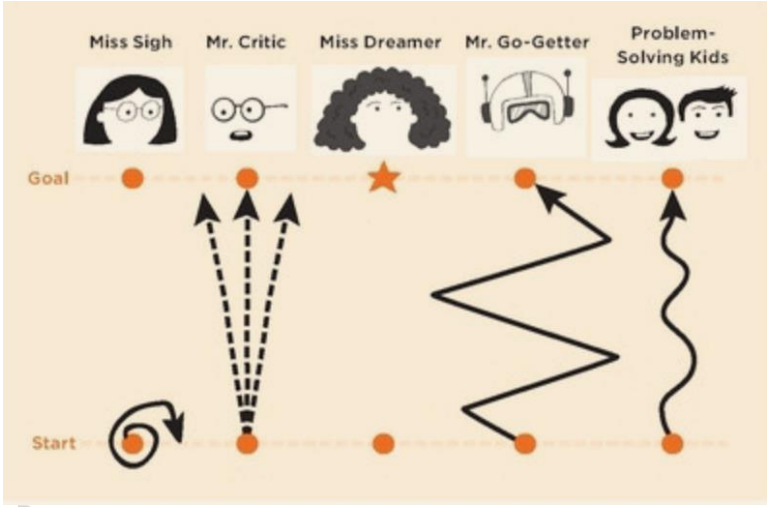
Apply Problem Solving in Data Analysis

Strategies & Methods

Problem Solving

Definition:

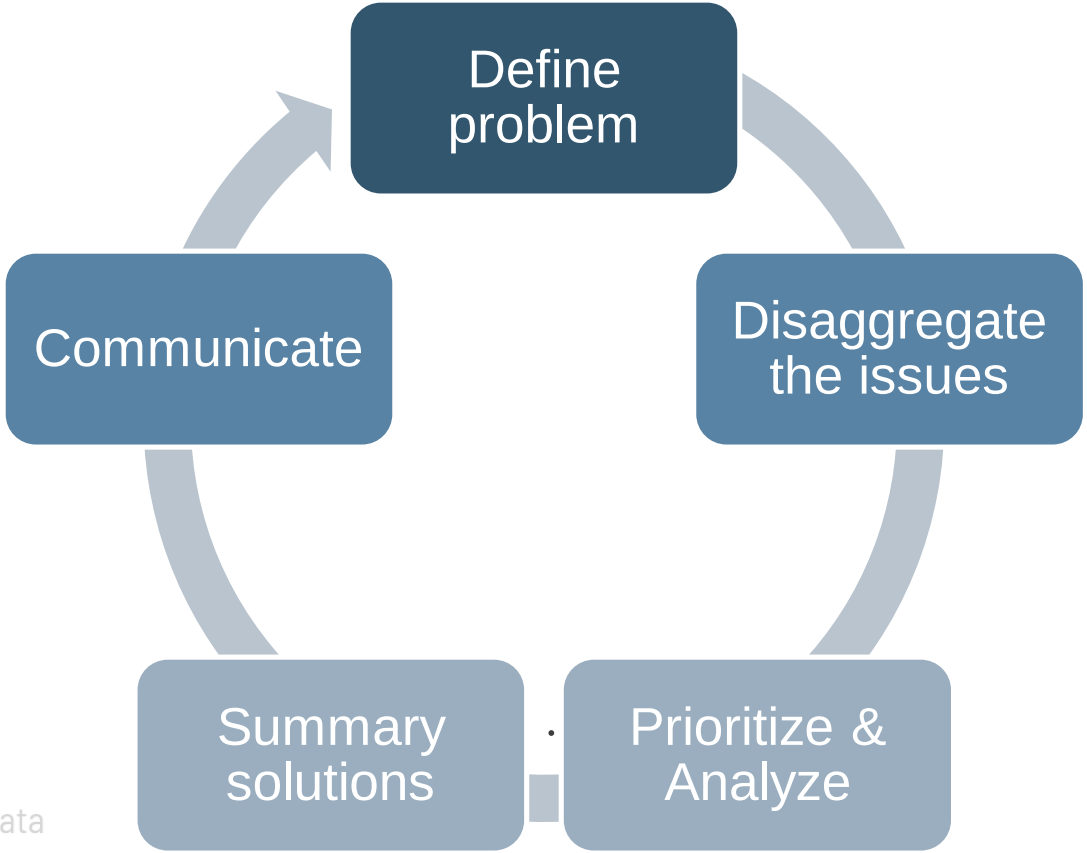
Problem solving is decision making when there is complexity and uncertainty that rules out obvious answers, and where there are consequences that make the work to get good answers worth it. Problem solving means the process of making better decisions on the complicated challenges of personal life, our workplaces, and the policy sphere.



TOP 10 SKILLS IN 2020

1. Complex Problem Solving
2. Critical Thinking
3. Creativity
4. People Management
5. Coordinating with Others
6. Emotional Intelligence
7. Judgment and Decision Making
8. Service Orientation
9. Negotiation
10. Cognitive Flexibility

Problem solving strategy in Data Analysis



Define problem

6 common problem in data analysis

1. Making predictions

2. Categorizing things

3. Spotting something unusual

4. Identifying themes

5. Discovering connections

6. Finding patterns

Define problem

S M A R T

				
S-pecific	**M-easurable**	**A-ction-oriented**	**R-elevant**	**T-ime-bound**
What is the problem? Can it be solved with data? If so, what data?	Where is this data? Which are the metrics for evaluation?	Which are the solutions? Which is solution prioritized first?	Are your solutions realistics?	What is the time range for analysis?

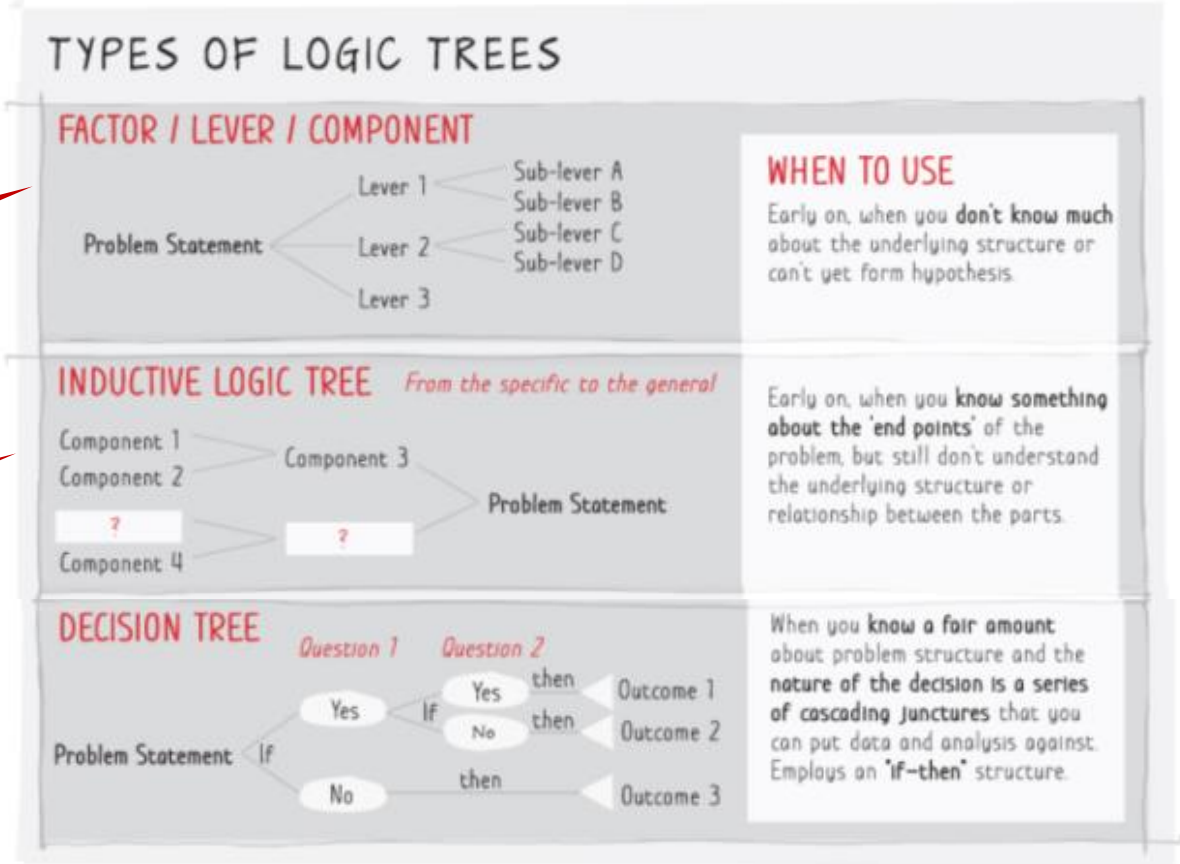
Disaggregate the issues

Methods:

1. Logic Tree

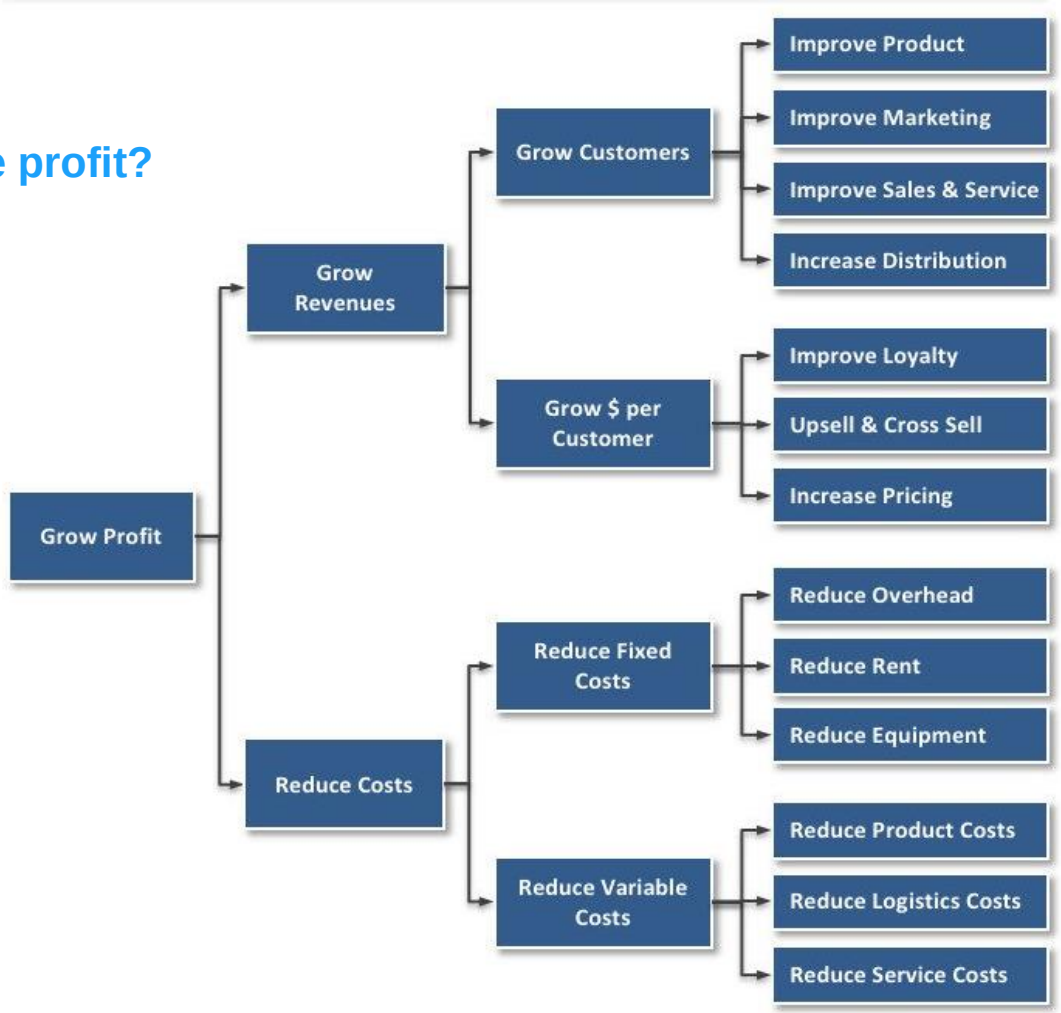
Top → down

Bottom → up

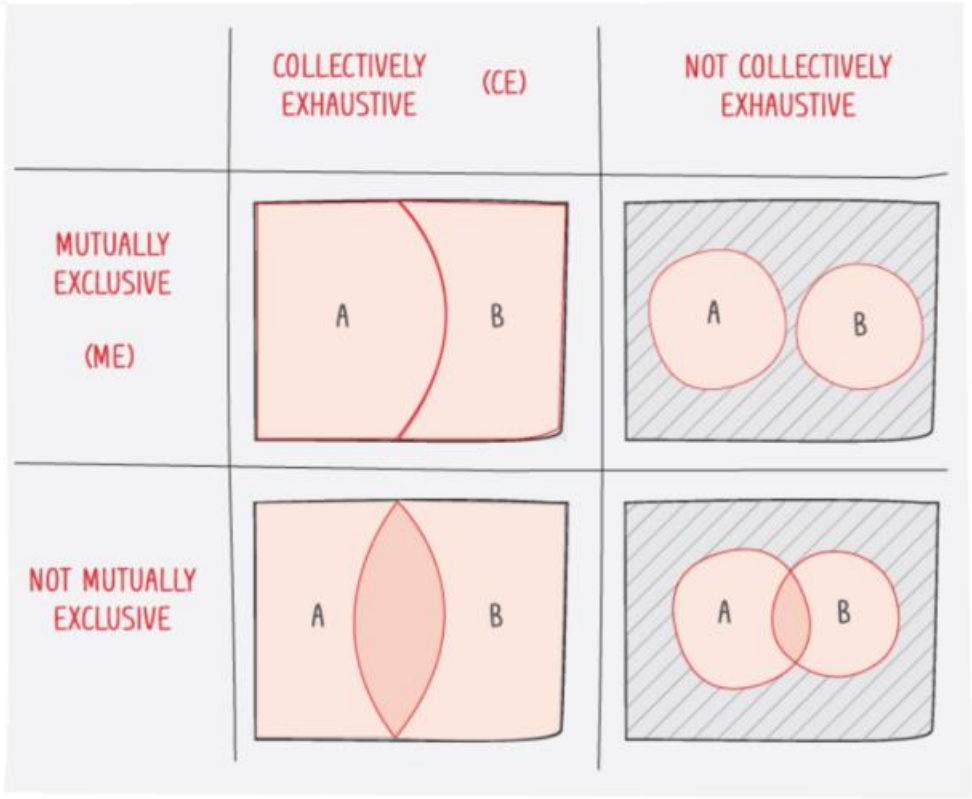


Logic Tree

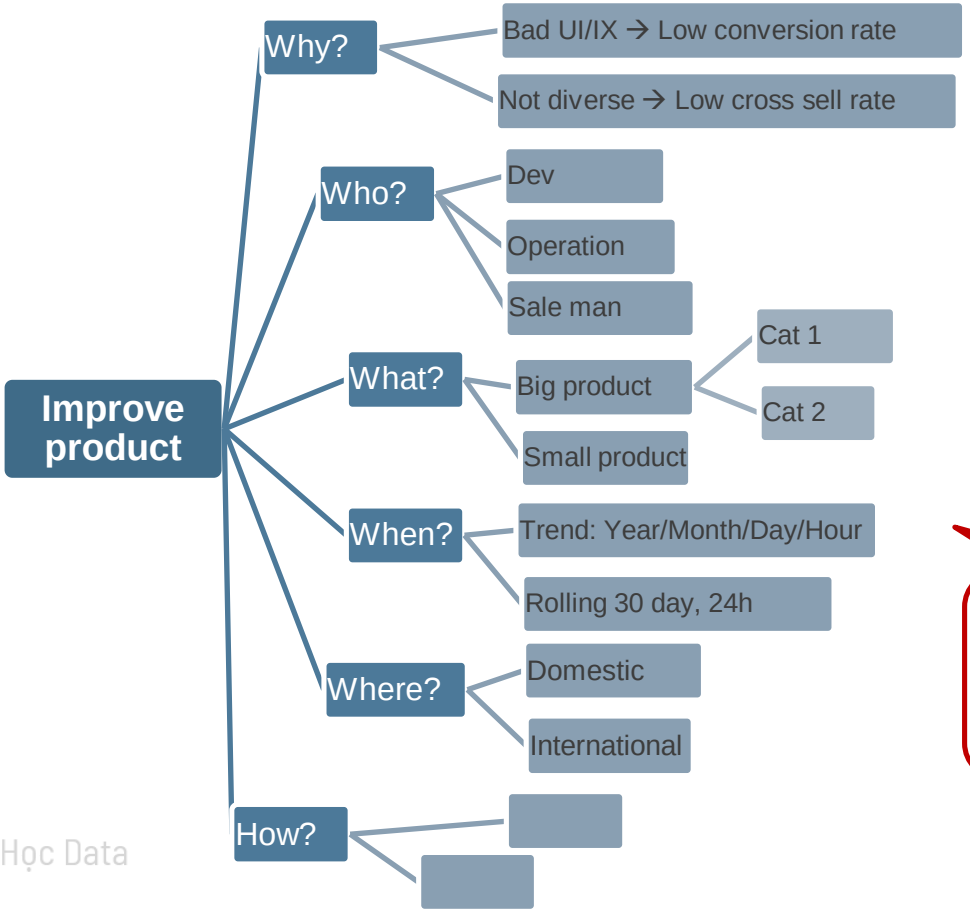
How to increase profit?



MECE (Mutually exclusive & Collectively exhaustive)



Define problem with 5W – 1H



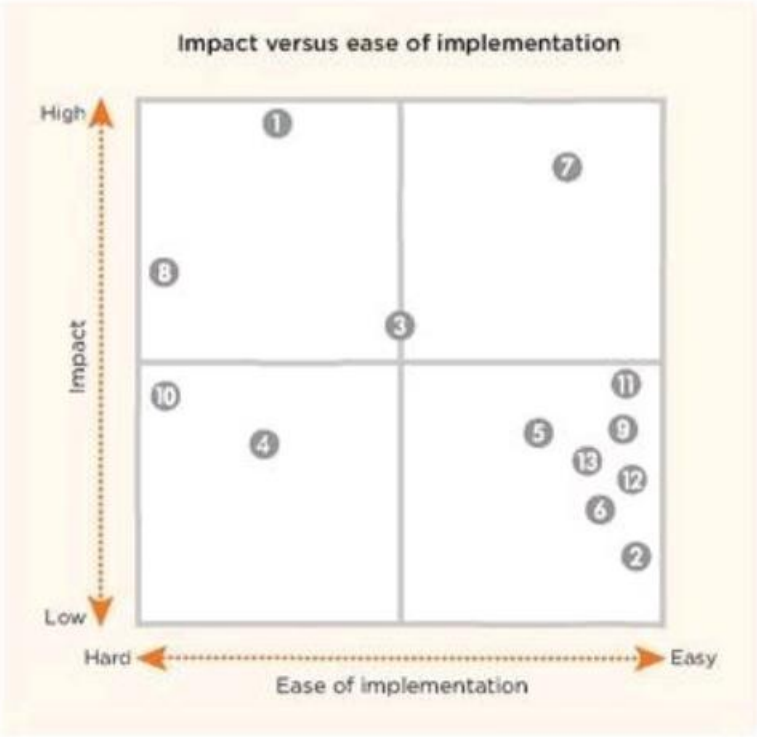
Based on your domain knowledge to define some **hypothesis** and **prioritize** them

Define all data you need to collect

Define solution

What should you do:

- List down all insights → Define solution for each of them
- List down all solution on the Impact & Implementation metrix



09

Review & Next action plan

Summary – Learning resources

What did we learn?

SQL Basics Cheat Sheet

LearnSQL.com

SQL

SQL, or *Structured Query Language*, is a language to talk to databases. It allows you to select specific data and to build complex reports. Today, SQL is a universal language of data. It is used in practically all technologies that process data.

SAMPLE DATA

COUNTRY

id	name	population	area
1	France	66600000	640680
2	Germany	80700000	357000
...	...	...	...

CITY

id	name	country_id	population	rating
1	Paris	1	2243000	5
2	Berlin	2	3460000	3
...	...	...	...	...

QUERYING SINGLE TABLE

Fetch all columns from the `country` table:

```
SELECT *
FROM country;
```

Fetch `id` and `name` columns from the `city` table:

```
SELECT id, name
FROM city;
```

Fetch city names sorted by the `rating` column in the default ASCending order:

```
SELECT name
FROM city
ORDER BY rating [ASC];
```

Fetch city names sorted by the `rating` column in the DESCending order:

```
SELECT name
FROM city
ORDER BY rating DESC;
```

ALIASES

COLUMNS

```
SELECT name AS city_name
FROM city;
```

TABLES

```
SELECT co.name, ci.name
FROM city AS ci
JOIN country AS co
  ON ci.country_id = co.id;
```

FILTERING THE OUTPUT

COMPARISON OPERATORS

Fetch names of cities that have a rating above 3:

```
SELECT name
FROM city
WHERE rating > 3;
```

Fetch names of cities that are neither Berlin nor Madrid:

```
SELECT name
FROM city
WHERE name != 'Berlin'
  AND name != 'Madrid';
```

TEXT OPERATORS

Fetch names of cities that start with a 'P' or end with an 's':

```
SELECT name
FROM city
WHERE name LIKE 'P%'
   OR name LIKE '%s';
```

Fetch names of cities that start with any letter followed by 'ublin' (like Dublin in Ireland or Lublin in Poland):

```
SELECT name
FROM city
WHERE name LIKE '_ublin';
```

OTHER OPERATORS

Fetch names of cities that have a population between 500K and 5M:

```
SELECT name
FROM city
WHERE population BETWEEN 500000 AND 5000000;
```

Fetch names of cities that don't miss a rating value:

```
SELECT name
FROM city
WHERE rating IS NOT NULL;
```

Fetch names of cities that are in countries with IDs 1, 4, 7, or 8:

```
SELECT name
FROM city
WHERE country_id IN (1, 4, 7, 8);
```

QUERYING MULTIPLE TABLES

INNER JOIN

`JOIN` (or explicitly `INNER JOIN`) returns rows that have matching values in both tables.

```
SELECT city.name, country.name
FROM city
[INNER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	3	Iceland

FULL JOIN

`FULL JOIN` (or explicitly `FULL OUTER JOIN`) returns all rows from both tables – if there's no matching row in the second table, `NULL`s are returned.

```
SELECT city.name, country.name
FROM city
FULL [OUTER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL
NULL	NULL	NULL	3	Iceland

LEFT JOIN

`LEFT JOIN` returns all rows from the left table with corresponding rows from the right table. If there's no matching row, `NULL`s are returned as values from the second table.

```
SELECT city.name, country.name
FROM city
LEFT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL

CROSS JOIN

`CROSS JOIN` returns all possible combinations of rows from both tables. There are two syntaxes available.

```
SELECT city.name, country.name
FROM city
CROSS JOIN country;

SELECT city.name, country.name
FROM city, country;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
1	Paris	1	2	Germany
2	Berlin	2	1	France
2	Berlin	2	2	Germany

RIGHT JOIN

`RIGHT JOIN` returns all rows from the right table with corresponding rows from the left table. If there's no matching row, `NULL`s are returned as values from the left table.

```
SELECT city.name, country.name
FROM city
RIGHT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
NULL	NULL	NULL	3	Iceland

NATURAL JOIN

`NATURAL JOIN` will join tables by all columns with the same name.

```
SELECT city.name, country.name
FROM city
NATURAL JOIN country;
```

CITY			COUNTRY	
country_id	id	name	name	id
6	6	San Marino	San Marino	6
7	7	Vatican City	Vatican City	7
5	9	Greece	Greece	9
10	11	Monaco	Monaco	10

`NATURAL JOIN` used these columns to match rows: `city.id, city.name, country.id, country.name`

`NATURAL JOIN` is very rarely used in practice.

MAZ

What did we learn?

SQL Basics Cheat Sheet

LearnSQL.com

AGGREGATION AND GROUPING

GROUP BY **groups** together rows that have the same values in specified columns. It computes summaries (aggregates) for each unique combination of values.

CITY

id	name	country_id
1	Paris	1
101	Marseille	1
102	Lyon	1
2	Berlin	2
103	Hamburg	2
104	Munich	2
3	Warsaw	4
105	Cracow	4

→

CITY

country_id	count
1	3
2	3
4	2

AGGREGATE FUNCTIONS

- **avg**(expr) – average value for rows within the group
- **count**(expr) – count of values for rows within the group
- **max**(expr) – maximum value within the group
- **min**(expr) – minimum value within the group
- **sum**(expr) – sum of values within the group

EXAMPLE QUERIES

Find out the number of cities:

```
SELECT COUNT(*)
FROM city;
```

Find out the number of cities with non-null ratings:

```
SELECT COUNT(rating)
FROM city;
```

Find out the number of distinctive country values:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

Find out the smallest and the greatest country populations:

```
SELECT MIN(population), MAX(population)
FROM country;
```

Find out the total population of cities in respective countries:

```
SELECT country_id, SUM(population)
FROM city
GROUP BY country_id;
```

Find out the average rating for cities in respective countries if the average is above 3.0:

```
SELECT country_id, AVG(rating)
FROM city
GROUP BY country_id
HAVING AVG(rating) > 3.0;
```

SUBQUERIES

A subquery is a query that is nested inside another query, or inside another subquery. There are different types of subqueries.

SINGLE VALUE

The simplest subquery returns exactly one column and exactly one row. It can be used with comparison operators =, <, <=, >, or >=.

This query finds cities with the same rating as Paris:

```
SELECT name FROM city
WHERE rating = (
    SELECT rating
    FROM city
    WHERE name = 'Paris'
);
```

MULTIPLE VALUES

A subquery can also return multiple columns or multiple rows. Such subqueries can be used with operators IN, EXISTS, ALL, or ANY.

This query finds cities in countries that have a population above 20M:

```
SELECT name
FROM city
WHERE country_id IN (
    SELECT country_id
    FROM country
    WHERE population > 20000000
);
```

CORRELATED

A correlated subquery refers to the tables introduced in the outer query. A correlated subquery depends on the outer query. It cannot be run independently from the outer query.

This query finds cities with a population greater than the average population in the country:

```
SELECT *
FROM city main_city
WHERE population > (
    SELECT AVG(population)
    FROM city average_city
    WHERE average_city.country_id = main_city.country_id
);
```

This query finds countries that have at least one city:

```
SELECT name
FROM country
WHERE EXISTS (
    SELECT *
    FROM city
    WHERE country_id = country.id
);
```

SET OPERATIONS

Set operations are used to combine the results of two or more queries into a single result. The combined queries must return the same number of columns and compatible data types. The names of the corresponding columns can be different.

CYCLING

id	name	country
1	YK	DE
2	ZG	DE
3	WT	PL
...	...	...

SKATING

id	name	country
1	YK	DE
2	DF	DE
3	AK	PL
...	...	...

UNION

UNION combines the results of two result sets and removes duplicates. UNION ALL doesn't remove duplicate rows.

This query displays German cyclists together with German skaters:

```
SELECT name
FROM cycling
WHERE country = 'DE'
UNION / UNION ALL
SELECT name
FROM skating
WHERE country = 'DE';
```

INTERSECT

INTERSECT returns only rows that appear in both result sets.

This query displays German cyclists who are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
INTERSECT
SELECT name
FROM skating
WHERE country = 'DE';
```

EXCEPT

EXCEPT returns only the rows that appear in the first result set but do not appear in the second result set.

This query displays German cyclists unless they are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
EXCEPT / MINUS
SELECT name
FROM skating
WHERE country = 'DE';
```

What did we learn?

SQL Window Functions Cheat Sheet

LearnSQL.com

WINDOW FUNCTIONS

compute their result based on a sliding window frame, a set of rows that are somehow related to the current row.

AGGREGATE FUNCTIONS VS. WINDOW FUNCTIONS

unlike aggregate functions, window functions do not collapse rows.

SYNTAX

```
SELECT city, month,
   sum(sold) OVER (
      PARTITION BY city
      ORDER BY month
      RANGE UNBOUNDED PRECEDING) total
FROM sales;
```

```
SELECT <column_1>, <column_2>,
   <window_function>() OVER (
      PARTITION BY <...>
      ORDER BY <...>
      <window_frame>) <window_column_alias>
FROM <table_name>;
```

Named Window Definition

```
SELECT country, city,
    rank() OVER country_sold_avg
FROM sales
WHERE month BETWEEN 1 AND 6
GROUP BY country, city
HAVING sum(sold) > 10000
WINDOW country_sold_avg AS (
    PARTITION BY country
    ORDER BY avg(sold) DESC)
ORDER BY country, city;
```

```
SELECT <column_1>, <column_2>,
   <window_function>() OVER <window_name>
FROM <table_name>
WHERE <...>
GROUP BY <...>
HAVING <...>
WINDOW <window_name> AS (
   PARTITION BY <...>
   ORDER BY <...>
   <window_frame>)
ORDER BY <...>;
```

PARTITION BY, ORDER BY, and window frame definition are all optional.

LOGICAL ORDER OF OPERATIONS IN SQL

1. FROM, JOIN
2. WHERE
3. GROUP BY
4. aggregate functions
5. HAVING
6. **window functions**
7. SELECT
8. DISTINCT
9. UNION/INTERSECT/EXCEPT
10. ORDER BY
11. OFFSET
12. LIMIT/FETCH/TOP

You can use window functions in SELECT and ORDER BY. However, you can't put window functions anywhere in the FROM, WHERE, GROUP BY, or HAVING clauses.

PARTITION BY

divides rows into multiple groups, called **partitions**, to which the window function is applied.

month	city	sold
1	Rome	200
2	Paris	500
1	London	100
1	Paris	300
2	Rome	300
2	London	400
3	Rome	400

PARTITION BY city

month	city	sold	sum
1	Paris	300	800
2	Paris	500	800
1	Rome	200	900
2	Rome	300	900
3	Rome	400	900
1	London	100	500
2	London	400	500

Default Partition: with no PARTITION BY clause, the entire result set is the partition.

ORDER BY

specifies the order of rows in each partition to which the window function is applied.

sold	city	month
200	Rome	1
500	Paris	2
100	London	1
300	Paris	1
300	Rome	2
400	London	2
400	Rome	3

PARTITION BY city ORDER BY month

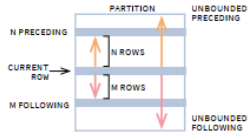
sold	city	month
300	Paris	1
500	Paris	2
200	Rome	1
300	Rome	2
400	Rome	3
100	London	1
400	London	2

Default ORDER BY: with no ORDER BY clause, the order of rows within each partition is arbitrary.

WINDOW FRAME

is a set of rows that are somehow related to the current row. The window frame is evaluated separately within each partition.

ROWS | RANGE | GROUPS BETWEEN lower_bound AND upper_bound



The bounds can be any of the five options:

- UNBOUNDED PRECEDING
- n PRECEDING
- CURRENT ROW
- n FOLLOWING
- UNBOUNDED FOLLOWING

The lower_bound must be BEFORE the upper_bound

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row →	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

1 row before the current row and 1 row after the current row

RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row →	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

values in the range between 3 and 5 ORDER BY must contain a single expression

GROUPS BETWEEN 1 PRECEDING AND 1 FOLLOWING

	city	sold	month
	Paris	300	1
	Rome	200	1
	Paris	500	2
	Rome	100	4
current row →	Paris	200	4
	Paris	300	5
	Rome	200	5
	London	200	5
	London	100	6
	Rome	300	6

1 group before the current row and 1 group after the current row regardless of the value

As of 2020, GROUPS is only supported in PostgreSQL 11 and up.

ABBREVIATIONS

Abbreviation	Meaning
UNBOUNDED PRECEDING	BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
n PRECEDING	BETWEEN n PRECEDING AND CURRENT ROW
CURRENT ROW	BETWEEN CURRENT ROW AND CURRENT ROW
n FOLLOWING	BETWEEN AND CURRENT ROW AND n FOLLOWING
UNBOUNDED FOLLOWING	BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

DEFAULT WINDOW FRAME

If ORDER BY is specified, then the frame is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

Without ORDER BY, the frame specification is ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

What did we learn?

SQL Window Functions Cheat Sheet

LearnSQL.com

LIST OF WINDOW FUNCTIONS

Aggregate Functions
- `avg()`
- `count()`
- `max()`
- `min()`
- `sum()`

Ranking Functions
- `row_number()`
- `rank()`
- `dense_rank()`

Distribution Functions
- `percent_rank()`
- `cume_dist()`

Analytic Functions
- `lead()`
- `lag()`
- `ntile()`
- `first_value()`
- `last_value()`
- `nth_value()`

AGGREGATE FUNCTIONS

- `avg(`*expr*`)` – average value for rows within the window frame
- `count(`*expr*`)` – count of values for rows within the window frame
- `max(`*expr*`)` – maximum value within the window frame
- `min(`*expr*`)` – minimum value within the window frame
- `sum(`*expr*`)` – sum of values within the window frame

ORDER BY and Window Frame: Aggregate functions do not require an ORDER BY. They accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

RANKING FUNCTIONS

- `row_number()` – unique number for each row within partition, with different numbers for tied values
- `rank()` – ranking within partition, with gaps and same ranking for tied values
- `dense_rank()` – ranking within partition, with no gaps and same ranking for tied values

city	price	row_number	rank	dense_rank
		over(order by price)		
Paris	7	1	1	1
Rome	7	2	1	1
London	8.5	3	3	2
Berlin	8.5	4	3	2
Moscow	9	5	5	3
Madrid	10	6	6	4
Oslo	10	7	6	4

ORDER BY and Window Frame: `rank()` and `dense_rank()` require ORDER BY, but `row_number()` does not require ORDER BY. Ranking functions do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

DISTRIBUTION FUNCTIONS

- `percent_rank()` – the percentile ranking number of a row—a value in [0, 1] interval: (rank - 1) / (total number of rows - 1)
- `cume_dist()` – the cumulative distribution of a value within a group of values, i.e., the number of rows with values less than or equal to the current row's value divided by the total number of rows; a value in (0, 1] interval

`percent_rank() OVER(ORDER BY sold)`

city	sold	percent_rank
Paris	100	0
Berlin	150	0.25
Rome	200	0.5
Moscow	200	0.5
London	300	1

← without this row 50% of values are less than this row's value

`cume_dist() OVER(ORDER BY sold)`

city	sold	cume_dist
Paris	100	0.2
Berlin	150	0.4
Rome	200	0.8
Moscow	200	0.8
London	300	1

← 80% of values are less than or equal to this one

ORDER BY and Window Frame: Distribution functions require ORDER BY. They do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

ANALYTIC FUNCTIONS

- `lead(`*expr, offset, default*`)` – the value for the row *offset* rows after the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = NULL
- `lag(`*expr, offset, default*`)` – the value for the row *offset* rows before the current; *offset* and *default* are optional; default values: *offset* = 1, *default* = NULL

`lead(sold) OVER(ORDER BY month)`

order by month	month	sold	
1	1	500	300
2	300		400
3	400		100
4	100		500
5	500		NULL

`lag(sold) OVER(ORDER BY month)`

order by month	month	sold	
1	1	500	NULL
2	300		500
3	400		300
4	100		400
5	500		100

`lead(sold, 2, 0) OVER(ORDER BY month)`

order by month	month	sold	
1	1	500	400
2	300		100
3	400		500
4	100		0
5	500		0

offset=2

`lag(sold, 2, 0) OVER(ORDER BY month)`

order by month	month	sold	
1	1	500	0
2	300		0
3	400		500
4	100		300
5	500		400

offset=2

- `first_value(`*expr*`)` – the value for the first row within the window frame
- `last_value(`*expr*`)` – the value for the last row within the window frame

`first_value(sold) OVER (PARTITION BY city ORDER BY month)`

city	month	sold	first_value
Paris	1	500	500
Paris	2	300	500
Paris	3	400	500
Rome	2	200	200
Rome	3	300	200
Rome	4	500	200

`last_value(sold) OVER (PARTITION BY city ORDER BY month RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)`

city	month	sold	last_value
Paris	1	500	400
Paris	2	300	400
Paris	3	400	400
Rome	2	200	500
Rome	3	300	500
Rome	4	500	500

Note: You usually want to use RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING with `last_value()`. With the default window frame for ORDER BY, RANGE UNBOUNDED PRECEDING, `last_value()` returns the value for the current row.

- `ntile(`*n*`)` – divide rows within a partition as equally as possible into *n* groups, and assign each row its group number.

`ntile(3)`

city	sold	
Rome	100	1
Paris	100	1
London	200	1
Moscow	200	2
Berlin	200	2
Madrid	300	2
Oslo	300	3
Dublin	300	3

ORDER BY and Window Frame: `ntile()`, `lead()`, and `lag()` require an ORDER BY. They do not accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

- `nth_value(`*expr, n*`)` – the value for the *n*-th row within the window frame; *n* must be an integer

`nth_value(sold, 2) OVER (PARTITION BY city ORDER BY month RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)`

city	month	sold	nth_value
Paris	1	500	300
Paris	2	300	300
Paris	3	400	300
Rome	2	200	300
Rome	3	300	300
Rome	4	500	300
Rome	5	300	300
London	1	100	NULL

ORDER BY and Window Frame: `first_value()`, `last_value()`, and `nth_value()` do not require an ORDER BY. They accept window frame definition (`ROWS`, `RANGE`, `GROUPS`).

Nice to have

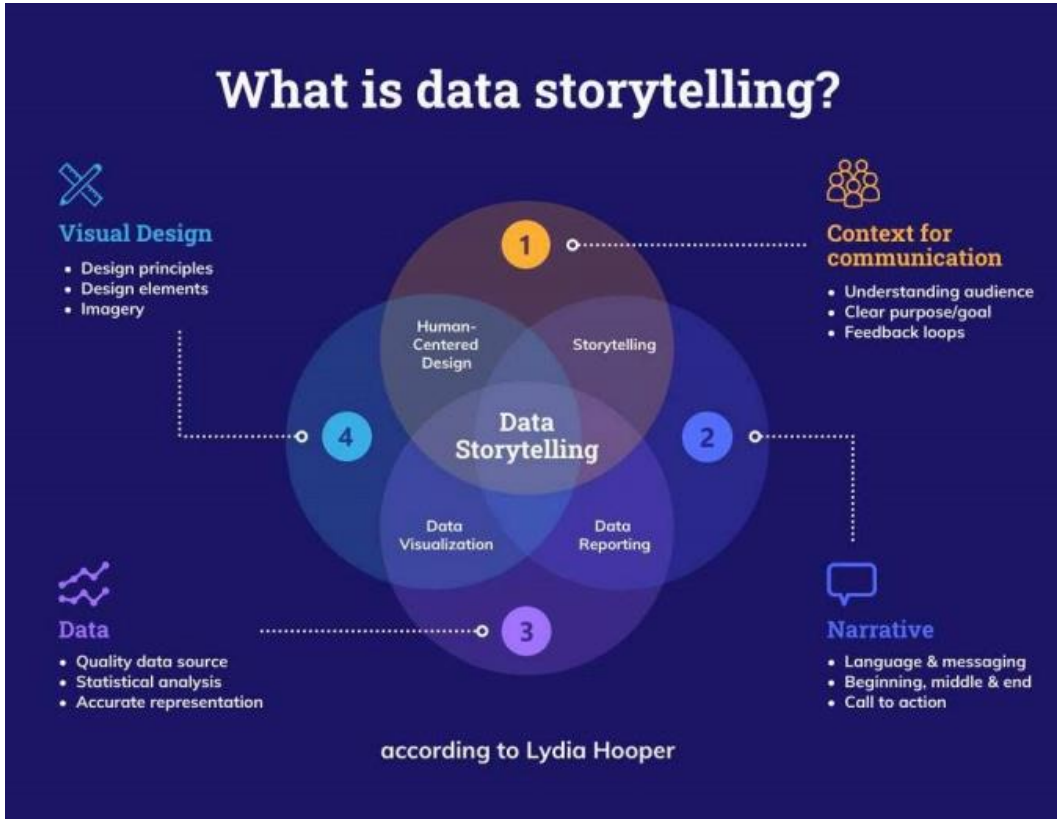
Modifying database

- Create database/table
- Insert value
- Drop database/table
- Delete rows

Variable & Stored Procedure

- Document

What is important for your next journey?



What is important for the next?

- **Statistics**
  - **Descriptive statistics**
    - Numerical summaries
    - Graphs
  - **Inferential statistics**
    - Hypothesis test
    - Statistical tests
    - ...