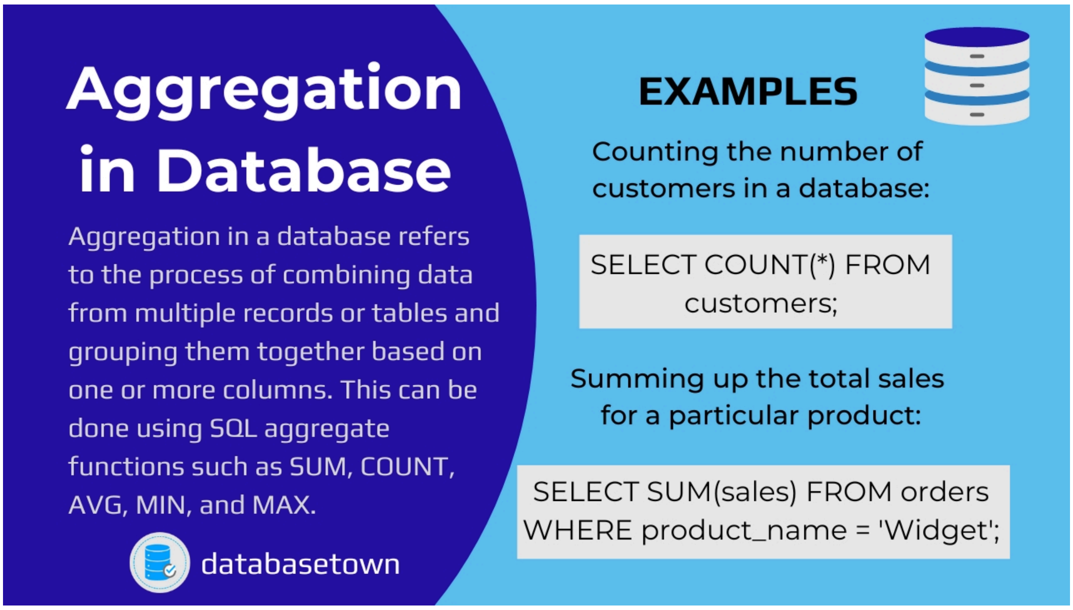
Author: Madhurima Rawat

Data Aggregation for Reporting

Data aggregation techniques were implemented to generate summarized views of large datasets, enhancing reporting and analytical efficiency.

Data Aggregation for Reporting



Aggregation in Database

Aggregation in a database refers to the process of combining data from multiple records or tables and grouping them together based on one or more columns. This can be done using SQL aggregate functions such as SUM, COUNT, AVG, MIN, and MAX.

databasetown

EXAMPLES

Counting the number of customers in a database:

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

Summing up the total sales for a particular product:

```
SELECT SUM(sales) FROM orders
WHERE product_name = 'Widget';
```

What is Data Aggregation?

Data aggregation is the process of collecting, summarizing, and organizing data from multiple sources to generate meaningful insights. It plays a crucial role in **reporting and business intelligence**, enabling organizations to analyze large datasets efficiently.

Aggregated data is often used in **dashboards, reports, and analytics tools** to provide high-level overviews of performance, trends, and patterns.

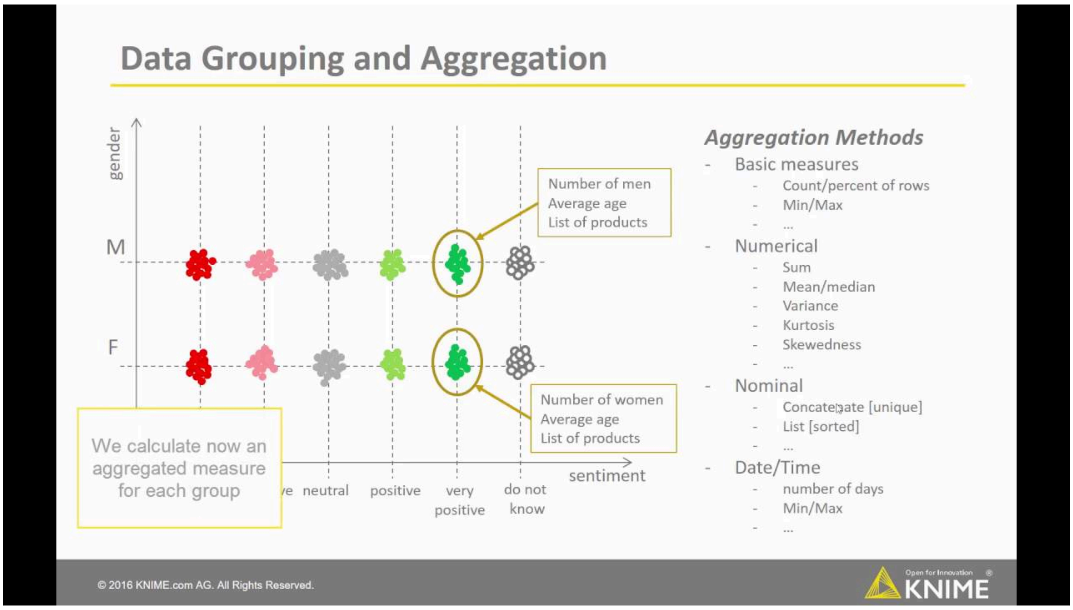
Why is Data Aggregation Important for Reporting?

- **Reduces Complexity** – Converts raw data into summarized, meaningful information.
- **Enhances Decision-Making** – Provides clear insights for business strategies.
- **Improves Performance** – Reduces processing time by storing precomputed results.
- **Enables Trend Analysis** – Helps track performance over time.

Types of Data Aggregation

1. **Summarization** – Computing totals, averages, or counts (e.g., total sales per region).
2. **Grouping** – Categorizing data into different segments (e.g., sales grouped by year).
3. **Filtering** – Selecting specific data based on conditions (e.g., sales above $50,000).
4. **Rollup and Drill-Down** – Aggregating data at different levels (e.g., country → state → city).
5. **Pivoting** – Restructuring data for multi-dimensional analysis (e.g., sales by product and region).

Flowchart: Data Aggregation Process



Data Grouping and Aggregation

Aggregation Methods

- Basic measures
  - Count/percent of rows
  - Min/Max
  - ...
- Numerical
  - Sum
  - Mean/median
  - Variance
  - Kurtosis
  - Skewedness
  - ...
- Nominal
  - Concatenate [unique]
  - List [sorted]
  - ...
- Date/Time
  - number of days
  - Min/Max
  - ...

Steps in Data Aggregation for Reporting:

1. **Data Collection** – Extract data from multiple sources (databases, APIs, logs).
2. **Data Cleaning** – Remove duplicates, correct errors, and standardize formats.
3. **Aggregation Operations** – Apply functions like SUM, AVG, COUNT, MAX, and MIN.
4. **Storage in Data Warehouse** – Save processed data in a structured format.
5. **Data Visualization** – Use dashboards, charts, and reports to present aggregated insights.

Real-Life Example: Retail Sales Reporting

Scenario:

A retail company needs to generate a **monthly sales report** for different stores and product categories.

Challenges:

- Large volumes of raw sales transactions.
- Slow reporting due to complex queries.
- Need for quick, summarized insights.

Aggregated Report Insights:

- **Total sales by region and store.**
- **Average revenue per product category.**
- **Top-selling products for the month.**
- **Sales trends compared to previous months.**

Outcome:

Using **data aggregation techniques**, the company **reduces processing time** and **improves decision-making** with clear, structured reports.

Conclusion

Data aggregation is **essential for reporting and analytics**, providing businesses with **actionable insights** by summarizing large datasets. By efficiently collecting, processing, and visualizing aggregated data, organizations can improve **strategic planning, trend analysis, and performance monitoring**.

Data Aggregation in MySQL

Step 1: Selecting Database and Viewing Tables

Input:

```
SHOW DATABASES;
USE company;
SHOW TABLES;
```

Command Breakdown:

- `SHOW DATABASES;` → Displays all available databases.
- `USE company;` → Switches to the `company` database.
- `SHOW TABLES;` → Lists all tables within the selected database.

Output:

```
+---------------------+
| Database            |
+---------------------+
| classdb             |
| company             |
| employee            |
| hospital            |
| information_schema  |
| movie               |
| mydatabase          |
| mysql               |
| olap                |
| performance_schema  |
| retaildatawarehouse |
| root                |
| shopping            |
| student             |
| sys                 |
| utd                 |
+---------------------+
16 rows in set (0.03 sec)

Database changed

+-------------------+
| Tables_in_company |
+-------------------+
| department        |
| dependent         |
| dept_locations    |
| employee_details  |
| project           |
| works_on          |
+-------------------+
6 rows in set (0.01 sec)
```

This confirms that the `company` database is active, and it contains six tables.

Step 2: Create and Insert Data into the Sales Table

Input:

```
CREATE TABLE Sales (
    Product VARCHAR(50),
    Region VARCHAR(50),
    Year INT,
    Sales_Amount DECIMAL(10,2)
);

INSERT INTO Sales VALUES
('Laptop', 'North', 2022, 50000),
('Laptop', 'South', 2022, 45000),
('Phone', 'North', 2022, 30000),
('Phone', 'South', 2022, 32000),
('Laptop', 'North', 2023, 52000),
('Laptop', 'South', 2023, 47000),
('Phone', 'North', 2023, 31000),
('Phone', 'South', 2023, 33000);
```

Command Breakdown:

- `CREATE TABLE Sales (...)` → Creates the `Sales` table with four columns: `Product`, `Region`, `Year`, and `Sales_Amount`.
- `INSERT INTO Sales VALUES (...)` → Populates the table with **sample sales data** for different products, regions, and years.

Output:

```
Query OK, 0 rows affected (0.02 sec)
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0
```

This confirms that the `Sales` table was created, and 8 records were successfully inserted.

Step 3: Total Sales Calculation

Input:

```
SELECT SUM(Sales_Amount) AS Total_Sales FROM Sales;
```

Command Breakdown:

- `SUM(Sales_Amount)` → Computes the total sum of all sales records in the table.

Output:

```
+-------------+
| Total_Sales |
+-------------+
|   320000.00 |
+-------------+
1 row in set (0.00 sec)
```

This confirms that the total revenue generated from all sales is **320,000.00**.

Step 4: Average Sales Per Product

Input:

```
SELECT Product, AVG(Sales_Amount) AS Avg_Sales
FROM Sales
GROUP BY Product;
```

Command Breakdown:

- `AVG(Sales_Amount)` → Calculates the average sales per product.
- `GROUP BY Product` → Groups results by product category to compute averages separately.

Output:

```
+---------+-------------+
| Product | Avg_Sales    |
+---------+-------------+
| Laptop  | 48500.000000 |
| Phone   | 31500.000000 |
+---------+-------------+
2 rows in set (0.00 sec)
```

This shows that **laptops have an average sales amount of 48,500** while **phones average 31,500**.

Step 5: Count of Sales by Region

Input:

```
SELECT Region, COUNT(*) AS Sales_Count
FROM Sales
GROUP BY Region;
```

Command Breakdown:

- `COUNT(*)` → Counts the total number of sales transactions for each region.
- `GROUP BY Region` → Groups sales data by region.

Output:

```
+--------+-------------+
| Region | Sales_Count |
+--------+-------------+
| North  |           4 |
| South  |           4 |
+--------+-------------+
2 rows in set (0.00 sec)
```

This shows that **both North and South regions had four recorded sales transactions**.

Step 6: Maximum and Minimum Sales per Year

Input:

```
SELECT Year, MAX(Sales_Amount) AS Max_Sales, MIN(Sales_Amount)
AS Min_Sales
FROM Sales
GROUP BY Year;
```

Command Breakdown:

- `MAX(Sales_Amount)` → Finds the highest sales value for each year.
- `MIN(Sales_Amount)` → Finds the lowest sales value for each year.
- `GROUP BY Year` → Groups data by year for separate calculations.

Output:

```
+------+-----------+-----------+
| Year | Max_Sales | Min_Sales |
+------+-----------+-----------+
| 2022 |  50000.00 |  30000.00 |
| 2023 |  52000.00 |  31000.00 |
+------+-----------+-----------+
2 rows in set (0.00 sec)
```

This reveals the **highest and lowest sales values for 2022 and 2023**.

Step 7: Sales Aggregation with `UNION ALL`

SQL Query:

```
SELECT Product, NULL AS Region, SUM(Sales_Amount) AS Total_Sales
FROM Sales
GROUP BY Product
UNION ALL
SELECT NULL, Region, SUM(Sales_Amount) AS Total_Sales
FROM Sales
GROUP BY Region
UNION ALL
SELECT NULL, NULL, SUM(Sales_Amount) AS Total_Sales
FROM Sales;
```

Explanation:

This query is **manually simulating the `GROUPING SETS` feature**, which is available in some databases but not in MySQL.

👉 How does this work?

- We perform **three separate `GROUP BY` queries**:
  1. **First query:** Aggregates sales **by Product**.
  2. **Second query:** Aggregates sales **by Region**.
  3. **Third query:** Computes the **overall total sales**.
- These results are **combined** using `UNION ALL` to create a single result set.

Breaking Down the Query:

🔹 First Query (Total Sales per Product)

```
SELECT Product, NULL AS Region, SUM(Sales_Amount) AS Total_Sales
FROM Sales
GROUP BY Product
```

✅ Output Example:

```
+---------+--------+-------------+
| Product | Region | Total_Sales |
+---------+--------+-------------+
| Laptop  | NULL   |   194000.00 |
| Phone   | NULL   |   126000.00 |
+---------+--------+-------------+
```

📌 Why `NULL AS Region` ?
We don't need the **Region** column for this part of the query, so we **fill it with NULL**.

🔹 Second Query (Total Sales per Region)

```
SELECT NULL, Region, SUM(Sales_Amount) AS Total_Sales
FROM Sales
GROUP BY Region
```

✅ Output Example:

```
+---------+--------+-------------+
| Product | Region | Total_Sales |
+---------+--------+-------------+
| NULL    | North  |   163000.00 |
| NULL    | South  |   157000.00 |
+---------+--------+-------------+
```

📌 Why `NULL AS Product` ?
Since we are grouping **by Region**, the **Product column is irrelevant** here, so we set it to NULL.

🔹 Third Query (Total Sales Overall)

```
SELECT NULL, NULL, SUM(Sales_Amount) AS Total_Sales
FROM Sales;
```

✅ Output Example:

```
+---------+--------+-------------+
| Product | Region | Total_Sales |
+---------+--------+-------------+
| NULL    | NULL   |   320000.00 |
+---------+--------+-------------+
```

📌 What does this do?
It **calculates the grand total of all sales**, without any grouping.

Final Combined Output (Using `UNION ALL`)

```
+---------+--------+-------------+
| Product | Region | Total_Sales |
+---------+--------+-------------+
| Laptop  | NULL   |   194000.00 |
| Phone   | NULL   |   126000.00 |
| NULL    | North  |   163000.00 |
| NULL    | South  |   157000.00 |
| NULL    | NULL   |   320000.00 |
+---------+--------+-------------+
```

✓ Summary of Each Row:

1. **Total sales for each product** (`Laptop` , `Phone`).
2. **Total sales for each region** (`North` , `South`).
3. **Overall total sales** (final row).

Step 8: Sales Aggregation with `ROLLUP`

SQL Query:

```sql
SELECT Region, Product, SUM(Sales_Amount) AS Total_Sales
FROM Sales
GROUP BY Region, Product WITH ROLLUP;
```

Explanation:

- `WITH ROLLUP` is an **automatic way to calculate subtotals and grand totals** in MySQL.
- Instead of writing **multiple queries with `UNION ALL`** , `ROLLUP` handles everything in one step.

How `WITH ROLLUP` Works:

- It **groups the data** at multiple levels:
    i. **Total sales per region and product.**
    ii. **Subtotal per region (NULL in `Product` column).**
    iii. **Grand total of all sales (NULL in both columns).**

Breaking Down the Output:

```
+--------+---------+-------------+
| Region | Product | Total_Sales |
+--------+---------+-------------+
| North  | Laptop  |   102000.00 |
| North  | Phone   |    61000.00 |
| North  | NULL    |   163000.00 | <-- (Subtotal for North)
| South  | Laptop  |    92000.00 |
| South  | Phone   |    65000.00 |
| South  | NULL    |   157000.00 | <-- (Subtotal for South)
| NULL   | NULL    |   320000.00 | <-- (Grand total)
+--------+---------+-------------+
```

✔ Row Analysis:

1. **Regular rows:** `Region` and `Product` are filled (specific product sales per region).
2. **Subtotal rows:** `Product` is `NULL` → Shows total sales per region.
3. **Grand Total row:** Both `Region` and `Product` are `NULL` → Shows total for all sales.

Key Differences Between `UNION ALL` and `ROLLUP`

Feature	`UNION ALL`	`WITH ROLLUP`
Manual or Automatic?	**Manual** (multiple queries combined)	**Automatic** (one query)
Flexibility	Can create **custom aggregations**	Follows a **fixed hierarchy**
Performance	**Less efficient** (runs multiple queries)	**More efficient** (single query)
Complexity	Requires explicit `NULL` handling	Automatically fills `NULL` values
Grouping Levels	Custom levels (product, region, total)	Fixed hierarchy (from more detail to summary)

Which One Should You Use?

- ✅ **Use `ROLLUP` if you want automatic subtotals & grand totals.**
- ✅ **Use `UNION ALL` if you need more flexibility in grouping.**

📌 Example Use Cases:

- **`WITH ROLLUP` is great for reports** where you need **subtotals and grand totals.**

- `UNION ALL` **is useful** if you need **custom grouping logic** that `ROLLUP` can't handle.

Final Summary

Query	Purpose
`UNION ALL` Aggregation	Manually groups data at **multiple levels** (Product, Region, Total).
`WITH ROLLUP` Aggregation	Automatically calculates **subtotals and grand total** in one query.

Both methods achieve **multi-level aggregation**, but `ROLLUP` **is more efficient and easier to use** in MySQL. 🚀

Final Summary

- **SUM, AVG, MAX, MIN, and COUNT** provide **essential sales insights**.
- **UNION ALL and ROLLUP** allow multi-level aggregations.
- **Data aggregation techniques** help businesses **analyze trends, forecast sales, and optimize decision-making**.