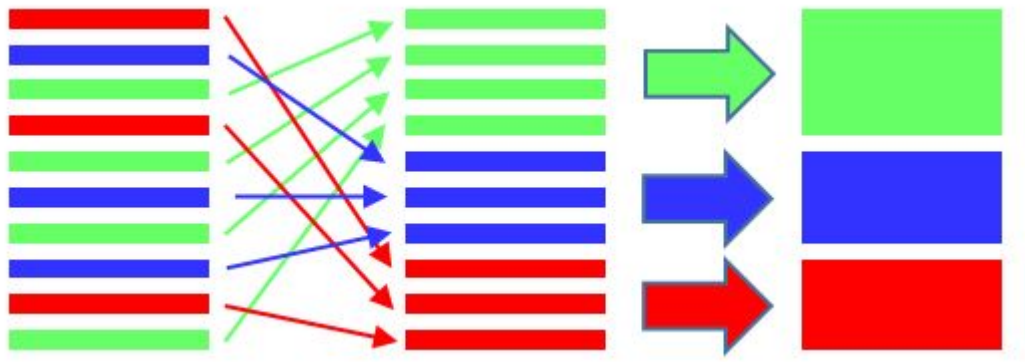
GROUP BY

Purpose of GROUP BY

- Aggregation functions: SUM, AVG, MIN, MAX, COUNT, COUNT DISTINCT
    - Describing characteristics of a dataset
- The need to divide a dataset into smaller groups for research
    - Average grade in one course vs average grade in another course
    - Weather data in different cities
    - Stock price patterns in one sector vs patterns in another sector

GROUP BY

- Group rows in a dataset into subsets
    - Putting rows of same characteristics into same processing unit



GROUP BY Syntax

- GROUP BY keyword is after table selection (FROM)
- Column order does not matter, but SELECT columns must either be a GROUP BY column or an aggregated column.

```
SELECT <group by columns>, <aggregated columns>
FROM <table>
GROUP BY <group by columns>
```

GROUP BY Example

```
SELECT Student_Name, SUM(Actual_Tuiton)
FROM Registration
GROUP BY Student_Name
```

Student Name	Course Name	Actual Tuition	SUM
Luke Skywalker	Intro Force	180	280
	Dagobah	100	
Leah Skywalker	Intro Force	200	700
	Public Speech	500	

Matching SELECT to GROUP BY

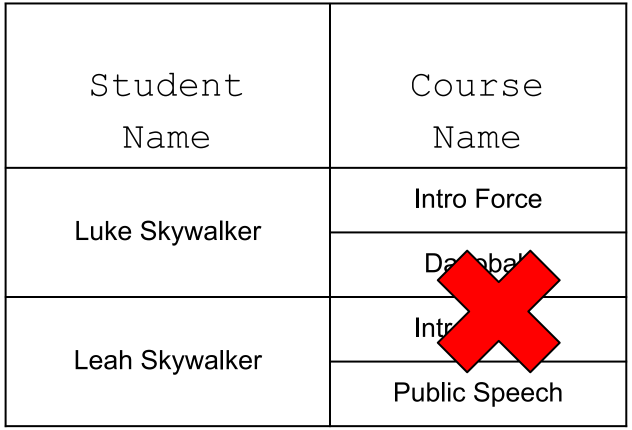
- All columns in SELECT must be either in GROUP BY or aggregated
    - Non aggregation columns that are not in GROUP BY are returned as repeating groups, which won’t be 1NF compliant. So, SQL does not allowing selecting columns directly without GROUP BY.

```
SELECT Student_Name, Course_Name
FROM Registration
GROUP BY Student_Name
```

SELECT without GROUP BY

- SELECT without GROUP BY and/or Aggregation: Repeating groups

```
SELECT Student_Name,
    Course_Name
FROM Registration
GROUP BY Student_Name
```



Student Name	Course Name
Luke Skywalker	Intro Force
	Da[illegible]ba[illegible]
Leah Skywalker	Intr[illegible]
	Public Speech

- It is possible that there may not be repeating groups based on data.

  But SQL chooses to enforce 1NF by default.

GROUP BY with Aggregation

Aggregations (SUM, COUNT, etc.) do not need to be in GROUP BY because these functions will only return one value per group. Thus it is 1NF compliant.

Student Name	Course Name	Actual Tuition	SUM
Luke Skywalker	Intro Force	180	280
	Dagobah	100	
Leah Skywalker	Intro Force	200	700
	Public Speech	500	

Multi-Column GROUP BY

- Multi-column GROUP BY: All GROUP BY columns must be in SELECT
  - GROUP BY is for all value combos. Column order not important.
  - The order in SELECT determines the order of display
  - GROUP BY on primary key is the same as a normal SELECT

```
SELECT Student_Name, Course_Name, AVG(Actual_Tuition)
FROM Registration
GROUP BY Student_Name, Course_Name

SELECT Student_Name, Course_Name, Actual_Tuition
FROM Registration
```

GROUP BY with ORDER BY

- GROUP BY with ORDER BY
  - The result set of GROUP BY can be ordered by the columns in this result set, but not columns in the original dataset, if they are not in the result set.
  - This includes the aggregation function columns too.

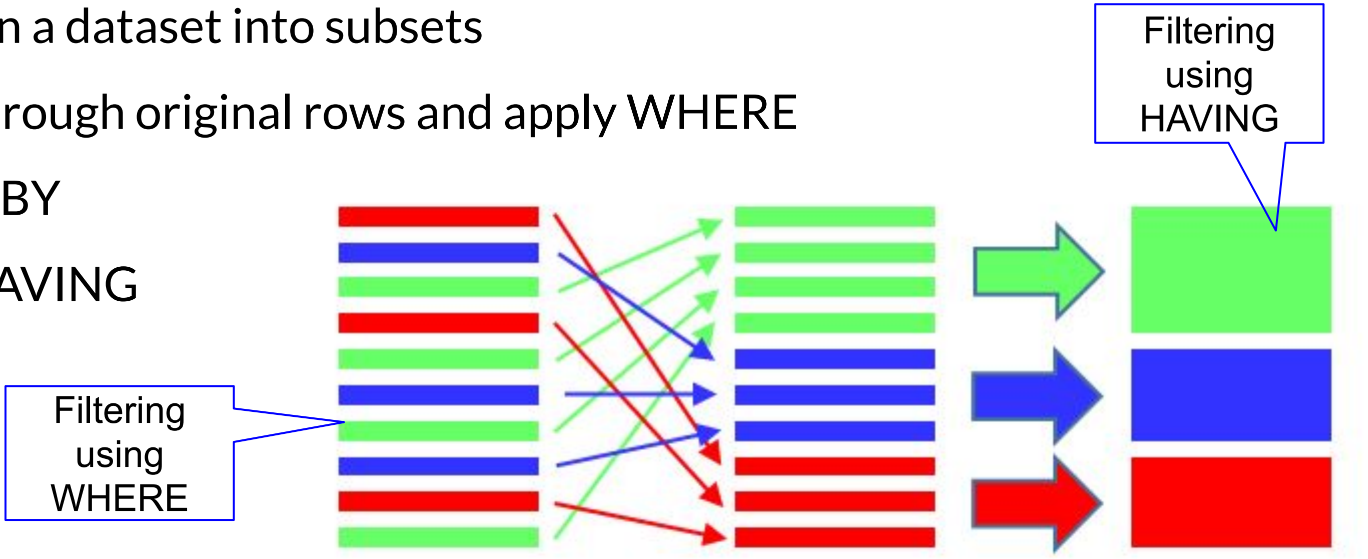
GROUP BY Syntax

```
SELECT <group by columns>, <aggregated columns>
FROM <table>
GROUP BY <group by columns>
ORDER BY <subset of columns in SELECT clause>
```

WHERE and HAVING in GROUP BY

GROUP BY

- Group rows in a dataset into subsets
    - Going through original rows and apply WHERE
    - GROUP BY
    - Apply HAVING



GROUP BY with Aggregation

HAVING SUM(Actual_Tuition) > 600

Student Name	Course Name	Actual Tuition	SUM
Luke Skywalker	Intro Force	180	280
	Dagobah	100	
Leah Skywalker	Intro Force	200	700
	Public Speech	500	

HAVING Example

```
SELECT Student_Name, SUM(ACTUAL_Tuition)
FROM Registration
GROUP BY Student_Name
HAVING SUM(ACTUAL_Tuition) > 600
```

HAVING

- HAVING Clause:
    - Key word `HAVING`, applied after GROUP BY clause, followed by filtering conditions
    - Conditions applied to the aggregated dataset
    - Only data in the aggregated dataset should be used
        - Aggregation functions
        - GROUP BY columns (poor performance, not recommended)

HAVING

- HAVING vs WHERE:
  - WHERE is applied on original dataset, thus cannot operate on the aggregated dataset
  - Having is applied on the aggregate dataset
  - Use WHERE whenever possible for performance reason

GROUP BY

- Execution order:

  1. FROM …
  2. WHERE …
  3. GROUP BY
  4. SELECT …
  5. HAVING
  6. ORDER BY

GROUP BY

- Execution order:

```
4 SELECT <group by columns>, <aggregated columns>
1 FROM <table>
2 WHERE <boolean expression of table columns>
3 GROUP BY <group by columns>
5 HAVING <boolean expression of group by columns>
6 ORDER BY <subset of group by columns>
```

GROUP BY

- At Step 1, DBMS can find Registration_Date in table.
- At Step 5 and 6, DBMS can only sees Student_Name and SUM() from GROUP BY datasets. So HAVING and ORDER BY can only use these two columns.

```
4.    SELECT Student_Name, SUM(Actual_Tuition)
1.    FROM Registration
2.    WHERE Registration_Date > ‘01-OCT-2019’
3.    GROUP BY Student_Name
5.    HAVING SUM(Actual_Tuition) > 600
6.    ORDER BY Student_Name
```

HAVING Summary

- WHERE is on detail (original table) level, and can use all source columns.
- HAVING is on aggregation level, and can use all SELECT columns

SELECT Statement with JOIN and GROUP BY

Full SELECT Statement Syntax

```
SELECT <qualified column list>
FROM <table 1> <alias 1>
JOIN <table 2> <alias 2>
  ON <matching column 1> = <matching column 1>
WHERE <boolean expression>
GROUP BY <GROUP BY column list>
HAVING <boolean expression>
ORDER BY <ORDER BY column list>
```

SELECT Execution Order

Execution order:

1. FROM … JOIN … WHERE will create a detailed dataset
2. GROUP BY on top of this detailed data set will form an aggregated dataset over the joined tables
3. Use HAVING to filter on aggregated dataset
4. Use ORDER BY to sort the result

SELECT Execution Order

```
    SELECT <qualified column list>
    FROM <table 1> <alias 1>
1   JOIN <table 2> <alias 2>
      ON <matching column 1> = <matching column 1>
    WHERE <boolean expression>
2   GROUP BY <GROUP BY column list>
3   HAVING <boolean expression>
4   ORDER BY <ORDER BY column list>
```

How to Write a SELECT Statement

1.a Find the tables involved

For each column required in the question, find its source table

1.b Starting from FROM

Starting from FROM: The table with most join keys should be in FROM clause (usually it is also the table with lowest level of details)

How to Write a SELECT Statement

1.c From FROM to JOIN

Add table name in the JOIN clause, then add join key in the ON clause.

- Each new table is a independent JOIN. Join one table at a time.

1.d Add WHERE condition -- Generates the detailed dataset

- This whole FROM...JOIN...WHERE chunk is one dataset (relation).

How to Write a SELECT Statement

2. Apply aggregation
    - Add GROUP BY for each “for each”
    - Add the same columns to SELECT
    - Add required aggregation functions to SELECT
3. Add HAVING if needed
4. Apply ORDER BY

Query as a Relation

SQL is Relational Algebra

- SQL operations are relational algebra operations
    - No matter how complex the query may be, the result of each SELECT query is another relation

Subquery to Select From

- Subqueries can be used as the dataset to be SELECTed from.

```
SELECT Instructor_Name FROM (
    SELECT * FROM Instructor
    WHERE Instructor_Affiliation = ‘Jedi’
) AS Jedi_Instructors
```

  - Need a table alias (`...AS...`) to designate this subquery

SQL is Relational Algebra

- Nested query: Main query with subquery, within which there are sub-subqueries
    - Work out each query as a relation, one level at a time.
    - Think about each query component as a relation

View

View definition

- A way to store a SELECT statement and make it easier to remember/use, aka a way to reuse subquery.

```
CREATE VIEW <view_name> (<column_list>) AS
SELECT <column_list> ...
```

  - Column list is used to define column names
- View can be altered and dropped

```
DROP VIEW <view_name>
```

View Usage

- Views can be used in any SELECT statement in the same way that tables are used
- The behavior of INSERT/UPDATE/DELETE on view is complex and may lead to error
    - In general, avoid INSERT/UPDATE/DELETE on views

Updating View

- Updating view is indeed updating the underlying table(s)
  - Updating row-and-column subset: NULL for un-included column.
  - Updating views with functions: Cannot update function results
  - Updating views whose underlying table has NO NULL or other restraints: Unpredictable
- In general, avoid INSERT/UPDATE/DELETE on views
  - Some DBMS don’t allow UPDATE on views.

Materialized View

- Views are saved as definition only. No data is saved
  - Every time a view is referenced, its underlying query is executed to fetch the result
- Materialized View: Save the data until been refreshed

```
CREATE MATERIALIZED VIEW mymatview
AS SELECT * FROM mytable;

REFRESH MATERIALIZED VIEW mymatview;
```

Non Permanent Relation

Temporary Table

- Exists only for the user session (from log-in to log-out)
    - Syntax is vendor specific

```
CREATE TEMPORARY TABLE my_temp_table (
  id            INT,
  description   VARCHAR(20)
)
```

Common Table Expression (CTE)

- Created as a part of the query. Effective only for the specific query.
    - Can be used to create recursive dataset

```
WITH
  cte1 AS (SELECT a, b FROM table1),
  cte2 AS (SELECT c, d FROM table2)
SELECT b, d
FROM cte1 JOIN cte2
```

Filtering from Another Table

Review IN Subquery

- How to use the result of a query in another query's IN clause?
  - Put the first query in parenthesis and place it in the second query

```
SELECT * FROM Course
WHERE Instructor_Name IN (
    SELECT Instructor_Name FROM Instructor
    WHERE Instructor_Affiliation = 'Jedi' )
```

  - Associating data from two tables via a common column
    - Which is what JOIN does too

Use JOIN for Filtering

- Associate two tables with JOIN

```
SELECT Course.*
FROM Course JOIN Instructor
on Course.Instructor_Name = Instructor.Instructor_Name
WHERE Instructor_Affiliation = ‘Jedi’
```

Filtering Data using Another Table

- Method 1: Use WHERE IN (subquery) clause
- Method 2: Join two tables together. Filter on the joined dataset
- Method 3: Use WHERE EXISTS ()

EXISTS

- Check the result of a correlated subquery
    - Subquery with correlation condition between main query and subquery
    - Correlation condition: same as the ON clause in JOIN between the two related tables

Use EXISTS for Filtering

- Associate two tables with JOIN

```
SELECT * FROM Course
WHERE EXISTS (
    SELECT * FROM Instructor
    WHERE Instructor_Affiliation = ‘Jedi’
    AND Course.Instructor_Name =
    Instructor.Instructor_Name
)
```

Filtering Condition Execution

- Same process for all
  - Outside query is executed row by row
  - For each row, go through the inner query (or joined table) row by row, matching outside query’s current row and apply WHERE clause

JOIN vs IN vs EXISTS

- Similarity in execution: Matching values in another table
  - SELECT contains information from both tables: JOIN
  - SELECT only from one main table: IN or EXISTS
  - In most cases, if you can use IN or EXISTS, you can also use JOIN
- Performance depends on DBMS setup. Use try-and-error to determine which one to use.

UPDATE From Multiple Tables

UPDATE Review

- Update a field in target table, one row at a time
    - Data can be literal or expression based on target table’s columns

```
UPDATE <table>
SET <column_1> = <value_1>, <column_2> = <value_2>, …
WHERE <filter_condition>
```

UPDATE From Join

- Can also based on a relation formed by joining target table to other tables
  - New value can be expression based on new relation’s columns

```
UPDATE <target_table>
SET <column_1> = <value_1>, <column_2> = <value_2>, …
FROM <table_2>, <table_3>, ...
WHERE <join_condition>
AND <filter_condition>
```

New Relation Definition

- New relation is the join of all tables in FROM clause plus target table
  - Joined with `<join_condition>` in WHERE clause

```
UPDATE <target_table>
SET <column_1> = <value_1>, <column_2> = <value_2>, …
FROM <table_2>, <table_3>, ...
WHERE <join_condition>
AND <filter_condition>
```

Duplicated Values

- If each row in target table corresponds to multiple values in the new relation, the result is unpredictable: Postgres will randomly pick one.
- For this reason, the tables in FROM clause must all have 1:M relationship with target table, i.e., the target table is functionally dependent on all the tables in the FROM clause.