



# Advanced SQL: Grouping, Aggregation and Views

**Abdu Alawini**

University of Illinois at Urbana-Champaign

CS411: Database Systems



# Learning Objectives

After this lecture, you should be able to:

- Write grouping and aggregate SQL queries
- Define SQL *views*, differentiate between *virtual* and *materialized* views, and know when a view can be updated.



# Aggregations

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.
- Also, COUNT(\*) counts the number of tuples.



# Example: Aggregation

- From Sells(caf , drink, price), find the average price of Mocha:

```
SELECT AVG(price)
FROM Sells
WHERE drink = 'Mocha';
```



# Eliminating Duplicates in an Aggregation

- DISTINCT inside an aggregation causes duplicates to be eliminated before the aggregation.
- Example: find the number of different prices charged for Mocha:

```
SELECT COUNT(DISTINCT price)
FROM Sells
WHERE drink = 'Mocha';
```



# NULL's Ignored in Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.
- But if all the values in a column are null, then the result of the aggregation is NULL.

# I

## Example: Effect of NULL's

The number of cafes  
that sell Mocha.

```
SELECT count(*)  
FROM Sells  
WHERE drink = 'Mocha';
```

```
SELECT count(price)  
FROM Sells  
WHERE drink = 'Mocha';
```

The number of cafes  
that sell Mocha at a  
known price.



# Grouping

- We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.
- The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group.





## Example: Grouping

- From Sells(cafe, drink, price), find the average price for each drink:

```
SELECT drink, AVG(price)
FROM Sells
GROUP BY drink;
```

# I

## Example: Grouping

- From Sells(cafe, drink, price) and Frequents(customer, cafe), find for each customer the average price of Mocha at the cafes they frequent:

```
SELECT customer, AVG(price)
FROM Frequents NATURAL JOIN Sells
WHERE drink = 'Mocha'
GROUP BY customer;
```

Compute combos  
of frequents  
and sells for  
Mocha selling cafés,  
then group  
by customer.



# Restriction on SELECT Lists With Aggregation

If any aggregation is used, then each element of the SELECT list must be either:

1. Aggregated, or
2. An attribute on the GROUP BY list.

Doesn't make sense to neither have be grouped or aggregated – how are you “collapsing” values down then?



# HAVING Clauses

- HAVING <condition> may follow a GROUP BY clause.
- If so, the condition applies to each group, and groups not satisfying the condition are eliminated.



# The HAVING clause: Example

```
SELECT drink, AVG(price)
FROM Sells
GROUP BY drink
HAVING COUNT(cafe) >= 3
```

*What does this query ask for?*



# The HAVING clause: Example

```
SELECT drink, AVG(price)
FROM Sells
GROUP BY drink
HAVING COUNT(cafe) >= 3
```

*What does this query ask for?*

*Average price of drinks that are sold at least at three cafes.*



# Requirements on HAVING Conditions

- These conditions may refer to any relation or tuple-variable in the FROM clause.
- They may refer to attributes of those relations, as long as the attribute makes sense within a group; i.e., it is either:
  1. A grouping attribute, or
  2. Aggregated.

# I General form of Grouping and Aggregation

SELECT S  
FROM  $R_1, \dots, R_n$   
WHERE  $C_1$   
GROUP BY  $a_1, \dots, a_k$   
HAVING  $C_2$

S = may contain attributes  $a_1, \dots, a_k$  and/or any aggregates but NO OTHER ATTRIBUTES

$C_1$  = is any condition on the attributes in  $R_1, \dots, R_n$

$C_2$  = is any condition on aggregate expressions or grouping attributes



# General form of Grouping and Aggregation

```
SELECT S  
FROM R1,...,Rn  
WHERE C1  
GROUP BY a1,...,ak  
HAVING C2
```

Evaluation steps:

1. Compute the FROM-WHERE part, obtain a table with all attributes in  $R_1, \dots, R_n$
2. Group by the attributes  $a_1, \dots, a_k$
3. Compute the aggregates in  $C_2$  and keep only groups satisfying  $C_2$
4. Compute aggregates in  $S$  and return the result



# Outline

- ✓ Aggregation and Grouping
- Views



# Views

- A view is a “virtual table,” a relation that is defined in terms of the contents of other tables and views.
- Declare by:

```
CREATE VIEW <name> AS <query>;
```
- Views are not stored in the database, but can be queried as if they existed.
  - We’ll talk about an exception later
- In contrast, a relation whose value is really stored in the database is called a *base table*.



## Example: View Definition

- CanDrink (customer, drink) is a view “containing” the customer-drink pairs such that the customer frequents at least one cafe that serves the drink on relations Frequents (customer, cafe) and Sells (cafe, drink, price)

```
CREATE VIEW CanDrink AS
SELECT customer, drink
FROM Frequents, Sells
WHERE Frequents.cafe = Sells.cafe;
```



## Example: Accessing a View

- You may query a view as if it were a base table.
- Example:

```
SELECT drink FROM CanDrink  
WHERE customer = 'Sally';
```



# What's Useful about Views

1. Can be used as relations in other queries
  - Allows the user to query things that make more sense
  - Can be stored (materialized) as appropriate
  - Sometimes can even be updated!



# What's Useful about Views

## 2. Can facilitate security/access control

- We can assign users permissions on different views
- Can select or project so we only reveal what we want!

## 3. Describe *transformations* or *mappings* from one schema (the base relations) to another (the output of the view)

- The basis of converting from different data models or representations
- Incredibly useful for logical data independence



# View Example

A company's database includes a relation:  
Part (PartID: Char(4), Weight:real)

- Weight is stored in Kilograms (kg)
- Company is purchased by a firm that uses imperial weights (lb)
- Databases must be integrated and use lb.
- But there's much old software using kilograms.
- 1 KG = ~2.2 LB





## Example (Cont.)

Create a view over the parts table so that it uses imperial weights

```
CREATE VIEW Part_lb AS  
SELECT PartID, Weight*2.2 as Weight_lb  
FROM Part
```



# Materialized Views

A **materialized view** is one that is computed once and its results are stored as a table

- Think of this as a cached answer
- These are incredibly useful!
- Techniques exist for using materialized views to answer other queries
- Materialized views are the basis of relating tables in different schemas

```
CREATE MATERIALIZED VIEW AS ...
```



# Views Should Stay Fresh

- Views (sometimes called *intensional relations*) behave, from the perspective of a query language, exactly like base relations (*extensional* relations)
- But there's an association that should be maintained:
  - If tuples change in the base relation, they should change in the view (whether it's materialized or not)
  - If tuples change in the view, that should reflect in the base relation(s)

# View Maintenance and Updates

- There exist algorithms to **incrementally** recompute a materialized view when the base relations change
- We can try to propagate view changes to the base relations
  - However, there are lots of views that aren't easily updatable:

R

A	B
1	2
2	2

S

B	C
2	4
2	3

R ⋈ S

A	B	C
1	2	4
1	2	3
2	2	4
2	2	3

delete?

↙



# The good news

We can ensure views are updatable by enforcing certain constraints:

- It is defined on a single base table
- Using only selection and projection
- No aggregates
- No DISTINCT

...but this limits the kinds of views we can have