

SQL: Aggregate Functions, GROUP BY, HAVING & Nested Queries

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. Aggregation
2. Nested Sub-Queries
3. Bonus Material: NULL

Aggregate Functions

We use **aggregate functions** to summarize information from multiple tuples into a single-tuple summary

- ▶ The following SQL functions operate on the **multiset of a single column's values**
 - ▶ They return a **scalar value**
- ▶ **avg**: average value
- ▶ **min**: minimum value
- ▶ **max**: maximum value
- ▶ **sum**: sum of values
- ▶ **count**: number of values

Aggregate Functions: Examples

Find the average salary of instructors in the Computer Science department

```
1 SELECT AVG (salary) AS avg_salary
2   FROM instructor
3   WHERE dept_name='Comp_Sci'
```

Find the total number of instructors who teach a course in the Spring 2018 semester

```
1 SELECT COUNT (DISTINCT ID)
2   FROM teaches WHERE semester = 'Spring' and year = 2018;
```

Find the number of tuples in the course relation

```
1 SELECT COUNT (*) FROM course;
```

Renaming Results of Aggregation

- This query returns a single row of computed values from an *Employee* table

```
1  SELECT
2    SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
3  FROM EMPLOYEE;
```

- You can rename the aggregations results using the AS clause

```
1  SELECT
2    SUM (Salary) AS Total_Sal, MAX (Salary) AS
      Highest_Sal,
3    MIN (Salary) AS Lowest_Sal, AVG (Salary) AS
      Average_Sal
4  FROM EMPLOYEE;
```

Review – Computation in SELECT clauses

```
SELECT location, time, celsius * 1.8 + 32 AS fahrenheit  
FROM SensorReading;
```

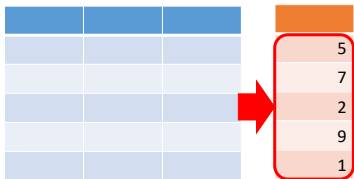
```
SELECT player_id, Floor(height) AS feet, (height – Floor(height)) * 12 AS inches  
FROM Player;
```

```
SELECT student_id, CASE WHEN lastname < 'N' THEN 1 ELSE 2 END AS group  
FROM Student;
```

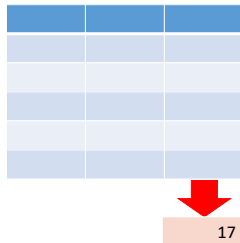
```
SELECT lastname || ', ' || firstname AS name  
FROM Student;
```

Horizontal vs. vertical computation

“Horizontal” – One result per row



“Vertical” – One result per column



Vertical – Aggregation

```
SELECT Sum(price)  
FROM Candy;
```

Candy

<u>candy_name</u>	price
Reese's Cup	\$0.50
5 th Avenue	\$1.20
Almond Joy	\$0.75
Jelly Babies	\$2.39



\$4.84

Aggregation combines with other features

```
SELECT Sum(price * quantity) AS total  
FROM Purchase  
WHERE product = 'bagel';
```

Purchase

product	date	price	quantity
bagel	2017-10-21	\$1.00	20
banana	2017-10-03	\$0.50	10
banana	2017-10-10	\$1.00	10
bagel	2017-10-25	\$1.50	20



total
\$50.00

Aggregate Functions & NULL

- ▶ **Q:** what behavior would you predict when an aggregate function is applied to an attribute containing NULL values?
- ▶ **A:** NULL values are discarded by aggregation (as you'd expect) ...
- ▶ But there is an important exception ☹ : COUNT
- ▶ COUNT(salary) will ignore NULL instances
- ▶ But: COUNT(*) is applied to **tuples, not attribute values!**
 - ▶ Therefore: tuples with NULL attributes (e.g., in the WHERE clause) will be counted

Aggregate Functions, COUNT & NULL (I)

```
1 SELECT COUNT(*) AS TotalRows, COUNT(region) AS NonNULLRows,  
2      COUNT(*) - COUNT(region) AS NULLRows  
3 FROM Northwind.dbo.suppliers
```

- This **example table** contains a mixture of rows: some with “region == NULL” and some where “region” does have a value

TotalRows	NonNULLRows	NULLRows
29	9	20

Aggregate Functions, COUNT & NULL (II)

```
1 SELECT COUNT(region) AS NULLRows
2 FROM Northwind.dbo.suppliers
3 WHERE region IS NULL
```

- ▶ This query returns “0” for the value of the “NULLRows” attribute!
- ▶ Q: why?
- ▶ A: it’s not magic! Let’s go through the SQL evaluation steps
 1. The WHERE clause explicitly eliminates every row in which region is not NULL
 2. The “value” expression in this example is simply a column name: “region”
 3. The COUNT(region) function implicitly eliminates every row in which region is NULL
- ▶ Result: 0 rows

Aggregate Functions, NULL, & Empty Relations

- ▶ A FROM/where clause may produce an **empty relation**
 - ▶ Because no tuples pass the **predicate test** ...
 - ▶ Possibly because NULL values are discarded when applying an aggregate function
- ▶ The behavior of aggregate functions when applied to an empty relation sort of, but doesn't quite, make sense 😊
 - ▶ `SELECT COUNT (*)` → returns 0
 - ▶ `SELECT COUNT` → returns 0
 - ▶ `SELECT MAX` → returns NULL
 - ▶ `SELECT MIN` → returns NULL
 - ▶ `SELECT AVG` → returns NULL
 - ▶ `SELECT SUM` → returns NULL
- ▶ IMHO the behavior of SUM is particularly bizarre 😞
 - ▶ But remember: NULL scalar arithmetic normally returns NULL

Aggregate Functions & Duplicates

```
1 SELECT AVG(DISTINCT cost) as "Average Cost"  
2 FROM products  
3 WHERE category = 'Clothing';
```

- ▶ If there are two cost values of \$25, only one of these values would be used during processing of AVG
 - ▶ You had better know the properties of your data-set before doing this, but this is probably not the semantics you have in mind
 - ▶ Default modifier is ALL: don't remove duplicates
 - ▶ Better to make the semantics explicit

Aggregation, DISTINCT, & NULLs

```
SELECT Count(price),  
       Count(DISTINCT price)  
FROM Candy;
```

Candy

<u>candy_name</u>	price
Reese's Cup	\$0.50
5 th Avenue	NULL
Almond Joy	\$0.75
Jelly Babies	\$2.39
Chocolate Orange	\$2.39
Jelly Nougats	NULL



4	3

Counting rows

Candy

<u>candy_name</u>	price
Reese's Cup	\$0.50
5 th Avenue	NULL
Almond Joy	\$0.75
Jelly Babies	\$2.39
Chocolate Orange	\$2.39
Jelly Nougats	NULL

```
SELECT Count(*)  
FROM Candy;
```

The only aggregate function that uses this *.



6

GROUP BY & Aggregation

- ▶ “Grouping” allows you to create “sub-groups” of tuples before summarizing
 - ▶ Result: the aggregate function is applied to each sub-group independently
- ▶ Think of each “group” as comprising a distinct “sub-relation” within the input relation

If you could execute this snippet of SQL

```
1      from instructor
2      group by dept_name
```

the result looks like

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

The power of GROUP BY manifests when prefixed by a SELECT clause (next slide)

Invoking SELECT On Results of GROUP BY

- ▶ Once the tuples are grouped, you can perform a SELECT query that is applied to the “grouped” sub-relations
- ▶ Any non-aggregated attribute in the SELECT CLAUSE **must be present** in the GROUP BY clause
 - ▶ Otherwise: which attribute value gets to represent the group?
- ▶ The following SQL is **invalid**: each instructor has a different *ID*, no single value can represent each group

```
1  select dept_name, ID, avg (salary)
2  from instructor
3  group by dept_name;
```

GROUP BY: An Example

Find the average instructor salary in each department

```
select dept_name, avg(salary) as  
       avg_salary  
from instructor  
group by dept_name
```

dept_name	salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

GROUP BY & NULL: Weirdness Alert

- ▶ If the “grouping attribute” contains NULL instances, SQL creates a **separate group** containing all tuples with a value for that attribute
- ▶ **Q:** why is that “weird”?
- ▶ **A:** because previous lectures explained that NULL semantics imply that “NULL ! = NULL” 😊
- ▶ The SQL designers know how we want GROUP BY to behave so they explicitly carved out an exception

Because SQL:2003 defines all Null markers as being unequal to one another, a special definition was required in order to group Nulls together when performing certain operations. SQL defines “any two values that are equal to one another, or any two Nulls”, as “not distinct”. This definition of not distinct allows SQL to group and sort Nulls when the GROUP BY clause (and other keywords that perform grouping) are used.

Source

GROUP BY & HAVING

- ▶ Think of the **HAVING clause** as a “WHERE clause for groups created by GROUP BY”
 - ▶ In other words: HAVING provides a condition to select or reject an entire group
- ▶ Despite the previous analogy, WHERE clause and HAVING clause **are not substitutable** for one another!
 - ▶ Selection conditions in a WHERE clause **constrain tuples selection**
 - ▶ Selection conditions in a HAVING clause **constrain group selection**
- ▶ Example: *For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project*

```
1 SELECT Pnumber, Pname, COUNT (*)
2 FROM project, works_on
3 WHERE Pnumber = Pno
4 GROUP BY Pnumber, Pname
5 HAVING COUNT (*) > 2;
```

Review – Can compute column totals

```
SELECT Sum(price * quantity) AS total  
FROM Purchase  
WHERE product = 'bagel';
```

Purchase

product	date	price	quantity
bagel	2017-10-21	\$1.00	20
banana	2017-10-03	\$0.50	10
banana	2017-10-10	\$1.00	10
bagel	2017-10-25	\$1.50	20



total
\$50.00

How to compute column *subtotals*?

```
SELECT product, Sum(price * quantity) AS subtotal  
FROM Purchase  
GROUP BY product;
```

Purchase

product	p_date	price	quantity
bagel	2017-10-21	\$1.00	20
banana	2017-10-03	\$0.50	10
banana	2017-10-10	\$1.00	10
bagel	2017-10-25	\$1.50	20



product	p_date	price	quantity
bagel	2017-10-21	\$1.00	20
	2017-10-25	\$1.50	20
banana	2017-10-03	\$0.50	10
	2017-10-10	\$1.00	10



product	subtotal
bagel	\$50.00
banana	\$15.00

Grouping + Aggregation

```
SELECT product,  
       Count(*) AS purchases,  
       Sum(quantity) AS items,  
       Sum(price * quantity) AS subtotal  
FROM Purchase  
GROUP BY product;
```

Can only SELECT
fields that you
GROUP BY.

Purchase

product	p_date	price	quantity
bagel	2017-10-21	\$1.00	20
banana	2017-10-03	\$0.50	10
banana	2017-10-10	\$1.00	10
bagel	2017-10-25	\$1.50	20



product	purchases	items	subtotal
bagel	2	40	\$50.00
banana	2	20	\$15.00

HAVING – Conditions on aggregates

```
SELECT product, Sum(price * quantity) AS subtotal  
FROM Purchase  
WHERE p_date > '2017-10-05'  
GROUP BY product  
HAVING Sum(quantity) >= 10;
```

Condition on individual rows.

Condition on aggregates.

Purchase

product	p_date	price	quantity
bagel	2017-10-21	\$1.00	20
banana	2017-10-03	\$0.50	10
banana	2017-10-10	\$1.00	10
bagel	2017-10-25	\$1.50	20
apple	2017-10-10	\$1.00	5



product	p_date	price	quantity
bagel	2017-10-21	\$1.00	20
	2017-10-25	\$1.50	20
banana	2017-10-10	\$1.00	10
apple	2017-10-10	\$1.00	5



product	subtotal
bagel	\$50.00
banana	\$10.00

Grouping without aggregation (uncommon)

```
SELECT product  
FROM Purchase  
GROUP BY product;
```



```
SELECT DISTINCT product  
FROM Purchase;
```

Purchase

product	p_date	price	quantity
bagel	2017-10-21	\$1.00	20
banana	2017-10-03	\$0.50	10
banana	2017-10-10	\$1.00	10
bagel	2017-10-25	\$1.50	20



product	p_date	price	Quantity
bagel	2017-10-21	\$1.00	20
	2017-10-25	\$1.50	20
banana	2017-10-03	\$0.50	10
	2017-10-10	\$1.00	10



product
bagel
banana

HAVING without GROUP BY (uncommon)

```
SELECT value, Count(*) AS count  
FROM Data  
HAVING count > 1;
```

Without GROUP BY,
the entire results
form one group.

Data

<u>item</u>	tag	value
1	a	2
2	a	2
3	b	1
4	b	4
5	c	4



value	count
2	5
2	5
1	5
4	5
4	5

Nested Sub-Queries

Introduction (I)

- ▶ Some queries require that tuples be retrieved so they be used as a “comparison condition” in a larger condition
- ▶ The ability to construct nested queries is a major convenience!
 - ▶ Definition: a complete `SELECT/FROM/WHERE` block within another SQL query
- ▶ Nested queries can appear in the outer query’s `SELECT`, `FROM`, or `WHERE` clause
- ▶ This generality is yet another testimonial to the power of set theory 😊
 - ▶ Queries operate on relations and return relations, so we can “plug-in” a nested query in any clause that produces a relation

- ▶ Two types of nested queries:
 - ▶ **Independent nested queries:** the nested query is executed independently of the outer query, and its result is used to execute the outer query
 - ▶ A new set of operators will allow us to test how the outer query's relation "relates" to the inner query's result
 - ▶ **Correlated nested queries:** the result of the inner query depends on the tuple currently being evaluated in the outer query

Let's Look At An Example (I)

- ▶ Consider this query: *Find the total number of employees whose salaries exceed \$40,000 in each department, but only for departments where more than five employees work*
- ▶ Q: will this SQL do the job?

```
1  SELECT Dno, COUNT (*)
2     FROM EMPLOYEE
3     WHERE Salary > 40000
4     GROUP BY Dno
5     HAVING COUNT (*) > 5;
```

- ▶ A: no, this SQL doesn't implement the desired semantics!
- ▶ The SQL will select only departments that have more than five employees all of whom earn more than \$40,000
 - ▶ The “natural language” query is ok with departments that have employees that earn less than \$40,000
 - ▶ All that's required is that those departments have five employees whose salary meets the threshold

Let's Look At An Example (II)

Find the total number of employees whose salaries exceed \$40,000 in each department, but only for departments where more than five employees work

ORDER	CLAUSE	FUNCTION
1	from	Choose and join tables to get base data.
2	where	Filters the base data.
3	group by	Aggregates the base data.
4	having	Filters the aggregated data.
5	select	Returns the final data.
6	order by	Sorts the final data.
7	limit	Limits the returned data to a row count.

```
1 SELECT Dno, COUNT (*)
2 FROM EMPLOYEE
3 WHERE Salary > 40000
4 GROUP BY Dno
5 HAVING COUNT (*) > 5;
```

- Per the figure on the left, the SQL rules for logical execution order imply that the WHERE clause is executed before the HAVING clause

Result: tuples have already been restricted to employees who earn more than \$40,000 before the HAVING constraint is applied 😞

Let's Look At An Example (III)

Find the total number of employees whose salaries exceed \$40,000 in each department, but only for departments where more than five employees work

- ▶ We can use a nested query (because it's a self-contained unit of execution) to implement the query properly

```
1 SELECT Dno, COUNT (*)
2 FROM EMPLOYEE
3 WHERE Salary > 40000 AND Dno IN
4 (SELECT Dno
5  FROM EMPLOYEE
6  GROUP BY Dno
7  HAVING COUNT (*) > 5)
8 GROUP BY Dno;
```

Segue to SQL Set Operations

- ▶ A common use of nested sub-queries in WHERE clauses is to perform tests for
 - ▶ set membership
 - ▶ set comparisons
 - ▶ set cardinality
- ▶ We have a new comparison operator: **IN**
 - ▶ Compares a (scalar or single tuple) value v with a set of values V (scalars or tuples)
 - ▶ Evaluates to TRUE *iff* $v \in V$
- ▶ For symmetry, another comparison operation: **NOT IN**

Set Operations

- ▶ If a nested query returns a **single attribute** and a **single tuple**, the query result is a single scalar value
 - ▶ SQL allows you to use **=** instead of **IN** when doing the comparison
- ▶ SQL allows the use of tuples of values in comparisons by placing them within parentheses.

```
1  SELECT DISTINCT Essn
2  FROM WORKS_ON
3  WHERE (Pno, Hours) IN
4         (SELECT Pno, Hours
5          FROM WORKS_ON
6          WHERE Essn = '123456789');
```

- ▶ A previous lecture showed how the **UNION**, **INTERSECT**, and **EXCEPT** operators allow us to formulate “set membership” queries
- ▶ We can get equivalent effect by using **nested queries** within a **WHERE** clause ...
 - ▶ In combination with our new **IN** and **NOT IN** operators

Set Intersection Semantics

“Find courses that ran in Fall 2009 and in Spring 2010”

Compare (set intersection approach):

```
1      (select course_id from section
2      where sem = 'Fall' and year = 2009)
3      intersect
4      (select course_id from section
5      where sem = 'Spring' and year = 2010)
```

To (nested sub-query approach):

```
1  select distinct course_id
2  from section
3  where semester = 'Fall' and year=2009 and
4  course_id in (select course_id from section
5  where semester = 'Spring' and year= 2010)
```

*“Find courses that ran in Fall 2009 **but not** in Spring 2010”*

Compare (set difference approach):

```
1      (select course_id from section
2      where sem = 'Fall' and year = 2009)
3      except
4      (select course_id from section
5      where sem = 'Spring' and year = 2010)
```

To (nested sub-query approach):

```
1  select distinct course_id
2  from section
3  where semester = 'Fall' and year= 2009 and
4  course_id not in (select course_id
5  from section
6  where semester = 'Spring' and year=2010)
```

Set Comparison Operators

- ▶ If you need to compare a **single value v** to a **set of values V** , SQL provides two new operators
 - ▶ SOME returns TRUE *iff* v is equal to **some value in the set V**
 - ▶ Implication: a comparison of = SOME is equivalent to IN
 - ▶ Important: SOME and ANY are **synonyms**!
 - ▶ ALL returns TRUE *iff* v is equal to **all values in the set V**

You can combine any of the standard relational operators with ALL or SOME: $>$, \geq , $<$, \leq , \neq

Set Comparison: SOME Example (I)

Find names of instructors with salary greater than that of some (“at least one”) instructor in the Biology department
Compare:

```
1 select distinct T.name
2 from instructor as T, instructor as S
3 where T.salary > S.salary and S.dept name = 'Biology';
```

The above query “works”, you may find this version more intuitive:

```
1 select name
2 from instructor
3 where salary > some
4 (select salary from instructor where dept name = 'Biology')
```


Set Comparison: SOME Example (II)

```
1 (select salary from instructor where dept name = 'Biology')
```

This sub-query returns a relation of “all instructor salaries in the Biology department”

By using *relop* SOME in the **outer** query

- ▶ **relop** (relational operator) is one of $<, \leq, >, \geq, =, \neq$

```
1 from instructor
2 where salary > some
3 (select salary from instructor where dept name = 'Biology')
```

We're asking: is there any tuple in the relation on the right

- ▶ Meaning: “*the one returned by the sub-query*”

for which the relational comparison between the relation on the left and the relation on the right is true?

- ▶ Equivalently: is there “some” tuple that meets the stated condition ...
- ▶ Note: in this example, “tuple” is a “scalar value”
- ▶ Per earlier slides, both types of comparisons are valid as long as the right-hand and left-hand parts of the comparison are the same “type”

Set Comparison: ALL Example

*Find names of instructors with salary greater than the salaries of **all the instructors** in the Biology department*

```
1 select name
2 from instructor
3 where salary > all (select salary
4 from instructor
5 where dept name = 'Biology')
```

The difference between SOME and ALL operators is exactly the same difference between \exists and \forall

We're Not Finished With "Nested Queries"

- ▶ I don't want to rush the presentation of this important topic
- ▶ But: we're out of time 😞
- ▶ Plan: continue the discussion next lecture

Now: quick review (different style) of the material thus far

...

Subqueries

Subquery – Used as an aggregated value

Find all the passing students (average score ≥ 60).

```
SELECT first_name, last_name  
FROM Student  
WHERE 60 <= (SELECT Avg(score)  
             FROM Assignment  
             WHERE Student.s_id = Assignment.s_id);
```

Subquery = inner query

*Correlated subquery,
since it depends on
outer query.*

Subquery – Used like a table

```
SELECT Sale.region_sales,  
       Quota.region_quota  
FROM (SELECT region_id,  
             SUM(total_due) AS region_sales  
      FROM Order  
      GROUP BY region_id) AS Sale  
INNER JOIN (SELECT region_id,  
                  SUM(quota_amount) AS region_quota  
           FROM Salesperson  
           GROUP BY region_id) AS Quota  
ON Sale.region_id = Quota.region_id;
```

Subqueries often
need aliases.

Compare **sales per region** to **sales quota per region**.

A version of aliasing useful with subqueries

```
SELECT Sale.region_sales,  
       Quota.region_quota  
FROM (SELECT order_region_id,  
             SUM(total_due)  
      FROM Order  
      GROUP BY region_id) AS Sale(region_id, region_sales)  
INNER JOIN (SELECT sales_region_id,  
                  SUM(quota_amount)  
           FROM Salesperson  
           GROUP BY region_id) AS Quota(region_id, region_quota)  
ON Sale.region_id = Quota.region_id;
```

Syntax summary

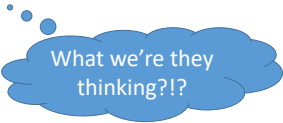
```
SELECT [DISTINCT] Computations
FROM   Table0
      INNER JOIN Table1 ON JoinCond1
      ...
WHERE   FilterCond
GROUP BY Attributes
HAVING  AggregateFilterCond
ORDER BY Computations
LIMIT  n;
```

Uses attributes or aggregates.

Semantics summary

```
SELECT [DISTINCT] Computations  
  
FROM      Table0  
      INNER JOIN Table1 ON JoinCond1  
      ...  
WHERE     FilterCond  
GROUP BY  Attributes  
HAVING    AggregateFilterCond  
ORDER BY  Computations  
LIMIT     n;
```

4. Computations, [eliminate duplicates]
8. Projections
1. Joins
2. Filter rows
3. Group by attributes
5. Filter groups
6. Sort
7. Use first n rows



What we're they thinking?!?

Bonus Material: NULL

Returning Briefly To The Topic of NULL

- ▶ Yes, you **must understand** three-valued logic and its implications
- ▶ Yes, you must be aware of how NULL can affect comparison, arithmetic, and aggregate function semantics
- ▶ All of these criticisms boil down to observing that
 - ▶ NULL is not a value, yet relational predicates must be applied to tuples with values
 - ▶ Because a predicate is truth statement about a set of values
 - ▶ Hence papers such as *SQL's Nulls Are A Disaster* and *The Final Null In The Coffin*

Q: Are so many people stupid? Is so much money invested in a technology that's broken?

What Alternative Do You Propose To NULL?

- ▶ (So far), solutions amount to
 - ▶ Figure out why you're using NULL
 - ▶ Eliminate those cases, if necessary by making many more relations to handle the different reasons why your tuple contains NULL
 - ▶ See Darwen on **vertical** and **horizontal** decomposition
- ▶ Problem: query execution now involves joins on many, many tables
 - ▶ Complexity → lack of clarity
 - ▶ Complexity → bad performance
 - ▶ Also: many more integrity constraints
- ▶ So in the commercial world
 - ▶ People shrug their shoulders
 - ▶ Try to avoid **nullable columns** as much as possible
 - ▶ And “live with it” ☺

Today's Lecture: Wrapping it Up

Aggregation

Nested Sub-Queries

Bonus Material: NULL

Readings

- ▶ Textbook discusses **aggregate functions** in Chapter 3.7
- ▶ Textbook discussed **nested queries** in Chapter 3.8