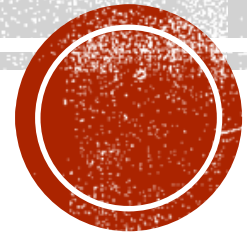


IT125 SQL: SUMMARY QUERIES & SUBQUERIES

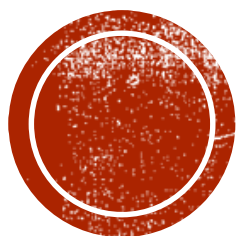
Bill Barry



TONIGHT

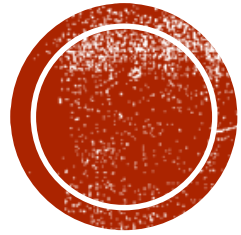
- Review a few **Proj05** issues
- Discuss **generating data** for Proj08
- Learn to write **summary queries** that do common math operations like count, sum, average, min, and max
 - Use these to return table-wide stats or apply them to grouped data
 - Add the HAVING clause to filter on grouped data
- Learn how to use **subqueries** in various SELECT clauses





PROJ05 ISSUES





PROJ08: GENERATING DATA



SAMPLE DATA FOR PROJ08

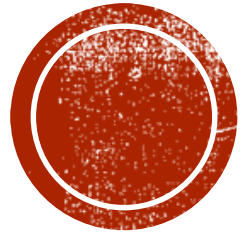
We want enough realistic data for a proof-of-concept demo. Our first task is to populate the more **independent tables**. Where do we get data?

- Find *real* data (internet?)
- Generate *mock* data, e.g., <https://www.mockaroo.com/>
- Make up *fake* data, or add fake data in certain columns
 - If you're handy with Excel, a random number plus IF or VLOOKUP, perhaps
- Munge data into usable form (parenthesized, comma-separated column values)
 - Excel formulas or a script (Python?)

Linking tables are harder

- For a small amount of demo data, make it up
- If bigger and/or fancier, some scripting might be required
 - Adding in some probabilities can make data feel more realistic





PART I: SUMMARY QUERIES



THE NEED FOR SUMMARY QUERIES

Queries Thus Far

- We've so far written queries that dump out raw data—perhaps joined, perhaps filtered, perhaps renamed—but raw row data nonetheless
- For example, we can easily answer questions like “How many customers live in Bellevue, WA?” or requests like “Get a list of customers who live in Kirkland, WA”

Wanting More

- What about “Get a count of customers in *each* WA city.” How many queries would you need to find that data? Yipes!
- Being a clever techie, you might paste into Excel and use formulas or Pivot Tables, but can't SQL be of more help?

The Bottom Line

- Raw data insufficient to handle many business requests; your manager will often want summaries to support or validate business decisions



WHAT SUMMARY QUERIES OFFER

- Summary Queries give us a way to apply common mathematical operations and show the *summarized results*
- These queries can not only show us the mathematical results on a *whole* table's data, but can also show us results on *groups* of data; this adds the GROUP BY clause to our SELECT toolbox
- On top of that, we can also filter based on those *grouped results*; we'll add the HAVING clause on top of the GROUP BY
- Order of clauses:
 - SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY

Applies to
individual rows

Applies to
grouped results

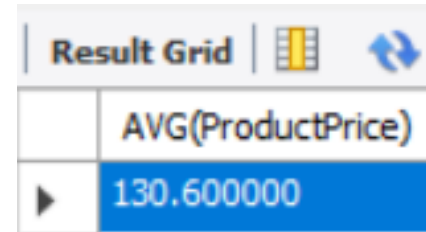
Applies to either individual or
grouped results



SAMPLE SUMMARY QUERY

- Using Cathy's Cakes, here's a typical summary query:

```
SELECT AVG(ProductPrice)
FROM Product;
```



The screenshot shows a 'Result Grid' window with a toolbar at the top containing a 'Result Grid' label, a grid icon, and a refresh icon. The grid has two columns: the first column is empty, and the second column is labeled 'AVG(ProductPrice)'. There is one row of data with the value '130.600000' in the second column. The row is highlighted with a blue background.

	AVG(ProductPrice)
▶	130.600000

- It's a **summary query** because it contains an aggregate function, AVG in this case
- As is typical with aggregate functions, this will return exactly **one cell of data** (never more, never less); this will be important later when we study subqueries



AGGREGATE FUNCTIONS

These are "column" functions; not scalar functions like we've seen before

- | | |
|---------------------|-----------------------------|
| ▪ AVG(expression) | average of non-null values |
| ▪ SUM(expression) | sum of non-null values |
| ▪ MIN(expression) | lowest non-null value |
| ▪ MAX(expression) | highest non-null value |
| ▪ COUNT(*) | number of rows in the query |
| ▪ COUNT(expression) | number of non-null values |

You'll use this version
of COUNT far less
frequently

Also available: STDDEV_POP(expression), STDDEV_SAMP(expression),
VAR_POP(expression), VAR_SAMP(expression)

Complete list: https://dev.mysql.com/doc/refman/5.7/en/group-by-functions.html#function_std





PRACTICE #1: SUMMARY OF TABLE DATA

- Aggregate function list: AVG, SUM, MIN, MAX, COUNT
- Set as the default database: Cathy's Cakes
- In one query, gather, from the **Product** table:
 - A count of the products in the table (call this "Count")
 - A total of all product prices (call this "Total") (doesn't really make sense, but it's practice)
 - The average product price (call this "Average") rounded to the nearest penny
 - The lowest product price in the table (call this "Lowest")
 - The highest product price in the table (call this "Highest")




GROUPING RESULT DATA

Instead of yielding one row of summary results for the whole table, it's common to want a list of *subtotals* instead. To do this, just add a GROUP BY after the FROM clause. Syntax:

- GROUP BY group_by_list [WITH ROLLUP]

This is usually a column of interest, e.g.,

- GROUP BY CustId // in Invoices table, the "many" side of a relationship
- GROUP BY AreaCode
- GROUP BY State, City
- The result set will have one row per group, e.g., one summary row for each Customer ID, Area Code, or State/City; all summary functions will be applied
- Include at WITH ROLLUP at the end of the SELECT line to request a grand total line
-  Note: don't include non-summary columns included in the SELECT clause unless they are used in the GROUP BY clause; they make no sense



GROUPING EXAMPLE (CATHY'S CAKES)

- No grouping: “How many cake orders are represented in ProductOrder?”

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

	COUNT(*)
▶	2047

- Grouping: “How many cake orders do we see within *each* ProductOrder?”

```
SELECT COUNT(*), CakeOrderId  
FROM ProductOrder  
GROUP BY CakeOrderId;
```

	COUNT(*)	CakeOrderId
▶	2	1001
	1	1002
	1	1003
	1	1004
	2	1005

Could also use
GROUP BY 2
(positions are allowed in this
clause, as in ORDER BY)

CakeOrderId makes
sense in SELECT;
other columns won't





PRACTICE #2: GROUPING

Here's a typical, reasonable request from Cathy (of Cathy's Cakes):

Find out how many customers live in each city

Show a grand total row at the bottom





FILTERING ANALOGY: CRUISE SHIP

On a cruise ship, there are two different kinds of “filters” you might experience:

- The first is the **security guard** who checks your id when you try to board the ship; if you don't have the right credentials, you aren't allowed to board
- The second is the **activity director** who might put you into groups based on what state you're from, then eliminate (or combine) groups smaller than 5 people

While these are both filtering activities, they happen at *different times*; the first says whether you're included at all, and the second happens later, *after you've been put into groups*

The first is an example of how **WHERE** works; it keeps non-matching rows from being part of the results to begin with; they aren't grouped, aren't totaled, etc. It works on *rows*

The second is an example of how **HAVING** works; it takes *grouped data* and applies a filter on the *results* of that grouping. It works on *grouped results*



FILTERING SUMMARY DATA

We've long known how to filter row data; we just use WHERE

But we may need to filter on *summary* (grouped) results. To accomplish that, we add a HAVING clause after the GROUP BY clause:

- HAVING condition
- It's legal to have both a WHERE and a HAVING clause; if so, put the WHERE in the usual spot but before GROUP BY and HAVING
 - ```
SELECT select_list
FROM table_name
WHERE row_filter_condition // selects which rows to include
GROUP BY group_by_list // creates summary grouping
HAVING summary_filter_condition // selects which summary data to include
ORDER BY order_by_list // sets sort order for results
```



# GROUP FILTERING EXAMPLE (CATHY'S CAKES)

- No filtering: “How many cake orders do we see within *each* ProductOrder?”

```
SELECT COUNT(*), CakeOrderId
FROM ProductOrder
GROUP BY CakeOrderId;
```

|   | COUNT(*) | CakeOrderId |
|---|----------|-------------|
| ▶ | 2        | 1001        |
|   | 1        | 1002        |
|   | 1        | 1003        |

- Group filtering: “...but show only Product Orders with more than 2 Cake Orders”

```
SELECT COUNT(*) AS `Cake Order Count`, CakeOrderId
FROM ProductOrder
GROUP BY CakeOrderId
HAVING `Cake Order Count` > 2;
```

|   | Cake Order Count | CakeOrderId |
|---|------------------|-------------|
| ▶ | 3                | 1026        |
|   | 3                | 1028        |
|   | 3                | 1037        |
|   | 3                | 1044        |
|   | 3                | 1064        |
|   | 2                | 1102        |

Could also write:  
HAVING COUNT(\*) > 2



# PRACTICE #3: FILTERING SUMMARY DATA



For Cathy's Cakes, get a list of the zip codes that have 7 or more customers in them, along with the number of customers in that zip code

Out of curiosity, use WITH ROLLUP and see why you might not want that here





# PRACTICE #4: SORTING SUMMARY DATA

Copy and paste the previous query and alter the copy

Sort so that the zip code with the highest number of customers is listed first





# PRACTICE #5: ALL TOGETHER NOW

- Now let's tie in what we've studied *before* with what we've learned *tonight*
  - For example, some joining of tables is required for more complex scenarios
- Get a list of Redmond customers who have ordered the most products
  - Note that this isn't the same as having placed *the most orders*; we're asking about products within those orders, for all time and all orders
- In the results list, show the customer id, customer first name, last name, and how many products they have ordered
- Show only the customers who have ordered over 5 products
- Sort by the number of product ordered, highest first



# SUMMARY QUERY HINTS AND TROUBLESHOOTING

- Don't try to do everything at once; get a basic query and it JOINS working first, then incrementally add other requirements
- It's only a summary query if you use an aggregate function in it
- Only summary queries should use GROUP BY or HAVING
- Only include in the SELECT statement aggregate functions or columns used in GROUP BY (or highly related ones); anything else yields puzzling data
- HAVING only makes sense if you're using GROUP BY
- Be clear on the purpose of WHERE vs. HAVING; both can participate
  - WHERE filters raw records, saying which ones even participate in the query
  - HAVING filters grouped results, saying which of the groups you want to see displayed



# OTHER USEFUL SUMMARY QUERY STUFF

- Use COUNT(DISTINCT *expr*) to find the number of different values, e.g., in OM:
  - SELECT COUNT(DISTINCT order\_id) FROM order\_details # result = 46
- Make a GROUP\_CONCAT column to return a string composed of all the different values in a column, e.g., in OM:
  - SELECT order\_id, GROUP\_CONCAT(item\_id) FROM Order\_Details GROUP BY order\_id

| order_id | group_concat(item_id) |
|----------|-----------------------|
| 606      | 8                     |
| 607      | 3,10                  |
| ...      |                       |
| 693      | 6,7,10                |
| ...      |                       |

Reference: <https://dev.mysql.com/doc/refman/5.7/en/group-by-functions.html>



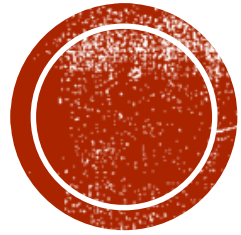


# MORE ON WITH ROLLUP

- If WITH ROLLUP is specified in the GROUP BY, you'll also get totaling of results
- If you group by a single column, the ROLLUP data is obvious; you get a total row at the bottom
- If you group by *multiple* columns, you get a total line every time the penultimate grouped column changes, another when the next major column changes, etc., through the most major grouped column
- WITH ROLLUP can't be used with ORDER BY; both do sorting
- See this site for a complete description:  
<https://dev.mysql.com/doc/refman/5.7/en/group-by-modifiers.html>

```
mysql> SELECT year, country, product, SUM(profit) AS profit
 FROM sales
 GROUP BY year ASC, country ASC, product ASC WITH ROLLUP;
```

| year | country | product    | profit |
|------|---------|------------|--------|
| 2000 | Finland | Computer   | 1500   |
| 2000 | Finland | Phone      | 100    |
| 2000 | Finland | NULL       | 1600   |
| 2000 | India   | Calculator | 150    |
| 2000 | India   | Computer   | 1200   |
| 2000 | India   | NULL       | 1350   |
| 2000 | USA     | Calculator | 75     |
| 2000 | USA     | Computer   | 1500   |
| 2000 | USA     | NULL       | 1575   |
| 2000 | NULL    | NULL       | 4525   |
| 2001 | Finland | Phone      | 10     |
| 2001 | Finland | NULL       | 10     |
| 2001 | USA     | Calculator | 50     |
| 2001 | USA     | Computer   | 2700   |
| 2001 | USA     | TV         | 250    |
| 2001 | USA     | NULL       | 3000   |
| 2001 | NULL    | NULL       | 3010   |
| NULL | NULL    | NULL       | 7535   |



# PART II: SUBQUERIES



# SUBQUERIES: WHAT AND WHERE

- What is a subquery?
  - A **subquery** is a SELECT query coded inside of another SELECT query
  - Most (but not all) subqueries could alternatively be done via other methods (e.g., JOINS)
- Where can you use a subquery? Within a SELECT statement...
  - ...in the SELECT clause
  - ...in the FROM clause
  - ...in the WHERE clause
  - ...in the HAVING clause



# SUBQUERIES: SHOULD I USE THEM?

Subqueries, especially nested ones, can make your SELECTs hard to read

- **Don't** Use Them...
  - ...if you can easily express the subquery using more traditional logic (e.g., a JOIN)
- **Do** Use Them...
  - ...when there is no other way to get what you want
  - ...when the subquery seems like the most natural way to express the logic
  - ...if your instructor asks you to use them 😊

If you do use them, make them readable with indentation and comments



# BILL'S RECOMMENDED STEPS FOR BUILDING A SUBQUERY

Problem: from the OM DB's Items table, we want to a list of unit prices that are higher than the average unit price. The way I'll usually phrase it in projects: "Find the average unit price; use that to get a list of unit prices that are higher than that average"

1. Write and test the subquery

```
SELECT AVG(unit_price) FROM Items; # result = 16.52
```

Always returns a single value; use in a WHERE clause in place of a single value

2. Use the value that are returned and hardcode them into another query (this is the outer query)

```
SELECT title, artist, unit_price
FROM Items
WHERE unit_price > 16.52
ORDER BY unit_price DESC;
```

3. Replace the hardcoded number with parentheses and the *entire* inner query (no semicolon)

```
SELECT title, artist, unit_price
FROM Items
WHERE unit_price >
 (SELECT AVG(unit_price) FROM Items)
ORDER BY unit_price DESC;
```

This method is foolproof; it **must** work if you follow these steps!



# SUBQUERY VS. JOIN: WHICH IS MORE READABLE?

```
SELECT invoice_number,
 invoice_date,
 invoice_total

FROM invoices
 JOIN vendors USING vendor_id

WHERE vendor_state = 'CA'

ORDER BY invoice_date;
```

```
SELECT invoice_number,
 invoice_date,
 invoice_total

FROM invoices

WHERE vendor_id IN
 (SELECT vendor_id
 FROM vendors
 WHERE vendor_state = 'CA')

ORDER BY invoice_date;
```

Returns a list



# RETURNS FROM SUBQUERIES

Some subqueries return a **single value**

- Use these where you'd use a **literal or expression**
- Example: WHERE invoice\_total > (subquery)

Some subqueries return a **list** (single column) of values

- Use these where you'd use a **list or set**
- Example: WHERE customer\_state **IN** (subquery)

Some subqueries return a **table** (more than one column) of values

- Use these where you'd use a **table**
- Note: you *must* give the resulting subquery table an *alias* even if you don't use it in your code
- Example: SELECT cust\_name FROM Customer **JOIN** (subquery) *alias* ON ...

Never return more than absolutely required; it makes more work! Hint: you shouldn't return any tables from subqueries in your assignment





# PRACTICE #6: SUBQUERIES THAT RETURN A VALUE



We'll use Cathy's Cakes for this exercise

Find the average price for a product

Use that data to get a list of product names and prices for which the product price is greater than the average product price

Sort by product price, highest first

The wording here is a helpful hint for your assignment; it'll often say, "Do this, then use the result to do the next thing." The "do this" part is the subquery



# PRACTICE #7: SUBQUERIES THAT RETURN LISTS



Use Cathy's Cakes for this exercise

Cathy has a new recipe for a Cinnamon Lover's cake batter; she wants to first promote it to customers who are likely the most interested

Get a list of product id's for products that contain cinnamon as an ingredient

Use that information to get a list of customers who have ordered those products; show customer name and email address

Don't duplicate customer names

Sort by customer name



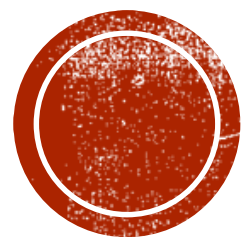
# READING

On your own, read starting on p. 208, tips for working with complex queries

We won't cover pp. 196-207, which includes...

- ALL and ANY keywords
  - <https://www.w3resource.com/mysql/subqueries/index.php#CR>
  - <https://dev.mysql.com/doc/refman/8.0/en/any-in-some-subqueries.html>
- Correlated subqueries
  - <https://dev.mysql.com/doc/refman/5.5/en/correlated-subqueries.html>
  - <http://www.geeksengine.com/database/subquery/correlated-subquery.php>
  - EXISTS operator
    - <https://dev.mysql.com/doc/refman/5.7/en/exists-and-not-exists-subqueries.html>
    - <http://www.geeksengine.com/database/subquery/exists.php>
- Subqueries in other clauses





# WRAP UP



# WHAT SHOULD I DO NEXT?

- Take this week's quiz
- Start Proj07
  - Summary queries and subqueries
- Before we meet again...
  - Submit Proj07
  - Read next week's material:
    - Chapter 12: Creating and using views
  - Keep working on Proj08
    - Presentations start in two weeks



**QUESTIONS?**

