IMEN891K / IDSC501 / AXGS-501 · DATA ENGINEERING · FALL 2026

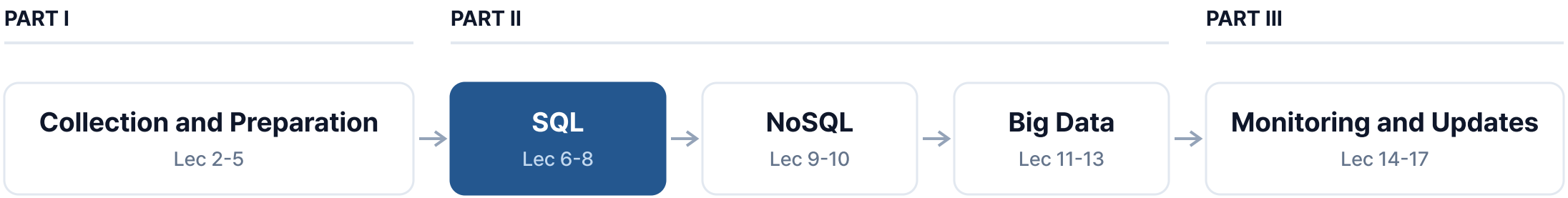
8.

SQL III: Aggregation

Sangwoo Mo · POSTECH

Industrial & Management Engineering

Where We Are


PART I
PART II
PART III
Collection and Preparation
Lec 2-5
SQL
Lec 6-8
NoSQL
Lec 9-10
Big Data
Lec 11-13
Monitoring and Updates
Lec 14-17

Outline: SQL III

01	**Calculate Measures**	calculate a quantity from detail rows
02	**Summarize Groups**	choose the key and keep one row per group
03	**Reuse Summaries**	use a computed result as another input
04	**Keep Detail Rows**	add group context without merging rows

01 · Calculate Measures

01	**Calculate Measures**	calculate a quantity from detail rows
02	**Summarize Groups**	choose the key and keep one row per group
03	**Reuse Summaries**	use a computed result as another input
04	**Keep Detail Rows**	add group context without merging rows

Analytical Result Shapes

Detail input

product	price
bagel	1.0
bagel	1.5
banana	0.5
banana	1.0

GROUP BY

product	mean
bagel	1.25
banana	0.75

Window

product	price	mean
bagel	1.0	1.25
bagel	1.5	1.25
banana	0.5	0.75
banana	1.0	0.75

Grouping reduces rows; a window adds context to each row

Case study: A Sales Report

product	date	price	quantity
bagel	2005-10-21	1.0	20
banana	2005-10-03	0.5	10
banana	2005-10-10	1.0	10
bagel	2005-10-25	1.5	20

Report total revenue, revenue per product and unusually expensive purchases

Each purchase records its product, date, unit price and quantity

Aggregate Functions

Function	Question
COUNT(*)	How many purchase rows?
SUM(quantity)	How many units?
SUM(price*quantity)	How much revenue?
AVG(price)	What is the average recorded unit price?
MIN(price), MAX(price)	What are the smallest and largest prices?

Choose the expression that measures the quantity named in the question

From Unit Price to Revenue

One purchase

price	quantity
1.5	20

→

Row calculation

1.5 × 20 = 30

→

Whole-table total

20 + 5 + 10 + 30 = 65

Calculate revenue per purchase before summing the rows

Revenue from Purchase Rows

```
SELECT COUNT(*) AS purchases,SUM(quantity) AS units,
  SUM(price*quantity) AS revenue FROM Purchase;
```

↑ revenue of each row, then summed

purchases	units	revenue
4	60	65.0

Four purchases contain 60 units and generate 65 in revenue

Caveat: Average Price per Purchase or Unit

```
SELECT AVG(price) AS mean_purchase_price,
  ↑ each purchase counts once
  SUM(price*quantity)/SUM(quantity) AS
mean_unit_price
    ↑ each unit counts once
FROM Purchase;
```

mean_purchase_price	mean_unit_price
1.0	1.083

Unequal purchase quantities make the two average prices differ

Caveat: Average of Averages

Batch	SUM	COUNT	AVG
A	10	1	10
B	180	9	20
A + B	190	10	19

```
SELECT AVG(1.0 * total / n) AS mean_of_means,
  ↑ each batch counts once
  1.0 * SUM(total) / SUM(n) AS combined_mean
    ↑ each item counts once
FROM batches;
```

mean_of_means	combined_mean
15.0	19.0

Unequal group sizes require SUM and COUNT; averaging the two means gives the wrong answer

Recap: Purchases for the Total

product	date	price	quantity
bagel	2005-10-21	1.0	20
banana	2005-10-03	0.5	10
banana	2005-10-10	1.0	10
bagel	2005-10-25	1.5	20

The two bagel purchases contribute 20 and 30 to revenue

Demo: Calculate a Sales Total

1. Return purchase count, total units and total revenue in one row
2. Calculate revenue for bagel purchases only

```
-- write your query here, then Run
```

QUERY RESULT

Write a query on the left, then Run.

Answer: Sales Totals

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

revenue
50.0

Filtering to bagel before SUM gives revenue of 50

COUNT and NULL

value
10.0
10.0
NULL

```
SELECT COUNT(*) AS rows, COUNT(value) AS known,
  ↑ COUNT(value) skips NULL
  COUNT(DISTINCT value) AS distinct_values,
  AVG(value) AS mean FROM readings;
```

rows	known	distinct_values	mean
3	2	1	10.0

Three rows contain two known values and one distinct value; NULL does not become zero

Empty Aggregate Inputs

```
SELECT COUNT(*) AS n, SUM(quantity) AS total,
AVG(quantity) AS mean
FROM Purchase WHERE quantity < 0;
```

↑ no row matches

n	total	mean
0	NULL	NULL

Empty input still yields one aggregate row: COUNT 0, SUM and AVG NULL

02 · Summarize Groups

01	Calculate Measures	calculate a quantity from detail rows
02	**Summarize Groups**	choose the key and keep one row per group
03	Reuse Summaries	use a computed result as another input
04	Keep Detail Rows	add group context without merging rows

One Row per Product

```
SELECT product,SUM(quantity) AS units,
  SUM(price*quantity) AS revenue
FROM Purchase GROUP BY product;
```

↑ one row per product

product	units	revenue
bagel	40	50.0
banana	20	15.0

GROUP BY turns four purchases into two product summaries

Grouping: Read the Input

1. Compute the **FROM** and **WHERE** clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '2005-10-01'
GROUP BY product
```

FROM →

Product	Date	Price	Quantity
bagel	2005-10-21	1	20
bagel	2005-10-25	1.5	20
banana	2005-10-03	0.5	10
banana	2005-10-10	1	10

Read the four purchases that pass the row condition

Grouping: Form the Groups

2. Group by the attributes in the **GROUP BY**

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '2005-10-01'
GROUP BY product
```

Product	Date	Price	Quantity
bagel	2005-10-21	1	20
bagel	2005-10-25	1.5	20
banana	2005-10-03	0.5	10
banana	2005-10-10	1	10

GROUP BY →

Product	Date	Price	Quantity
bagel	2005-10-21	1	20
	2005-10-25	1.5	20
banana	2005-10-03	0.5	10
	2005-10-10	1	10

Place purchases with the same product into the same group

Grouping: Calculate the Totals

3. Compute **SELECT**: keys and aggregates

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '2005-10-01'
GROUP BY product
```

Product	Date	Price	Quantity
bagel	2005-10-21	1	20
	2005-10-25	1.5	20
banana	2005-10-03	0.5	10
	2005-10-10	1	10

SELECT
→

Product	TotalSales
bagel	50.0
banana	15.0

Compute one revenue total for each product group

Choosing the Grouping Key

```
SELECT product,date,SUM(price*quantity) AS revenue
FROM Purchase GROUP BY product,date;
```

↑ one row per product and date

product	date	revenue
bagel	2005-10-21	20.0
bagel	2005-10-25	30.0
banana	2005-10-03	5.0
banana	2005-10-10	10.0

One row per product differs from one row per product and date

Adding date changes the report from product totals to product-by-date totals

A Name May Not Identify the Entity

```
WITH items(name,maker,units) AS (
  VALUES ('Plain','A',10),('Plain','B',20)
) SELECT name,SUM(units) AS units FROM items GROUP
BY name;
```

↑ two makers merge into one row

name	units
Plain	30

Two different products share a name and collapse into one group

Group by the Full Entity Key

```
WITH items(name,maker,units) AS (
  VALUES ('Plain','A',10),('Plain','B',20)
) SELECT name,maker,SUM(units) AS units
FROM items GROUP BY name,maker;
  ↑ the full key keeps them apart
```

name	maker	units
Plain	A	10
Plain	B	20

The complete product key preserves the two separate totals

WHERE and HAVING

WHERE price > 0.5

product	quantity
bagel	20
banana	10
bagel	20

→

GROUP BY product

product	units
bagel	40
banana	10

→

HAVING SUM(quantity) > 20

product	units
bagel	40

WHERE filters purchases; HAVING filters their completed groups

Compare Row and Group Filters

```
SELECT product,SUM(quantity) AS units
FROM Purchase WHERE price>0.5
  ↑ rows first
GROUP BY product HAVING SUM(quantity)>20;
  ↑ then groups
```

product	units
bagel	40

WHERE removes individual purchases; HAVING then removes a whole group

Recap: Products and Quantities

product	price	quantity
bagel	1.0	20
banana	0.5	10
banana	1.0	10
bagel	1.5	20

Bagel totals 40 units; banana totals 20 units

Demo: Build a Product Summary

1. Return units and revenue per product
2. Keep only product groups with more than 20 units sold

```
-- write your query here, then Run
```

QUERY RESULT

Write a query on the left, then Run.

Answer: Product Summary

```
SELECT product,SUM(quantity) AS units,
  SUM(price*quantity) AS revenue FROM Purchase
GROUP BY product HAVING SUM(quantity)>20;
```

product	units	revenue
bagel	40	50.0

HAVING keeps bagel after both product totals have been calculated

Caveat: Counting the Wrong Entity

Four matched pairs

cname	pname
GizmoWorks	Gizmo
GizmoWorks	PowerGizmo
Canon	SingleTouch
Hitachi	MultiTouch

Count pairs or companies?

pairs	companies
4	3

Four matching pairs contain only three distinct companies

Inspect the Rows Before Summing

```
WITH labels(product,label) AS (
  VALUES ('bagel','Baked'),('bagel','Breakfast'),
('banana','Fresh')
)
SELECT P.product,P.price*P.quantity AS
revenue,L.label
FROM Purchase P JOIN labels L ON
L.product=P.product;
```

product	revenue	label
bagel	20.0	Baked
bagel	30.0	Baked
bagel	20.0	Breakfast
bagel	30.0	Breakfast
banana	5.0	Fresh
banana	10.0	Fresh

A bagel purchase repeats once per label; its revenue repeats with it

Caveat: A Join Can Inflate Revenue

```
WITH labels(product,label) AS (
  VALUES ('bagel','Baked'),('bagel','Breakfast'),
('banana','Fresh')
)
SELECT SUM(P.price*P.quantity) AS repeated_revenue
  ↑ bagel rows counted twice
FROM Purchase P JOIN labels L ON
L.product=P.product;
```

repeated_revenue
115.0

Two bagel labels double bagel revenue and inflate the total to 115

Keep One Match per Product Key

```
WITH labels(product,label) AS (
  VALUES ('bagel','Baked'),('bagel','Breakfast'),
('banana','Fresh')
)
, products AS (SELECT DISTINCT product FROM labels)
  ↑ one row per product before the join
SELECT SUM(P.price*P.quantity) AS revenue
FROM Purchase P JOIN products L ON
L.product=P.product;
```

revenue
65.0

One matching label key per product restores the original revenue of 65

03 · Reuse Summaries

01	Calculate Measures	calculate a quantity from detail rows
02	Summarize Groups	choose the key and keep one row per group
03	**Reuse Summaries**	use a computed result as another input
04	Keep Detail Rows	add group context without merging rows

Named Product Summaries

```
WITH totals AS (
  ↑ a name for the per-product summary
  SELECT product,SUM(price*quantity) AS revenue
  FROM Purchase GROUP BY product
)
SELECT * FROM totals;
```

product	revenue
bagel	50.0
banana	15.0

The named totals table has one row per product

Aggregate a Summary

```sql
WITH totals AS (
  SELECT product,SUM(price*quantity) AS revenue
  FROM Purchase GROUP BY product
)
SELECT AVG(revenue) AS mean_product_revenue FROM
totals;
```

↑ average of the product totals

mean_product_revenue
32.5

The outer AVG reads the two product totals, 50 and 15

Scalar Aggregate Queries

```
WITH totals AS (
  SELECT product,SUM(price*quantity) AS revenue
  FROM Purchase GROUP BY product
)
SELECT product,revenue FROM totals
WHERE revenue>(SELECT AVG(revenue) FROM totals);
```

↑ compare with one number

product	revenue
bagel	50.0

The scalar average is 32.5; only bagel revenue exceeds it

Recap: Two Summary Rows

product	revenue
bagel	50.0
banana	15.0

The next query reads two product summaries rather than four purchase rows

Demo: Compare Product Summaries

1. Use a named product-revenue summary to calculate mean product revenue
2. Return products whose revenue exceeds that mean

```
-- write your query here, then Run
```

QUERY RESULT

Write a query on the left, then Run.

Answer: Above-Mean Product Revenue

```sql
WITH totals AS (
  SELECT product,SUM(price*quantity) AS revenue
  FROM Purchase GROUP BY product
)
SELECT product,revenue FROM totals
WHERE revenue>(SELECT AVG(revenue) FROM totals);
```

product	revenue
bagel	50.0

Four purchases become two totals; the final comparison keeps bagel

04 · Keep Detail Rows

01	Calculate Measures	calculate a quantity from detail rows
02	Summarize Groups	choose the key and keep one row per group
03	Reuse Summaries	use a computed result as another input
04	**Keep Detail Rows**	add group context without merging rows

Detail and Group Context

The four purchase rows

product	price
bagel	1.0
banana	0.5
banana	1.0
bagel	1.5

One row per product

product	mean_price
bagel	1.25
banana	0.75

Which **original purchases** cost more than their product average?

Grouping gives the means but removes the individual purchases needed for comparison

Keeping the Purchase Rows

Group summary

product	mean_price
bagel	1.25
banana	0.75

Detail with context

product	price	mean_price
bagel	1.0	1.25
bagel	1.5	1.25
banana	0.5	0.75
banana	1.0	0.75

Each purchase remains available for comparison with its product mean

Window Functions

```
SELECT product, date, price,
  AVG(price) OVER (PARTITION BY product) AS
product_mean
```

↑ rows stay; each gets its group mean

```
FROM Purchase ORDER BY product,date;
```

product	date	price	product_mean
bagel	2005-10-21	1.0	1.25
bagel	2005-10-25	1.5	1.25
banana	2005-10-03	0.5	0.75
banana	2005-10-10	1.0	0.75

Four purchase rows remain; each row receives the average of its own product partition

Ranking Within Groups

```
SELECT product,date,price,
  ROW_NUMBER() OVER (PARTITION BY product ORDER BY
price DESC,date) AS position
     ↑ rank inside each product
FROM Purchase ORDER BY product,position;
```

product	date	price	position
bagel	2005-10-25	1.5	1
bagel	2005-10-21	1.0	2
banana	2005-10-10	1.0	1
banana	2005-10-03	0.5	2

Partitioning restarts the row number for each product; ordering defines which purchase comes first

Filter a Window Result

```
WITH ranked AS (
  SELECT product,date,price,ROW_NUMBER() OVER (
    PARTITION BY product ORDER BY price DESC,date)
AS position
  FROM Purchase
)
SELECT product,date,price FROM ranked WHERE
position=1;
```

↑ top row of each product

product	date	price
bagel	2005-10-25	1.5
banana	2005-10-10	1.0

Rank the original purchases first, then keep position 1 for each product

Analytical Queries Have Stages

Stage	Result
FROM / JOIN → WHERE	Selected detail rows
GROUP BY → HAVING	Selected group summaries
Window expressions	Context on the remaining rows
ORDER BY → LIMIT	Final display order and selected rows

A later calculation needs another query block when an earlier filter must read its result

Recap: Purchases for Window Calculations

product	date	price	quantity
bagel	2005-10-21	1.0	20
bagel	2005-10-25	1.5	20
banana	2005-10-03	0.5	10
banana	2005-10-10	1.0	10

Window calculations keep each purchase available for comparison or ranking

Demo: Keep Purchases and Add Context

1. Add each product’s average price to every original purchase row
2. Return the highest-priced purchase per product using a row number and a named query

```
-- write your query here, then Run
```

QUERY RESULT

Write a query on the left, then Run.

Answer: Product Context on Purchases

```
SELECT product,date,price,
  AVG(price) OVER (PARTITION BY product) AS
product_mean
FROM Purchase ORDER BY product,date;
```

product	date	price	product_mean
bagel	2005-10-21	1.0	1.25
bagel	2005-10-25	1.5	1.25
banana	2005-10-03	0.5	0.75
banana	2005-10-10	1.0	0.75

The four original purchases receive their product averages without being grouped away

Answer: Highest-Priced Purchase

```
WITH ranked AS (
  SELECT product,date,price,ROW_NUMBER() OVER (
    PARTITION BY product ORDER BY price DESC,date)
AS position
  FROM Purchase
)
SELECT product,date,price FROM ranked WHERE
position=1;
```

product	date	price
bagel	2005-10-25	1.5
banana	2005-10-10	1.0

The final rows retain the dates of the highest-priced bagel and banana purchases

Takeaway: SQL III

1. A measure needs an explicit **entity, unit and denominator**
2. Grouping changes the **result grain**; repeated join matches can inflate totals
3. A summary can feed another query; a window adds context while **keeping detail rows**