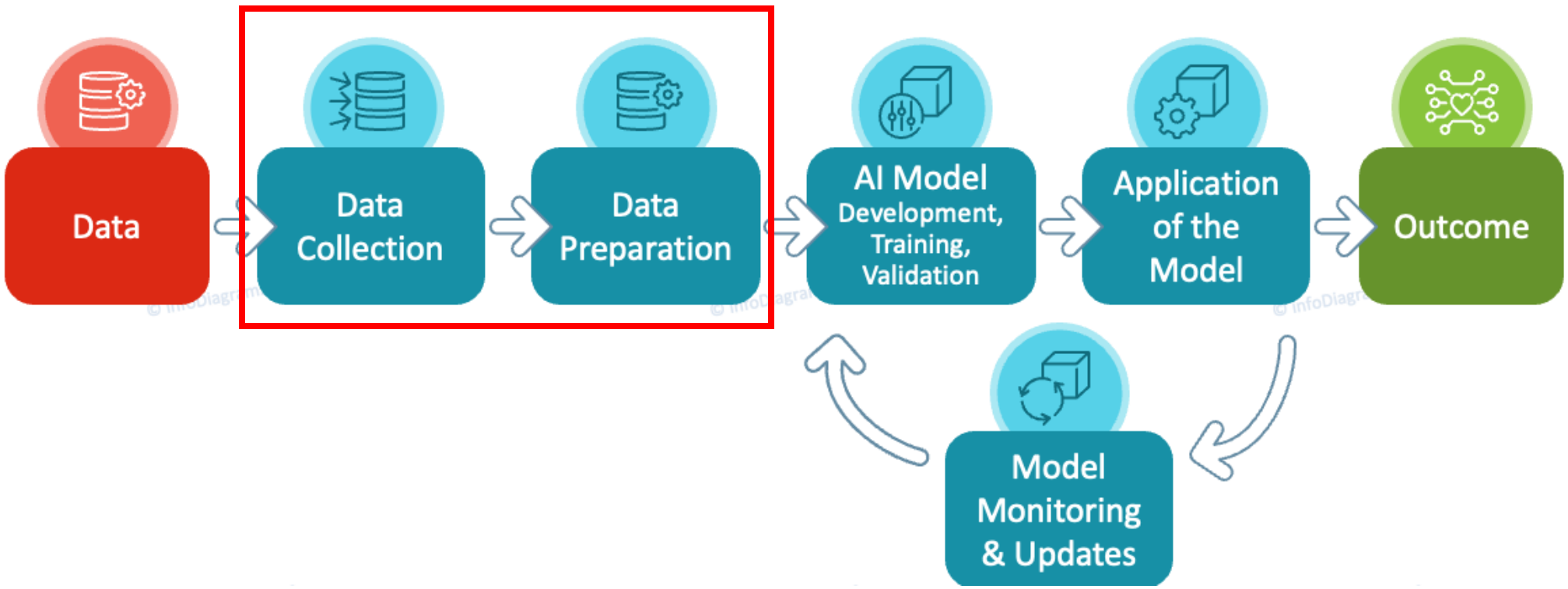
IMEN891K/IDSC501 Data Engineering

9. **SQL III**: Subqueries & Aggregations

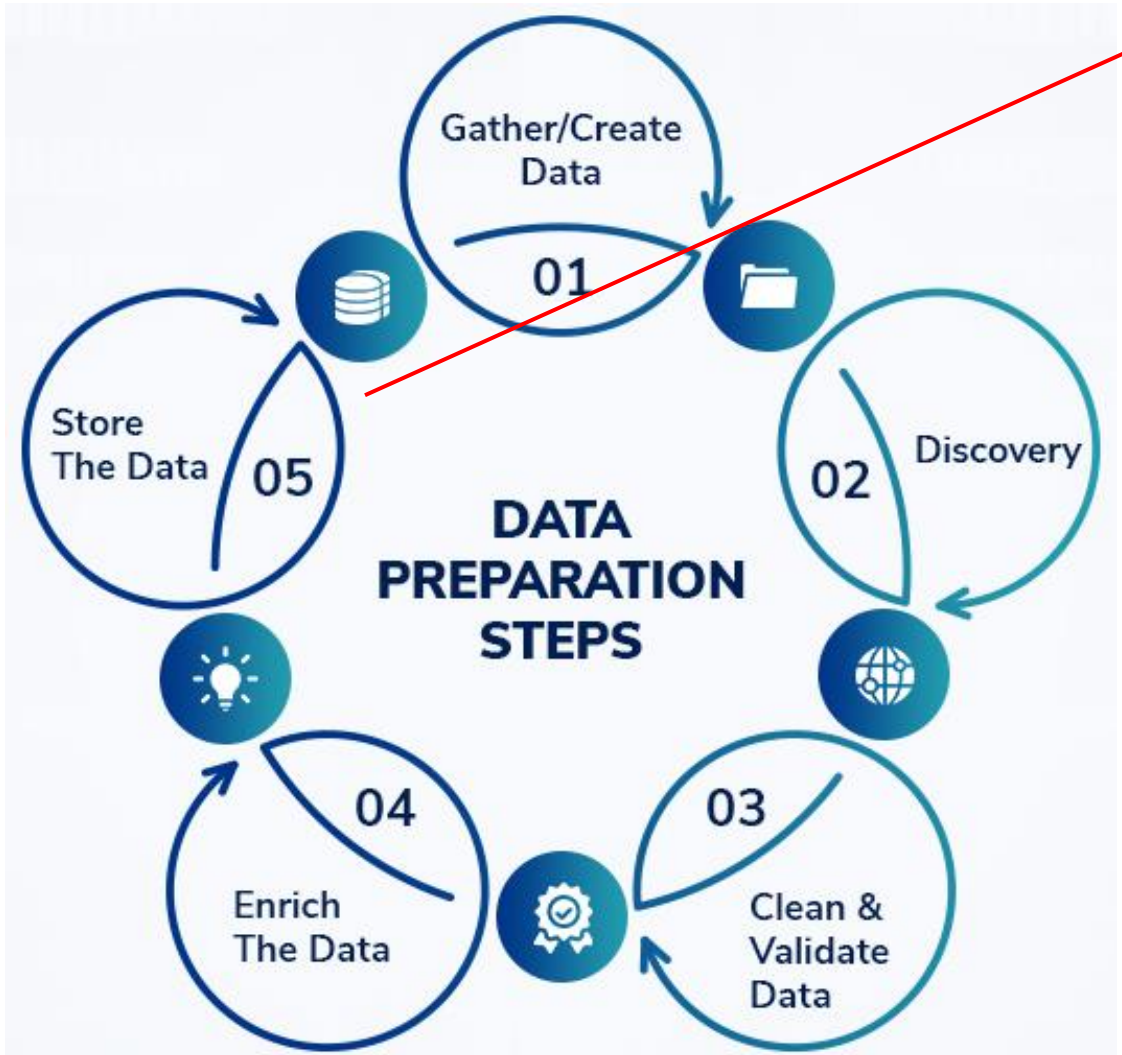
Fall 2025

Instructor: **Sangwoo Mo**

Recap: Course Overview



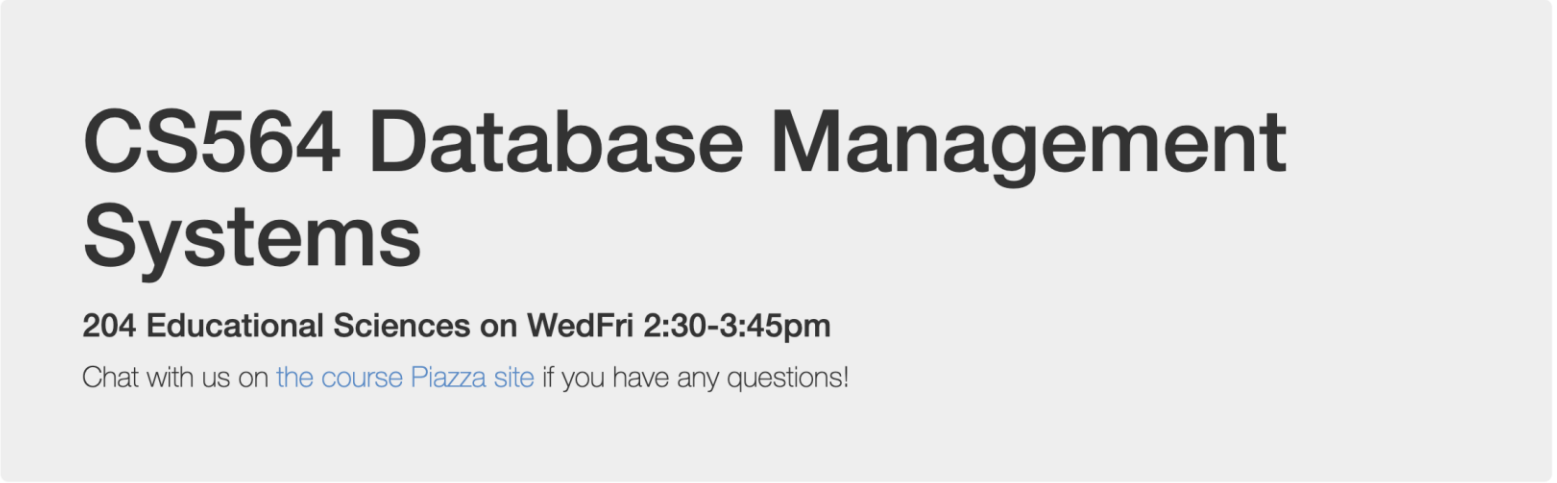
Course Overview



Lecture 7-15

Disclaimer

The SQL part was heavily copy-pasted from Wisconsin CS564



Recap: SQL Query

- Basic form (there are many many more bells and whistles)

```
SELECT <attributes>
FROM   <one or more relations>
WHERE  <conditions>
```

Call this a SFW query.

Recap: Declaring Keys

```
Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)

CREATE TABLE Enrolled(
      student_id CHAR(20),
      cid        CHAR(20),
      grade      CHAR(10),
      PRIMARY KEY (student_id, cid),

)
```

Recap: Declaring Foreign Keys

```
Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)

CREATE TABLE Enrolled(
      student_id CHAR(20),
      cid        CHAR(20),
      grade      CHAR(10),
      PRIMARY KEY (student_id, cid),
      FOREIGN KEY (student_id) REFERENCES Students(sid)
)
```

Recap: Declaring Foreign Keys

```
CREATE TABLE Company(
        cname           VARCHAR(100),
        stockprice      FLOAT,
        country                 VARCHAR(100),
        PRIMARY KEY (cname),
        FOREIGN KEY (cname) REFERENCES Product(pname, manufacturer)
)

CREATE TABLE Product(
        pname           VARCHAR(100),
        price           FLOAT,
        category                VARCHAR(100),
        manufacturer VARCHAR(100),
        PRIMARY KEY (pname, manufacturer)
)
```

If the **primary key** is a set of columns (a **composite key**), then the **foreign key** also must be a set of columns that corresponds to the **composite key**.

Recap: Declaring Foreign Keys

```
CREATE TABLE Company(
        cname        VARCHAR(100),
        stockprice   FLOAT,
        country            VARCHAR(100),
        PRIMARY KEY (cname),

)

CREATE TABLE Product(
        pname        VARCHAR(100),
        price        FLOAT,
        category           VARCHAR(100),
        manufacturer VARCHAR(100),          FOREIGN KEY (manufacturer)
        PRIMARY KEY (pname, manufacturer)   REFERENCES Company(cname)
)
```



More natural direction of reference

If the **primary key** is a set of columns (a **composite key**), then the **foreign key** also must be a set of columns that corresponds to the **composite key**.

Recap: Joins

```
Product(PName, Price, Category, Manufacturer)
Company(CName, StockPrice, Country)
```

Ex: Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
       AND Price <= 200
```

Recap: Joins

```
Product(PName, Price, Category, Manufacturer)
Company(CName, StockPrice, Country)
```

Ex: Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
       AND Price <= 200
```

```
SELECT PName, Price
FROM   Product
JOIN   Company ON Manufacturer = Cname
                  AND Country='Japan'
WHERE  Price <= 200
```

Recap: Joins

Product

PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

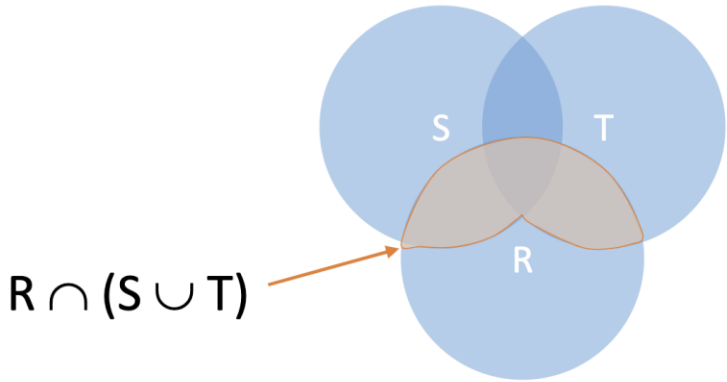


```
SELECT PName, Price
FROM   Product, Company
WHERE  Manufacturer = CName
       AND Country='Japan'
       AND Price <= 200
```

PName	Price
SingleTouch	$149.99

Recap: Multiset Operations

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```




```
output = {}

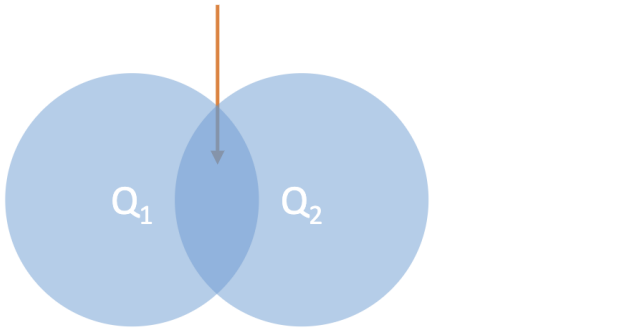
for r in R:
   for s in S:
      for t in T:
         if r['A'] == s['A'] or r['A'] == t['A']:
            output.add(r['A'])
return list(output)
```

Can you see now what happens if S = []?

Recap: Multiset Operations

```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
INTERSECT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \cap \{r.A \mid r.A = t.A\}$$



Nested Queries

```
Company(name, hq_city)
Product(pname, maker, factory_loc)
```

```
SELECT DISTINCT hq_city
FROM   Company, Product
WHERE  maker = name
       AND name IN (
            SELECT maker
            FROM   Product
            WHERE  factory_loc = 'US')
       AND name IN (
            SELECT maker
            FROM   Product
            WHERE  factory_loc = 'China')
```

"Headquarters of companies which make gizmos in US AND China"

Note: If we hadn't used DISTINCT here, how many copies of each hq_city would have been returned?

Nested Queries

- We can do nested queries because SQL is ***compositional:***
    - Everything (inputs / outputs) is represented as multisets- the output of one query can thus be used as the input to another (nesting)!
- This is extremely powerful!

Nested Queries

Another example:

```
Company(name, city)
Product(name, maker)
Purchase(id, product, buyer)
```

```
SELECT c.city
FROM   Company c
WHERE  c.name  IN (
     SELECT pr.maker
      FROM   Purchase p, Product pr
      WHERE  p.product = pr.name
        AND p.buyer = 'Joe Blow')
```

"Cities where one can find companies that manufacture products bought by Joe Blow"

Nested Queries

Is this query equivalent?

```
SELECT c.city
FROM   Company c,
       Product pr,
       Purchase p
WHERE  c.name = pr.maker
  AND  pr.name = p.product
  AND  p.buyer = 'Joe Blow'
```

Beware of duplicates!

Nested Queries

```
SELECT DISTINCT c.city
FROM   Company c,
       Product pr,
       Purchase p
WHERE  c.name = pr.maker
  AND  pr.name = p.product
  AND  p.buyer = 'Joe Blow'
```

```
SELECT DISTINCT c.city
FROM   Company c
WHERE  c.name  IN (
  SELECT pr.maker
  FROM   Purchase p, Product pr
  WHERE  p.product = pr.name
       AND p.buyer = 'Joe Blow')
```

Now they are equivalent

Subqueries Returning Relations

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

Ex: `Product(name, price, category, maker)`

```
SELECT name
FROM   Product
WHERE  price > ALL(
       SELECT price
       FROM   Product
       WHERE  maker = 'Gizmo-Works')
```

Find products that are more expensive than all those produced by "Gizmo-Works"

Subqueries Returning Relations

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

Ex: `Product(name, price, category, maker)`

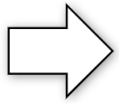
```
SELECT p1.name
FROM   Product p1
WHERE  p1.maker = 'Gizmo-Works'
   AND EXISTS(
      SELECT p2.name
      FROM   Product p2
      WHERE  p2.maker <> 'Gizmo-Works'
         AND p1.name = p2.name)
```

<> means !=

Find 'copycat' products, i.e. products made by competitors with the same names as products made by "Gizmo-Works"

Nested queries as alternatives to INTERSECT and EXCEPT

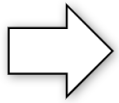
```
(SELECT R.A, R.B
 FROM   R)
INTERSECT
(SELECT S.A, S.B
 FROM   S)
```



```
SELECT R.A, R.B
FROM   R
WHERE EXISTS(
        SELECT *
        FROM S
        WHERE R.A=S.A AND R.B=S.B)
```

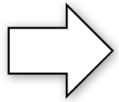
Nested queries as alternatives to INTERSECT and EXCEPT

```
(SELECT R.A, R.B
 FROM   R)
INTERSECT
(SELECT S.A, S.B
 FROM   S)
```



```
SELECT R.A, R.B
FROM   R
WHERE EXISTS(
        SELECT *
        FROM S
        WHERE R.A=S.A AND R.B=S.B)
```

```
(SELECT R.A, R.B
 FROM   R)
EXCEPT
(SELECT S.A, S.B
 FROM   S)
```



```
SELECT R.A, R.B
FROM   R
WHERE NOT EXISTS(
        SELECT *
        FROM S
        WHERE R.A=S.A AND R.B=S.B)
```

Correlated Queries

```
Movie(title, year, director, length)
```

```
SELECT DISTINCT title
FROM   Movie AS m
WHERE  year <> ANY(
           SELECT  year
           FROM    Movie
           WHERE   title =  m.title)
```

Find movies whose title appears more than once.

Note the scoping of the variables!

Note also: this can still be expressed as single SFW query…

Correlated Queries

```
SELECT title
FROM Movie
GROUP BY title
HAVING COUNT(*) > 1;
```

Find movies whose title appears more than once.

Note also: this can still be expressed as single SFW query…

Correlated Queries

```
Product(name, price, category, maker, year)
```

```
SELECT DISTINCT  x.name, x.maker
FROM    Product AS x
WHERE   x.price > ALL(
            SELECT y.price
            FROM    Product AS y
            WHERE   x.maker = y.maker
                AND y.year < 1972)
```

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

Can be very powerful (also much harder to optimize)

Aggregation

```
SELECT AVG(price)
FROM   Product
WHERE  maker = "Toyota"
```

```
SELECT COUNT(*)
FROM   Product
WHERE  year > 1995
```

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

Except COUNT, all aggregations apply to a single attribute

Aggregation

- COUNT applies to duplicates, unless otherwise stated

```
SELECT COUNT(category)
FROM   Product
WHERE  year > 1995
```

Note: Same as COUNT(*). Why?

We probably want:

```
SELECT COUNT(DISTINCT category)
FROM   Product
WHERE  year > 1995
```

Aggregation

```
Purchase(product, date, price, quantity)
```

```
SELECT SUM(price * quantity)
FROM   Purchase
```

What do these mean?

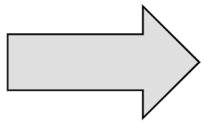
```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

Aggregation

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

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



50 (= 1*20 + 1.50*20)

Grouping and Aggregation

```
Purchase(product, date, price, quantity)
```

```
SELECT    product,
          SUM(price * quantity) AS TotalSales
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
```

Find total sales after 10/1/2005 per product.

Grouping and Aggregation

Semantics of the query:

1. Compute the FROM and WHERE clauses

2. Group by the attributes in the GROUP BY

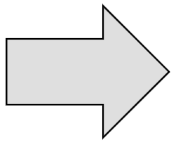
3. Compute the SELECT clause: grouped attributes and aggregates

Grouping and Aggregation

1. Compute the FROM and WHERE clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM



Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

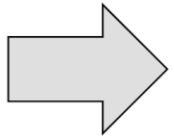
Grouping and Aggregation

2. Group by the attributes in the GROUP BY

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

GROUP BY



Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

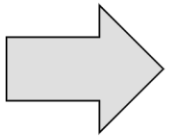
Grouping and Aggregation

3. Compute the SELECT clause: grouped attributes and aggregates

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT



Product	TotalSales
Bagel	50
Banana	15

HAVING Clause

```
SELECT    product, SUM(price*quantity)
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
HAVING    SUM(quantity) > 100
```

HAVING clauses contains conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples**…*

Same query as before, except that we consider only products that have more than 100 buyers

General form of Grouping and Aggregation

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

- S = Can ONLY contain attributes $a_1,\ldots,a_k$ and/or aggregates over other attributes
- C_1 = is any condition on the attributes in $R_1,\ldots,R_n$
- C_2 = is any condition on the aggregate expressions

General form of Grouping and Aggregation

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

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
3. Apply condition C_2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Group By vs. Nested Queries

```
SELECT   product, Sum(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

```
SELECT DISTINCT x.product,
       (SELECT Sum(y.price*y.quantity)
        FROM   Purchase y
        WHERE  x.product = y.product
           AND y.date > '10/1/2005') AS TotalSales
FROM   Purchase x
WHERE  x.date > '10/1/2005'
```

Group By vs. Nested Queries

```
Author(login, name)
Wrote(login, url)
```

- Find authors who wrote ≥ 10 documents:

Group By vs. Nested Queries

```
Author(login, name)
Wrote(login, url)
```

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM   Author
WHERE  COUNT(
     SELECT Wrote.url
     FROM   Wrote
     WHERE  Author.login = Wrote.login) > 10
```

This is SQL by a novice

Group By vs. Nested Queries

```
Author(login, name)
Wrote(login, url)
```

- Find authors who wrote ≥ 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT   Author.name
FROM     Author, Wrote
WHERE    Author.login = Wrote.login
GROUP BY Author.name
HAVING   COUNT(Wrote.url) > 10
```

This is SQL by an expert

No need for DISTINCT: automatically from GROUP BY

Group By vs. Nested Queries

Which way is more efficient?

- Attempt #1- *With nested:* How many times do we do a SFW query over all of the Wrote relations?

- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be <u>much</u> more efficient!

Practice: Group By

Consider a set of tables that describe the up-and-coming bagel startup industry; for now let's just look at two tables here, `bagel`, which describes types of bagels made by the different bagel companies:

> - name STRING
> - price FLOAT
> - made_by STRING

And `purchase`:

> - bagel_name STRING
> - franchise STRING
> - date INT
> - quantity INT
> - purchaser_age INT

Where `purchase.bagel_name` references `bagel.name` and `purchase.franchise` references `bagel.made_by`:

Practice: Group By

Consider a set of tables that describe the up-and-coming bagel startup industry; for now let's just look at two tables here, `bagel`, which describes types of bagels made by the different bagel companies:

> - name STRING
> - price FLOAT
> - made_by STRING

And `purchase`:

> - bagel_name STRING
> - franchise STRING
> - date INT
> - quantity INT
> - purchaser_age INT

Where `purchase.bagel_name` references `bagel.name` and `purchase.franchise` references `bagel.made_by`:

```
%sql SELECT * FROM bagel LIMIT 3;
```

```
Done.
```

name	price	made_by
Plain with shmear	1.99	Bobs Bagels
Egg with shmear	2.39	Bobs Bagels
eBagel Drinkable Bagel	27.99	eBagel

```
%sql SELECT * FROM purchase LIMIT 3;
```

```
Done.
```

bagel_name	franchise	date	quantity	purchaser_age
Plain with shmear	Bobs Bagels	1	12	28
Egg with shmear	Bobs Bagels	2	6	47
Plain with shmear	BAGEL CORP	2	12	24

Practice: Group By

Can you write a query to get the *total revenue* for each bagel type **which had an average purchaser age over 18**? Type your query below:

```
%%sql
SELECT b.name, SUM(p.quantity * b.price) AS Revenue
FROM bagel b, purchase p
WHERE b.name = p.bagel_name AND b.made_by = p.franchise
GROUP BY b.name
HAVING AVG(p.purchaser_age) > 18;
```

```
Done.
```

name	Revenue
Egg with shmear	14.34
Plain with shmear	84.51

Practice: Group By

Here we'll use a simplified version of the `precipitation_full` table, which just has *daily* rainfall *in CA only*, and has the following schema:

> - station_id
> - day
> - precipitation

```
%sql SELECT * FROM precipitation LIMIT 5;
```

Done.

station_id	day	precipitation
16102	1	10
16102	4	10
16102	24	30
21201	1	0
21201	20	10

Practice: Group By

We want to get station_ids which have average precipitations > 75. Try doing this first as a nested query:

```
%%sql
SELECT DISTINCT p.station_id
FROM precipitation p
WHERE (
    SELECT AVG(precipitation)
    FROM precipitation
    WHERE station_id = p.station_id) > 75;
```

Done.

station_id
88302
250002
335701
357302
488301

Practice: Group By

Now, try re-writing as a GROUP BY:

```
%%sql
SELECT station_id
FROM precipitation
GROUP BY station_id
HAVING AVG(precipitation) > 75;
```

Done.

station_id
88302
250002
335701
357302
488301

Practice: Group By

Now time it by using `%time` followed by single-line versions of your queries above (clunky, but will work) to see how they compare!

```
%time %sql SELECT DISTINCT p.station_id FROM precipitation p WHERE (SELECT AVG(precipitation) FROM precipitation WHERE station_id = p.station_id) > 75;
```

```
Done.
CPU times: user 41.9 ms, sys: 1.39 ms, total: 43.3 ms
Wall time: 42.5 ms
```

station_id
88302
250002
335701
357302
488301

Practice: Group By

Now time it by using `%time` followed by single-line versions of your queries above (clunky, but will work) to see how they compare!

```
%time %sql SELECT DISTINCT p.station_id FROM precipitation p WHERE (SELECT AVG(precipitation) FROM precipitation WHERE station_id = p.station_id) > 75;
```

```
Done.
CPU times: user 41.9 ms, sys: 1.39 ms, total: 43.3 ms
Wall time: 42.5 ms
```

station_id
88302
250002
335701
357302
488301

```
%time %sql SELECT p.station_id FROM precipitation p GROUP BY p.station_id HAVING AVG(p.precipitation) > 75;
```

```
Done.
CPU times: user 3.06 ms, sys: 1.43 ms, total: 4.49 ms
Wall time: 3.11 ms
```

station_id
88302
250002
335701
357302
488301

An ~ 10-20x difference in execution time!!

Quantifiers

```
Product(name, price, company)
Company(name, city)
```

```
SELECT DISTINCT Company.cname
FROM   Company, Product
WHERE  Company.name = Product.company
   AND Product.price < 100
```

Find all companies that make some products with price < 100

An existential quantifier is a logical quantifier (roughly) of the form “there exists”

Existential: easy ! ☺

Quantifiers

```
Product(name, price, company)
Company(name, city)
```

Find all companies with products all having price < 100

```
SELECT DISTINCT Company.cname
FROM    Company
WHERE   Company.name NOT IN(
        SELECT Product.company
        FROM Product.price >= 100)
```

Equivalent

Find all companies that make only products with price < 100

A universal quantifier is of the form "for all"

Universal: hard ! ☹

NULL Values

- Whenever we don’t have a value, we can put a NULL
- Can mean many things:
  - Value does not exists
  - Value exists but is unknown
  - Value not applicable
  - Etc.
- The schema specifies for each attribute if can be null (*nullable* attribute) or not
- How does SQL cope with tables that have NULLs?

NULL Values

- *For numerical operations,* NULL -> NULL:
  - If x = NULL then 4*(3-x)/7 is still NULL

- *For boolean operations,* in SQL there are three values:

FALSE = 0
UNKNOWN = 0.5
TRUE = 1

  - If x= NULL then x="Joe" is UNKNOWN

NULL Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM   Person
WHERE (age < 25)
  AND (height > 6 AND weight > 190)
```

Won't return e.g. (age=20 height=NULL weight=200)!

Rule in SQL: include only tuples that yield TRUE (1.0)

NULL Values

Unexpected behavior:

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
```

Some Persons are not included !

NULL Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
    OR age IS NULL
```

Now it includes all Persons!

Outer Joins

By default, joins in SQL are **"inner joins"**:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product
  JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

Both equivalent:
Both INNER JOINS!

Outer Joins

By default, joins in SQL are **"inner joins"**:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product
  JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

However: Products that never sold (with no Purchase tuple) will be lost!

Outer Joins

- An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations
  - I.e. If we join relations A and B on a.X = b.X, and there is an entry in A with X=5, but none in B with X=5…
    - A LEFT OUTER JOIN will return a tuple (a, NULL)!

- Left outer joins in SQL:

```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase ON
       Product.name = Purchase.prodName
```

Now we'll get products even if they didn't sell

Outer Joins

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
  INNER JOIN Purchase
       ON Product.name = Purchase.prodName
```



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Note: another equivalent way to write an INNER JOIN!

Outer Joins

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
  LEFT OUTER JOIN Purchase
       ON Product.name = Purchase.prodName
```



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

Outer Joins

- Left outer join:
  - Include the left tuple even if there's no match
- Right outer join:
  - Include the right tuple even if there's no match
- Full outer join:
  - Include the both left and right tuples even if there's no match

Practice: Outer Joins

Recall that the tables we just looked at:

`bagel`, which describes types of bagels made by the different bagel companies:

> - name STRING
> - price FLOAT
> - made_by STRING

And `purchase`:

> - bagel_name STRING
> - franchise STRING
> - date INT
> - quantity INT
> - purchaser_age INT

Where `purchase.bagel_name` references `bagel.name` and `purchase.franchise` references `bagel.made_by`.

Practice: Outer Joins

Can you find out if there were any purchases of products not on one of the company's official lists (i.e. the `bagel` table), using a single SQL query?

Write your query here:

```
%%sql
SELECT b.name AS "Official List Name", p.bagel_name AS "Purchase Item", franchise AS "Bought From", p.date, p.quantity
FROM purchase p
LEFT OUTER JOIN bagel b
     ON b.name = p.bagel_name AND b.made_by = p.franchise
WHERE b.name IS NULL;
```

```
Done.
```

Official List Name	Purchase Item	Bought From	date	quantity
None	Drugs	BAGEL CORP	7	1000