### Querying Data From a Table

**SELECT** c1, c2 FROM t;
Query data in columns c1, c2 from a table

**SELECT** * FROM t;
Query all rows and columns from a table

**SELECT** c1, c2 FROM t **WHERE** condition;
Query data and filter rows with a condition

**SELECT** **DISTINCT** c1 FROM t **WHERE** condition;
Query distinct rows from a table

**SELECT** c1, c2 FROM t **ORDER BY** c1 [ASC | DESC];
Sort the result set in ascending or descending order

**SELECT** c1, c2 FROM t **ORDER BY** c1 **LIMIT** n OFFSET offset;
Skip offset of rows and return the next n rows

**SELECT** c1, **aggregate**(c2) FROM t **GROUP BY** c1;
Group rows using an aggregate function

**SELECT** c1, **aggregate**(c2) FROM t **GROUP BY** c1 **HAVING** condition;
Filter groups using HAVING clause

### Querying From Multiple Tables

**SELECT** c1, c2 FROM t1 **INNER JOIN** t2 ON condition;
Inner join t1 and t2

**SELECT** c1, c2 FROM t1 **LEFT JOIN** t2 ON condition;
Left join t1 and t2

**SELECT** c1, c2 FROM t1 **RIGHT JOIN** t2 ON condition;
Right join t1 and t2

**SELECT** c1, c2 FROM t1 **FULL OUTER JOIN** t2 ON condition;
Perform full outer join

**SELECT** c1, c2 FROM t1 **CROSS JOIN** t2;
Produce a Cartesian product of rows in tables

**SELECT** c1, c2 FROM t1, t2;
Another way to perform cross join

**SELECT** c1, c2 FROM t1 A **INNER JOIN** t2 B ON condition;
Join t1 to itself using INNER JOIN clause

### Using SQL Operators

**SELECT** c1, c2 FROM t1 **UNION** [ALL];
Combine rows from two queries

**SELECT** c1, c2 FROM t1 **INTERSECT**
Return the intersection of two queries

**SELECT** c1, c2 FROM t1 **MINUS**;
Subtract a result set from another result set

**SELECT** c1, c2 FROM t1 **WHERE** c1 [NOT] LIKE pattern;
Query rows using pattern matching %, _

**SELECT** c1, c2 FROM t **WHERE** c1 [NOT] IN value_list;
Query rows in a list

**SELECT** c1, c2 FROM t **WHERE** c1 BETWEEN low AND high;
Query rows between two values

**SELECT** c1, c2 FROM t **WHERE** c1 IS [NOT] NULL;
Check if values in a table is NULL or not
Key concepts

- JOINs
- Aggregation & GROUP BY
Reminder on schemas

- Product(PName, Price, Category, Manufacturer)
- Company(CName, StockPrice, Country)
- Students(sid: string, name: string, gpa: float)
- Enrolled(student_id: string, cid: string, grade: string)

We’ll use different Tables/tuples, for examples to build ideas

Data about local areas (for real-world examples)

- SolarPanel(region_name: string, kw_total: float, carbon_offset_ton_metrics: float, ...)
- Census(zipcode: string, population: int, ...)
- Pollution(zipcode: string, Particle_count: int...)
- BikeShare(zipcode: string, trip_origin: float, trip_end: float, ...)
  ...
Option 1: ‘Good’ tables, with 10s-100s of columns

Option 2: ‘Frankenstein Table’ (with 1000s of columns)
Option 1: A few “good” tables (with 10s of columns)

- Option 2: ‘FrankenTable’ (with 1000s of columns)

Trade offs?
- Reads? Writes?
- 100s - thousands of applications reading/writing data
  ⇒ 1 column? all columns? [hybrid?]
Assume we have a set of “good” tables
  ▸ How do we “connect” (join) those tables?
  ▸ Next 2 weeks

Related important question
  ▸ How to break up a “Franken” Table into “good” Tables? (i.e, design “good” schema)
  ▸ Study in Week 8, 9
Joins

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

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

A join between tables returns all unique combinations of their tuples which meet specified join condition
Joins

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

Several equivalent ways to write a basic join in SQL:

```
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
WHERE Price <= 200
    AND Country='Japan'
```
## Joins

### Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

### Company

<table>
<thead>
<tr>
<th>CName</th>
<th>Stock Price</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

### SQL Query

```sql
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer = CName
    AND Country = 'Japan'
    AND Price <= 200
```
Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)
Company(name, address)

1. **SELECT DISTINCT** name, address
2. **FROM** Person, Company
3. **WHERE** worksfor = name

Which “address” does this refer to?

Which name”s??
Both equivalent ways to resolve variable ambiguity:

Person(name, address, worksfor)
Company(name, address)

\[
\begin{align*}
\text{SELECT DISTINCT} & \quad \text{Person.name, Person.address} \\
\text{FROM} & \quad \text{Person, Company} \\
\text{WHERE} & \quad \text{Person.worksfor = Company.name}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT DISTINCT} & \quad p.name, p.address \\
\text{FROM} & \quad \text{Person p, Company c} \\
\text{WHERE} & \quad p.worksfor = c.name
\end{align*}
\]
Semantics of JOINs

```
SELECT x_1.a_1, x_1.a_2, ..., x_n.a_k
FROM R_1 AS x_1, R_2 AS x_2, ..., R_n
AS x_n
WHERE Conditions(x_1, ..., x_n)
```

Answer = {}
for x_1 in R_1 do
  for x_2 in R_2 do
    ....
    for x_n in R_n do
      if Conditions(x_1, ..., x_n) then Answer = Answer \cup \{(x_1.a_1, x_1.a_2, ..., x_n.a_k)\}
return Answer

Note:
This is a multiset union
Semantics of JOINs (2 tables)

Recall: Cross product \((R \times S)\) is the set of all unique tuples in \(R,S\)

Ex: \(\{a,b,c\} \times \{1,2\}\)

\[\{(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)\}\]

- **Take cross product**
  \[V = R \times S\]

- **Apply selections/conditions**
  \[Y = \{(r, s) \in V \mid r.A == s.B\}\]

- **Apply projections** to get final output
  \[Z = (y.A) \text{ for } y \in Y\]

Remembering this order is critical to understanding the output of certain queries (see later on…)

---

```
SELECT R.A
FROM R, S
WHERE R.A = S.B
```
An example of SQL semantics

```
SELECT R.A
FROM R, S
WHERE R.A = S.B
```

Cross Product

Apply Selections / Conditions

Apply Projection

Output
Note: we say “semantics” not “execution order”

● The preceding slides show what a join means

● Not actually how the DBMS executes it under the covers
A Subtlety about Joins

Find all countries that manufacture some product in the ‘Gadgets’ category.

```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=CName AND Category='Gadgets'
```
A Subtlety about Joins

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PName</td>
<td>Price</td>
<td>Category</td>
<td>Manuf</td>
</tr>
<tr>
<td>Gizmo</td>
<td>$19</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Company</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cname</td>
<td>Stock</td>
<td>Country</td>
</tr>
<tr>
<td>GWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

SELECT Country
FROM Product, Company
WHERE Manufacturer=Cname
AND Category='Gadgets'

What is the Problem? What is the Solution?
Key concepts

- JOINs
- Aggregation & GROUP BY
Aggregation

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

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

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

COUNT counts all tuples (including duplicates), unless otherwise stated

\[
\text{SELECT COUNT(category)} \\
\text{FROM Product} \\
\text{WHERE year > 1995}
\]

versus

\[
\text{SELECT COUNT(DISTINCT category)} \\
\text{FROM Product} \\
\text{WHERE year > 1995}
\]
Simple Aggregations

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
</tbody>
</table>

Example 1

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

50

(= 1*20 + 1.50*20)

Example 2

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

65

(= 1*20 + 1.50*20 + 0.5*10 + 1*10)
Grouping and Aggregation

What GROUPings are possible?
- Type, Size, Color
- Number of holes
- Combination?
What GROUPings are possible?

Possible Groups
- **Product?** (e.g. SUM(quantity) by product) # product units sold
- **Date?** (e.g., SUM(price\*quantity) by date) # daily sales
- **Price?**
- **Product, Date?**
- **<various column combinations>**

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
</tbody>
</table>
Grouping and Aggregation

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

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

Let's see what this means…
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

(Why is Select on top? Similar to Functions -- f ⇒ output on top)

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

```
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```
2. Group by the attributes in the GROUP BY

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

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
3. Compute the **SELECT** clause: grouped attributes and aggregates

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

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>
HAVING Clause

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

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

HAVING clauses contain conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples**...*
General form of Grouping and Aggregation

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

\[
\begin{align*}
\text{SELECT} & \quad S \\
\text{FROM} & \quad R_1, \ldots, R_n \\
\text{WHERE} & \quad C_1 \\
\text{GROUP BY} & \quad a_1, \ldots, a_k \\
\text{HAVING} & \quad C_2
\end{align*}
\]
General form of Grouping and Aggregation

```
SELECT  S
FROM    R_1,...,R_n
WHERE   C_1
GROUP BY a_1,...,a_k
HAVING  C_2
```

Evaluation steps:
1. Evaluate **FROM-WHERE**: apply condition $C_1$ on the attributes in $R_1,...,R_n$
2. **GROUP BY** the attributes $a_1,...,a_k$
3. Apply condition $C_2$ to each group (may need to compute aggregates)
4. Compute aggregates in $S$ and return the result
NULL and NOT NULL

- To say “don’t know the value” we use **NULL**
  
  NULL has (sometimes painful) semantics, more detail later

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Bob</td>
<td>3.9</td>
</tr>
<tr>
<td>143</td>
<td>Jim</td>
<td>NULL</td>
</tr>
</tbody>
</table>

*Say, Jim just enrolled in his first class.*

In SQL, we may constrain a column to be NOT NULL, e.g., “name” in this table
NULLS in SQL

- Whenever we don’t have a value, we can put a NULL

- Can mean many things:
  - Value does not exist
  - 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:
  
<table>
<thead>
<tr>
<th>Value</th>
<th>Boolean Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>0.5</td>
</tr>
<tr>
<td>TRUE</td>
<td>1</td>
</tr>
</tbody>
</table>

  - If x = NULL then x == “Joe” is UNKNOWN (Is x equal to ‘Joe’?)
Null Values

- $C_1 \text{ AND } C_2 = \min(C_1, C_2)$
- $C_1 \text{ OR } C_2 = \max(C_1, C_2)$
- $\neg C_1 = 1 - C_1$

```
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:

```sql
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!
By default, joins in SQL are “inner joins”:

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

Both equivalent: Both INNER JOINS!

[Like Below]
Inner Joins + NULLS = Lost data?

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
```

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:

```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
LEFT OUTER JOIN

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

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>category</td>
</tr>
<tr>
<td>iphone</td>
<td>media</td>
</tr>
<tr>
<td>Tesla</td>
<td>car</td>
</tr>
<tr>
<td>Ford Pinto</td>
<td>car</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Purchase</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>prodName</td>
<td>store</td>
</tr>
<tr>
<td>iPhone</td>
<td>Apple store</td>
</tr>
<tr>
<td>Tesla</td>
<td>car</td>
</tr>
<tr>
<td>iPhone</td>
<td>Apple store</td>
</tr>
<tr>
<td>Ford Pinto</td>
<td>NULL</td>
</tr>
</tbody>
</table>

- `LEFT OUTER JOIN` combines rows from both tables, including rows where there is no match in the right table.
- In this example, the `iPhone` from the `Purchase` table matches the `iPhone` in the `Product` table.
- The `Ford Pinto` from the `Product` table does not have a match in the `Purchase` table, so its `store` value is NULL.
Other 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
Key concepts

- JOINs
- Aggregation & GROUP BY
## Querying Data From a Table

**SELECT** `c1`, `c2` FROM `t`;
Query data in columns `c1`, `c2` from a table

**SELECT** `*` FROM `t`;
Query all rows and columns from a table

**SELECT** `c1, c2` FROM `t` WHERE `condition`;
Query data and filter rows with a condition

**SELECT** `DISTINCT c1` FROM `t` WHERE `condition`;
Query distinct rows from a table

**SELECT** `c1, c2` FROM `t` ORDER BY `c1` ASC [DESC];
Sort the result set in ascending or descending order

**SELECT** `c1, c2` FROM `t` ORDER BY `c1` LIMIT `n` OFFSET `offset`;
Skip `offset` rows and return the next `n` rows

**SELECT** `c1, aggregate(c2)` FROM `t` GROUP BY `c1`;
Group rows using an aggregate function

**SELECT** `c1, aggregate(c2)` FROM `t` GROUP BY `c1` HAVING `condition`;
Filter groups using HAVING clause

## Querying From Multiple Tables

**SELECT** `c1, c2` FROM `t1` INNER JOIN `t2` ON `condition`;
Inner join `t1` and `t2`

**SELECT** `c1, c2` FROM `t1` LEFT JOIN `t2` ON `condition`;
Left join `t1` and `t2`

**SELECT** `c1, c2` FROM `t1` RIGHT JOIN `t2` ON `condition`;
Right join `t1` and `t2`

**SELECT** `c1, c2` FROM `t1` FULL OUTER JOIN `t2` ON `condition`;
Perform full outer join

**SELECT** `c1, c2` FROM `t1` CROSS JOIN `t2`;
Produce a Cartesian product of rows in tables

**SELECT** `c1, c2` FROM `t1` CROSS JOIN `t2` WHERE `c1` [NOT] IN `value_list`;
Query rows in a list

**SELECT** `c1, c2` FROM `t` WHERE `c1` BETWEEN `low` AND `high`;
Query rows between two values

**SELECT** `c1, c2` FROM `t` WHERE `c1` [IS NOT] NULL;
Check if values in a table is NULL or not

## Using SQL Operators

**SELECT** `c1, c2` FROM `t1` UNION [ALL] `t2`;
Combine rows from two queries

**SELECT** `c1, c2` FROM `t1` INTERSECT `t2`;
Return the intersection of two queries

**SELECT** `c1, c2` FROM `t1` MINUS `t2`;
Subtract a result set from another result set

**SELECT** `c1, c2` FROM `t1` WHERE `c1` [NOT] LIKE `pattern`;
Query rows using pattern matching `%`, `_`
Today's Lecture

SQL - Part 3

- SQL Sets operators
- When are two queries equivalent?
- How does SQL work?
  - Intro to Relational Algebra
  - A basic RDBMS query optimizer
**SQL CHEAT SHEET**

---

**QUERYING DATA FROM A TABLE**

SELECT c1, c2 FROM t;
- Query data in columns c1, c2 from a table

SELECT * FROM t;
- Query all rows and columns from a table

SELECT c1, c2 FROM t
WHERE condition;
- Query data and filter rows with a condition

SELECT DISTINCT c1 FROM t
WHERE condition;
- Query distinct rows from a table

SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
- Sort the result set in ascending or descending order

SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
- Skip offset of rows and return the next n rows

SELECT c1, aggregate(c2) FROM t
GROUP BY c1;
- Group rows using an aggregate function

SELECT c1, aggregate(c2) FROM t
GROUP BY c1
HAVING condition;
- Filter groups using HAVING clause

---

**QUERYING FROM MULTIPLE TABLES**

SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
- Inner join t1 and t2

SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
- Left join t1 and t2

SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
- Right join t1 and t2

SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
- Perform full outer join

SELECT c1, c2
FROM t1
CROSS JOIN t2;
- Produce a Cartesian product of rows in tables

SELECT c1, c2
FROM t1, t2;
- Another way to perform cross join

SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
- Join t1 to itself using INNER JOIN clause

---

**USING SQL OPERATORS**

SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
- Combine rows from two queries

SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
- Return the intersection of two queries

SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
- Subtract a result set from another result set

SELECT c1, c2 FROM t1
WHERE [NOT] LIKE pattern;
- Query rows using pattern matching %, _

SELECT c1, c2 FROM t1
WHERE [NOT] IN value_list;
- Query rows in a list

SELECT c1, c2 FROM t
WHERE c1 BETWEEN low AND high;
- Query rows between two values

SELECT c1, c2 FROM t
WHERE c1 [NOT] IS NULL;
- Check if values in a table is NULL or not

---

sqltutorial.org/sql-cheat-sheet
1. Multiset operators in SQL

2. Nested queries

Notation

\[ Q_1 \]

Result for Query Q1

\[ Q_2 \]

Result for Query Q2

What you will learn about in this section
Explicit Set Operators:
INTERSECT, UNIONS on results of Queries Q1, Q2

Q1

\[
\text{SELECT R.A FROM R, S WHERE R.A=S.A}
\]

Q2

\[
\text{SELECT R.A FROM R, T WHERE R.A=T.A}
\]

Q1 \ INTERSECT \ Q2

Q1 \ UNION \ Q2

By DEFAULT:
SQL retains Set semantics for Set Operators
Use ALL for Multiset

```
SELECT R.A
FROM R, S
WHERE R.A=S.A
UNION ALL
SELECT R.A
FROM R, T
WHERE R.A=T.A
```

Q1 Q2

*ALL indicates Multiset operations*
Recall Multisets

Multiset X

<table>
<thead>
<tr>
<th>Tuple</th>
<th>( \lambda(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, a)</td>
<td>2</td>
</tr>
<tr>
<td>(1, b)</td>
<td>1</td>
</tr>
<tr>
<td>(2, c)</td>
<td>3</td>
</tr>
<tr>
<td>(1, d)</td>
<td>2</td>
</tr>
</tbody>
</table>

\( \lambda(X) = "\text{Count of tuple in } X" \)  
(Items not listed have implicit count 0)
Generalizing Set Operations to Multiset Operations

<table>
<thead>
<tr>
<th>Multiset X</th>
<th>Tuple</th>
<th>$\lambda(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, a)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(1, b)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(2, c)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>(1, d)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

$\cap$

<table>
<thead>
<tr>
<th>Multiset Y</th>
<th>Tuple</th>
<th>$\lambda(Y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, a)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>(1, b)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(2, c)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(1, d)</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

= $\lambda(Z) = \min(\lambda(X), \lambda(Y))$

<table>
<thead>
<tr>
<th>Multiset Z</th>
<th>Tuple</th>
<th>$\lambda(Z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, a)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(1, b)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(2, c)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(1, d)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Generalizing Set Operations to Multiset Operations

Multiset $X$:

<table>
<thead>
<tr>
<th>Tuple</th>
<th>$\lambda(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, a)</td>
<td>2</td>
</tr>
<tr>
<td>(1, b)</td>
<td>0</td>
</tr>
<tr>
<td>(2, c)</td>
<td>3</td>
</tr>
<tr>
<td>(1, d)</td>
<td>0</td>
</tr>
</tbody>
</table>

Multiset $Y$:

<table>
<thead>
<tr>
<th>Tuple</th>
<th>$\lambda(Y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, a)</td>
<td>5</td>
</tr>
<tr>
<td>(1, b)</td>
<td>1</td>
</tr>
<tr>
<td>(2, c)</td>
<td>2</td>
</tr>
<tr>
<td>(1, d)</td>
<td>2</td>
</tr>
</tbody>
</table>

$\bigcup$ $\bigcup$ $\lambda(Z) = \lambda(X) + \lambda(Y)$
SQL is **compositional**

Can construct powerful query chains (e.g., \(f(g(...(x)))\))

Inputs / outputs are multisets

⇒ Output of one query can be input to another (nesting)!

\[ Q_1 \rightarrow Q_2 \rightarrow \ldots \rightarrow Q_n \rightarrow \text{Result Table} \]

⇒ Including on same table (e.g., self correlation)
Nested queries: Sub-queries Return Relations

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 = 'Mickey')

- Companies making products bought by Mickey
- Location of companies?
Subqueries Return Relations

You can also use operations of the form:
- \( s > \text{ALL} R \)
- \( s < \text{ANY} R \)
- \( \text{EXISTS} R \)

Ex: \text{Product}(\text{name, price, category, maker})

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

Find ‘copycat’ products, i.e. products made by competitors with the same names as products made by “Gizmo-Works”

\[
\text{SELECT name}
\text{FROM Product}
\text{WHERE price} > \text{ALL(}
\text{SELECT price}
\text{FROM Product}
\text{WHERE maker = ‘Gizmo-Works’})
\]

\[
\text{SELECT p1.name}
\text{FROM Product p1}
\text{WHERE p1.maker = ‘Gizmo-Works’}
\text{AND EXIST}(\text{SELECT p2.name}
\text{FROM Product p2}
\text{WHERE p2.maker <> ‘Gizmo-Works’}
\text{AND p1.name = p2.name})
\]

<> means !=

Note the scoping of the variables!
Example: Complex Correlated Query

```
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 < 2010)
```

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 2010

Can be very powerful (also much harder to optimize)
SQL is compositional

Can construct powerful query chains (e.g., f(g(x)))

Inputs / outputs are multisets

⇒ Output of one query can be input to another (nesting)!

⇒ Including on same table (e.g., self correlation)
Can write different SQL queries to solve same problem

Key:

-- Be careful with sets and multisets
-- Go back to semantics (1st principles)
Example1: Two equivalent queries?

Find all companies with products having price < 100

\[
\text{SELECT DISTINCT Company.cname}
\text{FROM Company, Product}
\text{WHERE Company.name = Product.company}
\text{AND Product.price < 100}
\]

VS

Find all companies that make only products with price < 100

A universal quantifier is of the form “for all”
Example 2: Headquarters of companies which make gizmos in US AND China

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

Option 1: With Nested queries

```
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')
```

Option 2: With Intersections

```
SELECT hq_city
FROM Company, Product
WHERE maker = name
  AND factory_loc='US'
INTERSECT
SELECT hq_city
FROM Company, Product
WHERE maker = name
  AND factory_loc='China'
```

<table>
<thead>
<tr>
<th>Company</th>
<th>hq_city</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Co.</td>
<td>Seattle</td>
</tr>
<tr>
<td>Y Inc.</td>
<td>Seattle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>pname</th>
<th>maker</th>
<th>factory_loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X Co.</td>
<td>U.S.</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y Inc.</td>
<td>China</td>
<td></td>
</tr>
</tbody>
</table>

X Co has a factory in the US (but not China)
Y Inc. has a factory in China (but not US)
But Seattle is returned by the query!
⇒ Option 1 and Option 2 are **NOT** equivalent
Example 3: Are these equivalent? [harder version]

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

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

---

**Step 1:** Construct some examples

**Step 2:** Test examples

Beware of duplicates!
Example 3: Are these equivalent?

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

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

Fix duplicates!

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

```
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 = 'Mickey')
```

Now they are equivalent (both use set semantics)
Example 4: Are these equivalent?

- Find students enrolled in > 5 classes

 Attempt 1: with nested queries

```sql
SELECT DISTINCT Students.sid
FROM Students
WHERE (SELECT COUNT (cid)
FROM Enrolled
WHERE Students.sid = Enrolled.student_id
) > 5
```

SQL by a novice

 Attempt 2: with GROUP BYs

```sql
SELECT Students.sid
FROM Students, Enrolled
WHERE Students.sid = Enrolled.student_id
GROUP BY Students.sid
HAVING COUNT(Enrolled.cid) > 5
```

1. SQL by an expert
2. No need for DISTINCT: automatic from GROUP BY
Group-by vs. Nested Query

Which way is more efficient?

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

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

With GROUP BY can be **much** more efficient!
Example 5: An Unintuitive Query

**Query**

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

**Computation**

Computes $R \cap (S \cup T)$

**Semantics**

What does it compute? But what if $S = \emptyset$?

Go back to the semantics!
What does this look like in Python?

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

Semantics:
1. Take **cross-product**
2. Apply **selections / conditions**
3. Apply **projection**

*Joins / cross-products are just nested for loops (in simplest implementation)!*

*If-then statements!*
What does this look like in Python?

```python
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 = []?
Can write different SQL queries to solve same problem

Key:

-- Be careful with sets and multisets
-- Go back to semantics (1st principles)
Basic SQL Summary

SQL is a high-level declarative language for manipulating data (DML)

○ The workhorse is the SFW block

○ Set operators are powerful but have some subtleties

○ Powerful, nested queries also allowed
### Querying Data from a Table

**SELECT c1, c2 FROM t;**
- Query data in columns c1, c2 from a table

**SELECT * FROM t;**
- Query all rows and columns from a table

**SELECT c1, c2 FROM t**
- Query data and filter rows with a condition

**SELECT DISTINCT c1 FROM t**
- Query distinct rows from a table

**SELECT c1, c2 FROM t**
- Sort the result set in ascending or descending order

**SELECT c1, c2 FROM t**
- Skip offset of rows and return the next n rows

**SELECT c1, aggregate(c2) FROM t**
- Group rows using an aggregate function

**SELECT c1, aggregate(c2) FROM t**
- Filter groups using HAVING clause

### Querying from Multiple Tables

**SELECT c1, c2 FROM t1**
- Inner join t1 and t2

**SELECT c1, c2 FROM t1**
- Left join t2 on condition; Left join t1 and t2

**SELECT c1, c2 FROM t1**
- Right join t2 on condition; Right join t1 and t2

**SELECT c1, c2 FROM t1**
- Full outer join t2 on condition; Perform full outer join

**SELECT c1, c2 FROM t1**
- Produce a Cartesian product of rows in tables

**SELECT c1, c2 FROM t1**
- Another way to perform cross join

**SELECT c1, c2 FROM t1**
- Join t1 to itself using INNER JOIN clause

### Using SQL Operators

**SELECT c1, c2 FROM t1**
- Combine rows from two queries

**SELECT c1, c2 FROM t1**
- Return the intersection of two queries

**SELECT c1, c2 FROM t1**
- Subtract a result set from another result set

**SELECT c1, c2 FROM t1**
- Query rows using pattern matching `%`, `_`

**SELECT c1, c2 FROM t1**
- Query rows in a list

**SELECT c1, c2 FROM t1**
- Query rows between two values

**SELECT c1, c2 FROM t1**
- Check if values in a table is NULL or not
How does it work?
RDBMS Architecture

How does a SQL engine work?

1. **SQL Query**
   - Declarative query (from user)

2. **Relational Algebra (RA) Plan**
   - Translate to relational algebra expression

3. **Optimized RA Plan**
   - Find logically equivalent - but more cost-efficient - RA expression

4. **Execution**
   - Execute each operator of the optimized plan!
Relational Algebra allows us to translate declarative (SQL) queries into precise and optimizable expressions!
Relational Algebra (RA)

Five **basic operators**:
1. Selection: \( \sigma \)
2. Projection: \( \Pi \)
3. Cartesian Product: \( \times \)
4. Union: \( \cup \)
5. Difference: \( - \)

**Derived or auxiliary operators**:
- Intersection
- Joins: \( \Join \) (natural, equi-join, semi-join)
- Renaming: \( \rho \)

What’s an **Algebra**? Why?
- For ex, in Math
  a) \((x + y) + z\) vs \(x + y + z\)
  b) \((x + y) + 2x\) vs \((x + y + 2)x\)
- Operators and rules
  - Basic notation for operators (`+`, `-`, `*`, `/`, `^` etc.)
  - Association, commutative, ...

⇒ Why?
  - What can you reorder, simplify?
  - Express complex equations and expressions, and reason about them
Converting SFW Query to RA

Students(sid,sname,gpa)
People(ssn,sname,address)

SELECT DISTINCT gpa, address
FROM Students S, People P
WHERE gpa > 3.5 AND
    sname = pname;

\[ \Pi_{gpa, address}(\sigma_{gpa > 3.5}(S \bowtie P)) \]

How do we represent this query in RA?
RDBMS Architecture

How does a SQL engine work?

We'll look at how to then optimize these plans now.
Visualize the plan as a tree

\[ \Pi_B(R(A, B) \bowtie S(B, C)) \]

Bottom-up tree traversal = order of operation execution!
One simple plan -- “Push down” projection

What SQL query does this correspond to?

Why might we prefer this plan?

Are there any logically equivalent RA expressions?
Logical Optimization

• Heuristically, we want selections and projections to occur as early as possible in the plan
  • Terminology: “push down selections and projections”

• Intuition: We will usually have fewer tuples in a plan.
  Exceptions
  ■ Could fail if the selection condition is very expensive (e.g., run image processing algorithm)
  ■ Projection could be a waste of effort, but more rarely

⇒ Cost-based Query Optimizers (e.g., Postgres/ BigQuery/ MySQL optimizers, SparkSQL’s Catalyst)
Optimizing the SFW RA Plan
Translating to RA

\[
\Pi_{A,D} (\sigma_{A<10} (T \bowtie (R \bowtie S)))
\]

\[
\text{SELECT } R.A, T.D \\
\text{FROM } R, S, T \\
\text{WHERE } R.B = S.B \\
\text{AND } S.C = T.C \\
\text{AND } R.A < 10;
\]

\[
\begin{align*}
R(A,B) & \quad S(B,C) \quad T(C,D) \\
\Pi_{A,D} & \quad \sigma_{A<10} \quad T(C,D) \\
\end{align*}
\]
Optimizing RA Plan

\[
\Pi_{A,D}(\sigma_{A<10}(T \Join (R \Join S)))
\]

```
SELECT R.A, S.D
FROM R, S, T
WHERE R.B = S.B
AND S.C = T.C
AND R.A < 10;
```

Push down selection on A so it occurs earlier.
Optimizing RA Plan

\[ \Pi_{A,D}(T \bowtie (\sigma_{A<10}(R) \bowtie S)) \]

Push down selection on \( A \) so it occurs earlier.
Optimizing RA Plan

\[
\Pi_{A,D}(T \bowtie (\sigma_{A<10}(R) \bowtie S))
\]

Push down projection so it occurs earlier

```
SELECT R.A, S.D
FROM R, S, T
WHERE R.B = S.B
AND S.C = T.C
AND R.A < 10;
```
Optimizing RA Plan

We eliminate B earlier!

In general, when is an attribute not needed…?

\[
\Pi_{A,D} \left( T \bowtie \Pi_{A,C} (\sigma_{A<10} (R) \bowtie S) \right)
\]
Basic RA commutators

- Push **projection** through (1) **selection**, (2) **join**
- Push **selection** through (3) **selection**, (4) **projection**, (5) **join**
- *Also*: Joins can be re-ordered!

⇒ Note that this is not an exhaustive set of operations
   This covers *local re-writes*; *global re-writes* possible but much harder

This simple set of tools allows us to greatly improve the execution time of queries by optimizing RA plans!
Takeaways

• This process is called logical optimization

• Many equivalent plans used to search for “good plans”

• Relational algebra is a simple and elegant abstraction