

SQL

Part-II

Preview

SQL queries

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 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

This Lecture

- Set operators & nested queries
- Aggregation & GROUP BY

Set Operators & Nested Queries

Set algebra (reminder)

List: [1, 1, 2, 3]

Set: {1, 2, 3}

Multiset: {1, 1, 2, 3}

A **multiset** is an unordered list (or: a set with multiple duplicate instances allowed)

UNIONs

Set: $\{1, 2, 3\} \cup \{2\} = \{1, 2, 3\}$

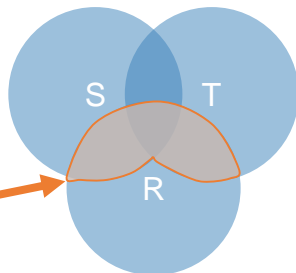
Multiset: $\{1, 1, 2, 3\} \cup \{2\} = \{1, 1, 2, 2, 3\}$

Cross-product

$$\begin{aligned} \{1, 1, 2, 3\} * \{y, z\} = \\ \{ \langle 1, y \rangle, \langle 1, y \rangle, \langle 2, y \rangle, \langle 3, y \rangle \\ \langle 1, z \rangle, \langle 1, z \rangle, \langle 2, z \rangle, \langle 3, z \rangle \\ \} \end{aligned}$$

An Unintuitive Query

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



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

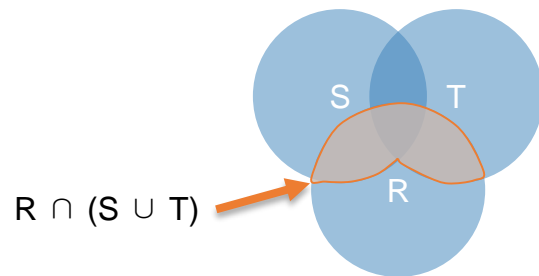
But what if $S = \phi$?

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
 1. Apply selections / conditions
 1. Apply projection

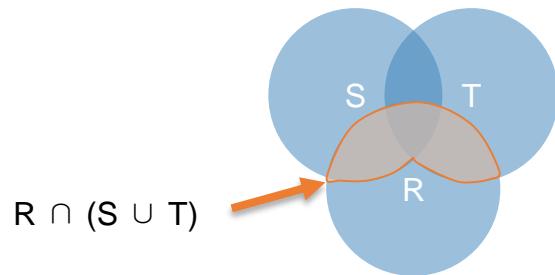


*Joins / cross-products are just **nested for loops** (in simplest implementation)!*

If-then statements!

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
```



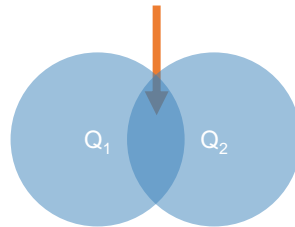
```
output = {}  
  
for r in R:  
    for s in S:  
        for t in T:  
            if r['A'] == s['A'] or r['A'] == t['A']:  
                output.add(r['A'])  
return list(output)
```

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

Explicit Set Operators: INTERSECT

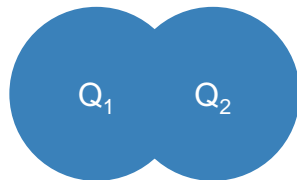
```
SELECT R.A  
FROM R, S  
WHERE R.A=S.A  
INTERSECT  
SELECT R.A  
FROM R, T  
WHERE R.A=T.A
```

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



UNION

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



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

Why aren't there duplicates?

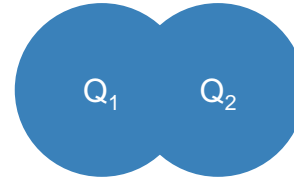
By default:
SQL retains **set** semantics for
UNIONs, INTERSECTs!

What if we want duplicates?

UNION ALL

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

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



ALL indicates Multiset operations

Nested queries: Sub-queries Return Relations

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 = 'Mickey')
```

“

- Companies making
products bought by
Mickey”

- Location of
companies?

”

Nested Queries

Are these queries equivalent?

```
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')
```

无重复项

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


有重复项

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 = '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)

Subqueries Return Relations

You can also use operations of the form:

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

ANY and ALL not supported by SQLite.

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:

- $s > \text{ALL } R$
- $s < \text{ANY } R$
- EXISTS R

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

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 !=

Nested queries as **alternatives** to INTERSECT and EXCEPT

INTERSECT and EXCEPT not in some DBMSs!

```
(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)
```

等价

```
(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 Using External Vars in Internal Subquery

Find movies whose title appears more than once.

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

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

Note the
scoping of the
variables!

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

What you will learn
about in this section

1. Aggregation operators
1. GROUP BY
1. GROUP BY: with HAVING, semantics

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

Aggregation: COUNT

- COUNT applies to duplicates, unless otherwise stated

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

Note: Same as COUNT(*). Why?

因：未去重与总行数
一样

We probably want:

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

More Examples

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

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

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

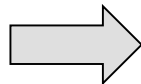
What do these mean?

Simple Aggregations

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

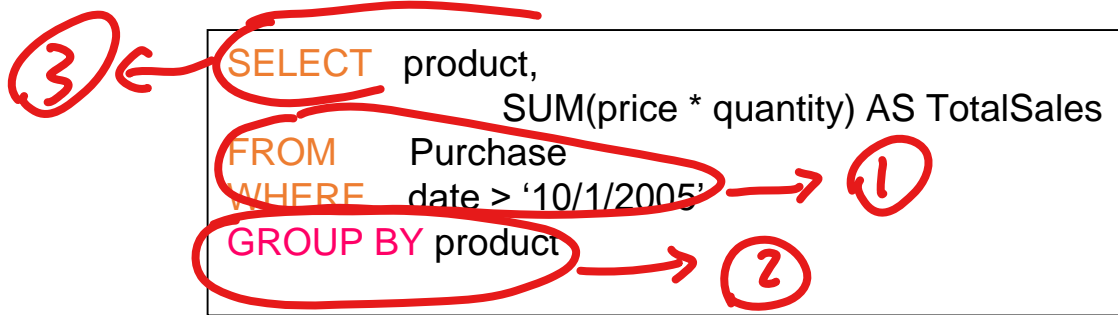
```
Purchase(product, date, price, quantity)
```

```
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.

Let's see what this means...

Grouping and Aggregation



A SQL query is shown within a rectangular box. The query is: `SELECT product, SUM(price * quantity) AS TotalSales FROM Purchase WHERE date > '10/1/2005' GROUP BY product`. Handwritten red annotations are present: a circled '3' with an arrow pointing to the `SELECT` clause; a circled '1' with an arrow pointing to the `WHERE` clause; and a circled '2' with an arrow pointing to the `GROUP BY` clause. The words `SELECT`, `FROM`, and `WHERE` are in orange, while `GROUP BY` is in pink.

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

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

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



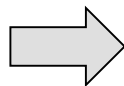
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

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



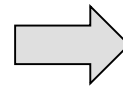
Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

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

GROUP BY v.s. Nested Queries

方①

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

✓ tx

方②

```
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'
```

HAVING Clause

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

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

HAVING clauses contains conditions on **aggregates**

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

General form of Grouping and Aggregation

SELECT	S
FROM	R_1, \dots, R_n
WHERE	C_1
GROUP BY	a_1, \dots, a_k
HAVING	C_2

Evaluation steps:

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

**THANK
YOU!**