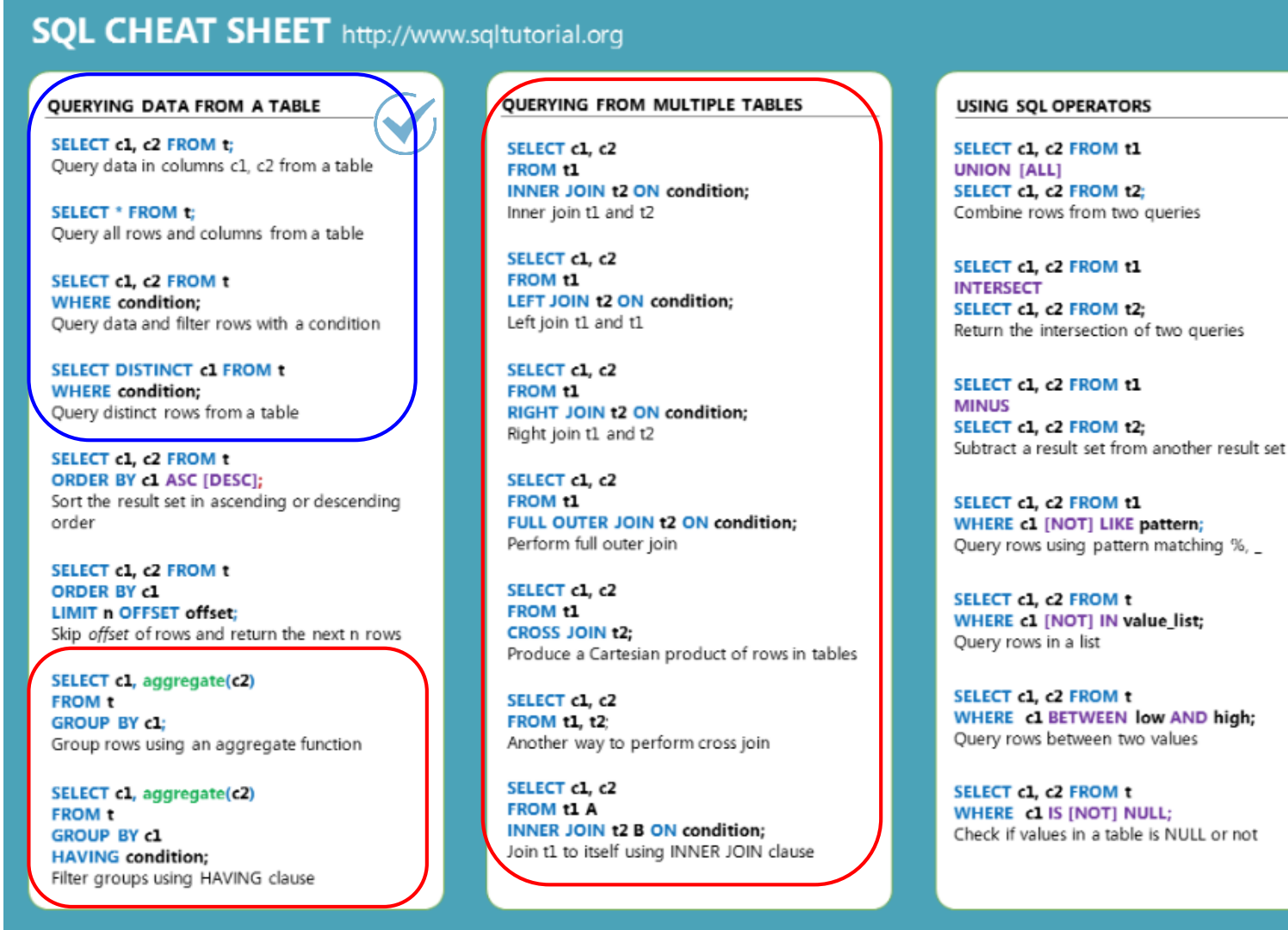
SQL

Part-II

Preview

SQL queries

sqltutorial.org/sql-cheat-sheet



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

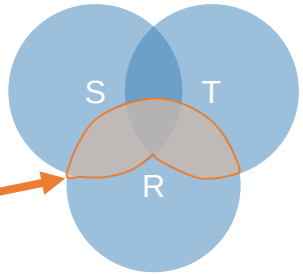
Set: {1, 2, 3} U { 2 } = { 1, 2, 3 }
Multiset: {1, 1, 2, 3} U { 2 } = { 1, 1, 2, 2, 3 }

Cross-product

{1, 1, 2, 3} * { y, z } =
{ <1, y>, <1, y>, <2, y>, <3, y>
<1, z>, <1, z>, <2, z>, <3, z>
}

An Unintuitive Query

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



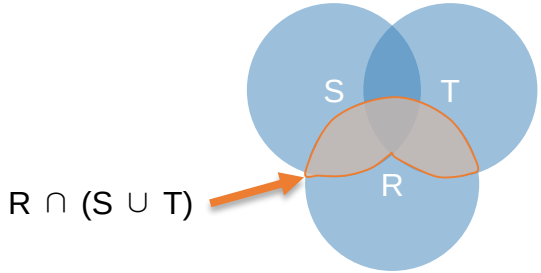
Computes R ∩ (S ∪ T)

But what if S = ϕ ?

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

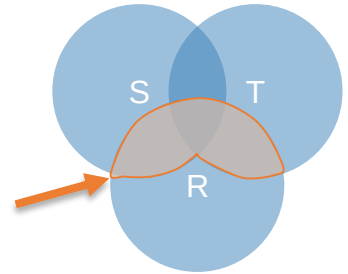
Joins / cross-products are just **nested for loops** (in simplest implementation)!

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
```

$R \cap (S \cup T)$



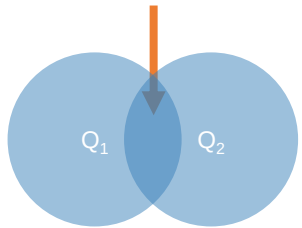

```
output = {}

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

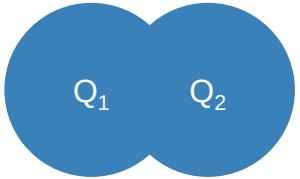
```
SELECT R.A
FROM   R, S
WHERE  R.A=S.A
INTERSECT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A
```

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



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?

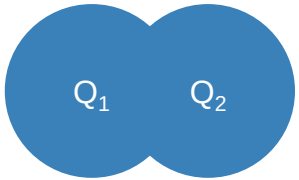
By default:
SQL retains set semantics for UNIONs, INTERSECTs!

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‘)
```

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

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 > ALL R
- s < 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)
```

```
(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?

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
```

What do these mean?

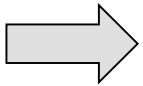
```
SELECT SUM(price * quantity)
FROM   Purchase
WHERE  product = 'bagel'
```

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

```
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

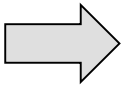
2. Group by the attributes in the GROUP BY

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



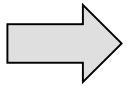
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

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



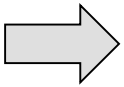
Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

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

```
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'
```

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       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 need to compute aggregates)
4. Compute aggregates in S and return the result

THANK YOU!