ICCS240 Database Management

SQL
(cont.)

Many slides in this lecture are either from or adapted from slides provided by

Theodoros Rekatsinas, UW-Madison

Kazuhiro Minami. UIUC

Multiset operations

Recall Multiset

$\lambda(X)$= *"Count of tuple in X"*
(Items not listed have implicit count 0)

Multiset X

Tuple
(1, a)
(1, a)
(1, b)
(2, c)
(2, c)
(2, c)
(1, d)
(1, d)

Equivalent representations of a <u>Multiset</u>

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	1
(2, c)	3
(1, d)	2

Note: In a set, $\forall x, \lambda(x) \in \{0,1\}$

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cap$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	2
(1, b)	0
(2, c)	2
(1, d)	0

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

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cup$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Multiset Z

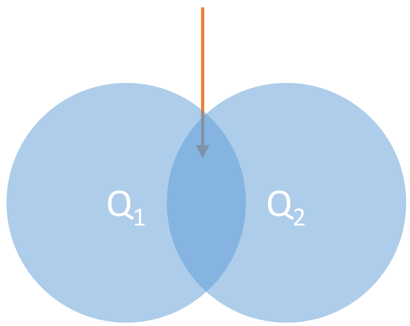
Tuple	$\lambda(Z)$
(1, a)	7
(1, b)	1
(2, c)	5
(1, d)	2

Multiset operations in SQL

(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
```

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

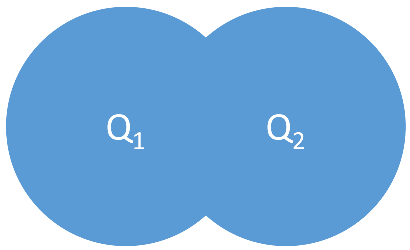


Be careful! Not all variations of SQL support INTERSECT operator.

(Explicit) *Set* Operator: UNION

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

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



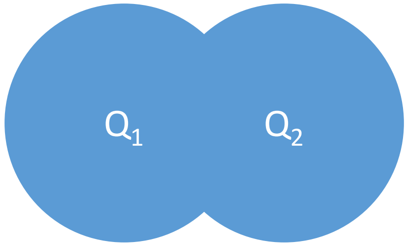
NOTE: These *set* operators return NO duplicates!

What if we want duplicates like in multiset?

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
```

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

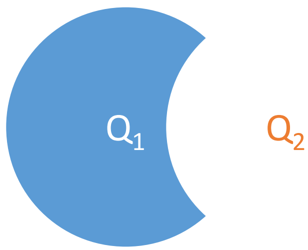


ALL indicates the Multiset disjoint union operation

(Explicit) *Set* Operator: EXCEPT

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

$$\{r.A \mid r.A = s.A\} \backslash \{r.A | r.A = t.A\}$$



What is the multiset version?

$$\boldsymbol{\lambda(Z) = \lambda(X) - \lambda(Y)}$$

For elements that are in X

Some subtle problems …

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

```
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'
```

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

What if two companies have HQ in US: BUT one has factory in China (but not US) and vice versa? What goes wrong?

Remember the semantics!

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

Example: `C JOIN P on maker=name`

```
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'
```

C.name	C.hq_city	P.pname	P.maker	P.factory_loc
X Co.	Seattle	X	X Co.	U.S.
Y Inc.	Seattle	X	Y Inc.	China

X Co has a factory in the US (but not China)
Y Inc. has a factor in China (but not US)

But Seattle is returned by the query!

We did the INTERSECT on the wrong attributes!

One Solution: 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 on 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!

Remember relational algebra is also compositional.

This property is extremely powerful!

Nested Queries: sub-queries return relations

```
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"

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!

Fix the 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

Nested Queries: sub-queries return relations

ANY and ALL are not supported by SQLite.

You can also use operations of the form:

- S > ALL R
- S < ANY 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"

Nested Queries: sub-queries return relations

ANY and ALL are not supported by SQLite.

You can also use operations of the form:

- S > ALL R
- S < ANY 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

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)
```

Grouping & AGGREGATION

Basic Aggregations

`SUM`, `AVG`, `COUNT`, `MIN`, and `MAX` can be applied to a column in a `SELECT` clause to produce that aggregation on the column.

Also, COUNT(*) counts the number of tuples.

Example: Aggregation

```
sells(bar, beer, price)
```

- Find average price of ‘Budweiser’:

```
SELECT  AVG(price)
FROM    sells
WHERE   beer = ‘Budweiser’
```

- Find number of different prices charged for ‘Budweiser’

```
SELECT COUNT(DISTINCT price)
FROM   sells
WHERE  beer = ‘Budweiser’
```

DISTINCT inside an aggregation causes duplicates to be eliminated before the aggregation.

NULL's are ignored in aggregation

NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.

But if there are no non-NULL values in a column,

then the result of the aggregation is NULL.

```
SELECT COUNT(*)
FROM   sells
WHERE  beer = 'Budweiser'
```

The number of bars that sell Budweiser.

```
SELECT COUNT(price)
FROM   sells
WHERE  beer = 'Budweiser'
```

The number of bars that sell Budweiser at *known* price.

Grouping

- We may follow a SELECT-FROM-WHERE expression by **GROUP BY** and a list of attributes

- The relation that results from the SELECT-FROM-WHERE is *grouped*

  according to the values of all those attributes,

  and any aggregation is applied only within each group.

```
sells(bar, beer, price)
```

```
SELECT beer, AVG(price)
FROM   sells
GROUP BY beer
```

Find average price for each beer

Example: Find bars that sells Bud at the least price?

```
sells(bar, beer, price)
```

```
SELECT bar, MIN(price)
FROM   sells
WHERE  beer = 'Budweiser'
GROUP BY bar
```

Does this query work as expected?

This doesn't quite work because we want to compare across bars, while here we're selecting the least price per bar.

Example: Find bars that sells Bud at the least price?

sells(bar, beer, price)

```
SELECT b     MIN(   ice)
FROM   sel
WHERE  bee      Budweiser'
GROUP BY   ar
```

```
SELECT bar
FROM   sells
WHERE  beer = 'Budweiser'
      AND price = (
          SELECT MIN(price)
          FROM sells
          WHERE beer = 'Budweiser')
```

HAVING Clauses

HAVING <condition> may follow a GROUP BY clause.

If so, the condition applies to each group, and groups not satisfying the condition are eliminated.

```
sells(bar, beer, price)
```

```
SELECT beer, AVG(price)
FROM   sells
GROUP BY beer
HAVING  COUNT(bar)>=3
```

What does this query ask for?

Requirements on HAVING conditions

- These conditions may refer to any relation or tuple-variable in the FROM clause.

- They may refer to attributes of those relations, as long as the attribute makes sense within a group; i.e., it is either:
    1. A grouping attribute, or
    2. Aggregated.

General form of Grouping & Aggregation

```
SELECT S
FROM   R1, …, Rn
WHERE  C1
GROUP BY A1, …, Ak
HAVING C2
```

S may contain attributes A1, …, Ak and/or any aggregation

but NO OTHER ATTRIBUTES

C1 is any condition on R1, …, Rn

C2 is any condition on aggregate expressions or grouping attributes

Evaluation steps

```
SELECT S
FROM   R1, …, Rn
WHERE  C1
GROUP BY A1, …, Ak
HAVING C2
```

1. Compute the `FROM-WHERE` part, obtain a table with all attributes in `R1`, …, `Rn`
2. Group by the attributes `A1`, …, `Ak`
3. Compute the aggregates in `C2` and keep only groups satisfying `C2`
4. Compute aggregates in `S`
5. Return the results

Your playtime:

Download this sample dataset:

sqlex_ddl3.sql

sqlex_data3.sql

You may execute the file in sql using command

```
source /path/to/your/file.sql;
```

Today’s Practice

Solve each of the following **using one query** (potentially with subqueries):

a) Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result

b) Find the highest salary of any instructor

c) For each department, find the highest salary of any instructor

d) Find the maximum enrollment, across all sections, in Fall 2009

e) Find the sections that had the maximum enrollment in Fall 2009

f) Find the IDs and names of all students who have not taken any course offering before Fall 2009

Views & Triggers

Views are like the temporary relations we declared and reused so often in relational algebra to make things easier.

```
CREATE VIEW view_name AS query;
```

Then use `view_name` in queries like a “real” relation, **even though its data isn’t actually stored**. It’s a virtual relation!

A relation whose instance is really stored in the database is a **base** table.

The DBMS replaces the view name by its definition at run time, essentially.

Example: view of my ICCS240 students

Enrollments(SID, course, semester, year, grade)

```
CREATE VIEW iccs240students AS
SELECT SID, grade
FROM Enrollments
WHERE course=‘ICCS240’ AND year=2020 AND semester=‘II’;
```

Sometimes it make sense to modify the tuples in a view.

Employee(ssn, name, department, project, alary)

```
CREATE VIEW  Developers AS
   SELECT name, project, department
   FROM   Employee
   WHERE  department = 'Development';
```

```
INSERT INTO  Developers
VALUES('Joe', 'Optimizer', 'Development');
```

Result:

```
INSERT INTO  Employee
VALUES(NULL, 'Joe', 'Development', 'Optimizer', NULL);
```

Warning: such insertions are prohibited if the null fields are part of the primary key.

Other times, the modification make no sense.

Employee(ssn, name, department, project, alary)

```
CREATE VIEW  Developers AS
   SELECT name, project, department
   FROM   Employee
   WHERE  department = 'Development';
```

```
INSERT INTO  Developers
VALUES('Joe', 'Optimizer');
```

Result:

```
INSERT INTO  Employee
VALUES(NULL, 'Joe', NULL, 'Optimizer', NULL);
```

Warning: Joe is NOT in the view, and your users are VERY confused!

Triggers are great for implementing view updates.

We cannot insert into view **`Developers(name,project)`**.

But we can use an INSTEAD OF trigger to turn a `(name, project)` tuple into an insertion of a tuple `(name, `**`` `Development' ``**`, project)` to `Employee`.

Example: Updating Developers

Employee(ssn, name, department, project, alary)

```
CREATE VIEW  Developers AS
   SELECT name, project, department
   FROM   Employee
   WHERE  department = 'Development';
```

If we make the following insertion:

```
INSERT INTO  Developers
VALUES('Joe', 'Optimizer');
```

This must be "Development"

It becomes:

```
INSERT INTO  Employee
VALUES(NULL, 'Joe', NULL, 'Optimizer', NULL);
```

So, have a TRIGGER to handle the tuple before insertion!

Allow insertions into Developers

```
CREATE TRIGGER AllowInsert
  INSTEAD OF INSERT ON Developers
  REFERENCING NEW ROW AS new
  FOR EACH ROW
  BEGIN
     INSERT INTO Empolyees(name, department,
      project) VALUES(new.name, `Development',
      new.project);
  END;
```

In general, a TRIGGER is an “ECA” rule:

- When **E**VENT occurs
- If **C**ONDITION doesn’t hold
- Then do **A**CTION

E.g., an INSERT / DELETE / UPDATE to relation *R*

Any SQL Boolean condition

Any SQL statements

Example: If someone inserts an unknown beer into Sells(bar,beer,price), add it to Beers with a NULL manufacturer.

```
CREATE TRIGGER BeerTrig
 AFTER INSERT ON Sells
 REFERENCING NEW ROW AS NewTuple
 FOR EACH ROW
 WHEN (NewTuple.beer NOT IN
     (SELECT name FROM Beers))
 INSERT INTO Beers(name)
     VALUES(NewTuple.beer);
```

The event

The condition

The action

SQL: Summary

- SQL provides a high-level declarative language for manipulating data
  - An attempt to implement Relational Algebra
- The workhorse is the Select-From-Where (SFW) block
- Set operators are powerful but have some subtleties
- Aggregation & grouping are also useful, esp. in building summary
- SQL is compositional, so powerful, nested queries are also allowed
- Views are like temporary relations we declared and reused to make things easier
- Triggers are special stored procedures that are executed automatically in response to certain actions, calls, events, etc.