Introduction to Data Management CSE 344

Lecture 6: Nested Queries in SQL

Friday June 30

Announcements

- Webquiz 2 is due on Sunday
  - Webquiz 3 will go out
- Homework 2 is due on Wednesday
  - Homework 3 uses Microsoft Azure Cloud services
    - (no more sqlite!)
  - Look for instructions on setting up your Azure Account next week.
    - Even if you already have a Microsoft Account you will create a new one for this class

Lecture Goals

- Today we will learn how to write (even) more powerful SQL queries

- Reading: Ch. 6.3

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, COUNT(*)
FROM   Product JOIN Purchase ON
       Product.name = Purchase.prodName
GROUP BY Product.name
```

What Changes?

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Projecting Columns with Grouping

```
SELECT product,
       max(quantity)
FROM   Purchase
GROUP BY product
```

```
SELECT   product, quantity
FROM     Purchase
GROUP BY product
```

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

Product
Bagel
Banana

+

Quantity
20
20
50
10
10

Can’t project a non-grouped / non-aggregated column!

Subqueries

- A subquery is a SQL query nested inside a larger query
- Such inner-outer queries are called nested queries
- A subquery may occur in:
  - A `SELECT` clause
  - A `FROM` clause
  - A `WHERE` clause

- Rule of thumb: avoid writing nested queries when possible
  - But sometimes it's impossible, as we will see

SQL Subqueries

Just because you can use them doesn’t mean you should.

Subqueries…

- Can appear as computed values in a `SELECT` clause
- Can appear in `FROM` clauses and aliased using a tuple variable that represents the tuples in the result of the subquery
- Can return a single constant to be compared with another value in a `WHERE` clause
- Can return relations to be used in `WHERE` clauses

1. Subqueries in SELECT

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

For each product return the city where it is manufactured

```
SELECT X.pname, (SELECT Y.city
                 FROM Company Y
                 WHERE Y.cid=X.cid) as City
FROM  Product X
```

What happens if the subquery returns more than one city?

We get a runtime error
… as usual, SQLite simply ignores the extra values

1. Subqueries in SELECT

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

For each product return the city where it is manufactured

```
SELECT X.pname, (SELECT Y.city
                 FROM Company Y
                 WHERE Y.cid=X.cid) as City
FROM  Product X
```

"correlated subquery"

Correlated Subquery: a sub-query that uses values from the outer query. In this case the inner query has to be executed for every row of outer query.

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

1. Subqueries in SELECT

Whenever possible, don’t use a nested queries:

```
SELECT X.pname, (SELECT Y.city
                 FROM Company Y
                 WHERE Y.cid=X.cid) as City
FROM  Product X
```

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

1. Subqueries in SELECT

Whenever possible, don’t use a nested queries:

```
SELECT X.pname, (SELECT Y.city
                 FROM Company Y
                 WHERE Y.cid=X.cid) as City
FROM  Product X
```

```
SELECT X.pname, Y.city
FROM   Product X, Company Y
WHERE  X.cid=Y.cid
```

We have “unnested” the query

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

1. Subqueries in SELECT

Compute the number of products made by each company

```
SELECT DISTINCT C.cname, (SELECT count(*)
                          FROM Product P
                          WHERE P.cid=C.cid)
FROM  Company C
```

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

1. Subqueries in SELECT

Compute the number of products made by each company

```
SELECT DISTINCT C.cname, (SELECT count(*)
                          FROM Product P
                          WHERE P.cid=C.cid)
FROM  Company C
```

Better: we can unnest using a `GROUP BY`

```
SELECT C.cname, count(*)
FROM   Company C, Product P
WHERE  C.cid=P.cid
GROUP BY C.cname
```

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

1. Subqueries in SELECT

But are these really equivalent?

```
SELECT DISTINCT C.cname, (SELECT count(*)
                          FROM Product P
                          WHERE P.cid=C.cid)
FROM  Company C
```

```
SELECT C.cname, count(*)
FROM   Company C, Product P
WHERE  C.cid=P.cid
GROUP BY C.cname
```

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

1. Subqueries in SELECT

But are these really equivalent?

```
SELECT DISTINCT C.cname, (SELECT count(*)
                          FROM Product P
                          WHERE P.cid=C.cid)
FROM  Company C
```

```
SELECT C.cname, count(*)
FROM   Company C, Product P
WHERE  C.cid=P.cid
GROUP BY C.cname
```

No! Different results if a company has no products

```
SELECT C.cname, count(pname)
FROM Company C LEFT OUTER JOIN Product P
ON   C.cid=P.cid
GROUP BY C.cname
```

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

2. Subqueries in FROM

Find all products whose prices is > 20 and < 500

```
SELECT X.pname
FROM (SELECT *
      FROM Product AS Y
      WHERE price > 20) as X
WHERE X.price < 500
```

Side note: This is not a correlated subquery. (why?)

Try unnest this query !

2. Subqueries in FROM

Use the result of the inner query as a new table in the FROM clause.

- We will see that sometimes we really need a subquery
  - will see most compelling examples next lecture
  - in that case, we can put it in the FROM clause

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

3. Subqueries in WHERE

Find all companies that make **some** products with price < 100

Existential quantifiers: there exists an x such that P(x)

$$\exists x \in \mathbf{X}\, P(x)$$

Useful Keyword: EXISTS , IN , ANY , ALL

Using EXISTS:

```
SELECT DISTINCT  C.cname
FROM     Company C
WHERE  EXISTS (SELECT *
                          FROM Product P
                          WHERE C.cid = P.cid and P.price < 100)
```

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

3. Subqueries in WHERE

Find all companies that make **some** products with price < 200

Existential quantifiers

Using IN

```
SELECT DISTINCT  C.cname
FROM     Company C
WHERE C.cid IN (SELECT P.cid
                FROM Product P
                WHERE P.price < 100)
```

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

3. Subqueries in WHERE

Find all companies that make **some** products with price < 200

Existential quantifiers

Using ANY:

```
SELECT DISTINCT  C.cname
FROM  Company C
WHERE 200 > ANY (SELECT price
                 FROM Product P
                 WHERE P.cid = C.cid)
```

Not supported in sqlite

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

3. Subqueries in WHERE

Find all companies that make **some** products with price < 200

Existential quantifiers

Now let’s unnest it:

```
SELECT DISTINCT  C.cname
FROM   Company C, Product P
WHERE  C.cid = P.cid and P.price < 200
```

Existential quantifiers are easy!

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

3. Subqueries in WHERE

Find all companies where **all** their products have price < 100

Same as:

Find all companies that make **only** products with price < 100

Universal quantifiers: for all x, P(x) holds

$$\forall x \in \mathbf{X}\; P(x)$$

Universal quantifiers are hard !

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

3. Subqueries in WHERE

Find all companies where all their products have price < 100

Step 1: Find *the other* companies: i.e. with some product >= 100

```
SELECT DISTINCT  C.cname
FROM     Company C
WHERE  C.cid IN (SELECT P.cid
                          FROM Product P
                          WHERE P.price >= 100)
```

Step 2: Find all companies where all their products have price < 1

```
SELECT DISTINCT  C.cname
FROM     Company C
WHERE  C.cid NOT IN (SELECT P.cid
                                FROM Product P
                                WHERE P.price >= 100)
```

Product (pname, price, cid)
Company(cid, cname, city)

3. Subqueries in WHERE

Find all companies where all their products have price < 100

Universal quantifiers

Using EXISTS:

```
SELECT DISTINCT  C.cname
FROM     Company C
WHERE NOT EXISTS (SELECT *
                  FROM Product P
                  WHERE P.cid = C.cid and P.price >= 100)
```

Product (pname, price, cid)
Company(cid, cname, city)

3. Subqueries in WHERE

Find all companies where all their products have price < 100

Universal quantifiers

Using ALL:

```
SELECT DISTINCT  C.cname
FROM     Company C
WHERE 100 >= ALL  (SELECT price
                   FROM Product P
                   WHERE P.cid = C.cid)
```

Not supported
in sqlite

Question for Database Theory Fans and their Friends

- Can we unnest the *universal quantifier* query?
- We need to first discuss the concept of *monotonicity*

```
Product (pname,  price, cid)
Company (cid, cname, city)
```

Monotone Queries

- Definition: A query Q is monotone if:
  - Whenever we add tuples to one or more input tables, the answer to the query will not lose any of the tuples
- Similar to Monotone Functions

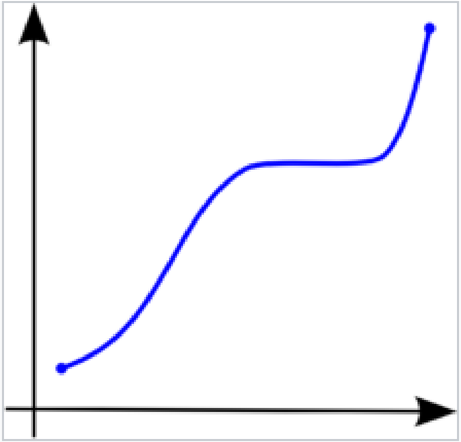


Figure 1. A monotonically increasing function. It is strictly increasing on the left and right while just *monotonic* (unchanging) in the middle.

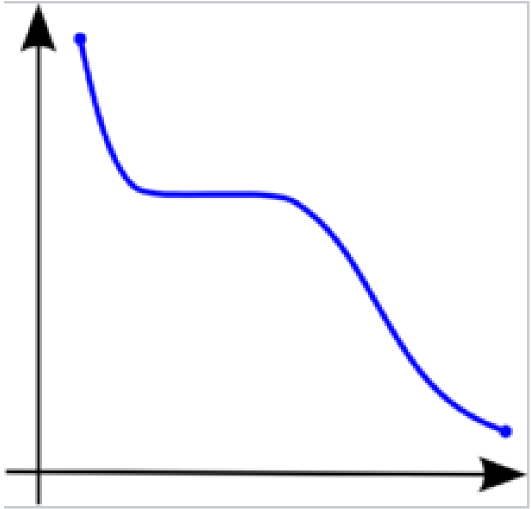


Figure 2. A monotonically decreasing function

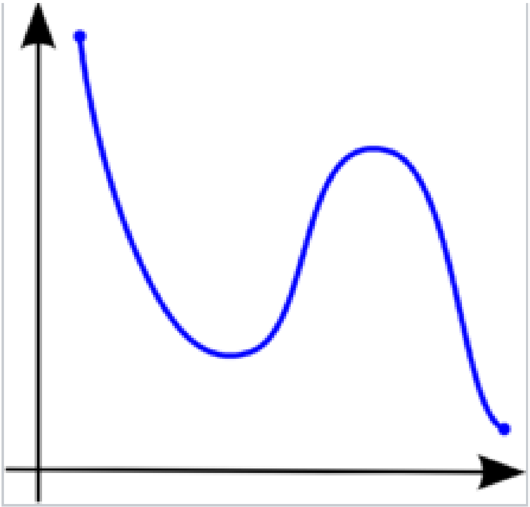


Figure 3. A function that is not monotonic

Product (pname, price, cid)
Company (cid, cname, city)

Monotone Queries

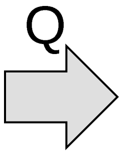
- Definition: A query Q is monotone if:
  - Whenever we add tuples to one or more input tables, the answer to the query will not lose any of the tuples

Product

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c004
Camera	149.99	c003

Company

cid	cname	city
c002	Sunworks	Bonn
c001	DB Inc.	Lyon
c003	Builder	Lodtz



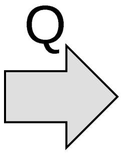
A	B
Gizmo	Lyon
Camera	Lodtz

Product

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c004
Camera	149.99	c003
iPad	499.99	c001

Company

cid	cname	city
c002	Sunworks	Bonn
c001	DB Inc.	Lyon
c003	Builder	Lodtz



A	B
Gizmo	Lyon
Camera	Lodtz
iPad	Lyon

Monotone Queries

- Theorem: If Q is a SELECT-FROM-WHERE query that does not have subqueries, and no aggregates, then it is monotone.

Monotone Queries

- Theorem: If Q is a SELECT-FROM-WHERE query that does not have subqueries, and no aggregates, then it is monotone.

- Proof. We use the nested loop semantics: if we insert a tuple in a relation R_i, this will not remove any tuples from the answer

```
SELECT $a_1$, $a_2$, …, $a_k$
FROM   $R_1$ AS $x_1$, $R_2$ AS $x_2$, …, $R_n$ AS $x_n$
WHERE  Conditions
```

```
for $x_1$ in $R_1$ do
  for $x_2$ in $R_2$ do
    …
    for $x_n$ in $R_n$ do
      if Conditions
        output ($a_1$,…,$a_k$)
```

Product (pname, price, cid)
Company (cid, cname, city)

Monotone Queries

- The query:

Find all companies where all their products have price < 100

is not monotone

pname	price	cid
Gizmo	19.99	c001

cid	cname	city
c001	Sunworks	Bonn

⇒

cname
Sunworks

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c001

cid	cname	city
c001	Sunworks	Bonn

⇒

cname

- Consequence: we cannot write it as a SELECT-FROM-WHERE query without nested subqueries

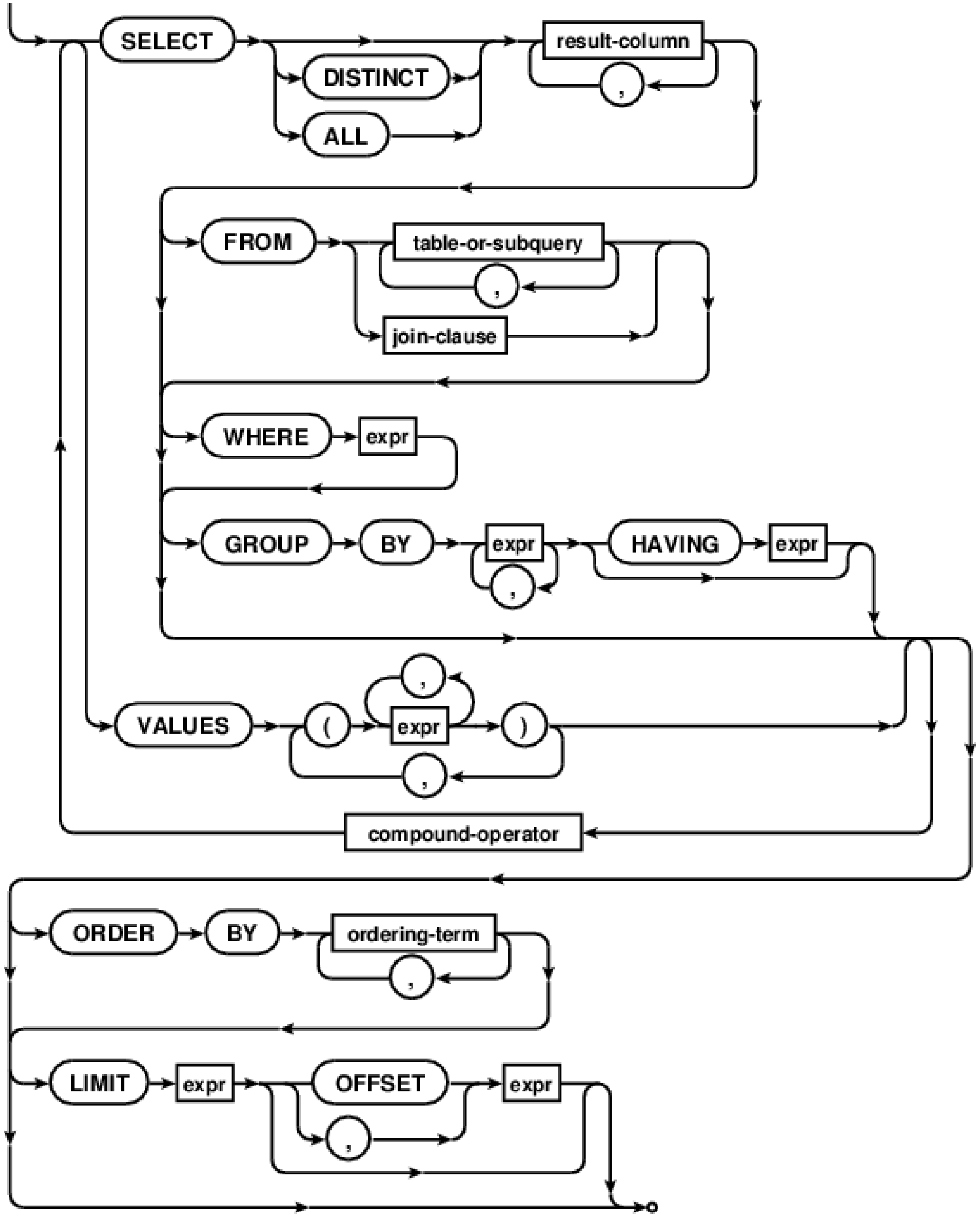
Queries that must be nested

- Queries with universal quantifiers or with negation

- Queries that use aggregates in certain ways
  - `sum(..)` and `count(*)` are NOT monotone, because they do not satisfy set containment
  - `select count(*) from R` is not monotone!

That is, cannot be SFW queries

SQLite SELECT https://sqlite.org/lang_select.html


SELECT
DISTINCT
ALL
result-column
,
FROM
table-or-subquery
,
join-clause
WHERE
expr
GROUP
BY
expr
,
HAVING
expr
VALUES
(
,
expr
)
,
compound-operator
ORDER
BY
ordering-term
,
LIMIT
expr
OFFSET
,
expr