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 nested queries when possible
  - But sometimes it’s impossible, as we will see

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 (and SQLite simply ignores the extra values…)

Simple Aggregations

Five basic aggregate operations in SQL

- `select count(*) from Purchase`  
- `select sum(quantity) from Purchase`  
- `select avg(price) from Purchase`  
- `select max(quantity) from Purchase`  
- `select min(quantity) from Purchase`

Except count, all aggregations apply to a single attribute
Including Empty Groups

```
SELECT x.manufacturer, count(*)
FROM Product x LEFT OUTER JOIN Purchase y
ON x.pname = y.product
GROUP BY x.manufacturer
```

<table>
<thead>
<tr>
<th>pname</th>
<th>manufacturer</th>
<th>product</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera</td>
<td>Canon</td>
<td>Camera</td>
<td>150</td>
</tr>
<tr>
<td>Camera</td>
<td>Hitachi</td>
<td>Camera</td>
<td>300</td>
</tr>
<tr>
<td>OneClick</td>
<td>Hitachi</td>
<td>Camera</td>
<td>180</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>Gizmo</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**Final results**

<table>
<thead>
<tr>
<th>manufacturer</th>
<th>Count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon</td>
<td>2</td>
</tr>
<tr>
<td>Hitachi</td>
<td>1</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>1</td>
</tr>
</tbody>
</table>

```
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
```

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

Recall: count of an empty table is 0

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
```

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
```

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!

A subquery whose result we called myTable

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

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Existential quantifiers

Find all companies that make some products with price < 200
3. Subqueries in WHERE

Find all companies that make some products with price < 200

Using EXISTS:

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

Existential quantifiers

Existential quantifiers are easy!

Not supported in sqlite

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!
3. Subqueries in WHERE
Find all companies s.t. all their products have price < 200
same as:
Find all companies that make only products with price < 200

Universal quantifiers are hard! 😊

3. Subqueries in WHERE
Find all companies s.t. all their products have price < 200

1. Find the other companies that make some product ≥ 200

2. Find all companies s.t. all their products have price < 200

Using EXISTS:

Universal quantifiers
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

\[ x \iff \forall y \in Y \]

Using ALL:

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

Question for Database Theory

Fans and their Friends

- Can we unnest the universal quantifier query?

- We need to first discuss the concept of monotonicity

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

So far it looks monotone...)
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.

**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.

**Consequence**: If a query is not monotonic, then 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 must be nested

• Queries with universal quantifiers or with negation
• Queries that use aggregates in certain ways
  – \( \text{sum}(..) \) and \( \text{count}(*) \) are NOT monotone, because they do not satisfy set containment
  – \( \text{select count}(*) \) from \( R \) is not monotone!

SQL Idioms

Finding Witnesses

For each city, find the most expensive product made in that city

Product \((\text{pname}, \text{price}, \text{cid})\)
Company \((\text{cid}, \text{name}, \text{city})\)

Finding Witnesses

To find the witnesses, compute the maximum price in a subquery (in FROM or in WITH)

\[
\text{WITH CityMax AS (}
\begin{align*}
\text{SELECT} & \quad \text{x.city, max(y.price)} \\
\text{FROM} & \quad \text{Company x, Product y} \\
\text{WHERE} & \quad \text{x.cid = y.cid} \\
\text{GROUP BY} & \quad \text{x.city}
\end{align*}
\)
\]

\[
\text{SELECT DISTINCT u.city, v.pname, v.price} \\
\text{FROM Company u, Product v, CityMax w} \\
\text{WHERE} \quad \text{u.cid = v.cid} \\
\text{and u.city = w.city} \\
\text{and v.price = w.maxprice;}
\]
Finding Witnesses

To find the witnesses, compute the maximum price in a subquery (in FROM or in WITH):

```sql
SELECT DISTINCT u.city, v.pname, v.price
FROM Company u, Product v,
    (SELECT x.city, max(y.price) as maxprice
     FROM Company x, Product y
     WHERE x.cid = y.cid
     GROUP BY x.city) w
WHERE u.cid = v.cid
    and u.city = w.city
    and v.price = w.maxprice;
```

Or we can use a subquery in where clause:

```sql
SELECT u.city, v.pname, v.price
FROM Company u, Product v
WHERE u.cid = v.cid
    and v.price >= ALL (SELECT y.price
                          FROM Company x, Product y
                          WHERE u.city = x.city
                          and x.cid = y.cid);
```

There is a more concise solution here:

```sql
SELECT u.city, v.pname, v.price
FROM Company u, Product v, Company x, Product y
WHERE u.cid = v.cid
    and u.city = x.city
    and v.price = max(y.price)
GROUP BY u.city, v.pname, v.price
HAVING v.price = max(y.price);
```

SQL: Our first language for the relational model

- Projections
- Selections
- Joins (inner and outer)
- Inserts, updates, and deletes
- Aggregates
- Grouping
- Ordering
- Nested queries