Lecture Overview

1. Aggregation & GROUP BY

2. Set operators & nested queries

3. Advanced SQL-izing (NULL, Outer Joins, etc.)
AGGREGATION, GROUP BY AND HAVING CLAUSE
Aggregation

- SQL supports several **aggregation** operations:
  - SUM, COUNT, MIN, MAX, AVG

```sql
SELECT AVG(price)
FROM   Product
WHERE  maker = 'Toyota'
```

```sql
SELECT COUNT(*)
FROM   Product
WHERE  year > 1995
```

Except for COUNT, all aggregations apply to a single attribute!
Aggregation: COUNT

COUNT counts the number of tuples including duplicates.

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

We probably want:

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

Note: Same as COUNT(*). Why?
More Examples

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

SELECT SUM(price * quantity)
FROM   Purchase

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

### Purchase

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
</tbody>
</table>

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

50 (= 1*20 + 1.50*20)
Grouping and Aggregation

Find total sales after 10/1 per product.

Note: Be very careful with dates! Use date/time related functions!

Purchase(product, date, price, quantity)

```
SELECT product,
       SUM(price * quantity) AS TotalSales
FROM Purchase
WHERE date > '10/01'
GROUP BY product
```

Let’s see what this means...
Grouping and Aggregation

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 > '10/01'
GROUP BY product
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
2. Group by the attributes in the **GROUP BY**

```sql
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/01'
GROUP BY product
```
3. Compute the **SELECT** clause: grouped attributes and aggregates

```
SELECT  product, SUM(price*quantity) AS TotalSales
FROM    Purchase
WHERE   date > '10/01'
GROUP BY product
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>
Activity

1) What do the next two queries calculate?

```sql
SELECT SUM(price) AS total, SUM(price) * 1.08 AS totalPlusTax
FROM Product pr
JOIN Purchase p ON pr.PName = p.product
WHERE p.buyer = 'Joe Blow'
```

```sql
SELECT p.buyer, SUM(price) AS total, SUM(price) * 1.08 AS totalPlusTax
FROM Product pr
JOIN Purchase p ON pr.PName = p.product
GROUP BY p.buyer
ORDER BY 1
```

2) Write a query to find the price of the most expensive product in each category.
HAVING Clause

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

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

HAVING clauses contain conditions on aggregates.

Whereas WHERE clauses condition on individual tuples...
General form of Grouping and Aggregation

\[
\begin{align*}
\text{SELECT} & \quad S \\
\text{FROM} & \quad R_1,\ldots,R_n \\
\text{WHERE} & \quad C_1 \\
\text{GROUP BY} & \quad a_1,\ldots,a_k \\
\text{HAVING} & \quad C_2
\end{align*}
\]

• \( S = \) Can ONLY contain attributes \( a_1,\ldots,a_k \) and/or aggregates over other attributes

• \( C_1 = \) is any condition on the attributes in \( R_1,\ldots,R_n \)

• \( C_2 = \) is any condition on the aggregate expressions

Why?
General form of Grouping and Aggregation

```
SELECT S
FROM R_1,...,R_n
WHERE C_1
GROUP BY a_1,...,a_k
HAVING C_2
```

Evaluation steps:

1. Evaluate **FROM-WHERE**: apply condition $C_1$ on the attributes in $R_1,...,R_n$
2. **GROUP BY** the attributes $a_1,...,a_k$
3. **Apply condition** $C_2$ to each group (may have aggregates)
4. Compute aggregates in $S$ and do projection (**SELECT**)
Activity

1) What does this query do?

```sql
SELECT p.buyer, SUM(price) AS total, COUNT(*) AS purchases
FROM Product pr
JOIN Purchase p ON pr.PName = p.product
GROUP BY p.buyer
HAVING purchases > 2
ORDER BY 1
```

2) What is the revenue per product in the DB?
SET OPERATORS & NESTED QUERIES
Multisets

Multiset X

<table>
<thead>
<tr>
<th>Tuple</th>
<th>( \lambda(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, a)</td>
<td>2</td>
</tr>
<tr>
<td>(1, a)</td>
<td>2</td>
</tr>
<tr>
<td>(1, b)</td>
<td>1</td>
</tr>
<tr>
<td>(2, c)</td>
<td>3</td>
</tr>
<tr>
<td>(2, c)</td>
<td>3</td>
</tr>
<tr>
<td>(2, c)</td>
<td>3</td>
</tr>
<tr>
<td>(1, d)</td>
<td>2</td>
</tr>
</tbody>
</table>

Equivalent Representations of a Multiset

\( \lambda(X) = \) “Count of tuple in X” (Items not listed have implicit count 0)

Note: In a set all counts are \{0, 1\}. 
Explicit Set Operators: INTERSECT

**INTERSECT** clause/operator is used to combine two SELECT statements, but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.

**Example:** Find all departments with an ID >= 25 that do not have an employee by the name of Anderson.

```sql
SELECT department_id
FROM departments
WHERE department_id >= 25
INTERSECT
SELECT department_id
FROM employees
WHERE last_name <> 'Anderson';
```
**UNION**

The **UNION** operator is used to combine the result sets of 2 or more SELECT statements. It *removes duplicate rows* between the various SELECT statements.

**Example:** Selects all the different cities (only distinct values) for customers and suppliers.

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City
```

Result does not contain duplicates!
UNION ALL

Same as UNION, but it does not remove duplicate rows. So we could later count how many customers/suppliers we have in each city.

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
ORDER BY City
```

**ALL indicates Multiset operations!**
The **EXCEPT** operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement.

**Example:** Find cities where we have customers, but no suppliers.

```sql
SELECT City FROM Customers
EXCEPT
SELECT City FROM Suppliers
ORDER BY City
```
NESTED QUERIES
High-level note on nested queries

- We can do nested queries because SQL is *compositional*:
  - Everything (inputs / outputs) is represented as multisets (a relation). The output of one query can thus be used as the input to another (nesting)!

- This is **extremely** powerful!
Nested Queries

```
SELECT c.country
FROM   Company c
WHERE  c.name  IN (
    SELECT pr.manufacturer
    FROM   Purchase p, Product pr
    WHERE  p.product = pr.name
    AND p.buyer = 'Joe Blow')
```

“Countries where one can find companies that manufacture products bought by Joe Blow”
Nested Queries

Is the query equivalent to this 3-way join?

```
SELECT c.country
FROM   Company c,
       Product pr,
       Purchase p
WHERE  c.name = pr.manufacturer
       AND  pr.name = p.product
       AND  p.buyer = 'Joe Blow'
```

“Cities where one can find companies that manufacture products bought by Joe Blow”

Beware of duplicates!
Nested Queries

```
SELECT DISTINCT c.country
FROM   Company c,
       Product pr,
       Purchase p
WHERE  c.name = pr.manufacturer
       AND  pr.name = p.product
       AND  p.buyer = 'Joe Blow'
```

```
SELECT DISTINCT c.country
FROM   Company c
WHERE  c.name  IN (
    SELECT pr.manufacturer
    FROM   Purchase p, Product pr
    WHERE  p.product = pr.name
    AND    p.buyer = 'Joe Blow'
)
```

Now they are equivalent
Subqueries Returning Relations

You can also use operations of the form:

- \( s \ <\text{operator}\> \ \text{ALL} \ R \)
- \( s \ <\text{operator}\> \ \text{ANY} \ R \)
- \( \text{EXISTS} \ R \)

Ex: Product(name, price, category, maker)

```
SELECT name
FROM    Product
WHERE price > ALL(
    SELECT price
    FROM    Product
    WHERE maker = 'Gizmo-Works')
```

SQLite: ANY and ALL are not supported!
They can be typically implemented using max/min and grouping.

EXISTS is supported.

Find products that are more expensive than all those produced by “Gizmo-Works”
Subqueries Returning Relations

You can also use operations of the form:

- \( s \ <\text{operator}> \ \text{ALL} \ R \)
- \( s \ <\text{operator}> \ \text{ANY} \ R \)
- \( \text{EXISTS} \ R \)

Ex:

\[
\text{Product(name, price, category, maker)}
\]

\[
\text{SELECT p1.name} \\
\text{FROM Product p1} \\
\text{WHERE p1.maker = ‘Gizmo-Works’} \\
\text{AND EXISTS (} \\
\text{SELECT p2.name} \\
\text{FROM Product p2} \\
\text{WHERE p2.maker <> ‘Gizmo-Works’} \\
\text{AND p1.name = p2.name)}
\]

Find ‘copycat’ products, i.e. products made by competitors with the same names as products made by “Gizmo-Works”

\(<>\) means \(!=\) (not equal)
Correlated Queries

A **correlated subquery** (or synchronized subquery) is a subquery (a query nested inside another query) that uses values from the outer query. Because the subquery is evaluated once for each row processed by the outer query, it can be inefficient.

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

Find movies whose title appears more than once (in different years).
Complex Correlated Query

Product(name, price, category, maker, year)

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972.

Can be very powerful but is much harder to optimize. It is best to avoid correlated queries.
Subqueries in other places

SELECT *
FROM (SELECT product, COUNT(product) AS count
     FROM Purchase GROUP BY product)
WHERE count > 2

SELECT *, (SELECT count(*) FROM Product p1
           WHERE p1.category = p2.category) AS '# Prod. in Cat.'
FROM Product p2

Subqueries can appear wherever a table or a value is needed.
Basic SQL Summary

• SQL provides a high-level declarative language for manipulating data (DML)

• Set operators are powerful but have some subtleties (duplicates).

• Powerful, nested queries also allowed.
Activity

What products did Joe Blow purchase and how much were they?

1) Solve as a multiple join.
2) Solve as a correlated subquery (in SELECT).
ADVANCED SQL: NULLS, CASTING AND OUTER JOINS
NULL Values

• Whenever we do not have a value, we can use NULL

• Can mean many things:
  – Value does not exist
  – Value exists but is unknown (n/a, not available)
  – Value not applicable

• The schema specifies for each attribute if it can be null (*nullable* attribute) or not with NOT NULL
Null Values and Operators

For numerical operations, NULL → NULL:
- If x = NULL then 4*(3-x)/7 is also NULL

For boolean operations, in SQL there are three values:

FALSE = 0
TRUE = 1
UNKNOWN

If x = NULL then x = ‘Joe’ is UNKNOWN

Note: comparison in SQL is a single ‘=’

SQLite does not have a boolean datatype. It uses Integer instead!
Try:
• SELECT 2>1
• SELECT 2>NULL
• SELECT 1+NULL
Null Values in the WHERE Clause

```sql
SELECT *
FROM   Person
WHERE  (age < 25)
       AND (height > 6 AND weight > 190)
```

Will not return age=20, height=NULL, weight=200
Since NULL > 6 is UNKNOWN!
Null Values in WHERE Clauses

Unexpected behavior:

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
```

Should return all persons, but persons with NULL as age are not included!

You can use CASE with IS NULL, ISNULL(), IFNULL() or COALESCE() to handle NULL values.
CASTing Data Types

SQL is a typed language. I.e., values and columns have a data type.

SELECT 3/2
SELECT 3.0/2
SELECT 3/2.0
SELECT CAST(3 AS DOUBLE)/2

Typecasting rules are similar to other typed languages like C++. 

**RECAP: Inner Joins**

**Inner joins** select all rows from both tables as long as there is a match between the columns in both tables. Inner joins are the default in SQL.

**Example:** What stores sell what products?

Product(name, category)
Purchase(prodName, store)

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

Both equivalent: Both INNER JOINS!

```sql
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```
Inner Joins + NULLS = Lost data?

However: Products that were never sold in any store (with no Purchase tuple) will be lost!
Outer Joins

An **outer join** returns also tuples from the joined relations that do not have a corresponding tuple in the other relations (filled with NULL values).

Left outer joins in SQL:

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

Now we’ll get products even if they didn’t sell
INNER JOIN:

Product

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

Purchase

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

SELECT Product.name, Purchase.store
FROM Product
INNER JOIN Purchase
ON Product.name = Purchase.prodName
LEFT OUTER JOIN:

### Product

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

### Purchase

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

**SELECT** Product.name, Purchase.store

**FROM** Product

**LEFT OUTER JOIN** Purchase

**ON** Product.name = Purchase.prodName
Other Outer Joins

• **Left outer join:**
  – Include the left tuple even if there’s no match

• **Right outer join:**
  – Include the right tuple even if there’s no match

• **Full outer join:**
  – Include the both left and right tuples even if there’s no match

SQLite currently only supports LEFT OUTER JOIN, but you can easily just change the order of the tables in the query.
Adding Data

**INSERT INTO** TABLE_NAME

[(column1, column2, column3,...columnN)]

**VALUES** (value1, value2, value3,...valueN);

Note: column names are optional.

**INSERT INTO** Product

**VALUES** ('Gizmo', 19, 'Gadgets', 'GWorks')
Adding Data

The data can also come from an existing table.

```sql
INSERT INTO first_table_name [(column1, column2, ... columnN)]
SELECT column1, column2, ...columnN
FROM second_table_name
[WHERE condition];
```
Removing a Table

DROP TABLE database_name.table_name
Select Syntax Diagram (SQLite)

http://www.sqlite.org/lang.html
Activity

Review (http://www.tutorialspoint.com/sqlite/):

• Transaction control
• Views
• Indexes
• Date & Time