Basic SQL and Relational Algebra

- The `SELECT` statement can be mapped directly to relational algebra.

```
SELECT A_1, A_2, ..., A_n /* this is projection */
FROM R_1, R_2, ..., R_m /* this is the selection op */
WHERE P /* this is cartesian product */
```

is equivalent to:

\[ \Pi_{A_1, A_2, ..., A_n}(\sigma_P(R_1 \times R_2 \times ... \times R_m)) \]

SQL...review

- Select clause
  - Need to specify Join condition
- Concept of ‘aliasing’ to rename relation using AS keyword
  - Rename an attribute...in Select clause
- Tuple variables
- Nested queries
- Set operations:
  - Union
  - Set membership – IN, NOT IN
  - Existence of query results – EXISTS, NOT EXISTS
  - Compare values with values in set (generated by subquery)
    - ALL, > ANY, <, >....

What we’ve seen in SQL so far is equivalent in power to RA and TRC
Some MySQL goodies…

- INTO clause
  - Variation on aliasing
  - Pipe output of SELECT into another table
- SELECT in FROM clause…..use as derived table later in query
  ```sql
  SELECT name
  FROM deposit
  (select name, custid
   from customer) dep-cust
  ON deposit.custid= dep-cust.custid;
  ```

- INNER JOIN
- LEFT (OUTER) JOIN, and RIGHT (OUTER) JOIN

Even more SQL….

- Functions on sets of tuples
  - Aggregate functions: Max, sum,….
- Operating on partitions of sets/relations
  - GROUPBY
- Update operations

- Security, Views, Transactions….not today
  - Maybe later!

SQL— Aggregate Operations

- Thus far SQL (and Relational Algebra/Calculus) only fetched data stored in database tables
- What if we need some basic ‘statistics’ on the data ?
  - Number of rows?
  - Maximum value in a field ?
- Aggregate Operators: apply a function to a set of tuples
  - Function defined on one (or more) field
  - Number of customers with loans
  - Average balance for a customer
  - Number of tuples in a relation
  - …...

Aggregate Operators

- Compute functions on set of tuples selected by where clause
  - Operate on a single column
- Semantics: if SELECT clause contains aggregate operations then it can contain only aggregate operations
  - Except when groupby construct is used
- Functions on sets of values but result is single value
  - Average, minimum, maximum, sum, count(size)
- These functions operate on a single column of a table and return a single value.
Aggregate Operators

- Significant extension of relational algebra.

<table>
<thead>
<tr>
<th>Aggregate Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT (*)</td>
</tr>
<tr>
<td>COUNT ([DISTINCT] A)</td>
</tr>
<tr>
<td>SUM ([DISTINCT] A)</td>
</tr>
<tr>
<td>AVG ([DISTINCT] A)</td>
</tr>
<tr>
<td>MAX (A)</td>
</tr>
<tr>
<td>MIN (A)</td>
</tr>
</tbody>
</table>

Examples

Aggregate operators and computed columns
-arithmetic on column values

_Purchase_(product, date, price, quantity)

\[
\text{SELECT} \quad \text{Sum(price * quantity)} \\
\from \quad \text{Purchase}
\]

What do they mean?

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>

\[
\text{SELECT} \quad \text{Sum(price * quantity)} \\
\from \quad \text{Purchase} \\
\where \quad \text{product} = \text{‘bagel’}
\]

50  (= 20+30)
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>

\[
\text{SELECT } \text{Sum(price} \times \text{quantity)} \text{ FROM Purchase WHERE product = 'bagel'} \Rightarrow 50 \ (= 20+30)
\]

Aggregate Function Example

- Return the number of employees and their average salary.

\[
\text{SELECT COUNT(eno) AS numEmp, AVG(salary) AS avgSalary FROM emp}
\]

Result

<table>
<thead>
<tr>
<th>numEmp</th>
<th>avgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>38750</td>
</tr>
</tbody>
</table>

Just a little bit more SQL…..

- Grouping
  - Operating on groups (partitions) of tuples

Motivation for Grouping

- So far, we’ve applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several groups of tuples.
- Consider: Find the average balance for each branch in the bank.
  - In general, we don’t know how many branches exist, and what the balances are!
  - Suppose we know that 10 branchnames exists; then we can write 10 queries that look like this (!):

    \[
    \text{SELECT AVG(balance) FROM Deposit D WHERE D.branchname='x'}
    \]

    For \( x = 1, 2, \ldots, 10 \):
    \[
    \text{SELECT AVG(balance) FROM Deposit D WHERE D.branchname='x'}
    \]

    Oops…no For loops in SQL!!
**GROUP BY Clause**

- Aggregate functions are most useful when combined with the `GROUP BY` clause. The `GROUP BY` clause groups the tuples based on the values of the attributes specified.

- When used in combination with aggregate functions, the result is a table where each tuple consists of unique values for the group by attributes and the result of the aggregate functions applied to the tuples of that group.

---

**Grouping and Aggregation**

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

Let's see what this means...

---

**Grouping and Aggregation**

1. Compute the `FROM` and `WHERE` clauses.

2. Group by the attributes in the `GROUP BY` bar

3. Compute the `SELECT` clause: grouped attributes and aggregates.

---

**1&2. FROM-WHERE-GROUPBY**

<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>

---

**Grouping and Aggregation**

- Aggregate functions are most useful when combined with the `GROUP BY` clause. The `GROUP BY` clause groups the tuples based on the values of the attributes specified.

- When used in combination with aggregate functions, the result is a table where each tuple consists of unique values for the group by attributes and the result of the aggregate functions applied to the tuples of that group.
3. SELECT

```sql
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
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>

<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>

GROUP BY Example

- For each employee title, return the number of employees with that title, and the minimum, maximum, and average salary.

```sql
SELECT title, COUNT(eno) AS numEmp,
      MIN(salary) as minSal,
      MAX(salary) as maxSal,
      AVG(salary) AS avgSal
FROM emp
GROUP BY title
```

<table>
<thead>
<tr>
<th>title</th>
<th>numEmp</th>
<th>minSal</th>
<th>maxSal</th>
<th>avgSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE</td>
<td>2</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
</tr>
<tr>
<td>SA</td>
<td>3</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>ME</td>
<td>2</td>
<td>40000</td>
<td>40000</td>
<td>40000</td>
</tr>
<tr>
<td>PR</td>
<td>1</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
</tr>
</tbody>
</table>

GROUP BY Clause Rules

- There are a few rules for using the `GROUP BY` clause:
  - 1) A column name cannot appear in the `SELECT` part of the query unless it is part of an aggregate function or in the list of group by attributes.
     Note that the reverse is allowed: a column can be in the `GROUP BY` without being in the `SELECT` part.
  - 2) Any `WHERE` conditions are applied before the `GROUP BY` and aggregate functions are calculated.

Condition on the Groups

- What if we are only interested in groups that satisfy a condition?
**HAVING Clause**
- The **HAVING** clause is applied *AFTER* the **GROUP BY** clause and aggregate functions are calculated.
- It is used to filter out entire *groups* that do not match certain criteria.
- The **HAVING** clause can contain any condition that references aggregate functions and the group by attributes themselves.
  - However, any conditions on the **GROUP BY** attributes should be specified in the **WHERE** clause if possible due to performance reasons.

**Grouping and Aggregation: Evaluation Steps**

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

Find total sales after 10/1/2005 per product.

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

Let's see what this means...

**Grouping and Aggregation**

<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>

What if we are only interested in products that sold quantity >30?

```sql
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING Sum(quantity) > 30
```

Find total sales after 10/1/2005 per product, except that we consider only products that had at least 30 buyers.

**HAVING Clause**

```sql
SELECT product, Sum(price * quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.
General form of Grouping and Aggregation

**SELECT** $S$
**FROM** $R_1, \ldots, R_n$
**WHERE** $C_1$
**GROUP BY** $a_1, \ldots, a_k$
**HAVING** $C_2$

$S$ = may contain attributes $a_1, \ldots, a_k$ and/or any aggregate function but NO OTHER ATTRIBUTES
$C_1 = $ is any condition on the attributes in $R_1, \ldots, R_n$
$C_2 = $ is any condition on aggregate expressions

**Why?**

Evaluation steps:
1. Evaluate FROM-WHERE, apply condition $C_1$
2. Group by the attributes $a_1, \ldots, a_k$
3. Apply condition $C_2$ to each group (may have aggregates)
4. Compute aggregates in $S$ and return the result

Generalized SELECT: Queries With GROUP BY and HAVING

**SELECT** [DISTINCT] attribute-list
**FROM** relation-list
**WHERE** qualification/predicate
**GROUP BY** grouping-list
**HAVING** group-qualification/predicate

- The attribute-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (balance)).
- The attribute list must be a subset of grouping-list. Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group. (A group is a set of tuples that have the same value for all attributes in grouping-list.)

Conceptual Evaluation

- The cross-product of relation-list is computed, tuples that fail qualification in WHERE clause are discarded, “unnecessary” fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- The group-qualification specified in the HAVING clause is then applied to eliminate some groups. Expressions in HAVING clause must have a single value per group!
  - In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.
- Any aggregate function can be applied to a group
  - Final SELECT can have function over each selected group
A quick Note: Group-by v.s. Nested Query

Author(login, name)
Wrote(login, url)

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM Author
WHERE count(SELECT Wrote.url
FROM Wrote
WHERE Author.login=Wrote.login)
> 10
```

This is SQL by a novice

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT Author.name
FROM Author, Wrote
WHERE Author.login=Wrote.login
GROUP BY Author.name
HAVING count(wrote.url) > 10
```

No need for DISTINCT: automatically from GROUP BY

…..more SQL

- Finally, updates/modifications to database
- INSERT, DELETE and UPDATE can be result of queries!

INSERT

- Give all customers with a Loan at Downtown branch a $200 savings account with same account number as Loan number

```
INSERT INTO Deposit
SELECT CustID, Loan-number, Branch-name, 200
FROM Loan
WHERE branch-name = ‘Downtown’;
```
DELETE r
WHERE P
Predicate in P can be as complex as any select clause
Delete all accounts located in New York
DELETE Deposit
WHERE branchname in (SELECT branchname
FROM Branch
WHERE branchcity='New York');

How about this query?
DELETE Deposit
WHERE balance < (SELECT avg(balance)
FROM Deposit);

Delete anomalies
- If delete/update request contains embedded select (sub-query) that references relation where deletions/update take place
- SQL standard disallows such requests
  - Alternate implementation: mark tuples in first round, and actual delete in second round

- INSERT
  - Can insert tuple with specified values
  - Can insert set of tuples resulting from query
- UPDATE
  - Change a value in tuple without changing all values in the tuple
  - Can update set of tuples by using query to select the set
Now, we are done.....kind of!

- More components to SQL:
  - Views
  - Constraints, Triggers
  - …will get to these!

- Next....building 3-tier (full stack) application: PHP+MySQL