SQL Queries

- SQL queries use the `SELECT` statement.

- General form is:
  
  ```sql
  SELECT A_1, A_2, ... 
  FROM r_1, r_2, ... 
  WHERE P;
  ```

  - $r_i$ are the relations (tables).
  - $A_i$ are attributes (columns).
  - $P$ is the selection predicate.

- Equivalent to: $\Pi_{A_1, A_2, ...}(\sigma_P(r_1 \times r_2 \times ...))$
Ordered Results

- SQL query results can be ordered by particular attributes
- Two main categories of query results:
  - “Not ordered by anything”
    - Tuples can appear in any order
  - “Ordered by attributes $A_1, A_2, \ldots$”
    - Tuples are sorted by specified attributes
    - Results are sorted by $A_1$ first
    - Within each value of $A_1$, results are sorted by $A_2$
    - etc.
- Specify an ORDER BY clause at end of SELECT statement
Ordered Results (2)

- Find bank accounts with a balance under $700:
  
  ```sql
  SELECT account_number, balance
  FROM account
  WHERE balance < 700;
  ```

- Order results in increasing order of bank balance:
  
  ```sql
  SELECT account_number, balance
  FROM account
  WHERE balance < 700
  ORDER BY balance;
  ```

- Default order is ascending order
Ordered Results (3)

- Say **ASC** or **DESC** after attribute name to specify order
  - **ASC** is redundant, but can improve readability in some cases
- Can list multiple attributes, each with its own order

  “Retrieve a list of all bank branch details, ordered by branch city, with each city’s branches listed in reverse order of holdings.”

  ```sql
  SELECT * FROM branch
  ORDER BY branch_city ASC, assets DESC;
  ```

<table>
<thead>
<tr>
<th>branch_name</th>
<th>branch_city</th>
<th>assets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pownal</td>
<td>Bennington</td>
<td>400000.00</td>
</tr>
<tr>
<td>Brighton</td>
<td>Brooklyn</td>
<td>7000000.00</td>
</tr>
<tr>
<td>Downtown</td>
<td>Brooklyn</td>
<td>9000000.00</td>
</tr>
<tr>
<td>Round Hill</td>
<td>Horseneck</td>
<td>8000000.00</td>
</tr>
<tr>
<td>Perryridge</td>
<td>Horseneck</td>
<td>1700000.00</td>
</tr>
<tr>
<td>Mianus</td>
<td>Horseneck</td>
<td>400200.00</td>
</tr>
<tr>
<td>Redwood</td>
<td>Palo Alto</td>
<td>2100000.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Aggregate Functions in SQL

- SQL provides grouping and aggregate operations, just like relational algebra.

- Aggregate functions:
  - `SUM`: sums the values in the collection.
  - `AVG`: computes average of values in the collection.
  - `COUNT`: counts number of elements in the collection.
  - `MIN`: returns minimum value in the collection.
  - `MAX`: returns maximum value in the collection.

- `SUM` and `AVG` require numeric inputs (obvious).
Aggregate Examples

- Find average balance of accounts at Perryridge branch

```sql
SELECT AVG(balance) FROM account
WHERE branch_name = 'Perryridge';
```

```
+----------------------------+
<table>
<thead>
<tr>
<th>AVG(balance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>650.000000</td>
</tr>
</tbody>
</table>
+----------------------------+
```

- Find maximum amount of any loan in the bank

```sql
SELECT MAX(amount) AS max_amt FROM loan;
```

```
+-------+
<table>
<thead>
<tr>
<th>max_amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>7500.00</td>
</tr>
</tbody>
</table>
+-------+
```

- Can name computed values, like usual
Aggregate Examples (2)

- This query produces an error:
  
  ```sql
  SELECT branch_name,
         MAX(amount) AS max_amt
  FROM loan;
  ```

- Aggregate functions compute a single value from a multiset of inputs
  
  Doesn’t make sense to combine individual attributes and aggregate functions like this

- This does work:
  
  ```sql
  SELECT MIN(amount) AS min_amt,
         MAX(amount) AS max_amt
  FROM loan;
  ```

  +---------+---------+
  | min_amt | max_amt |
  +---------+---------+
  | 500.00  | 7500.00 |
  +---------+---------+
Eliminating Duplicates

- Sometimes need to eliminate duplicates in SQL queries
  - Can use DISTINCT keyword to eliminate duplicates
- Example:
  “Find the number of branches that currently have loans.”
  ```sql
  SELECT COUNT(branch_name) FROM loan;
  ```
  - Doesn’t work, because branches may have multiple loans
  - Instead, do this:
    ```sql
    SELECT COUNT(DISTINCT branch_name) FROM loan;
    ```
  - Duplicates are eliminated from input multiset before aggregate function is applied
Computing Counts

- Can count individual attribute values
  \[ \text{COUNT}(\text{branch\_name}) \]
  \[ \text{COUNT}((\text{DISTINCT branch}\_\text{name}) \]

- Can also count the total number of tuples
  \[ \text{COUNT}(\ast) \]
  - If used with grouping, counts total number of tuples in each group
  - If used without grouping, counts total number of tuples

- Counting a specific attribute is useful when:
  - Need to count (possibly distinct) values of a particular attribute
  - Cases where some values in input multiset may be \text{NULL}
    - As before, COUNT ignores NULL values (more on this next week)
Grouping and Aggregates

- Can also perform grouping on a relation before computing aggregates
  - Specify a `GROUP BY A_1, A_2, ...` clause at end of query
- Example:
  “Find the average loan amount for each branch.”
  ```sql
  SELECT branch_name, AVG(amount) AS avg_amt
  FROM loan GROUP BY branch_name;
  ```
  - First, tuples in `loan` are grouped by `branch_name`
  - Then, aggregate functions are applied to each group

<table>
<thead>
<tr>
<th>branch_name</th>
<th>avg_amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central</td>
<td>570.000000</td>
</tr>
<tr>
<td>Downtown</td>
<td>1250.000000</td>
</tr>
<tr>
<td>Mianus</td>
<td>500.000000</td>
</tr>
<tr>
<td>North Town</td>
<td>7500.000000</td>
</tr>
<tr>
<td>Perryridge</td>
<td>1400.000000</td>
</tr>
<tr>
<td>Redwood</td>
<td>2000.000000</td>
</tr>
<tr>
<td>Round Hill</td>
<td>900.000000</td>
</tr>
</tbody>
</table>
Grouping and Aggregates (2)

- Can group on multiple attributes
  - Each group has unique values for the entire set of grouping attributes

- Example:
  - “How many accounts does each customer have at each branch?”
  - Group by both customer name and branch name
  - Compute count of tuples in each group
  - Can write the SQL statement yourself, and try it out
Note the difference between relational algebra notation and SQL syntax

Relational algebra syntax:

\[ G_{G_1, G_2, \ldots, G_n}^{F_1(A_1), F_2(A_2), \ldots, F_m(A_m)}(E) \]

Grouping attributes only appear on left of \( G \)

SQL syntax:

```sql
SELECT G_1, G_2, \ldots, F_1(A_1), F_2(A_2), \ldots
FROM r_1, r_2, \ldots WHERE P
GROUP BY G_1, G_2, \ldots
```

Frequently, grouping attributes are specified in both the SELECT clause and GROUP BY clause
Grouping and Aggregates (4)

- SQL doesn’t require that you specify the grouping attributes in the `SELECT` clause
  - Only requirement is that the grouping attributes are specified in the `GROUP BY` clause
  - e.g. if you only want the aggregated results, could do this:
    ```sql
    SELECT F_1(A_1), F_2(A_2), ...
    FROM r_1, r_2, ... WHERE P
    GROUP BY G_1, G_2, ...
    ```
- Also, can use expressions for grouping and aggregates
  - Example (very uncommon, but also valid):
    ```sql
    SELECT MIN(a + b) - MAX(c)
    FROM t GROUP BY d * e;
    ```
The **WHERE** clause is applied *before* any grouping occurs.

```sql
SELECT G_1, G_2, ..., F_1(A_1), F_2(A_2), ...
FROM r_1, r_2, ... WHERE P
GROUP BY G_1, G_2, ...
```

- Translates into relational algebra expression:
  \[ \Pi_{G_1, G_2, ...} G_{F_1(A_1), F_2(A_2), ...} (\sigma_P (r_1 \times r_2 \times ...)) \]

- A **WHERE** clause constrains the set of tuples that grouping and aggregation are applied to.
Filtering Results

- To apply filtering to the results of grouping and aggregation, use a **HAVING** clause.
  - Exactly like **WHERE** clause, except applied after grouping and aggregation.

```sql
SELECT G_1, G_2, ..., F_1(A_1), F_2(A_2), ...
FROM r_1, r_2, ..., WHERE P_W
GROUP BY G_1, G_2, ...
HAVING P_H
```

- Translates into:

\[
\prod \ldots (\sigma_{P_H} (G_1, G_2, \ldots G_{F_1(A_1)}, F_2(A_2), \ldots (\sigma_{P_W} (r_1 \times r_2 \times \ldots))))
\]
The **HAVING** Clause

- The **HAVING** clause can use aggregate functions in its predicate
  - It’s applied after grouping/aggregation, so those values are available
  - The **WHERE** clause cannot do this, of course

- Example:
  
  “Find all customers with more than one loan.”
  
  ```sql
  SELECT customer_name, COUNT(*) AS num_loans
  FROM borrower
  GROUP BY customer_name
  HAVING COUNT(*) > 1;
  ```

<table>
<thead>
<tr>
<th>customer_name</th>
<th>num_loans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>3</td>
</tr>
</tbody>
</table>
Nested Subqueries

- SQL provides broad support for nested subqueries
  - A SQL query is a “select-from-where” expression
  - Nested subqueries are “select-from-where” expressions embedded within another query

- Can embed queries in WHERE clauses
  - Sophisticated selection tests

- Can embed queries in FROM clauses
  - Issuing a query against a derived relation

- Can even embed queries in SELECT clauses!
  - Appeared in SQL:2003 standard; many DBs support this
  - Makes many queries easier to write, but can be slow too
Kinds of Subqueries

- Some subqueries produce only a single result
  ```sql
  SELECT MAX(assets) FROM branch;
  ```
  Called a `scalar subquery`
  Still a relation, just with one attribute and one tuple

- Most subqueries produce a relation containing multiple tuples
  Nested queries often produce relation with single attribute
    - Very common for subqueries in `WHERE` clause
  Nested queries can also produce multiple-attribute relation
    - Very common for subqueries in `FROM` clause
    - Can also be used in the `WHERE` clause in some cases
Subqueries in WHERE Clause

- **Widely used:**
  - Direct comparison with scalar-subquery results
  - Set-membership tests: `IN`, `NOT IN`
  - Empty-set tests: `EXISTS`, `NOT EXISTS`

- **Less frequently used:**
  - Set-comparison tests: `ANY`, `SOME`, `ALL`
  - Uniqueness tests: `UNIQUE`, `NOT UNIQUE`

- (Can also use these in the `HAVING` clause)
Comparison with Subquery Result

- Can use scalar subqueries in `WHERE` clause comparisons

- Example:
  - Want to find the name of the branch with the smallest number of assets.
  - Can easily find the smallest number of assets:
    
    ```
    SELECT MIN(assets) FROM branch;
    ```
  - This is a scalar subquery; can use it in `WHERE` clause:
    
    ```
    SELECT branch_name FROM branch
    WHERE assets = (SELECT MIN(assets) FROM branch);
    ```

<table>
<thead>
<tr>
<th>branch_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pownal</td>
</tr>
</tbody>
</table>
Set Membership Tests

- Can use **IN (...)** and **NOT IN (...)** for set membership tests

- Example:
  - Find customers with both an account and a loan.
  - Before, did this with a **INTERSECT** operation
  - Can also use a set-membership test:
    “Select all customer names from depositor relation, that also appear somewhere in borrower relation.”
    
    ```sql
    SELECT DISTINCT customer_name FROM depositor
    WHERE customer_name IN (SELECT customer_name FROM borrower)
    ```

    - **DISTINCT** necessary because a customer might appear multiple times in **depositor**
Set Membership Tests (2)

- **IN ( . . . )** and **NOT IN ( . . . )** support subqueries that return multiple columns (!!!)

- Example: “Find the ID of the largest loan at each branch, including the branch name and the amount of the loan.”
  - First, need to find the largest loan at each branch
    ```sql
    SELECT branch_name, MAX(amount)
    FROM loan GROUP BY branch_name
    ```
  - Use this result to identify the rest of the loan details
    ```sql
    SELECT * FROM loan
    WHERE (branch_name, amount) IN (  
        SELECT branch_name, MAX(amount)  
        FROM loan GROUP BY branch_name
    );
    ```
Empty-Set Tests

- Can test whether or not a subquery generates any results at all
  - EXISTS (...)
  - NOT EXISTS (...)

- Example:
  “Find customers with an account but not a loan.”
  ```sql
  SELECT DISTINCT customer_name FROM depositor d 
  WHERE NOT EXISTS ( 
    SELECT * FROM borrower b 
    WHERE b.customer_name = d.customer_name); 
  ```
  - Result includes every customer that appears in depositor table, that doesn’t also appear in the borrower table.
“Find customers with an account but not a loan.”

```
SELECT DISTINCT customer_name FROM depositor d
WHERE NOT EXISTS (  
    SELECT * FROM borrower b  
    WHERE b.customer_name = d.customer_name);
```

- Inner query refers to an attribute in outer query’s relation

- In general, nested subqueries can refer to enclosing queries’ relations.
- However, enclosing queries cannot refer to the nested queries’ relations.
“Find customers with an account but not a loan.”

```
SELECT DISTINCT customer_name FROM depositor d
WHERE NOT EXISTS (  
    SELECT * FROM borrower b  
    WHERE b.customer_name = d.customer_name)
```

- When a nested query refers to an enclosing query’s attributes, it is a correlated subquery
  - The inner query must be evaluated once for each tuple considered by the enclosing query
  - Generally to be avoided! Very slow.
Many correlated subqueries can be restated using a join or a Cartesian product
- Often the join operation will be much faster
- More advanced DBMSes will automatically decorrelate such queries, but some can’t…

Certain conditions, e.g. EXISTS/NOT EXISTS, usually indicate presence of a correlated subquery

If it’s easy to decorrelate the subquery, do that! 😊

If not, test the query for its performance.
- If the database can decorrelate it, you’re done!
- If the database can’t decorrelate it, may need to come up with an alternate formulation.
Set Comparison Tests

- Can compare a value to a set of values
  - Is a value larger/smaller/etc. than some value in the set?

- Example:
  “Find all branches with assets greater than at least one branch in Brooklyn.”

```sql
SELECT branch_name FROM branch
WHERE assets > SOME (SELECT assets FROM branch
WHERE branch_name='Brooklyn');
```
Set Comparison Tests (2)

- General form of test:
  \[ \text{attr compare\_op SOME ( subquery )} \]
  - Can use any comparison operation
    - \( = \) SOME is same as \( \text{IN} \)
    - \( \text{ANY} \) is a synonym for SOME
  - Can also compare a value with \textit{all} values in a set
    - Use \textit{ALL} instead of SOME
      - \( <> \) ALL is same as \( \text{NOT IN} \)
Example:

“Find branches with assets greater than all branches in Brooklyn.”

```sql
SELECT branch_name FROM branch
WHERE assets > ALL (
    SELECT assets FROM branch
    WHERE branch_name='Brooklyn');
```

Could also write this with a scalar subquery

```sql
SELECT branch_name FROM branch
WHERE assets >
    (SELECT MAX(assets) FROM branch
    WHERE branch_name='Brooklyn');
```
Uniqueness Tests

- Can test whether a nested query generates any duplicate tuples
  - `UNIQUE (...)`
  - `NOT UNIQUE (...)`
- Not widely implemented
  - Expensive operation!
- Can emulate in a number of ways
  - `GROUP BY ... HAVING COUNT(*) = 1` or
  - `GROUP BY ... HAVING COUNT(*) > 1` is one approach
Subqueries in **FROM** Clause

- Often need to compute a result in multiple steps
- Can query against a subquery’s results
  - Called a *derived relation*
- A trivial example:
  - A **HAVING** clause can be implemented as a nested query in the **FROM** clause
HAVING vs. Nested Query

“Find all cities with more than two customers living in the city.”

```
SELECT customer_city, COUNT(*) AS num_customers
FROM customer GROUP BY customer_city
HAVING COUNT(*) > 2;
```

- Or, can write:

```
SELECT customer_city, num_customers
FROM (SELECT customer_city, COUNT(*)
     FROM customer GROUP BY customer_city)
     AS counts (customer_city, num_customers)
WHERE num_customers > 2;
```

- Grouping and aggregation is computed by inner query
- Outer query selects desired results generated by inner query
Derived Relation Syntax

- Subquery in `FROM` clause must be given a name
  - Many DBMSes also require attributes to be named
    ```sql
    SELECT customer_city, num_customers
    FROM (SELECT customer_city, COUNT(*)
          FROM customer GROUP BY customer_city)
    AS counts (customer_city, num_customers)
    WHERE num_customers > 2;
    ```
  - Nested query is called `counts`, and specifies two attributes
  - Syntax varies from DBMS to DBMS...
    - MySQL requires a name for derived relations, but doesn’t allow attribute names to be specified.
More typical is a query against aggregate values

Example:

“Find the largest total account balance of any branch.”

Need to compute total account balance for each branch first.

```sql
SELECT branch_name, SUM(balance) AS total_bal
FROM account GROUP BY branch_name;
```

Then we can easily find the answer:

```sql
SELECT MAX(total_bal) AS largest_total
FROM (SELECT branch_name,
        SUM(balance) AS total_bal
    FROM account GROUP BY branch_name)
    AS totals (branch_name, tot_bal);
```
**Aggregates of Aggregates**

- **Always** take note when computing aggregates of aggregates!
  - “Find the largest total account balance of any branch.”
  - Two nested aggregates: max of sums

- **A very common mistake:**

  ```sql
  SELECT branch_name, SUM(balance) AS tot_bal
  FROM account GROUP BY branch_name
  HAVING tot_bal = MAX(tot_bal)
  ```

  - A **SELECT** query can only perform one level of aggregation
  - Need a second **SELECT** to find the maximum total
  - Unfortunately, MySQL accepts this and returns bogus result
More Data Manipulation Operations

- SQL provides many other options for inserting, updating, and deleting tuples
- All commands support SELECT-style syntax
- Can insert individual tuples into a table:
  ```sql
  INSERT INTO table VALUES (1, 'foo', 50);
  ```
- Can also insert the result of a query into a table:
  ```sql
  INSERT INTO table SELECT ...;
  ```
- Only constraint is that generated results must have a compatible schema
Deleting Tuples

- SQL DELETE command can use a WHERE clause
  
  ```sql
  DELETE FROM table;
  ```
  
  Deletes all rows in the table
  
  ```sql
  DELETE FROM table WHERE ...;
  ```
  
  Only deletes rows that satisfy the conditions
  
- The WHERE clause can use anything that SELECT’s WHERE clause supports
  
  - Nested queries, in particular!
Updating Tables

- SQL also has an **UPDATE** command for modifying existing tuples in a table.

- General form:
  
  ```
  UPDATE table
  SET attr1=val1, attr2=val2, ...
  WHERE condition;
  ```

  - Must specify the attributes to update.
  - Attributes being modified *must* appear in table being updated (obvious).
  - The **WHERE** clause is optional! If unspecified, *all* rows are updated.
  - **WHERE** condition can contain nested queries, etc.
Values in **UPDATE** can be arithmetic expressions

- Can refer to any attribute in table being updated

**Example:**

- Add 2% interest to all bank account balances with a balance of $500 or less.

```sql
UPDATE account
SET balance = balance * 1.02
WHERE balance <= 500;
```
Review

- SQL query syntax is very rich
  - Can state a wide range of complex queries
  - Many ways to state a particular query

- SQL supports nested queries
  - Often essential for computing particular results
  - Can sometimes be very inefficient

- SQL also provides similar capability for inserting, deleting, and updating tables
Next Time

- **NULL** values in SQL
- Additional SQL join operations
  - Natural join
  - Outer joins
- SQL views