SQL QUERIES

CS121: Introduction to Relational Database Systems

Fall 2016 – Lecture 5

SQL Queries

- SQL queries use the **SELECT** statement
- General form is:

```
SELECT A1, A2, ...
  FROM r1, r2, ...
  WHERE P;
```

  - r_i are the relations (tables)
  - A_i are attributes (columns)
  - P is the selection predicate
- Equivalent to: $\Pi_{A_1, A_2, \ldots}(\sigma_P(r_1 \times r_2 \times \ldots))$

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, …"
    - 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:

```
SELECT account_number, balance
  FROM account
  WHERE balance < 700;
```

account_number	balance
A-102	400.00
A-101	500.00
A-444	625.00
A-305	350.00

- Order results in increasing order of bank balance:

```
SELECT account_number, balance
  FROM account
  WHERE balance < 700
  ORDER BY balance;
```

account_number	balance
A-305	350.00
A-102	400.00
A-101	500.00
A-444	625.00

  - 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."

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

```
+-------------+-------------+------------+
| branch_name | branch_city | assets     |
+-------------+-------------+------------+
| Pownal      | Bennington  |  400000.00 |
| Brighton    | Brooklyn    | 7000000.00 |
| Downtown    | Brooklyn    |  900000.00 |
| Round Hill  | Horseneck   | 8000000.00 |
| Perryridge  | Horseneck   | 1700000.00 |
| Mianus      | Horseneck   |  400200.00 |
| Redwood     | Palo Alto   | 2100000.00 |
| ...         | ...         |        ... |
```

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

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

```
+--------------+
| AVG(balance) |
+--------------+
|   650.000000 |
+--------------+
```

- Find maximum amount of any loan in the bank

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

  - Can name computed values, like usual

```
+---------+
| max_amt |
+---------+
| 7500.00 |
+---------+
```

Aggregate Examples (2)

- This query produces an error:

```
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:

```
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."

  ```
  SELECT COUNT(branch_name) FROM loan;
  ```

  - Doesn't work, because branches may have multiple loans
  - Instead, do this:

  ```
  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

  `COUNT(branch_name)`

  `COUNT(DISTINCT branch_name)`
- Can also count the total number of tuples

  `COUNT(*)`
  - 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 **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** $A_1, A_2, \ldots$ clause at end of query
- Example:

  "Find the average loan amount for each branch."

```
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

branch_name	avg_amt
Central	570.000000
Downtown	1250.000000
Mianus	500.000000
North Town	7500.000000
Perryridge	1400.000000
Redwood	2000.000000
Round Hill	900.000000

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

Grouping and Aggregates (3)

- Note the difference between relational algebra notation and SQL syntax
- Relational algebra syntax:

  $_{G_1, G_2, \ldots, G_n}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots, F_m(A_m)}(E)$

  - Grouping attributes only appear on left of $\mathcal{G}$
- SQL syntax:

  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 <u>and</u> **GROUP BY** clause

Grouping and Aggregates (4)

- SQL doesn't <u>require</u> 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:

```
SELECT F1(A1), F2(A2), ...
  FROM r1, r2, ... WHERE P
  GROUP BY G1, G2, ...
```

$$\texttt{SELECT } F_1(A_1), F_2(A_2), \ldots \quad \texttt{FROM } r_1, r_2, \ldots \texttt{ WHERE P} \quad \texttt{GROUP BY } G_1, G_2, \ldots$$

- Also, can use expressions for grouping and aggregates
  - Example (very uncommon, but also valid):

```
SELECT MIN(a + b) - MAX(c)
FROM t GROUP BY d * e;
```

Filtering Tuples

- The **WHERE** clause is applied *before* any grouping occurs

```
SELECT G1,G2,..., F1(A1),F2(A2),...
  FROM r1,r2,... WHERE P
  GROUP BY G1,G2,...
```

  - Translates into relational algebra expression:

    $\Pi_{\ldots}(_{G_1, G_2, \ldots}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots}(\sigma_P(r_1 \times r_2 \times \ldots)))$
  - 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

SELECT $G_1, G_2, \ldots, F_1(A_1), F_2(A_2), \ldots$
FROM $r_1, r_2, \ldots$ **WHERE** P_W
GROUP BY $G_1, G_2, \ldots$
HAVING P_H

  - Translates into:

$$\Pi_{\ldots}(\sigma_{P_H}({}_{G_1, G_2, \ldots}\mathcal{G}_{F_1(A_1), F_2(A_2), \ldots}(\sigma_{P_W}(r_1 \times r_2 \times \ldots))))$$

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."

```
SELECT customer_name, COUNT(*) AS num_loans
  FROM borrower GROUP BY customer_name
  HAVING COUNT(*) > 1;
```

```
+---------------+-----------+
| customer_name | num_loans |
+---------------+-----------+
| Smith         |         3 |
+---------------+-----------+
```

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

```
SELECT MAX(assets) FROM branch;
```

  - Called a <u>scalar subquery</u>
  - 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);
```

```
+-------------+
| branch_name |
+-------------+
| Pownal      |
+-------------+
```

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."

```
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

    ```
    SELECT branch_name, MAX(amount)
      FROM loan GROUP BY branch_name
    ```

  - Use this result to identify the rest of the loan details

    ```
    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."

```
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.

Empty-Set Tests (2)

"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.

Correlated Subqueries

"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.

Correlated Subqueries (2)

- 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."

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

Set Comparison Tests (2)

- General form of test:

  `attr` *compare_op* **`SOME ( `***subquery***` )`**

  - Can use any comparison operation

    `= SOME` is same as **`IN`**

  - **`ANY`** is a synonym for **`SOME`**
- Can also compare a value with *all* values in a set
  - Use **`ALL`** instead of **`SOME`**

    `<> ALL` is same as **`NOT IN`**

Set Comparison Tests (3)

- Example:

  "Find branches with assets greater than *all* branches in Brooklyn."

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

  - Could also write this with a scalar subquery

```
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

```
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.

Using Derived Relations

- 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.

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

  - Then we can easily find the answer:

    ```
    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:

```
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:

```
INSERT INTO table VALUES (1, 'foo', 50);
```

- Can also insert the result of a query into a table:

```
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

DELETE FROM table;

  - Deletes all rows in the table

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.

Updating Tables (2)

- 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.

```
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