Relational Database System Implementation

CS122 – Lecture 5

Winter Term, 2017-2018

Last Time: SQL Join Expressions

- Last time, began discussing SQL join syntax
- Original SQL form:
  - SELECT ... FROM t1, t2, ... WHERE P
  - Any join conditions are specified in WHERE clause
- FROM clause produces a Cartesian product of t1, t2, ...
  - $t1 \times t2 \times \ldots$
  - Schema produced by FROM clause is $t1.* \cup t2.* \cup \ldots$
- ANSI-standard SQL: WHERE clause may only refer to the columns generated by the FROM clause
  - Aliases in SELECT clause shouldn't be visible (although many databases make them visible in WHERE clause)

SQL Join Expressions (2)

- SELECT … FROM t1, t2, … WHERE P
  - $t1 \times t2 \times \ldots$
  - Schema of FROM clause is $t1.* \cup t2.* \cup \ldots$ (in that order)
- To avoid ambiguity, column names in schema also include corresponding table names, e.g. t1.a, t1.b, t2.a, t2.c, etc.
  - If column name is unambiguous, predicate can just use column name by itself
  - If column name is ambiguous, predicate must specify both table name and column name
- Example: SELECT * FROM t1, t2 WHERE a > 5 AND c = 20;
  - Not valid: column name a is ambiguous (given above schema)
- Valid: SELECT * FROM t1, t2 WHERE t1.a > 5 AND c = 20;

Additional SQL Join Syntax

- SQL-92 introduced several new forms:
  - SELECT … FROM t1 JOIN t2 ON t1.a = t2.a
  - SELECT … FROM t1 JOIN t2 USING (a1, a2, …)
  - SELECT … FROM t1 NATURAL JOIN t2
  - Can specify INNER, [LEFT|RIGHT|FULL] OUTER JOIN
    - Also CROSS JOIN, but cannot specify ON, USING, or NATURAL
- ON clause is not that challenging
  - Similar to original syntax, but allows inner/outer joins
  - Schema of "FROM t1 JOIN t2 ON …" is $t1.* \cup t2.*$

Additional SQL Join Syntax (2)

- USING and NATURAL joins are more complicated
  - SELECT … FROM t1 JOIN t2 USING (a1, a2, …)
  - SELECT … FROM t1 NATURAL JOIN t2
  - Join condition is inferred from the common column names (NATURAL JOIN), or generated from the USING clause
  - Also includes a project to eliminate duplicate column names (project is part of the FROM clause; affects WHERE predicate)
- For SELECT * FROM t1 NATURAL JOIN t2, or SELECT * FROM t1 JOIN t2 USING (a1, a2, …):
  - Denote the join columns as JC. These have no table name.
    - For natural join, JC = $t1 \cap t2$; otherwise, JC = attrs in USING clause
  - FROM clause's schema is JC $\cup$ ($t1$ – JC) $\cup$ ($t2$ – JC)

Additional SQL Join Syntax (3)

- For SELECT * FROM t1 NATURAL [???] JOIN t2:
  - Schemas: $t1(a, b)$ and $t2(a, c)$
  - FROM schema: $(a, t1.b, t2.c)$
- For natural inner join:
  - Project can use either $t1.a$ or $t2.a$ to generate a
- For natural left outer join:
  - Project should use $t1.a$; $t2.a$ may be NULL for some rows
  - (Similar for natural right outer join, except $t2.a$ is used)
- For natural full outer join:
  - Project should use COALESCE(t1.a, t2.a), since either $t1.a$ or $t2.a$ could be NULL

Additional SQL Join Syntax (4)

- SELECT t1.a FROM t1 NATURAL JOIN t2
  - Schemas: $t1(a, b)$ and $t2(a, c)$
  - FROM schema: $(a, t1.b, t2.c)$
- This query is not valid under the ANSI standard, because there is no t1.a outside the FROM clause
  - Some databases (e.g. MySQL) will allow this query
- This query is valid:
  - SELECT a, t2.c FROM t1 NATURAL JOIN t2
  - (Technically, can also say "SELECT a, c" because *c* won't be ambiguous)

Additional SQL Join Syntax (5)

- SELECT * FROM t1 NATURAL JOIN t2 NATURAL JOIN t3
  - Schemas: $t1(a, b)$, $t2(a, c)$, $t3(a, d)$
  - FROM schema: $(a, t1.b, t2.c, t3.d)$
- This query presents another challenge
- Step 1: t1 NATURAL JOIN t2
  - Join condition is: $t1.a = t2.a$
  - Result schema is $(a, t1.b, t2.c)$
- Step 2: natural-join this result with t3
  - Join condition is: $a = t3.a$
  - Problem: column-reference a is ambiguous

Additional SQL Join Syntax (6)

- SELECT * FROM t1 NATURAL JOIN t2 NATURAL JOIN t3
  - Schemas: $t1(a, b)$, $t2(a, c)$, $t3(a, d)$
  - FROM schema: $(a, t1.b, t2.c, t3.d)$
- Generate placeholder table names to avoid ambiguities
- Step 1 (revised): t1 NATURAL JOIN t2
  - Join condition is: $t1.a = t2.a$
  - Result schema is $\#R1(a, t1.b, t2.c)$
- Step 2 (revised): natural-join this result with t3
  - Join condition is: $\#R1.a = t3.a$
  - Result schema is $\#R2(a, t1.b, t2.c, t3.d)$

Mapping SQL Joins into Plans

- Summary: translating SQL joins has its own challenges
- Primarily center around natural joins, and joins with the USING clause:
  - Must generate an appropriate schema to eliminate duplicate columns
  - Must use COALESCE() operations on join-columns used in full outer joins
  - May need to deal with ambiguous column names when more than two tables are natural-joined together
- (All surmountable; just annoying…)

Nested Subqueries

- SQL queries can also include nested subqueries
- Subqueries can appear in the SELECT clause:
  - ```
    SELECT customer_id,
                (SELECT SUM(balance)
                 FROM loan JOIN borrower b
                 WHERE b.customer_id = c.customer_id) tot_bal
    FROM customer c;
    ```
  - *(Compute total of each customer's loan balances)*
- Must be a scalar subquery
  - Must produce exactly one row and one column
- This is almost always a correlated subquery
  - Inner query refers to an enclosing query's values
  - Requires correlated evaluation to compute the results
    ```

Nested Subqueries (2)

- Subqueries can also appear in the FROM clause:
  - ```
    SELECT u.username, email, max_score
    FROM users u,
           (SELECT username, MAX(score) AS max_score
            FROM game_scores GROUP BY username) AS s
    WHERE u.username = s.username;
    ```
- Called a *derived relation*
  - The table is produced by a subquery, instead of being read from a file (a.k.a. a *base relation*)
- Cannot be a correlated subquery
  - …at least, not with respect to the immediately enclosing query
  - Could still be correlated with a query further out, if parent appears in a SELECT expression, or a WHERE predicate, etc.
    ```

Nested Subqueries (3)

- Subqueries can also appear in the WHERE clause:
  - SELECT employee_id, last_name, first_name
    FROM employees e WHERE e.is_manager = 0 AND
    EXISTS (SELECT * FROM employees m
    WHERE m.department = e.department AND
    m.is_manager = 1 AND m.salary < e.salary);
  - *(Find non-manager employees who make more money than some manager in the same department)*
- Also, IN/NOT IN operators, ANY/SOME/ALL queries, and scalar subqueries as well
- Again, could be a correlated subquery, and often is. ☹

Nested Subqueries (4)

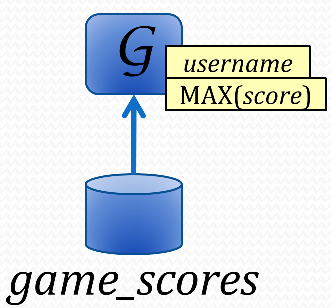
- Previous example:
  - SELECT employee_id, last_name, first_name
    FROM employees e WHERE is_manager = 0 AND
    EXISTS (SELECT * FROM employees m
    WHERE m.department = e.department AND
    m.is_manager = 1 AND m.salary < e.salary);
- Note that EXISTS/NOT EXISTS can complete after only one row is generated by subquery
  - Don't need to evaluate entire subquery result…
  - Definitely want to optimize subplan to produce first row as quickly as possible

Subqueries in FROM Clause

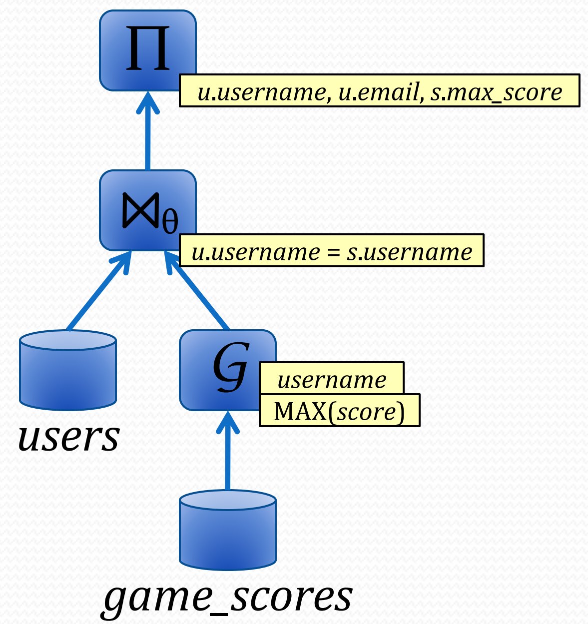
- FROM subqueries are the easiest to deal with! ☺
  - ```
    SELECT u.username, email, max_score
    FROM users u,
           (SELECT username, MAX(score) AS max_score
            FROM game_scores GROUP BY username) AS s
    WHERE u.username = s.username;
    ```
- To generate execution plan for full query:
  - Simply generate execution plan for the derived relation (e.g. recursive call to planner with subquery's AST)
  - Use the subquery's plan as an input into the outer query (as if it were another table in the FROM clause)
    ```

Subqueries in FROM Clause (2)

- Our example:
  - SELECT u.username, email, max_score
    FROM users u,
    (SELECT username, MAX(score) AS max_score
    FROM game_scores GROUP BY username) AS s
    WHERE u.username = s.username;
- Subquery plan:



- Full plan:



FROM Subqueries and Views

- Views will also create subqueries in the FROM clause
  - ```
    CREATE VIEW top_scores AS
    SELECT username, MAX(score) AS max_score
    FROM game_scores GROUP BY username;
    ```
  - ```
    SELECT u.username, email, max_score
    FROM users u, top_scores s
    WHERE u.username = s.username;
    ```
- Simple substitution of view's definition creates a nested subquery in the FROM clause:
  - ```
    SELECT u.username, email, max_score
    FROM users u, (SELECT username, MAX(score) AS max_score
                   FROM game_scores GROUP BY username) s
    WHERE u.username = s.username;
    ```
    ```

FROM Subqueries and Views (2)

- Two options as to how this is done
- Option 1:
  - When view is created, database can construct a relational algebra plan for the view, and save it.
  - When a query references the view, simply use the view's plan as a subplan in the referencing query.
- Option 2:
  - When view is created, database parses and verifies the SQL, but doesn't generate a relational algebra plan.
  - When a query references the view, modify the query's SQL to use the view's definition, then generate a plan.
- Second option requires more work during planning, but potentially allows for greater optimizations to be applied

Subqueries in SELECT Clause

- Subqueries in the SELECT clause must be scalar subqueries:
  - SELECT customer_id,
                (SELECT SUM(balance) FROM loan JOIN borrower b
                 WHERE b.customer_id = c.customer_id) tot_bal
    FROM customer c;
  - Must produce exactly one row and one column
- An easy, generally useful approach:
  - Represent scalar subquery as special kind of expression
  - During query planning, generate a plan for the subquery
  - When select-expression is evaluated, recursively invoke the query executor to evaluate the subquery to generate a result
  - (Report an error if doesn't produce exactly one row/column!)

Subqueries in SELECT Clause (2)

- Subqueries in the SELECT clause must be scalar subqueries:
  - ```
    SELECT customer_id,
           (SELECT SUM(balance) FROM loan JOIN borrower b
            WHERE b.customer_id = c.customer_id) tot_bal
    FROM customer c;
    ```
  - Must produce exactly one row and one column
- If scalar subquery is correlated:
  - Must reevaluate the subquery for each row in outer query
- If scalar subquery isn't correlated:
  - Can evaluate subquery once and cache the result
  - (This is an optimization; correlated evaluation will also work, although it is obviously unnecessarily slow.)
    ```

Subqueries in SELECT Clause (3)

- Correlated scalar subqueries in the SELECT clause can frequently be restated as a decorrelated outer join:
  - ```
    SELECT customer_id,
                 (SELECT SUM(balance) FROM loan JOIN borrower b
                  WHERE b.customer_id = c.customer_id) tot_bal
    FROM customer c;
    ```
- Equivalent to:
  - ```
    SELECT c.customer_id, tot_bal
    FROM customer c LEFT OUTER JOIN
         (SELECT b.customer_id, SUM(balance) tot_bal
          FROM loan JOIN borrower b GROUP BY b.customer_id) t
         ON t.customer_id = c.customer_id);
    ```
- Usually, outer join is cheaper than correlated evaluation

Scalar Subqueries in Other Clauses

- Scalar subqueries can also appear in other predicates, e.g. WHERE clauses, HAVING clauses, ON clauses, etc.
- These cases are more likely to be uncorrelated, which means they can be evaluated once and then cached
- If they are correlated, they can also often be restated as a join in an appropriate part of the execution plan
  - But, it can get significantly more complicated…

Subqueries in WHERE Clause

- IN/NOT IN clauses and EXISTS/NOT EXISTS predicates can also appear in WHERE and HAVING clauses
- Example: Find bank customers with accounts at any bank branch in Los Angeles
  - ```
    SELECT * FROM customer c
    WHERE customer_id IN
                (SELECT customer_id FROM depositor
                   NATURAL JOIN account NATURAL JOIN branch
                 WHERE branch_city = 'Los Angeles');
    ```
- Is this query correlated?
  - No; inner query doesn't reference enclosing query values
    ```

Subqueries in WHERE Clause (2)

- Again, can implement IN/EXISTS in a simple and generally useful way:
  - Create special IN and EXISTS expression operators that include a subquery
  - During planning, an execution plan is generated for each subquery in an IN or EXISTS expression
  - When IN or EXISTS expression is evaluated, recursively invoke the executor to evaluate subquery and test required condition
    - e.g. IN scans the generated results for the LHS value
    - e.g. EXISTS returns true if a row is generated by subquery, or false if no rows are generated by the subquery

Subqueries in WHERE Clause (3)

- IN/NOT IN clauses and EXISTS/NOT EXISTS predicates can also be correlated
  - EXISTS/NOT EXISTS subqueries are almost always correlated
- If subquery is not correlated, can materialize subquery results and reuse them
  - …but they may be large; we may still end up being *verrry* slow
- Previous approach isn't anywhere near ideal
  - IN operator effectively implements a join operation, but without any optimizations
  - EXISTS is a bit faster, but subquery is frequently correlated
- Would greatly prefer to evaluate subquery using joins, particularly if we can eliminate correlated evaluation!

Semijoin and Antijoin

- Two useful relational algebra operations in the context of IN/NOT IN and EXISTS/NOT EXISTS queries
- Relations $r(R)$ and $s(S)$
- The *semijoin* $r \ltimes s$ is the collection of all rows in r that can join with some corresponding row in s
  - $\{ t_r \mid t_r \in r \land \exists\, t_s \in s \,(\, join(t_r, t_s) \,)\, \}$
  - $join(t_r, t_s)$ is the join condition
- $r \ltimes s$ equivalent to $\Pi_R(r \bowtie s)$, but only with sets of tuples
  - If r and s are multisets, these expressions are not equivalent, since a tuple in r that matches multiple tuples in s will become duplicated in the natural join's result

Semijoin and Antijoin (2)

- The *antijoin* $r \triangleright s$ is the collection of all rows in r that don't join with some corresponding row in s
  - $\{ t_r \mid t_r \in r \wedge \neg\exists\, t_s \in s \,(\, join(t_r, t_s)\,)\,\}$
- Also called *anti-semijoin*, since $r \triangleright s$ is equivalent to $r - r \ltimes s$ ($\triangleright$ is the complement of $\ltimes$)
- Both semijoin and antijoin operations are easy to compute with our various join algorithms
  - Can incorporate into theta-join implementations easily
- Can use these operations to restate many IN/NOT IN and EXISTS/NOT EXISTS queries

Example IN Subquery

- Find all bank customers who have an account at any bank branch in the city they live in
  - SELECT * FROM customer c WHERE c.customer_city IN
    ```
    (SELECT b.branch_city
     FROM branch b NATURAL JOIN account a
           NATURAL JOIN depositor d
     WHERE d.customer_id = c.customer_id);
    ```
  - Recall: branches have a branch_name and a branch_city
- Inner query is clearly correlated with outer query
- Naïve correlated evaluation would be very slow ☹
  - Join three tables in inner query for every bank customer!

Example IN Subquery (2)

- Example query:
  - ```
    SELECT * FROM customer c WHERE c.customer_city IN
      (SELECT b.branch_city
        FROM branch b NATURAL JOIN account a
                 NATURAL JOIN depositor d
        WHERE d.customer_id = c.customer_id);
    ```
- Can decorrelate by extracting inner query, modifying it to find all branches for all customers, in one shot:
  - ```
    SELECT *
    FROM branch b NATURAL JOIN account a
             NATURAL JOIN depositor d
    ```
  - Includes tuples for each branch that each customer has accounts at

Example IN Subquery (3)

- Could take our inner query and join it against customer
  - ```
    SELECT c.* FROM customer c JOIN
        (SELECT * FROM branch b NATURAL JOIN account a
                NATURAL JOIN depositor d) AS t
        ON (t.customer_id = c.customer_id AND
              c.customer_city = t.branch_city);
    ```
- Problems?
  - If a customer has multiple accounts at local branches, the customer will appear multiple times in the result
- Cause: the outermost join will duplicate customer rows for each matching row in nested query
- Solution: use a semijoin to join customers to the subquery
    ```

Example IN Subquery (4)

- Our original correlated query:
  - SELECT * FROM customer c WHERE c.customer_city IN
    (SELECT b.branch_city
    FROM branch b NATURAL JOIN account a
    NATURAL JOIN depositor d
    WHERE d.customer_id = c.customer_id);
- The decorrelated query:
  - SELECT * FROM customer c SEMIJOIN
    (SELECT * FROM branch b NATURAL JOIN account a
    NATURAL JOIN depositor d) AS t
    ON (t.customer_id = c.customer_id AND
    c.customer_city = t.branch_city);
- (Note: writing a semijoin in SQL isn't widely supported…)

Example NOT EXISTS Subquery

- A simpler query: find customers who have no bank branches in their home city
  - SELECT * FROM customer c
    WHERE NOT EXISTS (SELECT * FROM branch b
    WHERE b.branch_city = c.customer_city);
- Again, this query requires correlated evaluation
  - Not as bad as previous query, since NOT EXISTS only has to produce one row from inner query, not all the rows…
  - If there's an index on branch_city, this won't be horribly slow, but again, we are implementing a join here
  - *(We have fast equijoin algorithms; why not use them?)*

Example NOT EXISTS Subquery (2)

- Example query:
  - ```
    SELECT * FROM customer c
    WHERE NOT EXISTS (SELECT * FROM branch b
          WHERE b.branch_city = c.customer_city);
    ```
- This query is very easy to write with an antijoin:
  - ```
    SELECT * FROM customer c ANTIJOIN branch b
       ON branch_city = customer_city;
    ```
- Could also write with an outer join:
  - ```
    SELECT c.* FROM customer c LEFT JOIN branch b
       ON branch_city = customer_city
    WHERE branch_city IS NULL;
    ```
  - This approach won't create duplicates of customers, like our previous IN example would have…
    ```

Summary: Nested Subqueries

- Only scratched the surface of subquery translation and optimization
  - An incredibly rich topic – tons of interesting research!
- Can use basic tools we discussed today to decorrelate and optimize a pretty broad range of subqueries
  - Outer joins, sometimes against group/aggregate results
  - Semijoins and antijoins for set-membership subqueries
- An important question, not considered for now:
  - **Is the translated version actually faster?**
    (Or when multiple options, which option is fastest?)
  - A planner/optimizer must make that decision