Relational Database System Implementation

CS122 – Lecture 5

Winter Term, 2018-2019

Last Time: SQL Query Translation

- Began discussing SQL query translation
- <u>Basic</u> SQL syntax maps easily to relational algebra
  - Explored this in CS121
- SELECT * FROM t1, t2, …
  - $t1 \times t2 \times \ldots$
- SELECT * FROM t1, t2, … WHERE P
  - $\sigma_P(t1 \times t2 \times \ldots)$
- SELECT e1 AS a1, e2 AS a2, … FROM t1, t2, … WHERE P
  - $\Pi_{e1 \text{ as } a1, e2 \text{ as } a2, \ldots}(\sigma_P(t1 \times t2 \times \ldots))$

SQL Grouping/Aggregation

- Grouping and aggregation are significantly more difficult
- SELECT g1, g2, ..., e1, e2, ... FROM t1, t2, ... WHERE Pw GROUP BY g1, g2, ... HAVING Ph
  - $g1, g2$, ... are expressions whose values are grouped on
  - $e1, e2$, ... are expressions involving aggregate functions
    - e.g. MIN(), MAX(), COUNT(), SUM(), AVG()
  - Approximately maps to: $\sigma_{Ph}({}_{g1,g2,\ldots}\mathcal{G}_{e1,e2,\ldots}(\sigma_{Pw}(t1 \times t2 \times \ldots)))$
- What makes this challenging:
  - $g1, g2$, ... are not required to be simple column refs
  - $e1, e2$, ... are not required to be single aggregate fns
  - Ph can also contain aggregate function calls not in e_i

SQL Grouping/Aggregation (2)

- This is an acceptable grouping/aggregate query:
  - SELECT a - b AS g, 3 * MIN(c) + MAX(d * e) FROM t GROUP BY a - b HAVING SUM(f) < 20
- Clearly can't use our mapping from last slide:
  - $\sigma_{Ph}({}_{g1,g2,\ldots}\mathcal{G}_{e1,e2,\ldots}(\sigma_{Pw}(t1 \times t2 \times \ldots)))$
  - e.g. *Ph* is SUM(f) < 20, but we don't compute SUM(f) in $\mathcal{G}$ step
- Problem: SQL mixes grouping/aggregation, projection and selection parts of the query together
- Need to rewrite query to separate these different parts
  - Makes translation into relational algebra straightforward

SQL Grouping/Aggregation (3)

- Our initial query:
  - SELECT a - b AS g, 3 * MIN(c) + MAX(d * e) FROM t GROUP BY a - b HAVING SUM(f) < 20
- Step 1: Identify and extract all aggregate functions
  - Replace with auto-generated column references
  - (Use names that users can't enter, e.g. starting with "#")
- Rewrite the query:
  - SELECT a - b AS g, 3 * "#A1" + "#A2" FROM t GROUP BY a - b HAVING "#A3" < 20
    - #A1 = MIN(c) #A2 = MAX(d * e) #A3 = SUM(f)
- Now we know what aggregates we need to compute

SQL Grouping/Aggregation (4)

- Rewritten query:
  - SELECT a - b AS g, 3 * "#A1" + "#A2" FROM t
    GROUP BY a - b HAVING "#A3" < 20
    - #A1 = MIN(c) #A2 = MAX(d * e) #A3 = SUM(f)
- Now we can translate grouping/aggregation and HAVING clause into relational algebra:
  - $\sigma_{\#A3\ <\ 20}({}_{a\ -\ b}\mathcal{G}_{\mathrm{MIN}(c)\ \mathbf{as}\ \#A1,\ \mathrm{MAX}(d\ *\ e)\ \mathbf{as}\ \#A2,\ \mathrm{SUM}(f)\ \mathbf{as}\ \#A3}(t))$
- Finally, wrap this with a suitable project, based on SELECT clause contents
  - $\Pi_{a\ -\ b\ \mathbf{as}\ g,\ 3\ *\ \#A1\ +\ \#A2\ \mathrm{as}\ \text{“3 * MIN(c) + MAX(d * e)”}}(\ \ldots\ )$
  - Note: second expression’s name is implementation-specific
  - Can assign a placeholder name, e.g. “unnamed1”, …
  - Or, can generate a name based on expression being computed

SQL Grouping/Aggregation (5)

- Unfortunately, we still have a problem…
- Our translation: $\Pi_{a\ -\ b\ \mathbf{as}\ g,\ \ldots}(\sigma_{\#A3\ <\ 20}({}_{a\ -\ b}\mathcal{G}_{\ldots}(t)))$
- The project operation can't compute expression a - b
  - a - b is already computed in grouping/aggregation phase
- Before attempting to project, we really also need to substitute in placeholders for grouping expressions
  - SELECT a - b AS g, 3 * "#A1" + "#A2" FROM t
    GROUP BY a - b HAVING "#A3" < 20
    - #A1 = MIN(c) #A2 = MAX(d * e) #A3 = SUM(f)
    - #G1 = a - b

SQL Grouping/Aggregation (6)

- Finally, replace instances of grouping expressions in the SELECT clause with the corresponding names
- Translated:
  - SELECT "#G1" AS g, 3 * "#A1" + "#A2" FROM t GROUP BY a - b [AS "#G1"] HAVING "#A3" < 20
    - #A1 = MIN(c) #A2 = MAX(d * e) #A3 = SUM(f)
    - #G1 = a - b
- Now we can carry on with our project, as before
  - $\Pi_{\#G1\ \mathbf{as}\ g,\ \ldots}(\sigma_{\#A3<20}({}_{a\text{-}b\ \mathbf{as}\ \#G1}\mathcal{G}_{\ldots}(t)))$
- Aside: this also allows us to handle crazy SQL like SELECT 3 * (a - b) AS g, ... GROUP BY a - b ...

SQL Grouping/Aggregation (7)

- Finally, this is an ANSI SQL query:
  - SELECT a - b AS g, 3 * MIN(c) + MAX(d * e) FROM t GROUP BY a - b HAVING SUM(f) < 20
  - GROUP BY and HAVING clauses cannot use SELECT aliases
- Some databases allow the nonstandard "GROUP BY g" instead of requiring the ANSI-standard "GROUP BY a - b"
  - Similarly, HAVING can refer to renamed aggregate expressions
- Can use our alias techniques from earlier
  - e.g. traverse SELECT, record alias: $g = a - b$
  - If query says "GROUP BY g", substitute in definition of g
  - (Apply similar techniques to HAVING clause)

Join Expressions

- Original SQL form:
  - SELECT … FROM t1, t2, … WHERE P
  - List of relations in FROM clause
  - Any join conditions specified in WHERE clause
  - Can't specify outer joins
- 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

Join Expressions (2)

- SQL FROM clauses can be much more complex:
  - SELECT * FROM t1, t2 LEFT JOIN t3 ON (t2.a = t3.a) WHERE t1.b > t2.b;
  - FROM clause is comma-separated list of join expressions
- JOIN expressions are binary operations…
  - Operate on two relations; left-associative
- Similarly, interpret FROM *join_expr*, *join_expr* as a binary operation
  - A Cartesian product between two join expressions
  - Expressions themselves may involve JOIN operations (the "," operator is lower precedence than JOIN keyword)

Join Expressions (3)

- FROM clause is parsed into a binary tree of join exprs
  - Can use parentheses to override precedence, where necessary
- This binary tree is straightforward to translate
  - Translate left subtree into relational algebra plan
  - Translate right subtree into relational algebra plan
  - Create a new plan from these subtrees based on the kind of join being performed
- Note: This is a naïve translation of the join expression, and probably horribly inefficient
  - Will discuss solutions for this in the future

Join Expression Details

- 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)

Join Expression Details (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;

Join Expression Details (3)

- SQL-92 join syntax:
  - 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 $t1.* \cup t2.*$

Join Expression Details (4)

- 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)

Join Expression Details (5)

- 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 values of 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

Join Expression Details (6)

- 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)

Join Expression Details (7)

- 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: $a = t3.a$
  - <u>Problem</u>: column-reference *a* is ambiguous

Join Expression Details (8)

- 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)

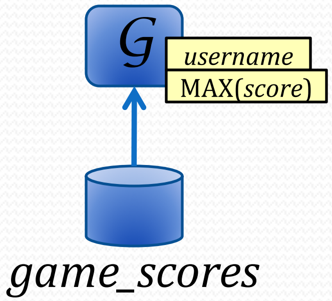
- Subqueries can also appear in the WHERE clause:
  - SELECT employee_id, last_name, first_name
    FROM employees e WHERE e.is_manager = 0 AND
    EXISTS (SELECT * FROM employees m
    WHERE m.department = e.department AND
    m.is_manager = 1 AND m.salary < e.salary);
  - *(Find non-manager employees 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. ☹

Subqueries in FROM Clause

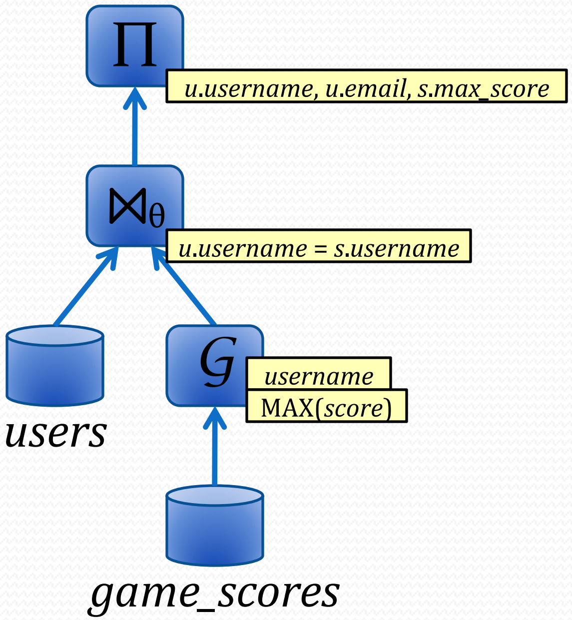
- FROM subqueries are the easiest to deal with! ☺
  - SELECT u.username, email, max_score
    FROM users u,
    (SELECT username, MAX(score) AS max_score
    FROM game_scores GROUP BY username) AS s
    WHERE u.username = s.username;
- To generate execution plan for full query:
  - Simply generate execution 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