Relational Database System Implementation

CS122 – Lecture 6

Winter Term, 2018-2019

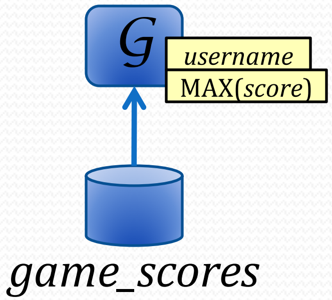
Last Time: Subqueries

- Began discussing translation of SQL subqueries
- FROM subqueries are the easiest to deal with
- 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)

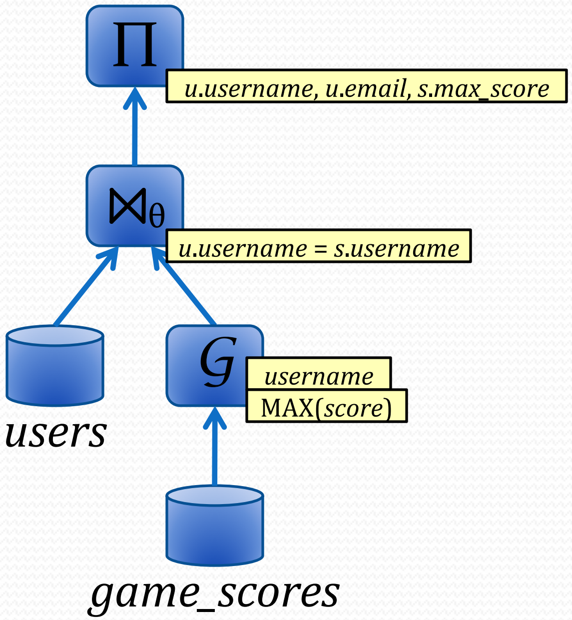
3

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:



4

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
  - $\{ t_r \mid t_r \in r \wedge \exists\, t_s \in s\ (\, join(t_r, t_s)\ )\ \}$
  - $join(t_r, t_s)$ is the join condition
- $r \ltimes s$ equivalent to $\Pi_R(r \bowtie s)$, but only with <u>sets</u> 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 * 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 <u>very</u> 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 branch_city, customer_id
    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 branch_city, customer_id
     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 branch_city, customer_id
     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

Plan-Node Implementations

- Previously: To evaluate SQL queries, we must…

1. Implement relational algebra operations in some way
2. Translate the SQL abstract syntax tree (AST) into a corresponding relational algebra plan
   - Covered a variety of naïve translations that will work
3. Figure out how to evaluate plan and generate results
   - We will use a pull-based, pipelined evaluation of query plans

- Still need implementations of our plan nodes

Select Implementations

- Select σ plan-nodes are easy to implement
  - Retrieve tuples from child plan-node (or from a table) until predicate is true, then pass the tuple to parent
    - An unspecified predicate is treated as *true*
- Several different kinds, based on source of tuples
  - File-scan through a table – no children; reads from table
  - Simple filter plan-node – one child plan-node
  - *(Also index-scans – will discuss in a later lecture…)*
- If select predicate has an equality condition on a key, it can stop once it returns its first row
  - Halves the expected cost of the select operation

Project Implementations

- Project Π plan-nodes are also easy to implement
  - Retrieve next tuple from child plan-node, and compute an output tuple based on the project criteria
- Project expressions are evaluated in the context of the child node's schema and tuple data
  - Child schema specifies variable names; tuples specify values

- Both selects and projects can have a "hidden" cost:
  - If planner/optimizer is not able to rewrite subqueries in the SELECT clause, or in a WHERE/HAVING clause, either of these plan-nodes could end up doing correlated evaluation

Group/Aggr. Implementations

- Implementing grouping and aggregation is similarly straightforward
- If input tuples are sorted on grouping attributes, can implement a sort-based grouping/aggregation node
- For each input tuple:
  - If grouping-attribute values changed from previous input (or child plan-node finishes producing tuples) then the current group is completed
  - Output a tuple containing grouping-attribute values, and also aggregate function values
  - Reset aggregates, store new group-attribute values, and begin calculating the new group's aggregates

Group/Aggr. Implementations (2)

- Sketch of sort-based implementation: ${}_{g1,g2,\ldots}\mathcal{G}_{e1,e2,\ldots}(E)$

```
current_group = []
current_aggregates = []
do:
   t := next tuple from E
   if t != null:
      group := compute g1, g2, … using t
   if t == null or group != current_group:            // Current group is done
      add join(current_group, current_aggregates) to result
      current_group := group
      reset current_aggregates
   update current_aggregates using t
while t != null
```

Group/Aggr. Implementations (3)

- Aggregate functions work differently from simple scalar functions
  - Simple functions take inputs and return an output
- Aggregate functions are fed a sequence of input values, and update their aggregate state with each input
- Example: MIN(x) aggregate function
  - As a group of input tuples is being consumed, MIN(x) function is handed each input value in sequence
  - When group of input tuples is completed, MIN(x) function can be queried for its aggregate result

Group/Aggr. Implementations (4)

- If input tuples aren't sorted on grouping attributes then a hash-based implementation must be used
- Plan-node maintains a hash-table that maps distinct values of $\langle g1, g2, \ldots\rangle$ to aggregate functions $\langle e1, e2, \ldots\rangle$
- No way of knowing when all tuples for a given group have been seen…
  - Hash-based implementation can't output any results until all input tuples have been seen
- This can have serious memory implications for large data sets with large numbers of distinct groups
  - Must use external memory if internal memory overflows

Group/Aggr. Implementations (5)

- Sketch of hash-based implementation: ${}_{g1,g2,\ldots}\mathcal{G}_{e1,e2,\ldots}(E)$

  // Compute all groups, and their corresponding aggregates

  group_aggregates = {}

  while E has more tuples:

     t := next tuple from E

     group := compute $g1$, $g2$, … using t

     aggregates := *group_aggregates*[*group*] // Add entry if missing

     update *aggregates* using t

  // Output all of our computed groups and aggregates as tuples

  for *group*, *aggregates* in *group_aggregates*:

     add join(*group*, *aggregates*) to result