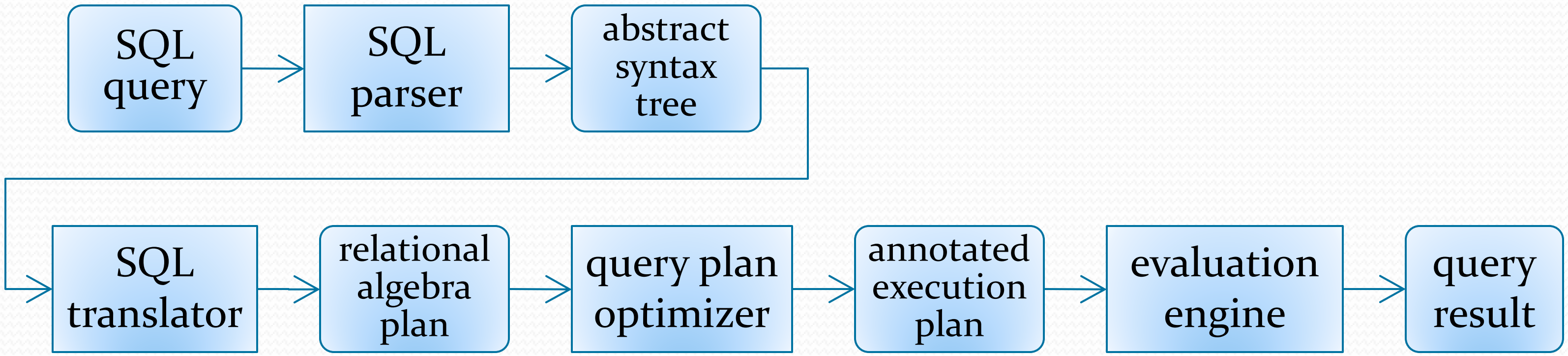
Relational Database System Implementation

CS122 – Lecture 4

Winter Term, 2017-2018

SQL Query Translation

- Last time, introduced query evaluation pipeline

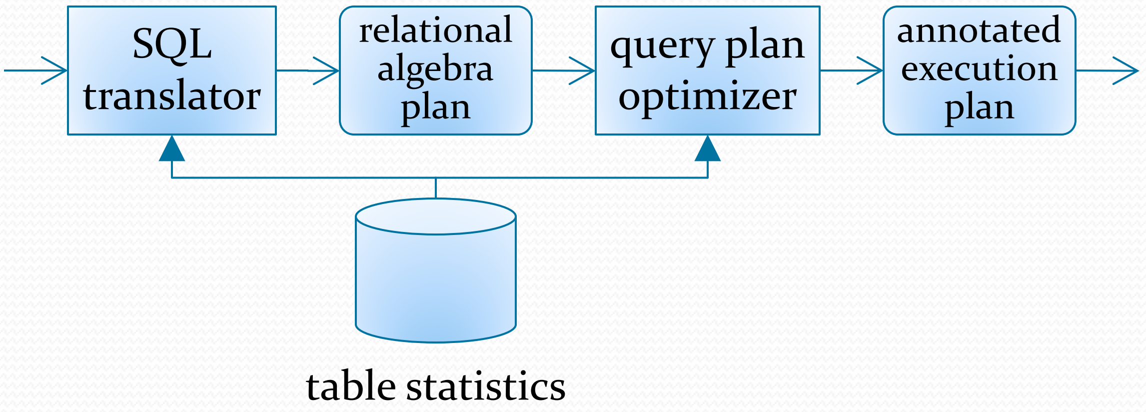


- To evaluate SQL queries, must solve several problems:

1. Implement relational algebra operations in some way
2. Translate the SQL abstract syntax tree (AST) into a corresponding relational algebra plan
3. Figure out how to evaluate plan and generate results

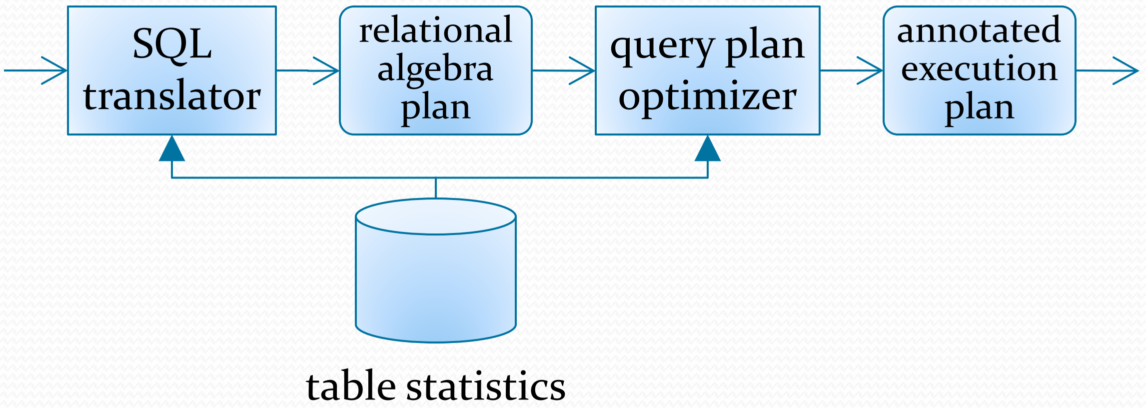
Plan Creation and Optimization

- Some databases use slightly different representations between initial query plan and optimized plan
  - e.g. initial plan uses abstract relational algebra expressions without any implementation details at all
  - Query optimizer adds in these details as annotations
- Annotated plan nodes are called *evaluation primitives*
  - They can be directly used to evaluate the query plan



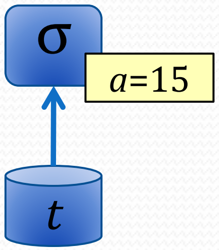
Plan Creation and Optimization

- Other databases use the same representation for both
  - All generated plans contain implementation details
  - Initial query plans may be very unoptimized and use the slowest, most general implementations
  - Optimizations can replace slow implementations with faster ones, and/or apply other transformations
- (NanoDB uses this approach)



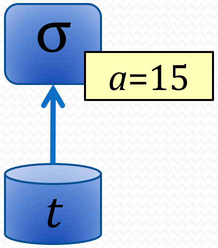
Evaluation Primitives

- Implementations of relational algebra operations are called evaluation primitives
- Don't always correspond directly to relational algebra
- Example:
  - SELECT * FROM t WHERE a = 15
  - $\sigma_{a=15}(t)$
- If t is a heap file:
  - Could create two components, a select node, and another file-scan node that produces all tuples in t

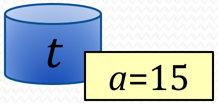


Evaluation Primitives (2)

- Example:
  - SELECT * FROM t WHERE a = 15
  - $\sigma_{a=15}(t)$



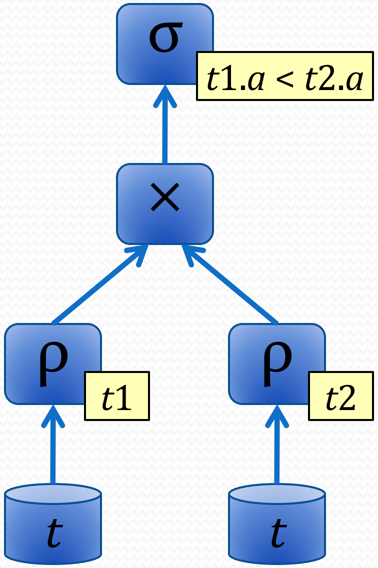
- What if t is ordered or hashed on attribute a?
  What if t has an (ordered or hashed) index on a?
  - Can't really take advantage of file organization or other access paths if select-predicate is applied separately



- Can also create a file-scan node with a predicate
- Evaluation primitives are often more powerful than their corresponding relational algebra operations
  - Allows us to optimize the implementations, then use the optimizations when constructing our plans

Evaluation Primitives (3)

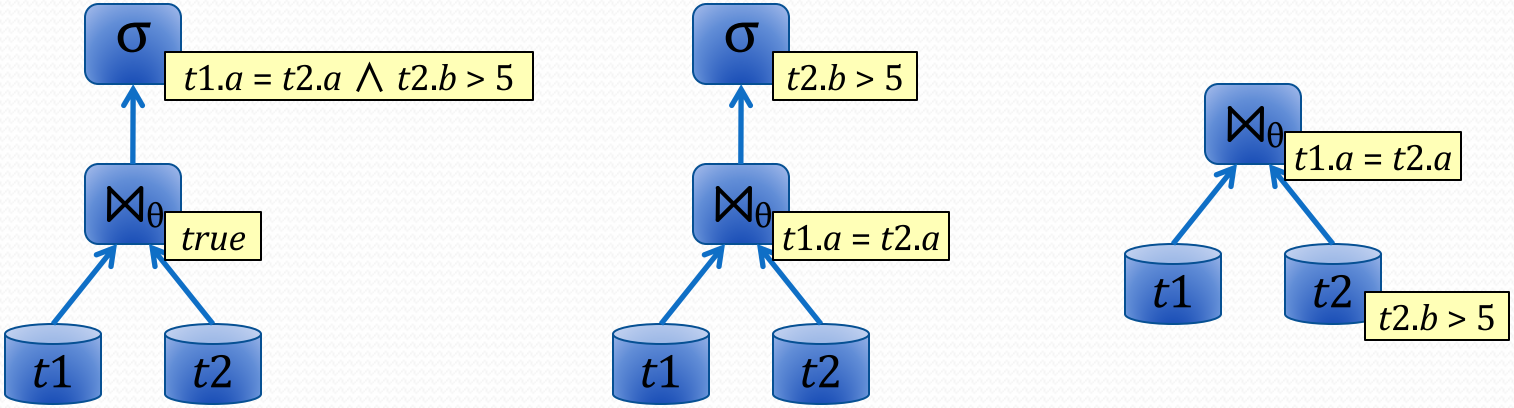
- Example:
  - SELECT * FROM t AS t1, t AS t2
    WHERE t1.a < t2.a
- Table *t* is accessed twice, and is renamed in query plan
- Insert extra rename nodes into plan?
  - Sole operation is to rename table in node's output schema…
  - (This is NanoDB's approach.)



- Or, give plan nodes ability to handle simple renaming ops?
  - When plan nodes produce their schemas, can easily apply renaming at that point

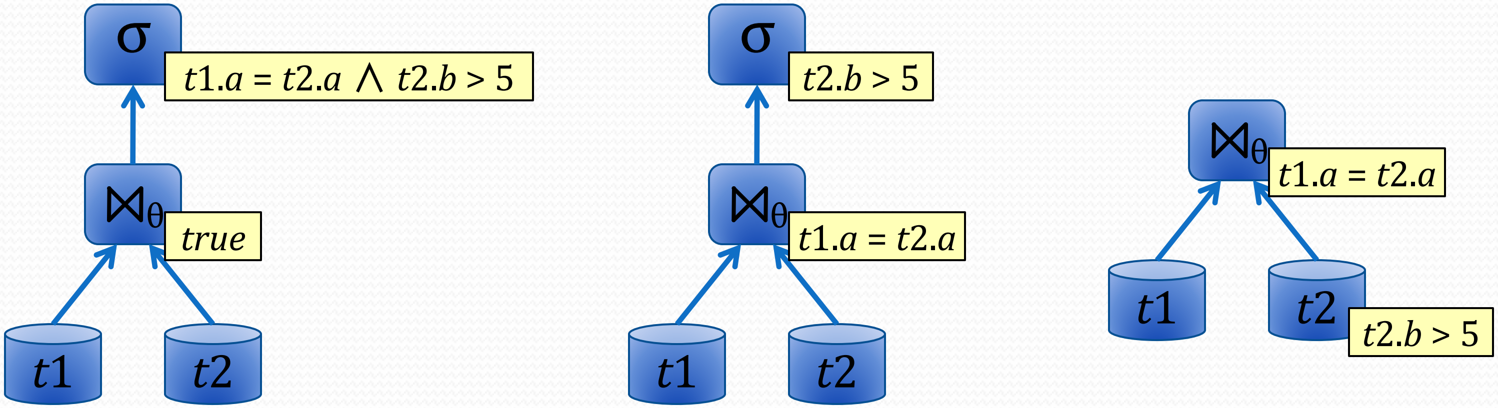
Evaluation Primitives (4)

- Join operations usually implemented with theta-join
  - More advanced/flexible than simple translation using Cartesian product, or simple natural-join operator
  - Implementation can also be configured to produce inner join, or left/right/full outer join, where supported
- SELECT * FROM t1, t2 WHERE t1.a = t2.a AND t2.b > 5;
- Can evaluate in multiple ways:



Evaluation Primitives (5)

- SELECT * FROM t1, t2 WHERE t1.a = t2.a AND t2.b > 5;

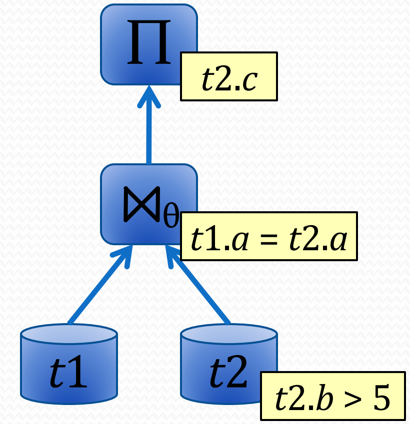


- Ideally, can implement theta-join to take advantage of join condition when possible
  - Perform equijoins more quickly
  - Take advantage of ordered data, or indexes on inputs

Evaluation Primitives (6)

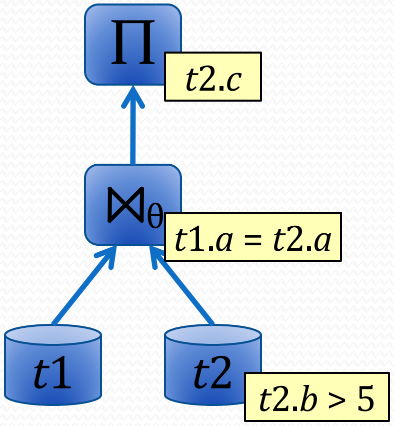
- Many join implementations can do several kinds of join
  - Implement inner join, left outer join, full outer join
  - Implement semijoin and antijoin operations as well (will discuss more in a future lecture)
  - Configure plan node to do the required operation in plan
- By combining multiple operations in plan nodes:
  - Can implement wide range of queries without needing large, complicated plans, or many kinds of plan nodes
  - Can take advantage of certain cases to implement the operation in a much faster way

Plan Evaluation



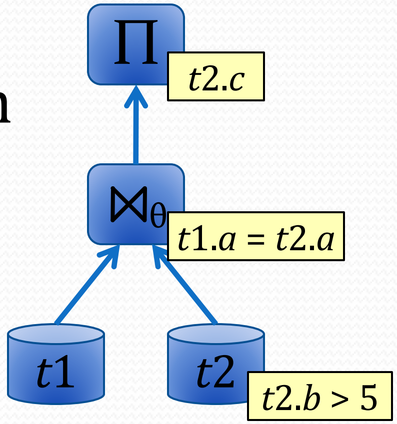
- Previous example, slightly altered:
  - SELECT c FROM t1, t2
    WHERE t1.a = t2.a AND t2.b > 5
- One evaluation approach:
  - Each node is evaluated completely, and its results are saved in a temporary table (postorder tree traversal)
    - "Evaluate" $t1$ → $t1$ (no-op)
    - Evaluate $\sigma_{b>5}(t2)$ → *temp1*
    - Evaluate $\bowtie_{t1.a=t2.a}(t1, temp1)$ → *temp2*
    - Evaluate $\Pi_{t2.c}(temp2)$ → result

Plan Evaluation (2)



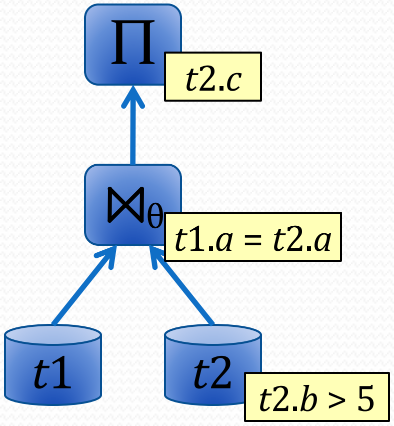
- Called *materialized evaluation*
  - Each node's results are *materialized* into a temporary table (possibly onto disk)
- Issues with this approach?
  - For large tables, causes many additional disk accesses on top of ones already required for plan-node evaluation!
  - (Small temporary results can be held in memory.)
- Another evaluation approach: *pipelined evaluation*
  - Evaluate multiple plan nodes simultaneously
  - Results are passed tuple-by-tuple to the next plan node

Plan Evaluation (3)



- Several ways to implement pipelined evaluation
- *Demand-driven* pipeline:
  - Rows are requested (pulled) from top of plan
  - When a plan-node must produce a row, it requests rows from its child nodes until it can produce one
- Example:
  - Top-level output loop requests a row from $\Pi_{t2.c}$ node
  - $\Pi_{t2.c}$ node requests the next row from $\bowtie_{t1.a=t2.a}$ node
  - $\bowtie_{t1.a=t2.a}$ node requests rows from its children until it can produce a joined row
  - $\sigma_{t2.b>5}$ node scans through $t2$ until it finds a row with $b > 5$

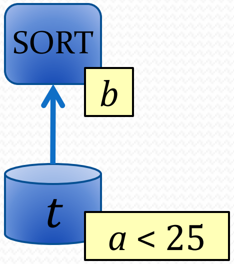
Plan Evaluation (4)



- *Producer-driven* pipeline:
  - Each plan-node independently generates rows and pushes them up the plan
  - Plan nodes communicate via queues
- Primarily used in parallel databases
  - Planner hands subplans (or individual plan nodes) to different processors to compute
  - Processors independently evaluate plan components and push tuples to the next stage in the plan
- Sequential databases generally use demand-driven pipelines for query evaluation

Blocking Operations

- Not all operations can be pipelined
- An obvious one: sorting
  - SELECT * FROM t WHERE a < 25 ORDER BY b;
  - Sort plan-node must completely consume its input before it can produce any rows



- These are called *blocking operations*
- Some databases take blocking operations into account
  - e.g. PostgreSQL's planner computes two estimates for each plan node:
    - the cost to produce all rows
    - the cost to produce the first row
  - For e.g. EXISTS subquery, want to minimize time to first row

Blocking Operations (2)

- Some operations can be implemented in blocking or in pipelined ways
- Grouping/aggregation operation
  - SELECT username, SUM(score) AS total_score
    FROM game_scores GROUP BY username;

    $_{username}\mathcal{G}_{\textbf{sum}(score)\ \textbf{as}\ total\_score}(game\_scores)$
- If incoming tuples are already sorted on *username*:
  - Can apply aggregate function to runs of tuples with same *username* value, and produce output rows along the way
- If incoming tuples are not sorted on *username*:
  - Must either use a hash-table, or must sort internally
  - Either way, the operation will be blocking

SQL Query Translation (2)

- For now, ignore the question of how to implement specific relational algebra operations
  - (Most are straightforward anyway)
- SQL doesn't map directly to the relational algebra
  - Nested subqueries!!!! Correlated evaluation!!!!
  - Grouping and aggregation is also complicated
- <u>Basic</u> SQL syntax maps easily to relational algebra
  - Explored this in CS121

Mapping Basic SQL Queries

- 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, …
  - $e1, e2, \ldots$ are expressions using columns in $t1, t2, \ldots$
  - $a1, a2, \ldots$ are aliases (alternate names) for $e1, e2, \ldots$
  - $\Pi_{e1 \text{ as } a1, e2 \text{ as } a2, \ldots}(t1 \times t2 \times \ldots)$
- SELECT e1 AS a1, e2 AS a2, … FROM t1, t2, … WHERE P
  - $\Pi_{e1 \text{ as } a1, e2 \text{ as } a2, \ldots}(\sigma_P(t1 \times t2 \times \ldots))$

Mapping Basic SQL Queries (2)

- SELECT e1 AS a1, e2 AS a2, … FROM t1, t2, … WHERE P
  - $\Pi_{e1,e2,\ldots}(\sigma_P(t1 \times t2 \times \ldots))$
- This mapping is somewhat confusing, because many DBs accept queries that don't work with this translation
- Example: SELECT a + c AS v FROM t WHERE v < 25;
  - Following the above mapping: $\Pi_{a+c\ \mathbf{as}\ v}(\sigma_{v<25}(t))$
  - Doesn't make sense; *v* isn't defined in select predicate!

- The SQL standard is very clear (and simple!):
  - P is only allowed to refer to columns in the FROM clause
  - (ignoring correlated evaluation for the time being)

Mapping Basic SQL Queries (3)

- Can easily support non-standard syntax by recording select-clause aliases in the AST representation
- Example: SELECT a + c AS v FROM t WHERE v < 25;
  - Traverse SELECT clause; record alias: $v = a + c$
  - In the WHERE predicate: anytime v is used, replace it with expression $a + c$
    - Also do this with ON clauses in joins, HAVING clauses, etc.
  - Allows us to follow previous mapping: $\Pi_{a+c\ \mathbf{as}\ v}(\sigma_{a+c<25}(t))$

- Other techniques as well, but same idea

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:
  - $\sigma_{\#A3\ <\ 20}({}_{a\ -\ b}\mathcal{G}_{\text{MIN}(c)\ \textbf{as}\ \#A1,\ \text{MAX}(d\ *\ e)\ \textbf{as}\ \#A2,\ \text{SUM}(f)\ \textbf{as}\ \#A3}(t))$
- Finally, wrap this with a suitable project, based on SELECT clause contents
  - $\Pi_{a\ -\ b\ \textbf{as}\ g,\ 3\ *\ \#A1\ +\ \#A2\ \text{as “3 * MIN(c) + MAX(d * e)”}}$ (…)
  - 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