Compilation

A query is first parsed into a parse tree based on the grammar for SQL.

The parse tree is then transformed into a query plan, which is a tree with leaves labeled as relations and non-leaves labeled as relational algebra operations (with subscripts); the children of a node are the arguments of the operation labeling that node.

The query plan is then optimized by transforming it into one that is more efficient.

Query Plans

A simple query like SELECT S FROM R1, R2 WHERE C; is translated into a query plan like this:

π_S

|

σ_C

|

x

/ \

R1 R2

Subqueries

SELECT S FROM R WHERE A IN SELECT B ...; if inner select is uncorrelated:

π_S

|

$\sigma_{R.A=B}$

|

x

/ \ (may need to insert δ)

R π_B

| (rest of subquery)

Correlated Subqueries

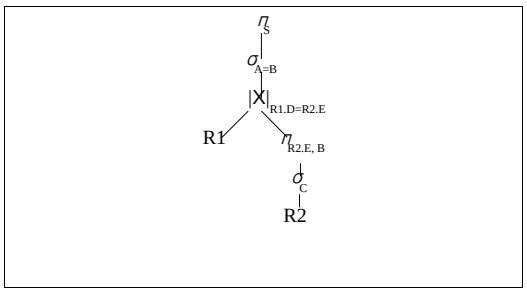
Correlated subqueries must be modified pulling out the part of the condition containing the correlation variables, extend the tuples returned by the subqueries with extra attributes, and selecting on the basis of the pulled-out condition after joining the relation(s) with the results of the subqueries.

Example: SELECT S FROM R1

WHERE A IN SELECT B FROM R2

WHERE C AND R1.D=R2.E

Query Plan for Example



Transformations

One way to improve query plans is to reduce the size and number of tuples being processed as soon as possible. This can be done with transformations based on equivalences.

Some transformations are also used to break a query down into a combination of simpler queries that can be more efficiently computed. Example: a selection involving required values of two attributes (A1=v1 AND A2=v2) can be restructured to do an efficient selection using only one of the attributes, followed by a selection based on the other attribute.

Selection Transformations

$\sigma_{C1 \text{ AND } C2}(R) = \sigma_{C1}(\sigma_{C2}(R)) = \sigma_{C2}(\sigma_{C1}(R))$

$\sigma_{C1 \text{ OR } C2}(R) = \sigma_{C1}(R) \cup \sigma_{C2}(R)$

$\sigma_{C}(R \cup S) = \sigma_{C}(R) \cup \sigma_{C}(S)$

$\sigma_{C}(R - S) = \sigma_{C}(R) - S$

$\sigma_{C}(R - S) = \sigma_{C}(R) - \sigma_{C}(S)$

If C depends only on attributes in R, [similar equations if C depends only on S]

$\sigma_{C}(R \cap S) = \sigma_{C}(R) \cap S$

$\sigma_{C}(R \times S) = \sigma_{C}(R) \times S$ (similarly for joins)

More Selection

If R has the attributes in C and S has the attributes in C,

$\sigma_{C}(R \cap S) = \sigma_{C}(R) \cap \sigma_{C}(S)$

$\sigma_{C}(R \,|X|\, S) = \sigma_{C}(R) \,|X|\, \sigma_{C}(S)$

Note: sometimes several equations can be combined to get an improvement. Example:

$\sigma_{C}(R) \,|X|\, S = \sigma_{C}(R \,|X|\, S) = \sigma_{C}(R) \,|X|\, \sigma_{C}(S)$

if R has the attributes in C and S has the attributes in C.

Projection

In subscript $_{expr \rightarrow x}$ the attributes in $_{expr}$ are input attributes and $_{x}$ is an output attribute. In subscript $_{x}$ x is both an input and output attribute.

If L is a projection list, M is the join attributes common to R and S and the input attributes of L that are in R, and N is the join attributes common to R and S and the input attributes of L that are in S, then

$\pi_{L}(R \,|X|\, S) = \pi_{L}(\pi_{M}(R) \,|X|\, \pi_{N}(S))$

Similarly for other joins, and for product.

More Projection

$\pi_{L}(R \cup_{B} S) = \pi_{L}(R) \cup_{B} \pi_{L}(S)$

(Won't work for set union, for intersection or difference)

The following is sometimes useful

$\pi_{L}(\sigma_{C}(R)) = \pi_{L}(\sigma_{C}(\pi_{M}(R)))$

where M lists the input attributes of L and the attributes in C.

Joins, Products

$R \,|X|_{C}\, S = \sigma_{C}(R \times S)$

$R \,|X|\, S = \pi_{L}(\sigma_{C}(R \times S))$ where C equates the attributes that are common to R and S, and L lists all the attributes of R and S but without duplication of the common attributes.

The above equations are usually used to transform products into joins, which are more efficient to compute.

Duplication Elimination

$\delta(R) = R$

Useful when we can prove that R will have no duplicates (a stored relation with a primary key, the result of a grouping operation, or a set operation).

$\delta(R \times S) = \delta(R) \times \delta(S)$

Similarly for joins.

$\delta(\sigma_{C}(R)) = \sigma_{C}(\delta(R))$

$\delta(R \cap_{B} S) = \delta(R) \cap_{B} S = R \cap_{B} \delta(S) = \delta(R) \cap_{B} \delta(S)$

Grouping

$\delta(\gamma_L(R)) = \gamma_L(R)$

$\gamma_L(R) = \gamma_L(\delta(R))$ provided L involves only MIN and MAX aggregate operators.

$\gamma_L(R) = \gamma_L(\pi_M(R))$ where M contains at least the attributes of R that are in L.

Other Transformations

Operators that are associative and commutative can be rearranged for increased efficiency.

If natural joins and theta joins are combined, the natural joins can be converted to theta joins so that the joins can be rearranged.

Estimating Size of a Selection

$S = \sigma_{a=v}(R)$ Assume $T(S) = T(R)/V(R,a)$

$S = \sigma_{a<v}(R)$ Assume $T(S) = T(R)/3$ (a heuristic)

$S = \sigma_{a\neq v}(R)$ Assume $T(S) = T(R)$ or assume $T(S) = T(R)*(V(R,a) - 1)/V(R,a)$

$S = \sigma_{\text{not } C}(R)$ Assume $T(S) = T(R) - T(\sigma_C(R))$

$S = \sigma_{C1 \text{ or } C2}(R)$ Assume $T(S) = T(\sigma_{\text{not(not C1 and not C2)}}(R))$

Remember that $\sigma_{C1 \text{ and } C2}(R) = \sigma_{C1}(\sigma_{C2}(R))$

Estimating Size of a Join

Assume $T(R \mid X \mid S) = T(R)*T(S)/\max(V(R,a),V(S,a))$ when R and S have one attribute (a) in common. To see this, assume uniform distribution of values for attribute a. For a given value of a, there are T(R)/V(R,a) tuples in R with that value, and T(S)/V(S,a) tuples in S with that value, so the number of join tuples with that value is (T(R)/V(R,a))*(T(S)/V(S,a)). There are V(R,a) distinct values of a in R, so the total number of join tuples is V(R,a)*(T(R)/V(R,a))*(T(S)/V(S,a)) = T(R)*T(S)/V(S,a). Alternatively, multiply by V(S,a) to get T(R)*T(S)/V(R,a). Use the smaller estimate (divide T(R)*T(S) by max(V(R,a),V(S,a))).

Alternative Argument

For each of the T(R) tuples in R, there are T(S)/V(S,a) tuples in S with the same value for a, so the total number of join tuples is estimated to be T(R)*T(S)/V(S,a). Or, starting with S, for each of the T(S) tuples in S, there are T(R)/V(R,a) tuples in R with the same value for a, so the total number of join tuples is estimated to be T(R)*T(S)/V(R,a). Take the smaller estimate.

Multiple Join Attributes

Suppose that the attributes common to R and S are $a_1, ..., a_k$. Estimate $T(R \mid X \mid S)$ this way:

$T = T(R)*T(S)$

for i from 1 to k, $T = T/\max(V(R,a_i), V(S,a_i))$

$T(R \mid X \mid S) = T$

This assumes that the distributions of the attribute values are uniform and independent of each other.

Multiple Joins

$T(R_1|X| \cdots |X| R_n)$ can be estimated this way:

$T = T(R_1) * \cdots * T(R_n)$

for each attribute a that is common to two or more of the relations, if a is common to k relations,

T = T/ the product of the k-1 largest values in $\{V(R_1,a), ..., V(R_n,a)\}$

$T(R_1|X| \cdots |X| R_n) = T$

Heuristics for Other Operations

Estimate $T(R \cup_B S) = T(R) + T(S)$

Estimate $T(R \cup_S S)$ = The larger + ½ of the smaller

Estimate $T(R \cap S)$ = ½ of the smaller

Estimate $T(R - S) = T(R) - T(S)/2$

Estimate $T(\delta(R(a_1,...,a_n)))$ = smaller of T(R)/2 and

$V(R,a_1) * \cdots * V(R,a_n)$

Estimate $T(\gamma_L(R))$ = smaller of T(R)/2 and $V(R,a_1) * \cdots * V(R,a_n)$

where the a_i are the grouping attributes.

Enumeration of Physical Query Plans

1. Exhaustive enumeration – Try all possible transformations on the original query plan, assign all possible algorithms for computing the individual operations, compute the disc access estimates.
2. Heuristic selection – Use heuristics to choose which algorithm to use for each operation.

Examples of Heuristics

- To compute $\sigma_{a=c}(R)$, if R has an index on a, use the index to look up the tuples with a = c.
- To compute $\sigma_{a=c \text{ and } C}(R)$, if R has an index on a, use the index to look up the tuples with a = c, and immediately test whether C is true on each tuple as it is retrieved.
- To compute a join, if one relation has an index on the join attributes, use an index-join with that relation in inner loop.
- To compute a join, if one relation is sorted on the join attributes, choose a sort-join over a hash-join.
- For union or intersection of 3 or more relations, combine the smallest relations first.

More Enumerations

3. Branch-and-bound – Abandon a physical plan as soon as part of it costs more than the best plan found so far. If the whole plan costs less than the best plan so far, it becomes the new best plan found so far.
4. Hill climbing – Start from one plan. Consider several changes to it, choose the change that has the least overall cost. Common changes are choice of algorithm for an operator, or order of doing associative operations.
5. Dynamic programming – Working from leaves to root, choose the best way to compute the operation at a node given the previously computed best way for computing the operations at its child nodes.

More Enumerations cont.

6. Selinger-style optimization – Keep several plans for computing the operation at a node, the best as in dynamic programming, and some that produce tuples in an order that might be useful higher up in the plan.

Heuristics for Joins

- For the join of two relations, let the smaller relation be the build relation and the larger relation be the probe relation.
- For the join of three or more relations, use only left-deep (left-branching) join trees. There are fewer of them and they tend to be more efficient.
- Use a greedy algorithm to do join of three or more relations – Start with the smallest relation, join it with the next smallest relation, join the result with the next smallest relation, etc.

Choose Selection Methods

- For $\sigma_{C}(R)$ and no useful index on R, scan all of R, testing for C. Cost is B(R) or T(R).
- For $\sigma_{a=c \text{ and } C}(R)$ and an index on a, retrieve tuples for a = c and filter them with C test. Cost is B(R)/V(R,a) or T(R)/V(R,a).
- For $\sigma_{a<c \text{ and } C}(R)$ and other inequalities on a, and an index on a, retrieve the tuples satisfying the inequality and filter them with C test. Cost is B(R)/3 or T(R)/3.

Choose Join Methods

- Choose among sort-joins, hash-joins, indexed joins if we have enough info about the size of the tables and the space available in memory for buffers.
- Choose a one-pass join (all of the smaller table is put in memory and accessed by the join attributes), or if the smaller table is not much larger than memory, use a nested-loop join, with the smaller table in the outer loop.
- Choose a sort-join if one or more tables are already sorted on the join attributes or two or more joins use the same join attributes.
- If one table is very small and the other table is indexed on the join attributes, use an indexed join.

Choose Join Methods cont.

- If tables are not already sorted or indexed and a multi-pass method is needed, choose a hash-join.

Pipelining, Materialization

Writing the output of an operation to disc is called materialization. It is more efficient to pipeline; use the output buffer block as an input buffer block for the next operation. Iterators facilitate the necessary interleaving of code for the two operations.

Split the physical query plan into subtrees at the boundaries where intermediate tables get materialized. Generate code for each subtree using iterators. Put the blocks of code in a bottom-up, left-to-right order.