An Overview of Query Optimization

Why? Improving query processing time. Fast response is important in several db applications.

Is it possible? Yes, but relative. A query can be evaluated in several ways. We can list all of them and then find the best among them. Sometime, we might not be able to do it because of the number of possible ways to evaluate the query is huge. We need to settle for a compromise: we try to find one that is reasonably good.

Typical Query Evaluation Steps in DBMS. The DBMS has the following components for query evaluation: SQL parser, quey optimizer, cost estimator, query plan interpreter. The SQL parser accepts a SQL query and generates a relational algebra expression (sometime, the system catalog is considered in the generation of the expression). This expression is fed into the query optimizer, which works in conjunction with the cost estimator to generate a query execution plan (with hopefully a reasonably cost). This plan is then sent to the plan interpreter for execution.

How? The query optimizer uses several heuristics to convert a relational algebra expression to an equivalent expression which (hopefully!) can be computed efficiently. Different heuristics are (see explaination in textbook):

(i) transformation between selection and projection;

1. *Cascading of selection:* $\sigma_{C_1 \wedge C_2}(R) \equiv \sigma_{C_1}(\sigma_{C_2}(R))$
2. *Commutativity of selection:* $\sigma_{C_1}(\sigma_{C_2}(R)) \equiv \sigma_{C_2}(\sigma_{C_1}(R))$
3. *Cascading of projection:* $\pi_{Atts}(R) = \pi_{Atts}(\pi_{Atts'}(R))$ if $Atts \subseteq Atts'$
4. *Commutativity of projection*: $\pi_{Atts}(\sigma_C(R)) = \sigma_C(\pi_{Atts}((R))$ if $Atts$ includes all attributes used in C.

(ii) transformation between Cartesian product and join;

1. $R \bowtie S \equiv S \bowtie R$
2. $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$
3. $R \times S \equiv S \times R$
4. $R \times (S \times T) \equiv (R \times S) \times T$

(iii) pushing selection and projection through join and Cartesian product:

1. $\sigma_C(R \times S) \equiv R \bowtie_C S$ if C uses attributes of both R and S
2. $\sigma_C(R \times S) \equiv \sigma_C(R) \times S$ if C uses attributes of only R
3. $\sigma_C(R \bowtie S) \equiv \sigma_C(R) \bowtie S$ if C uses attributes of only R
4. $\pi_{Atts}(R \times S) \equiv \pi_{Atts}(\pi_{Atts'}(R) \times S)$ if $Atts'$ are attributes in R and $(Atts \cap Attributes(R)) \subseteq Atts'$
5. $\pi_{Atts}(R \bowtie_C S) \equiv \pi_{Atts}(\pi_{Atts'}(R) \bowtie_C S)$ if $Atts'$ are attributes in R and $(Atts \cap Attributes(R)) \subseteq Atts'$

These algebraic equivalence rules are applied in the order (6 steps, book).

1. Use cascading of selection to break up conjunction in selection condition
2. Use commutative of selection to push selection through joins to propage selection as far inside the query as possible
3. Combine Cartesian product and selection to form joins
4. Use associativity rules to rearrange the order of join operations
5. Use cascading of projection to propage projection as far inside the query as possible
6. Identify the operations that can be processed at the same pass (*pipelining*)

Notes: The order of the rule application determines the query execution plan. The cost of execution of a query execution plan can be estimated based on the information in the system catalog.

What do you need for this chapter? You will need to know the basics of query processing: how are the basic operators evaluated, the cost associated to the execution of each operator, what is the role of indexes in query processing, what is access path, which access path should be selected for which query?

Estimating the size of the output: Use the system catalog to determine the *weight* of an attribute in a relation. You need to understand the basics of it. The system catalog contains the following information:

- $Blocks(R)$ – the number of blocks (pages) occupied by R;
- $Tuples(R)$ – the number of tuples in the instance of R;
- $Values(R.A)$ – the number of distinct values of attribute A in the instance of R;
- $MaxVal(R.A)$ – the maximum value of attribute A in the instance of R;
- $MinVal(R.A)$ – the maximum value of attribute A in the instance of R;

For the query:

$Query$ = `SELECT` $TargetList$ `FROM` R_1 V_1`, ...,` R_nV_n `WHERE Condition`

the **reduction factor** is defined by

$$\frac{Tuples(ResultSet)}{Tuples(R_1) \times \ldots \times Tuples(R_n)}$$

that can be computed by

$$reduction(Query) = reduction(TargetList) \times reduction(Condition)$$

where for $Condition = Condition_1$ `AND` $Condition_2$,

$$reduction(Condition) = reduction(Condition_1) \times reduction(Condition_2)$$

and for atomic conditions:

- $reduction(R_i.A = value) = \frac{1}{Values(R_i.A)}$ (assumption: all values are equally probable)
- $reduction(R_i.A = R_j.B) = \frac{1}{\max(Values(R_i.A), Values(R_j.B)}$
- $reduction(R_i.A > Value) = \frac{MaxVal(R_i.A) - value}{Values(R_i.A)}$
- $reduction(TargetList) = \frac{number-of-attributes(TargetList)}{number-of-attributes(R_i)}$

Finally,

$$weight(R_i.A) = Tuples(R_i) \times reduction(R_i.A = value)$$

Choosing a plan: Difficult for different reasons: the number of possible query execution plans is large. Steps:

1. *Choosing a logical plan:* choose a query tree — only query trees in which selections and projections are *fully* or *nearl fully* down to the leaf and Cartesian products and selections are combined to joins are considered;
2. *Reducing the search space:* settle for a fixed form of tree shape (i.e. left-deep tree); not only this reduces the search space but also allows pipelining;
3. *Choosing a heuristic search algorithm:* assigns the relations to the leafs of the tree; use dynamic programming to accomplish this task; see algorithm (Fig. 14.6)

Query Design: current DBMS provides tools for query anlyzing; looking at it, one can decide whether to create additional indices or change certain things;