Appendix B
Centralized Query Processing

Our discussion in this appendix is a survey of query processing in centralized relational DBMSs. The distributed approach discussed in Chapter 4 is an extension of these techniques.

The success of relational database technology in data processing is due, in part, to the availability of non-procedural languages (i.e., SQL), which can significantly improve application development and end-user productivity. By hiding the low-level details about the physical organization of the data, relational database languages allow the expression of complex queries in a concise and simple fashion. In particular, to construct the answer to the query, the user does not precisely specify the procedure to follow. This procedure is actually devised by a DBMS module, usually called a *query processor*. This relieves the user from query optimization, a time-consuming task that is best handled by the query processor, since it can exploit a large amount of useful information about the data.

B.1 Characterization of Query Processors

It is quite difficult to evaluate and compare query processors in the context of both centralized systems and distributed systems because they may differ in many aspects. In what follows, we list important characteristics of query processors that can be used as a basis for comparison.

B.1.1 Languages

Initially, most of the work on query processing was done in the context of relational DBMSs because their high-level languages give the system many opportunities for optimization. The input language to the query processor is thus based on relational

calculus. As noted in Appendix A, we primarily use SQL as the user language in this book.

The former requires an additional phase to decompose a query expressed in relational calculus into relational algebra. In a distributed context, the output language is generally some internal form of relational algebra augmented with communication primitives. The operators of the output language are implemented directly in the system. Query processing must perform efficient mapping from the input language to the output language.

B.1.2 Types of Optimization

Conceptually, query optimization aims at choosing the “best” point in the solution space of all possible execution strategies. An immediate method for query optimization is to search the solution space, exhaustively predict the cost of each strategy, and select the strategy with minimum cost. Although this method is effective in selecting the best strategy, it may incur a significant processing cost for the optimization itself. The problem is that the solution space can be large; that is, there may be many equivalent strategies, even with a small number of relations. The problem becomes worse as the number of relations increases (e.g., becomes greater than 1-). Having high optimization cost is not necessarily bad, particularly if query optimization is done once for many subsequent executions of the query. Therefore, an “exhaustive” search approach is often used whereby (almost) all possible execution strategies are considered [Selinger et al., 1979].

To avoid the high cost of exhaustive search, *randomized* strategies, such as *iterative improvement* [Swami, 1989] and *simulated annealing* [Ioannidis and Wong, 1987] have been proposed to be used for query optimization. They try to find a very good solution, not necessarily the best one, but avoid the high cost of optimization, in terms of memory and time consumption.

Another popular way of reducing the cost of exhaustive search is the use of heuristics, whose effect is to restrict the solution space so that only a few strategies are considered. In both centralized and distributed systems, a common heuristic is to minimize the size of intermediate relations. This can be done by performing unary operators first, and ordering the binary operators by the increasing sizes of their intermediate relations.

B.1.3 Optimization Timing

A query may be optimized at different times relative to the actual time of query execution. Optimization can be done *statically* before executing the query or *dynamically* as the query is executed. Static query optimization is done at query compilation time. Thus the cost of optimization may be amortized over multiple query executions.

Therefore, this timing is appropriate for use with the exhaustive search method. Since the sizes of the intermediate relations of a strategy are not known until run time, they must be estimated using database statistics. Errors in these estimates can lead to the choice of suboptimal strategies.

Dynamic query optimization proceeds at query execution time. At any point of execution, the choice of the best next operator can be based on accurate knowledge of the results of the operators executed previously. Therefore, database statistics are not needed to estimate the size of intermediate results. However, they may still be useful in choosing the first operators. The main advantage over static query optimization is that the actual sizes of intermediate relations are available to the query processor, thereby minimizing the probability of a bad choice. The main shortcoming is that query optimization, an expensive task, must be repeated for each execution of the query. Therefore, this approach is best for ad-hoc queries.

Hybrid query optimization attempts to provide the advantages of static query optimization while avoiding the issues generated by inaccurate estimates. The approach is basically static, but dynamic query optimization may take place at run time when a high difference between predicted sizes and actual size of intermediate relations is detected.

B.1.4 Statistics

The effectiveness of query optimization relies on *statistics* on the database. Dynamic query optimization requires statistics in order to choose which operators should be done first. Static query optimization is even more demanding since the size of intermediate relations must also be estimated based on statistical information. In a distributed database, statistics for query optimization typically bear on fragments, and include fragment cardinality and size as well as the size and number of distinct values of each attribute. To minimize the probability of error, more detailed statistics such as histograms of attribute values are sometimes used at the expense of higher management cost. The accuracy of statistics is achieved by periodic updating. With static optimization, significant changes in statistics used to optimize a query might result in query reoptimization.

B.2 Query Processing Methodology

The centralized query processing methodology in relational databases is given in Figure B.1. We discuss each of these steps below.

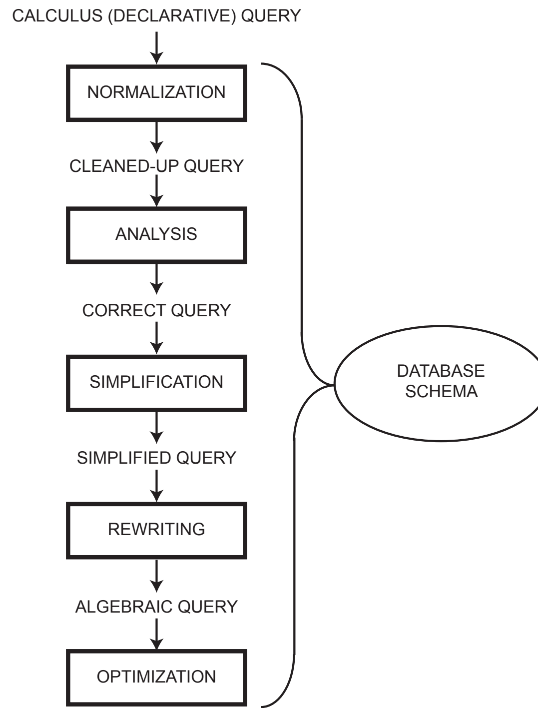


Fig. B.1: Centralized Query Processing Methodology

B.2.1 Normalization

The input query may be arbitrarily complex, depending on the facilities provided by the language. It is the goal of normalization to transform the query to a normalized form to facilitate further processing. With relational languages such as SQL, the most important transformation is that of the query qualification (the **WHERE** clause), which may be an arbitrarily complex, quantifier-free predicate, preceded by all necessary quantifiers (∀ or ∃). There are two possible normal forms for the predicate, one giving precedence to the AND (∧) and the other to the OR (∨). The *conjunctive normal form* is a conjunction (∧ predicate) of disjunctions (∨ predicates) as follows:

$$(p_{11} \vee p_{12} \vee \cdots \vee p_{1n}) \wedge \cdots \wedge (p_{m1} \vee p_{m2} \vee \cdots \vee p_{mn})$$

where p_{ij} is a simple predicate. A qualification in *disjunctive normal form*, on the other hand, is as follows:

$$(p_{11} \wedge p_{12} \wedge \cdots \wedge p_{1n}) \vee \cdots \vee (p_{m1} \wedge p_{m2} \wedge \cdots \wedge p_{mn})$$

The transformation of the quantifier-free predicate is straightforward using the well-known equivalence rules for logical operations ($\wedge$, $\vee$, and $\neg$):

1. $p_1 \wedge p_2 \Leftrightarrow p_2 \wedge p_1$
2. $p_1 \vee p_2 \Leftrightarrow p_2 \vee p_1$
3. $p_1 \wedge (p_2 \wedge p_3) \Leftrightarrow (p_1 \wedge p_2) \wedge p_3$
4. $p_1 \vee (p_2 \vee p_3) \Leftrightarrow (p_1 \vee p_2) \vee p_3$
5. $p_1 \wedge (p_2 \vee p_3) \Leftrightarrow (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$
6. $p_1 \vee (p_2 \wedge p_3) \Leftrightarrow (p_1 \vee p_2) \wedge (p_1 \vee p_3)$
7. $\neg(p_1 \wedge p_2) \Leftrightarrow \neg p_1 \vee \neg p_2$
8. $\neg(p_1 \vee p_2) \Leftrightarrow \neg p_1 \wedge \neg p_2$
9. $\neg(\neg p) \Leftrightarrow p$

In the disjunctive normal form, the query can be processed as independent conjunctive subqueries linked by unions (corresponding to the disjunctions). However, this form may lead to replicated join and select predicates, as shown in the following example. The reason is that predicates are very often linked with the other predicates by AND. The use of rule 5 mentioned above, with p_1 as a join or select predicate, would result in replicating p_1. The conjunctive normal form is more practical since query qualifications typically include more AND than OR predicates. However, it leads to predicate replication for queries involving many disjunctions and few conjunctions, a rare case.

Example B.1. Let us consider the following query on the engineering database that we have been referring to:

> "Find the names of employees who have been working on project P1 for 12 or 24 months"

The query expressed in SQL is

```
SELECT ENAME
FROM   EMP, ASG
WHERE  EMP.ENO = ASG.ENO
AND ASG.PNO = "P1"
AND DUR = 12 OR DUR = 24
```

The qualification in conjunctive normal form is

EMP.ENO = ASG.ENO $\wedge$ ASG.PNO = "P1" $\wedge$ (DUR = 12 $\vee$ DUR = 24)

while the qualification in disjunctive normal form is

(EMP.ENO = ASG.ENO $\wedge$ ASG.PNO = "P1" $\wedge$ DUR = 12) $\vee$
(EMP.ENO = ASG.ENO $\wedge$ ASG.PNO = "P1" $\wedge$ DUR = 24)

In the latter form, treating the two conjunctions independently may lead to redundant work if common subexpressions are not eliminated. ♦

B.2.2 Analysis

Query analysis enables rejection of normalized queries for which further processing is either impossible or unnecessary. The main reasons for rejection are that the query is *type incorrect* or *semantically incorrect*. When one of these cases is detected, the query is simply returned to the user with an explanation. Otherwise, query processing is continued. Below we present techniques to detect these incorrect queries.

A query is type incorrect if any of its attribute or relation names are not defined in the global schema, or if operations are being applied to attributes of the wrong type. The technique used to detect type incorrect queries is similar to type checking for programming languages. However, the type declarations are part of the global schema rather than of the query, since a relational query does not produce new types.

Example B.2. The following SQL query on the engineering database is type incorrect for two reasons. First, attribute E\# is not declared in the schema. Second, the operation ">200" is incompatible with the type string of ENAME.

```
SELECT E#
FROM   EMP
WHERE  ENAME > 200
```

♦

A query is semantically incorrect if its components do not contribute in any way to the generation of the result. In the context of relational calculus, it is not possible to determine the semantic correctness of general queries. However, it is possible to do so for a large class of relational queries, those which do not contain disjunction and negation [Rosenkrantz and Hunt, 1980]. This is based on the representation of the query as a graph, called a *query graph* or *connection graph* [Ullman, 1982]. We define this graph for the most useful kinds of queries involving select, project, and join operators. In a query graph, one node indicates the result relation, and any other node indicates an operand relation. An edge between two nodes one of which does not correspond to the result represents a join, whereas an edge whose destination node is the result represents a project. Furthermore, a non-result node may be labeled by a select or a self-join (join of the relation with itself) predicate. An important subgraph of the query graph is the *join graph*, in which only the joins are considered. The join graph is particularly useful in the query optimization phase.

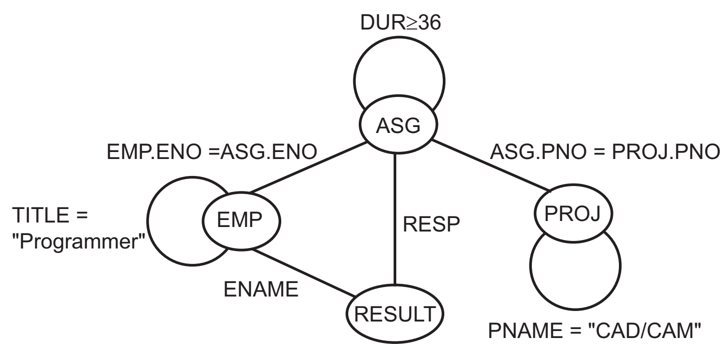
Example B.3. Let us consider the following query:

> "Find the names and responsibilities of programmers who have been working on the CAD/CAM project for more than 3 years."

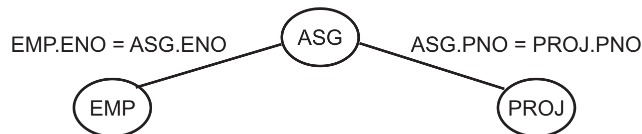
The query expressed in SQL is

```
SELECT ENAME, RESP
FROM EMP, ASG, PROJ
WHERE EMP.ENO = ASG.ENO
AND ASG.PNO = PROJ.PNO
AND PNAME = "CAD/CAM"
AND \dur+ ≥ 36
AND TITLE = "Programmer
```

The query graph for the query above is shown in Figure B.2a. Figure B.2b shows the join graph for the graph in Figure B.2a. ♦



(a) Query graph



(b) Corresponding join graph

Fig. B.2: Relation Graphs

The query graph is useful to determine the semantic correctness of a conjunctive multivariable query without negation. Such a query is semantically incorrect if its query graph is not connected. In this case one or more subgraphs (corresponding to subqueries) are disconnected from the graph that contains the result relation. The query could be considered correct (which some systems do) by considering the missing connection as a Cartesian product. But, in general, the problem is that join predicates are missing and the query should be rejected.

Example B.4. Let us consider the following SQL query:

```
SELECT ENAME, RESP
FROM EMP, ASG, PROJ
WHERE EMP.ENO = ASG.ENO
AND PNAME = "CAD/CAM"
AND DUR ≥ 36
AND TITLE = "Programmer
```

Its query graph, shown in Figure B.3, is disconnected, which tells us that the query is semantically incorrect. There are basically three solutions to the problem: (1) reject the query, (2) assume that there is an implicit Cartesian product between relations ASG and PROJ, or (3) infer (using the schema) the missing join predicate ASG.PNO = PROJ.PNO which transforms the query into that of Example B.3. ♦

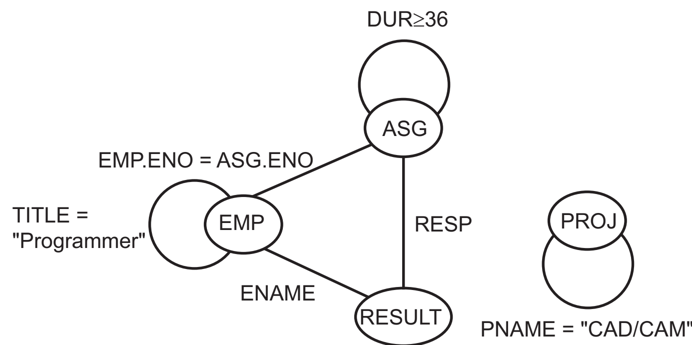


Fig. B.3: Disconnected Query Graph

B.2.3 Elimination of Redundancy

As discussed in Chapter 3, relational languages can be used uniformly for semantic data control. In particular, a user query typically expressed on a view may be enriched with several predicates to achieve view-relation correspondence, and ensure semantic integrity and security. The enriched query qualification may then contain redundant predicates. A naive evaluation of a qualification with redundancy can well lead to duplicated work. Such redundancy and thus redundant work may be eliminated by simplifying the qualification with the following well-known idempotency rules:

1. $p \land p \Leftrightarrow p$
2. $p \lor p \Leftrightarrow p$

3. $p \wedge true \Leftrightarrow p$
4. $p \vee false \Leftrightarrow p$
5. $p \wedge false \Leftrightarrow false$
6. $p \vee true \Leftrightarrow true$
7. $p \wedge \neg p \Leftrightarrow false$
8. $p \vee \neg p \Leftrightarrow true$
9. $p_1 \wedge (p_1 \vee p_2) \Leftrightarrow p_1$
10. $p_1 \vee (p_1 \wedge p_2) \Leftrightarrow p_1$

Example B.5. The SQL query

```
SELECT TITLE
FROM EMP
WHERE (NOT (TITLE = "Programmer")
AND (TITLE = "Programmer"
OR TITLE = "Elect. Eng.")
AND NOT (TITLE = "Elect. Eng."))
OR ENAME = "J. Doe
```

can be simplified using the previous rules to become

```
SELECT TITLE
FROM EMP
WHERE ENAME = "J. Doe"
```

The simplification proceeds as follows. Let p_1 be `TITLE = "Programmer"`, p_2 be `TITLE = "Elect. Eng."`, and p_3 be `ENAME = "J. Doe"`. The query qualification is

$$(\neg p_1 \wedge (p_1 \vee p_2) \wedge \neg p_2) \vee p_3$$

The disjunctive normal form for this qualification is obtained by applying rule 5 defined in Section B.2.1, which yields

$$(\neg p_1 \wedge ((p_1 \wedge \neg p_2) \vee (p_2 \wedge \neg p_2))) \vee p_3$$

and then rule 3 defined in Section B.2.1, which yields

$$(\neg p_1 \wedge p_1 \wedge \neg p_2) \vee (\neg p_1 \wedge p_2 \wedge \neg p_2) \vee p_3$$

By applying rule 7 defined above, we obtain

$$(false \wedge \neg p_2) \vee (\neg p_1 \wedge false) \vee p_3$$

By applying the same rule, we get

$false \vee false \vee p_3$

which is equivalent to p_3 by rule 4. ♦

B.2.4 Rewriting

The last step of query decomposition rewrites the query in relational algebra. For the sake of clarity it is customary to represent the relational algebra query graphically by an *operator tree*. An operator tree is a tree in which a leaf node is a relation stored in the database, and a non-leaf node is an intermediate relation produced by a relational algebra operator. The sequence of operations is directed from the leaves to the root, which represents the answer to the query.

The transformation of a tuple relational calculus query into an operator tree can easily be achieved as follows. First, a different leaf is created for each different tuple variable (corresponding to a relation). In SQL, the leaves are immediately available in the **FROM** clause. Second, the root node is created as a project operation involving the result attributes. These are found in the **SELECT** clause in SQL. Third, the qualification (SQL **WHERE** clause) is translated into the appropriate sequence of relational operations (select, join, union, etc.) going from the leaves to the root. The sequence can be given directly by the order of appearance of the predicates and operators.

Example B.6. The query

> "Find the names of employees other than J. Doe who worked on the CAD/CAM project for either one or two years"

whose SQL expression is

```
SELECT ENAME
FROM PROJ, ASG, EMP
WHERE ASG.ENO = EMP.ENO
AND ASG.PNO = PROJ.PNO
AND ENAME != "J. Doe"
AND PROJ.PNAME = "CAD/CAM"
AND (DUR = 12 OR DUR = 24)
```

can be mapped in a straightforward way in the tree in Figure B.4. The predicates have been transformed in order of appearance as join and then select operations. ♦

By applying *transformation rules*, many different trees may be found equivalent to the one produced by the method described above [Smith and Chang, 1975]. We now present the six most useful equivalence rules, which concern the basic relational algebra operators. The correctness of these rules has been proven [Ullman, 1982].

In the remainder of this section, R, S, and T are relations where R is defined over attributes $A = \{A_1, A_2, \ldots, A_n\}$ and S is defined over $B = \{B_1, B_2, \ldots, B_n\}$.

1. **Commutativity of binary operators.** The Cartesian product of two relations R and S is commutative:

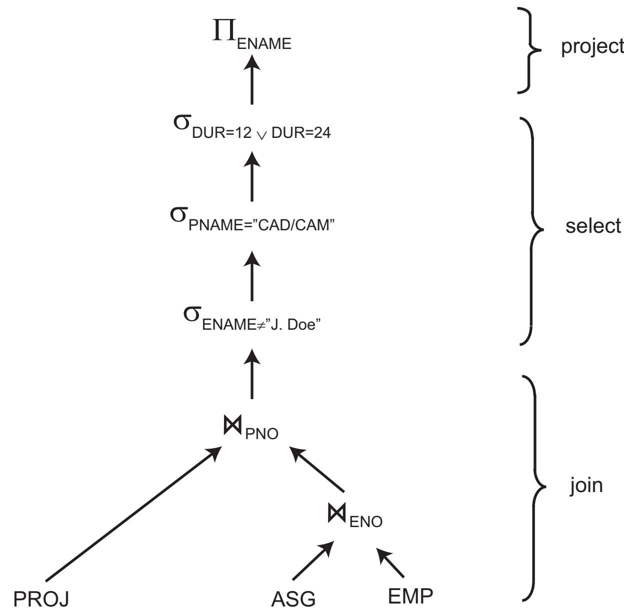


Fig. B.4: Example of Operator Tree

$$R \times S \Leftrightarrow S \times R$$

Similarly, the join of two relations is commutative:

$$R \bowtie S \Leftrightarrow S \bowtie R$$

This rule also applies to union but not to set difference or semijoin.

2. **Associativity of binary operators.** The Cartesian product and the join are associative operators:

$$(R \times S) \times T \Leftrightarrow R \times (S \times T)$$
$$(R \bowtie S) \bowtie T \Leftrightarrow R \bowtie (S \bowtie T)$$

3. **Idempotence of unary operators.** Several subsequent projections on the same relation may be grouped. Conversely, a single projection on several attributes may be separated into several subsequent projections. If R is defined over the attribute set A, and $A' \subseteq A$, $A'' \subseteq A$, and $A' \subseteq A''$, then

$$\Pi_{A'}(\Pi_{A''}(R)) \Leftrightarrow \Pi_{A'}(R)$$

Several subsequent selections $\sigma_{p_i(A_i)}$ on the same relation, where p_i is a predicate applied to attribute A_i, may be grouped as follows:

$$\sigma_{p_1(\mathtt{A}_1)}(\sigma_{p_2(\mathtt{A}_2)}(\mathtt{R})) = \sigma_{p_1(\mathtt{A}_1)\wedge p_2(\mathtt{A}_2)}(\mathtt{R})$$

Conversely, a single selection with a conjunction of predicates may be separated into several subsequent selections.

4. **Commuting selection with projection.** Selection and projection on the same relation can be commuted as follows:

$$\Pi_{\mathtt{A}_1,\ldots,\mathtt{A}_n}(\sigma_{p(\mathtt{A}_p)}(\mathtt{R})) \Leftrightarrow \Pi_{\mathtt{A}_1,\ldots,\mathtt{A}_n}(\sigma_{p(\mathtt{A}_p)}(\Pi_{\mathtt{A}_1,\ldots,\mathtt{A}_n,\mathtt{A}_p}(\mathtt{R})))$$

Note that if $\mathtt{A}_p$ is already a member of $\{\mathtt{A}_1, \ldots, \mathtt{A}_n\}$, the last projection on $[\mathtt{A}_1, \ldots, \mathtt{A}_n]$ on the right-hand side of the equality is useless.

5. **Commuting selection with binary operators.** Selection and Cartesian product can be commuted using the following rule (remember that attribute $\mathtt{A}_i$ belongs to relation $\mathtt{R}$):

$$\sigma_{p(\mathtt{A}_i)}(\mathtt{R} \times \mathtt{S}) \Leftrightarrow (\sigma_{p(\mathtt{A}_i)}(\mathtt{R})) \times \mathtt{S}$$

Selection and join can be commuted:

$$\sigma_{p(\mathtt{A}_i)}(\mathtt{R} \bowtie_{p(\mathtt{A}_j,\mathtt{B}_k)} \mathtt{S}) \Leftrightarrow \sigma_{p(\mathtt{A}_i)}(\mathtt{R}) \bowtie_{p(\mathtt{A}_j,\mathtt{B}_k)} \mathtt{S}$$

Selection and union can be commuted if $\mathtt{R}$ and $\mathtt{T}$are union compatible (have the same schema):

$$\sigma_{p(\mathtt{A}_i)}(\mathtt{R} \cup \mathtt{T}) \Leftrightarrow \sigma_{p(\mathtt{A}_i)}(\mathtt{R}) \cup \sigma_{p(\mathtt{A}_i)}(\mathtt{T})$$

Selection and difference can be commuted in a similar fashion.

6. **Commuting projection with binary operators.** Projection and Cartesian product can be commuted. If $\mathtt{C} = \mathtt{A}' \cup \mathtt{B}'$, where $\mathtt{A}' \subseteq \mathtt{A}$, $\mathtt{B}' \subseteq \mathtt{B}$, and $\mathtt{A}$ and $\mathtt{B}$ are the sets of attributes over which relations $\mathtt{R}$ and$\mathtt{S}$ respectively, are defined, we have

$$\Pi_{\mathtt{C}}(\mathtt{R} \times \mathtt{S}) \Leftrightarrow \Pi_{\mathtt{A}'}(\mathtt{R}) \times \Pi_{\mathtt{B}'}(\mathtt{S})$$

Projection and join can also be commuted.

$$\Pi_{\mathtt{C}}(\mathtt{R} \bowtie_{p(\mathtt{A}_i,\mathtt{B}_j)} S) \Leftrightarrow \Pi_{\mathtt{A}'}(\mathtt{R}) \bowtie_{p(\mathtt{A}_i,\mathtt{B}_j)} \Pi_{\mathtt{B}'}(\mathtt{S})$$

For the join on the right-hand side of the implication to hold we need to have $\mathtt{A}_i \in \mathtt{A}'$ and $\mathtt{B}_j \in \mathtt{B}'$. Since $\mathtt{C} = \mathtt{A}' \cup \mathtt{B}'$, $\mathtt{A}_i$ and $\mathtt{B}_j$ are in $\mathtt{C}$ and therefore we don't need a projection over $\mathtt{C}$ once the projections over $\mathtt{A}'$ and $\mathtt{B}'$ are performed. Projection and union can be commuted as follows:

$$\Pi_{\mathtt{C}}(\mathtt{R} \cup \mathtt{S}) \Leftrightarrow \Pi_{\mathtt{C}}(\mathtt{R}) \cup \Pi_{\mathtt{C}}(\mathtt{S})$$

Projection and difference can be commuted similarly.

The application of these six rules enables the generation of many equivalent trees. For instance, the tree in Figure B.5 is equivalent to the one in Figure B.4. However, the one in Figure B.5 requires a Cartesian product of relations `EMP` and `PROJ`, and may lead to a higher execution cost than the original tree. In the optimization phase, one can imagine comparing all possible trees based on their predicted cost. However, the excessively large number of possible trees makes this approach unrealistic. The rules presented above can be used to restructure the tree in a systematic way so that the "bad" operator trees are eliminated. These rules can be used in four different ways. First, they allow the separation of the unary operations, simplifying the query expression. Second, unary operations on the same relation may be grouped so that access to a relation for performing unary operations can be done only once. Third, unary operations can be commuted with binary operations so that some operations (e.g., selection) may be done first. Fourth, the binary operations can be ordered. This last rule is used extensively in query optimization. A simple restructuring algorithm uses a single heuristic that consists of applying unary operations (select/project) as soon as possible to reduce the size of intermediate relations [Ullman, 1982].

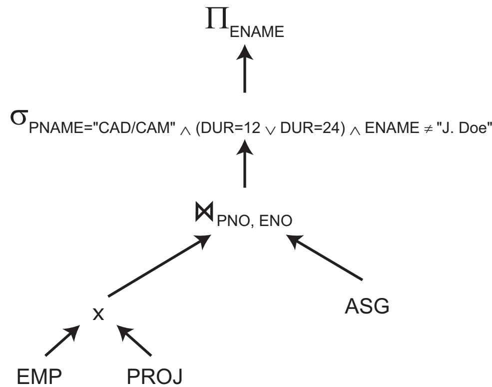


Fig. B.5: Equivalent Operator Tree

Example B.7. The restructuring of the tree in Figure B.4 leads to the tree in Figure B.6. The resulting tree is good in the sense that repeated access to the same relation (as in Figure B.4) is avoided and that the most selective operations are done first. However, this tree is far from optimal. For example, the select operation on `EMP` is not very useful before the join because it does not greatly reduce the size of the operand relation. ♦

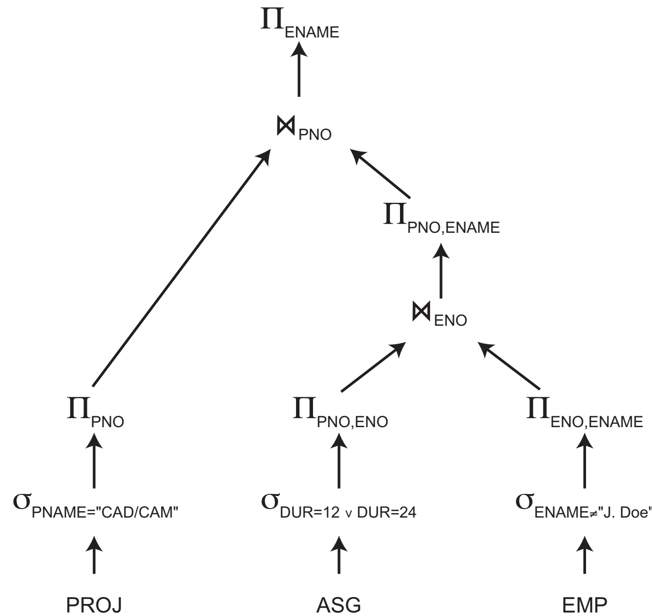


Fig. B.6: Rewritten Operator Tree

B.3 Query Optimization

Query optimization is one of the more important and involved aspects of query processors. Therefore, we devote a section to it. An important aspect of query optimization is the complexity of the relational algebra operators. Therefore, we start with a discussion of operator complexity. Then, we present the main components of a query optimizer. Finally, we introduce different optimization approaches as discussed early in this appendix.

B.3.1 Complexity of Relational Algebra Operations

In this book we consider relational algebra as a basis to express the output of query processing. Therefore, the complexity of relational algebra operators, which directly affects their execution time, dictates some principles useful to a query processor. These principles can help in choosing the final execution strategy.

The simplest way of defining complexity is in terms of relation cardinalities independent of physical implementation details such as fragmentation and storage structures. Figure B.7 shows the complexity of unary and binary operators in the order of increasing complexity, and thus of increasing execution time. Complexity is

$O(n)$ for unary operators, where n denotes the relation cardinality, if the resulting tuples may be obtained independently of each other. Complexity is $O(n * \log n)$ for binary operators if each tuple of one relation must be compared with each tuple of the other on the basis of the equality of selected attributes. This complexity assumes that tuples of each relation must be sorted on the comparison attributes. However, using hashing and enough memory to hold one hashed relation can reduce the complexity of binary operators $O(n)$ [Bratbergsengen, 1984]. Projects with duplicate elimination and grouping operators require that each tuple of the relation be compared with each other tuple, and thus also have $O(n * \log n)$ complexity. Finally, complexity is $O(n^2)$ for the Cartesian product of two relations because each tuple of one relation must be combined with each tuple of the other.

Operation	Complexity
Select Project (without duplicate elimination)	$O(n)$
Project (with duplicate elimination) Group by	$O(n*\log n)$
Join Semijoin Division Set Operators	$O(n*\log n)$
Cartesian Product	$O(n^2)$

Fig. B.7: Complexity of Relational Algebra Operations

This simple look at operator complexity suggests two principles. First, because complexity is relative to relation cardinalities, the most selective operators that reduce cardinalities (e.g., selection) should be performed first. Second, operators should be ordered by increasing complexity so that Cartesian products can be avoided or delayed.

B.3.2 Query Optimizer Components

Query optimization refers to the process of producing a query execution plan (QEP) which represents an execution strategy for the query. This QEP minimizes an objective cost function. A query optimizer, the software module that performs query

optimization, is usually seen as consisting of three components: a search space, a cost model, and a search strategy (see Figure B.8). The *search space* is the set of alternative execution plans that represent the input query. These plans are equivalent, in the sense that they yield the same result, but they differ in the execution order of operators and the way these operators are implemented, and therefore in their performance. The search space is obtained by applying transformation rules, such as those for relational algebra described in Section B.2.4. The *cost model* is used to predict the cost of any given execution plan, and to compare equivalent plans so as to choose the best one. To be accurate, the cost model must have good knowledge about the distributed execution environment. The *search strategy* explores the search space and selects the best plan, using the cost model. It defines which plans are examined and in which order. The details of the environment (centralized versus distributed) are captured by the search space and the cost model.

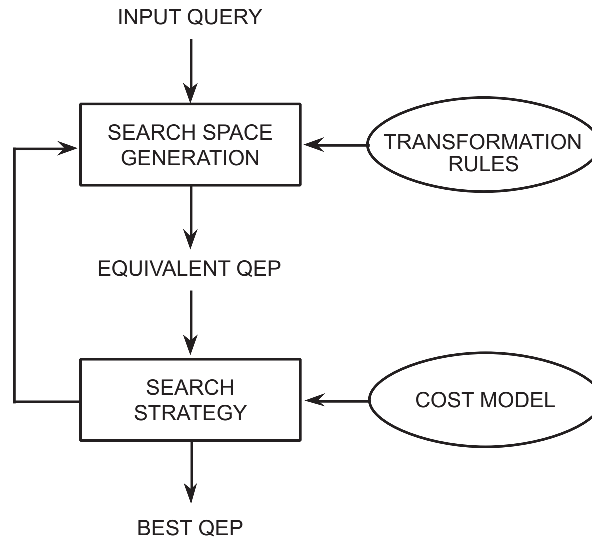


Fig. B.8: Query Optimization Process

B.3.2.1 Search Space

Query execution plans are typically abstracted by means of operator trees, which define the order in which the operators are executed. They are enriched with additional information, such as the best algorithm chosen for each operator. For a given query, the search space can thus be defined as the set of equivalent operator trees that can be produced using transformation rules. To characterize query optimizers, it is useful to concentrate on *join trees*, which are operator trees whose operators are join or

Cartesian product. This is because permutations of the join order have the most important effect on performance of relational queries.

Example B.8. Consider the following query:

```
SELECT ENAME, RESP
FROM EMP, ASG, PROJ
WHERE EMP.ENO=ASG.ENO
AND ASG.PNO=PROJ.PNO
```

Figure B.9 illustrates three equivalent join trees for that query, which are obtained by exploiting the associativity of binary operators. Each of these join trees can be assigned a cost based on the estimated cost of each operator. Join tree (c) which starts with a Cartesian product may have a much higher cost than the other join trees. ♦

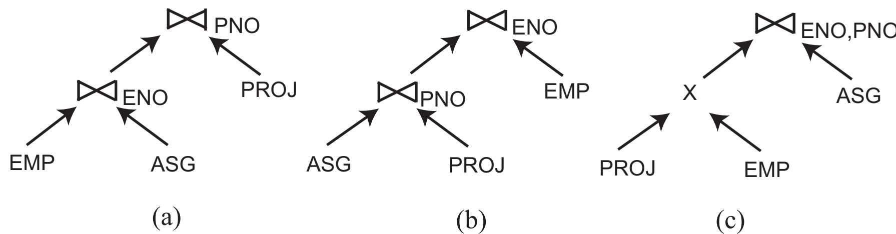
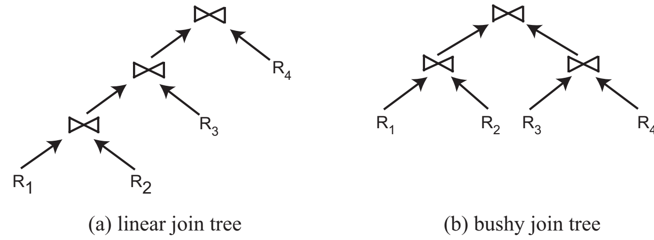


Fig. B.9: Equivalent Join Trees

For a complex query (involving many relations and many operators), the number of equivalent operator trees can be very high. For instance, the number of alternative join trees that can be produced by applying the commutativity and associativity rules is $O(N!)$ for N relations. Investigating a large search space may make optimization time prohibitive, sometimes much more expensive than the actual execution time. Therefore, query optimizers typically restrict the size of the search space they consider. The first restriction is to use heuristics. The most common heuristic is to perform selection and projection when accessing base relations. Another common heuristic is to avoid Cartesian products that are not required by the query. For instance, in Figure B.9, operator tree (c) would not be part of the search space considered by the optimizer.

Another important restriction is with respect to the shape of the join tree. Two kinds of join trees are usually distinguished: linear versus bushy trees (see Figure B.10). A *linear tree* is a tree such that at least one operand of each operator node is a base relation. A *bushy tree* is more general and may have operators with no base relations as operands (i.e., both operands are intermediate relations). By considering only linear trees, the size of the search space is reduced to $O(2^N)$. However, in a distributed environment, bushy trees are useful in exhibiting parallelism. For example, in join tree (b) of Figure B.10, operators $R_1 \bowtie R_2$ and $R_3 \bowtie R_4$ can be done in parallel.



(a) linear join tree (b) bushy join tree

Fig. B.10: The Two Major Shapes of Join Trees

B.3.2.2 Search Strategy

The most popular search strategy used by query optimizers is *dynamic programming*, which is *deterministic*. Deterministic strategies proceed by *building* plans, starting from base relations, joining one more relation at each step until complete plans are obtained, as in Figure B.11. Dynamic programming builds all possible plans, breadth-first, before it chooses the "best" plan. To reduce the optimization cost, partial plans that are not likely to lead to the optimal plan are *pruned* (i.e., discarded) as soon as possible. By contrast, another deterministic strategy, the greedy algorithm, builds only one plan, depth-first.

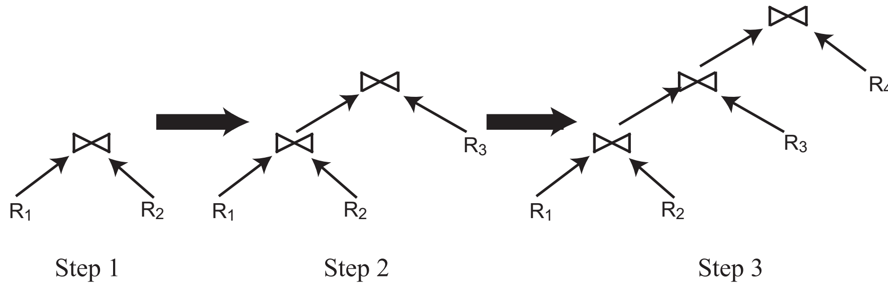


Fig. B.11: Optimizer Actions in a Deterministic Strategy

Dynamic programming is almost exhaustive and assures that the "best" of all plans is found. It incurs an acceptable optimization cost (in terms of time and space) when the number of relations in the query is small. However, this approach becomes too expensive when the number of relations is greater than 5 or 6. For more complex queries, *randomized* strategies have been proposed, which reduce the optimization complexity but do not guarantee the best of all plans. Unlike deterministic strategies, *randomized* strategies allow the optimizer to trade optimization time for execution time [Lanzelotte et al., 1993].

Randomized strategies, such as Simulated Annealing [Ioannidis and Wong, 1987] and Iterative Improvement [Swami, 1989] concentrate on searching for the optimal

solution around some particular points. They do not guarantee that the best solution is obtained, but avoid the high cost of optimization, in terms of memory and time consumption. First, one or more *start* plans are built by a greedy strategy. Then, the algorithm tries to improve the start plan by visiting its *neighbors*. A neighbor is obtained by applying a random *transformation* to a plan. An example of a typical transformation consists in exchanging two randomly chosen operand relations of the plan, as in Figure B.12. It has been shown experimentally that randomized strategies provide better performance than deterministic strategies as soon as the query involves more than several relations[Lanzelotte et al., 1993].

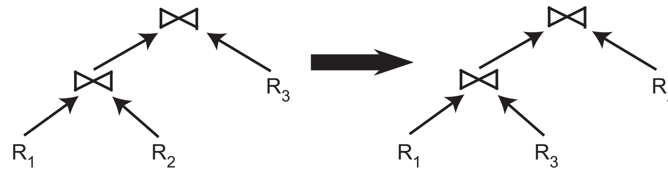


Fig. B.12: Optimizer Action in a Randomized Strategy

B.3.2.3 Cost Model

An optimizer's cost model includes cost functions to predict the cost of operators, statistics and base data, and formulas to evaluate the sizes of intermediate results. The cost is in terms of execution time, so a cost function represents the execution time of a query.

Cost Functions

The cost of a distributed execution strategy can be expressed with respect to either the total time or the response time. The total time is the sum of all time (also referred to as cost) components, while the response time is the elapsed time from the initiation to the completion of the query. A general formula for determining the total time can be specified as follows:

$$Total_time = T_{CPU} * \#insts + T_{I/O} * \#I/Os + T_{MSG} * \#msgs + T_{TR} * \#bytes$$

The two first components measure the local processing time, where T_{CPU} is the time of a CPU instruction and $T_{I/O}$ is the time of a disk I/O. Costs are generally expressed in terms of time units, which in turn, can be translated into other units (e.g., dollars).

Database Statistics

The main factor affecting the performance of an execution strategy is the size of the intermediate relations that are produced during the execution. When a subsequent operator is located at a different site, the intermediate relation must be transmitted over the network. Therefore, it is of prime interest to estimate the size of the intermediate results of relational algebra operators in order to minimize the size of data transfers. This estimation is based on statistical information about the base relations and formulas to predict the cardinalities of the results of the relational operators. There is a direct trade-off between the precision of the statistics and the cost of managing them, the more precise statistics being the more costly [Piatetsky-Shapiro and Connell, 1984]. For a relation R defined over the attributes $A = \{A_1, A_2, \ldots, A_n\}$ and fragmented as $R_1, R_2, \ldots, R_r$, the statistical data typically are the following:

1. For each attribute A_i, its length (in number of bytes), denoted by $length(A_i)$, and for each attribute A_i of each fragment R_j, the number of distinct values of A_i, with the cardinality of the projection of fragment R_j on A_i, denoted by $card(\Pi_{A_i}(R_j))$.
2. For the domain of each attribute A_i, which is defined on a set of values that can be ordered (e.g., integers or reals), the minimum and maximum possible values, denoted by $min(A_i)$ and $max(A_i)$.
3. For the domain of each attribute A_i, the cardinality of the domain of A_i, denoted by $card(dom[A_i])$. This value gives the number of unique values in the $dom[A_i]$.
4. The number of tuples in each fragment R_j, denoted by $card(R_j)$.

In addition, for each attribute A_i, there may be a histogram that approximates the frequency distribution of the attribute within a number of buckets, each corresponding to a range of values.

Sometimes, the statistical data also include the join selectivity factor for some pairs of relations, that is the proportion of tuples participating in the join. This is useful to predict the size of the joined relation which can then be used as input information for evaluating the cost of the next operator. The *join selectivity factor*, denoted $SF_{\bowtie}$, of relations R and S is a real value between 0 and 1:

$$SF_{\bowtie}(R, S) = \frac{card(R \bowtie S)}{card(R) * card(S)}$$

For example, a join selectivity factor of 0.5 corresponds to a very large joined relation, while 0.001 corresponds to a small one. We say that the join has bad (or low) selectivity in the former case and good (or high) selectivity in the latter case.

These statistics are useful to predict the size of intermediate relations. The size of an intermediate relation R is defined as follows:

$$size(R) = card(R) * length(R)$$

where $length(\mathtt{R})$ is the length (in bytes) of a tuple of R, computed from the lengths of its attributes. The estimation of $card(\mathtt{R})$, the number of tuples in R, requires the use of the formulas given in the following section.

Cardinalities of Intermediate Results

Database statistics are useful in evaluating the cardinalities of the intermediate results of queries. Two simplifying assumptions are commonly made about the database. The distribution of attribute values in a relation is supposed to be uniform, and all attributes are independent, meaning that the value of an attribute does not affect the value of any other attribute. These two assumptions are often wrong in practice, but they make the problem tractable. In what follows we give the formulas for estimating the cardinalities of the results of the basic relational algebra operators (selection, projection, Cartesian product, join, semijoin, union, and difference). The operand relations are denoted by R and S. The *selectivity factor* of an operator, that is, the proportion of tuples of an operand relation that participate in the result of that operator, is denoted SF_{OP}, where OP denotes the operator.

Selection.

The cardinality of selection is

$$card(\sigma_F(\mathtt{R})) = SF_\sigma(F) * card(\mathtt{R})$$

where $SF_\sigma(F)$ is dependent on the selection formula and can be computed as follows [Selinger et al., 1979], where $p(\mathtt{A}_i)$ and $p(\mathtt{A}_j)$ indicate predicates over attributes $\mathtt{A}_i$ and $\mathtt{A}_j$, respectively:

$$SF_\sigma(\mathtt{A} = value) = \frac{1}{card(\Pi_\mathtt{A}(\mathtt{R}))}$$

$$SF_\sigma(\mathtt{A} > value) = \frac{max(\mathtt{A}) - value}{max(\mathtt{A}) - min(\mathtt{A})}$$

$$SF_\sigma(\mathtt{A} < value) = \frac{value - min(\mathtt{A})}{max(\mathtt{A}) - min(\mathtt{A})}$$

$$SF_\sigma(p(\mathtt{A}_i) \wedge p(\mathtt{A}_j)) = SF_\sigma(p(\mathtt{A}_i)) * SF_\sigma(p(\mathtt{A}_j))$$

$$SF_\sigma(p(\mathtt{A}_i) \vee p(\mathtt{A}_j)) = SF_\sigma(p(\mathtt{A}_i)) + SF_\sigma(p(\mathtt{A}_j)) - (SF_\sigma(p(\mathtt{A}_i)) * SF_\sigma(p(\mathtt{A}_j)))$$

$$SF_\sigma(\mathtt{A} \in \{values\}) = SF_\sigma(\mathtt{A} = value) * card(\{values\})$$

Projection.

As indicated in Section A.3.1, projection can be with or without duplicate elimination. We consider projection with duplicate elimination. An arbitrary projection is difficult to evaluate precisely because the correlations between projected attributes are usually unknown [Gelenbe and Gardy, 1982]. However, there are two particularly useful cases where it is trivial. If the projection of relation R is based on a single attribute A, the cardinality is simply the number of tuples when the projection is performed. If one of the projected attributes is a key of R, then

$$card(\Pi_{\mathtt{A}}(\mathtt{R})) = card(\mathtt{R})$$

Cartesian product.

The cardinality of the Cartesian product of R and S is simply

$$card(\mathtt{R} \times \mathtt{S}) = card(\mathtt{R}) * card(\mathtt{S})$$

Join.

There is no general way to estimate the cardinality of a join without additional information. The upper bound of the join cardinality is the cardinality of the Cartesian product. It has been used in the earlier distributed DBMS (e.g. [Epstein et al., 1978]), but it is a quite pessimistic estimate. A more realistic solution is to divide this upper bound by a constant to reflect the fact that the join result is smaller than that of the Cartesian product [Selinger and Adiba, 1980]. However, there is a case, which occurs frequently, where the estimation is simple. If relation R is equijoined with S over attribute A from R, and B from S where A is a key of relation R, and B is a foreign key of relationS the cardinality of the result can be approximated as

$$card(\mathtt{R} \bowtie_{A=B} \mathtt{S}) = card(\mathtt{S})$$

because each tuple of S matches with at most one tuple of R. Obviously, the same thing is true if B is a key of S and A is a foreign key of R. However, this estimation is an upper bound since it assumes that each tuple of R participates in the join. For other important joins, it is worthwhile to maintain their join selectivity factor $SF_{\bowtie}$ as part of statistical information. In that case the result cardinality is simply

$$card(\mathtt{R} \bowtie \mathtt{S}) = SF_{\bowtie} * card(\mathtt{R}) * card(\mathtt{S})$$

Semijoin.

The selectivity factor of the semijoin of R by S gives the fraction (percentage) of tuples of R that join with tuples of S. An approximation for the semijoin selectivity factor is given by Hevner and Yao [1979] as

$$SF_{\ltimes}(\mathtt{R} \ltimes_{\mathtt{A}} \mathtt{S}) = \frac{card(\Pi_{\mathtt{A}}(\mathtt{S}))}{card(dom[\mathtt{A}])}$$

This formula depends only on attribute A of S. Thus it is often called the selectivity factor of attribute A of S, denoted $SF_{\ltimes}(\mathtt{S.A})$, and is the selectivity factor of S.A on any other joinable attribute. Therefore, the cardinality of the semijoin is given by

$$card(\mathtt{R} \ltimes_{\mathtt{A}} \mathtt{S}) = SF_{\ltimes}(\mathtt{S.A}) * card(\mathtt{R})$$

This approximation can be verified on a very frequent case, that of R.Abeing a foreign key of S (R.A is a primary key). In this case, the semijoin selectivity factor is 1 since $\Pi_{\mathtt{A}}(\mathtt{S})) = card(dom[\mathtt{A}])$ yielding that the cardinality of the semijoin is $card(\mathtt{R})$.

Union.

It is quite difficult to estimate the cardinality of the union of R and S because the duplicates between R and S are removed by the union. We give only the simple formulas for the upper and lower bounds, which are, respectively,

$$card(\mathtt{R}) + card(\mathtt{S})$$
$$max\{card(\mathtt{R}), card(\mathtt{S})\}$$

Note that these formulas assume that R and S do not contain duplicate tuples.

Difference.

Like the union, we give only the upper and lower bounds. The upper bound of $card(\mathtt{R} - \mathtt{S})$ is $card(\mathtt{R})$, whereas the lower bound is 0.

More complex predicates with conjunction and disjunction can also be handled by using the formulas given above.

Using Histograms for Selectivity Estimation

The formulae above for estimating the cardinalities of intermediate results of queries rely on the strong assumption that the distribution of attribute values in a relation is uniform. The advantage of this assumption is that the cost of managing the statistics

is minimal since only the number of distinct attribute values is needed. However, this assumption is not practical. In case of skewed data distributions, it can result in fairly inaccurate estimations and QEPs which are far from the optimal.

An effective solution to accurately capture data distributions is to use histograms. Today, most commercial DBMS optimizers support histograms as part of their cost model. Various kinds of histograms have been proposed for estimating the selectivity of query predicates with different trade-offs between accuracy and maintenance cost [Poosala et al., 1996]. To illustrate the use of histograms, we use the basic definition by Bruno and Chaudhuri [2002]. A *histogram* on attribute A from R is a set of buckets. Each bucket b_i describes a range of values of A, denoted by $range_i$, with its associated frequency f_i and number of distinct values d_i. f_i gives the number of tuples of R where $\texttt{R.A} \in range_i$. d_i gives the number of distinct values of A where $\texttt{R.A} \in range_i$. This representation of a relation's attribute can capture non-uniform distributions of values, with the buckets adapted to the different ranges. However, within a bucket, the distribution of attribute values is assumed to be uniform.

Histograms can be used to accurately estimate the selectivity of selection operators. They can also be used for more complex queries including selection, projection and join. However, the precise estimation of join selectivity remains difficult and depends on the type of the histogram [Poosala et al., 1996]. We now illustrate the use of histograms with two important selection predicates: equality and range predicate.

Equality predicate.

With $value \in range_i$, we simply have: $SF_\sigma(\texttt{A} = value) = 1/d_i$.

Range predicate.

Computing the selectivity of range predicates such as $\texttt{A} \leq value$, $\texttt{A} < value$ and $\texttt{A} > value$ requires identifying the relevant buckets and summing up their frequencies. Let us consider the range predicate $\texttt{R.A} \leq value$ with $value \in range_i$. To estimate the numbers of tuples of R that satisfy this predicate, we must sum up the frequencies of all buckets which precede bucket *i* and the estimated number of tuples that satisfy the predicate in bucket b_i. Assuming uniform distribution of attribute values in b_i, we have:

$$card(\sigma_{\texttt{A} \leq value}(\texttt{R})) = \sum_{j=1}^{i-1} f_j + (\frac{value - min(range_i)}{min(range_i)} - min(range_i) * f_i)$$

The cardinality of other range predicates can be computed in a similar way.

Example B.9. Figure B.13 shows a possible 4-bucket histogram for attribute DUR of a relation ASG with 300 tuples. Let us consider the equality predicate ASG.DUR=18. Since the value "18" fits in bucket b_3, the selectivity factor is 1/12. Since the cardinalty

of b_3 is 50, the cardinality of the selection is 50/12 which is approximately 5 tuples. Let us now consider the range predicate ASG.DUR ≤ 18. We have $min(range_3) = 12$ and $max(range_3) = 24$. The cardinality of the selection is: $100 + 75 + (((18 - 12)/(24 - 12)) * 50) = 200$ tuples. ♦

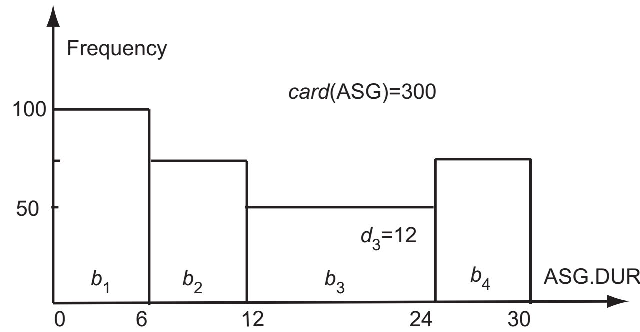


Fig. B.13: Histogram of Attribute ASG.DUR

B.3.3 Query Optimization Approaches

As discussed earlier in this Appendix, the optimization timing, which can be dynamic, static or hybrid, is a good basis for classifying query optimization techniques. Therefore, we present a representative technique of each class.

B.3.4 Dynamic Query Optimization

Dynamic query optimization combines the two phases of query decomposition and optimization with execution. The QEP is dynamically constructed by the query optimizer which makes calls to the DBMS execution engine for executing the query's operations. Thus, there is no need for a cost model.

The most popular dynamic query optimization algorithm is that of INGRES [Stonebraker et al., 1976], one of the first relational DBMS. In this section, we present this algorithm based on the detailed description by Wong and Youssefi [1976]. The algorithm recursively breaks up a query expressed in relational calculus (i.e., SQL) into smaller pieces which are executed along the way. The query is first decomposed into a sequence of queries having a unique relation in common. Then each monorelation query is processed by selecting, based on the predicate, the best

access method to that relation (e.g., index, sequential scan). For example, if the predicate is of the form $A = value$, an index available on attribute A would be used if it exists. However, if the predicate is of the form $A \neq value$, an index on A would not help, and sequential scan should be used.

The algorithm executes first the unary (monorelation) operations and tries to minimize the sizes of intermediate results in ordering binary (multirelation) operations. Let us denote by $q_{i-1} \rightarrow q_i$ a query q decomposed into two subqueries, q_{i-1} and q_i, where q_{i-1} is executed first and its result is consumed by q_i. Given an n-relation query q, the optimizer decomposes q into n subqueries $q_1 \rightarrow q_2 \rightarrow \cdots \rightarrow q_n$. This decomposition uses two basic techniques: *detachment* and *substitution*. These techniques are presented and illustrated in the rest of this section.

Detachment is the first technique employed by the query processor. It breaks a query q into $q' \rightarrow q''$, based on a common relation that is the result of q'. If the query q expressed in SQL is of the form

SELECT $R_2.A_2, R_3.A_3, \ldots, R_n.A_n$
FROM $R_1, R_2, \ldots, R_n$
WHERE $P_1(R_1.A_1^{'})$
AND $P_2(R_1.A_1, R_2.A_2, \ldots, R_n.A_n)$

where A_i and $A_i^{'}$ are lists of attributes of relation R_i, P_1 is a predicate involving attributes from relation R_1, and P_2 is a multirelation predicate involving attributes of relations $R_1, R_2, \ldots, R_n$. Such a query may be decomposed into two subqueries, q' followed by q'', by detachment of the common relation R_1:

q': **SELECT** $R_1.A_1$ **INTO** $R_1^{'}$
FROM R_1
WHERE $P_1(R_1.A_1^{'})$

where $R_1^{'}$ is a temporary relation containing the information necessary for the continuation of the query:

q'': **SELECT** $R_2.A_2, \ldots, R_n.A_n$
FROM $R_1^{'}, R_2, \ldots, R_n$
WHERE $P_2(R_1^{'}.A_1, \ldots, R_n.A_n)$

This step has the effect of reducing the size of the relation on which the query q'' is defined. Furthermore, the created relation $R_1^{'}$ may be stored in a particular structure to speed up the following subqueries. For example, the storage of $R_1^{'}$ in a hashed file on the join attributes of q'' will make processing the join more efficient. Detachment extracts the select operations, which are usually the most selective ones. Therefore, detachment is systematically done whenever possible. Note that this can have adverse effects on performance if the selection has bad selectivity.

Example B.10. To illustrate the detachment technique, we apply it to the following query:

"Names of employees working on the CAD/CAM project"

This query can be expressed in SQL by the following query q_1 on the engineering database example we have been using:

```
q1: SELECT EMP.ENAME
    FROM EMP, ASG, PROJ
    WHERE EMP.ENO=ASG.ENO
    AND ASG.PNO=PROJ.PNO
    AND PNAME="CAD/CAM"
```

After detachment of the selections, query q_1 is replaced by q_{11} followed by q', where JVAR is an intermediate relation.

```
q11: SELECT PROJ.PNO INTO JVAR
     FROM PROJ
     WHERE PNAME="CAD/CAM"
q':  SELECT EMP.ENAME
     FROM EMP, ASG, JVAR
     WHERE EMP.ENO=ASG.ENO
     AND ASG.PNO=JVAR.PNO
```

The successive detachments of q' may generate

```
q12: SELECT ASG.ENO INTO GVAR
     FROM ASG, JVAR
     WHERE ASG.PNO=JVAR.PNO
q13: SELECT EMP.ENAME
     FROM EMP, GVAR
     WHERE EMP.ENO=GVAR.ENO
```

Note that other subqueries are also possible.

Thus query q_1 has been reduced to the subsequent queries $q_{11} \rightarrow q_{12} \rightarrow q_{13}$. Query q_{11} is monorelation and can be executed. However, q_{12} and q_{13} are not monorelation and cannot be reduced by detachment. ♦

Multirelation queries, which cannot be further detached (e.g., q_{12} and q_{13}), are *irreducible*. A query is irreducible if and only if its query graph is a chain with two nodes or a cycle with k nodes where $k > 2$. Irreducible queries are converted into monorelation queries by tuple substitution. Given an n-relation query q, the tuples of one relation are substituted by their values, thereby producing a set of $(n-1)$-relation queries. Tuple substitution proceeds as follows. First, one relation in q is chosen for tuple substitution. Let R_1 be that relation. Then for each tuple t_{1i} in R_1, the attributes referred to by in q are replaced by their actual values in t_{1i}, thereby generating a query q' with $n-1$ relations. Therefore, the total number of queries q' produced by tuple substitution is $card(R_1)$. Tuple substitution can be summarized as follows:

$q(R_1, R_2, \ldots, R_n)$ is replaced by $\{q'(t_{1i}, R_2, R_3, \ldots, R_n), t_{1i} \in R_1\}$

For each tuple thus obtained, the subquery is recursively processed by substitution if it is not yet irreducible.

Example B.11. Let us consider the query q_{13}:

```
SELECT EMP.ENAME
FROM EMP, GVAR
WHERE EMP.ENO=GVAR.ENO
```

The relation GVAR is over a single attribute (ENO). Assume that it contains only two tuples: ⟨E1⟩ and ⟨E2⟩. The substitution of GVAR generates two one-relation subqueries:

q_{131}:
```
SELECT EMP.ENAME
FROM EMP
WHERE EMP.ENO="E1"
```

q_{132}:
```
SELECT EMP.ENAME
FROM EMP
WHERE EMP.ENO="E2"
```

These queries may then be executed. ♦

This dynamic query optimization algorithm (called Dynamic-QOA) is depicted in Algorithm B.1. The algorithm works recursively until there remain no more monorelation queries to be processed. It consists of applying the selections and projections as soon as possible by detachment. The results of the monorelation queries are stored in data structures that are capable of optimizing the later queries (such as joins). The irreducible queries that remain after detachment must be processed by tuple substitution. For the irreducible query, denoted by MRQ', the smallest relation whose cardinality is known from the result of the preceding query is chosen for substitution. This simple method enables one to generate the smallest number of subqueries. Monorelation queries generated by the reduction algorithm are executed after choosing the best existing access path to the relation, according to the query qualification.

B.3.5 Static Query Optimization

With static query optimization, there is a clear separation between the generation of the QEP at compile-time and its execution by the DBMS execution engine. Thus, an accurate cost model is key to predict the costs of candidate QEPs.

The most popular static query optimization algorithm is that of System R [Astrahan et al., 1976], also one of the first relational DBMS. In this section, we present this

Algorithm B.1: Dynamic-QOA

```
Input: MRQ: multirelation query with n relations
Output: output: result of execution
begin
    output ← φ ;
    if n = 1 then
        output ← run(MRQ)                     {execute the one relation query}
    end if
    {detach MRQ into m one-relation queries (ORQ) and one multirelation
     query} ORQ_1, ..., ORQ_m, MRQ' ← MRQ ;
    for i from 1 to m do
        output' ← run(ORQ_i) ;                {execute ORQ_i}
        output ← output ∪ output'             {merge all results}
    end for
    R ← CHOOSE_RELATION(MRQ') ;   {R chosen for tuple substitution}
    for each tuple t ∈ R do
        MRQ'' ← substitute values for t in MRQ' ;
        output' ← Dynamic-QOA(MRQ'') ;        {recursive call}
        output ← output ∪ output'             {merge all results}
    end for
end
```

algorithm based on the description by Selinger et al. [1979]. Most commercial relational DBMSs have implemented variants of this algorithm due to its efficiency and compatibility with query compilation.

The input to the optimizer is a relational algebra tree resulting from the decomposition of an SQL query. The output is a QEP that implements the "optimal" relational algebra tree.

The optimizer assigns a cost (in terms of time) to every candidate tree and retains the one with the smallest cost. The candidate trees are obtained by a permutation of the join orders of the n relations of the query using the commutativity and associativity rules. To limit the overhead of optimization, the number of alternative trees is reduced using dynamic programming. The set of alternative strategies is constructed dynamically so that, when two joins are equivalent by commutativity, only the cheapest one is kept. Furthermore, the strategies that include Cartesian products are eliminated whenever possible.

The cost of a candidate strategy is a weighted combination of I/O and CPU costs (times). The estimation of such costs (at compile time) is based on a cost model that provides a cost formula for each low-level operation (e.g., select using a B-tree index with a range predicate). For most operations (except exact match select), these cost formulas are based on the cardinalities of the operands. The cardinality information for the relations stored in the database is found in the database statistics. The cardinality

of the intermediate results is estimated based on the operation selectivity factors discussed in Section B.3.2.3.

The optimization algorithm consists of two major steps. First, the best access method to each individual relation based on a select predicate is predicted (this is the one with the least cost). Second, for each relation R, the best join ordering is estimated, where R is first accessed using its best single-relation access method. The cheapest ordering becomes the basis for the best execution plan.

In considering the joins, there are two basic algorithms available, with one of them being optimal in a given context. For the join of two relations, the relation whose tuples are read first is called the *external*, while the other, whose tuples are found according to the values obtained from the external relation, is called the *internal relation*. An important decision with either join method is to determine the cheapest access path to the internal relation.

The first method, called *nested-loop*, performs two loops over the relations. For each tuple of the external relation, the tuples of the internal relation that satisfy the join predicate are retrieved one by one to form the resulting relation. An index or a hashed table on the join attribute is a very efficient access path for the internal relation. In the absence of an index, for relations of n_1 and n_2 tuples, respectively, this algorithm has a cost proportional to $n_1 * n_2$, which may be prohibitive if n_1 and n_2 are high. Thus, an efficient variant is to build a hashed table on the join attribute for the internal relation (chosen as the smallest relation) before applying nested-loop. If the internal relation is itself the result of a previous operation, then the cost of building the hashed table can be shared with that of producing the previous result.

The second method, called *merge-join*, consists of merging two sorted relations on the join attribute. Indices on the join attribute may be used as access paths. If the join criterion is equality, the cost of joining two relations of n_1 and n_2 tuples, respectively, is proportional to $n_1 + n_2$. Therefore, this method is always chosen when there is an equijoin, and when the relations are previously sorted. If only one or neither of the relations are sorted, the cost of the nested-loop algorithm is to be compared with the combined cost of the merge join and of the sorting. The cost of sorting n pages is proportional to $n \log n$. In general, it is useful to sort and apply the merge join algorithm when large relations are considered.

The simplified version of the static optimization algorithm, for a select-project-join query, is shown in Algorithm B.2. It consists of two loops, the first of which selects the best single-relation access method to each relation in the query, while the second examines all possible permutations of join orders (there are $n!$ permutations with n relations) and selects the best access strategy for the query. The permutations are produced by the dynamic construction of a tree of alternative strategies. First, the join of each relation with every other relation is considered, followed by joins of three relations. This continues until joins of n relations are optimized. Actually, the algorithm does not generate all possible permutations since some of them are useless. As we discussed earlier, permutations involving Cartesian products are eliminated, as are the commutatively equivalent strategies with the highest cost. With these two heuristics, the number of strategies examined has an upper bound of 2^n rather than $n!$.

Algorithm B.2: Static-QOA

```
Input: QT: query tree with n relations
Output: output: best QEP
begin
    for each relation R_i ∈ QT do
        for each access path AP_ij to R_i do
            compute cost(AP_ij)
        end for
        best_AP_i ← AP_ij with minimum cost ;
        for each order (R_i1, R_i2, ··· , R_in) with i = 1, ··· , n! do
            build QEP (...((best AP_i1 ⋈ R_i2) ⋈ R_i3) ⋈ ... ⋈ R_in) ;
            compute cost (QEP)
        end for
        output ← QEP with minimum cost
    end for
end
```

Example B.12. Let us illustrate this algorithm with the query q_1 (see Example B.10) on the engineering database. The join graph of q_1 is given in Figure B.14. For short, the label ENO on edge EMP–ASG stands for the predicate EMP.ENO=ASG.ENO and the label PNO on edge ASG–PROJ stands for the predicate ASG.PNO=PROJ.PNO. We assume the following indices:

EMP has an index on ENO
ASG has an index on PNO
PROJ has an index on PNO and an index on PNAME

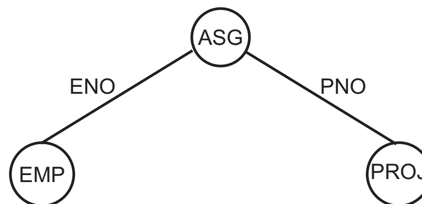


Fig. B.14: Join Graph of Query q_1

We assume that the first loop of the algorithm selects the following best single-relation access paths:

EMP: sequential scan (because there is no selection on EMP)
ASG: sequential scan (because there is no selection on ASG)
PROJ: index on PNAME (because there is a selection on PROJ based on PNAME)

The dynamic construction of the tree of alternative strategies is illustrated in Figure B.15. Note that the maximum number of join orders is 3!; dynamic search considers fewer alternatives, as depicted in Figure B.15. The operations marked "pruned" are dynamically eliminated. The first level of the tree indicates the best single-relation access method. The second level indicates, for each of these, the best join method with any other relation. Strategies (EMP $\times$ PROJ) and (PROJ $\times$ EMP) are pruned because they are Cartesian products that can be avoided (by other strategies). We assume that (EMP $\bowtie$ ASG) and (ASG $\bowtie$ PROJ) have a cost higher than (ASG $\bowtie$ EMP) and (PROJ $\bowtie$ ASG), respectively. Thus they can be pruned because there are better join orders equivalent by commutativity. The two remaining possibilities are given at the third level of the tree. The best total join order is the least costly of ((ASG $\bowtie$ EMP) $\bowtie$ PROJ) and ((PROJ $\bowtie$ ASG) $\bowtie$ EMP). The latter is the only one that has a useful index on the select attribute and direct access to the joining tuples of ASG and EMP. Therefore, it is chosen with the following access methods:

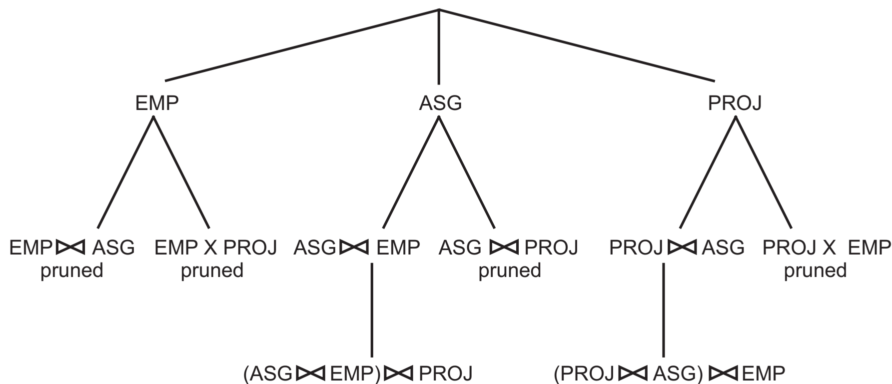


Fig. B.15: Alternative Join Orders

Select PROJ using index on PNAME
Then join with ASG using index on PNO
Then join with EMP using index on ENO

♦

The performance measurements substantiate the important contribution of the CPU time to the total time of the query[Mackert and Lohman, 1986b]. The accuracy of the optimizer's estimations is generally good when the relations can be contained in the main memory buffers, but degrades as the relations increase in size and are

written to disk. An important performance parameter that should also be considered for better predictions is buffer utilization.

B.3.6 Hybrid Query Optimization

Dynamic and static query optimimization both have advantages and drawbacks. Dynamic query optimization mixes optimization and execution and thus can make accurate optimization choices at run-time. However, query optimization is repeated for each execution of the query. Therefore, this approach is best for ad-hoc queries. Static query optimization, done at compilation time, amortizes the cost of optimization over multiple query executions. The accuracy of the cost model is thus critical to predict the costs of candidate QEPs. This approach is best for queries embedded in stored procedures, and has been adopted by all commercial DBMSs.

However, even with a sophisticated cost model, there is an important problem that prevents accurate cost estimation and comparison of QEPs at compile-time. The problem is that the actual bindings of parameter values in embedded queries is not known until run-time. Consider for instance the selection predicate "**WHERE** R.A = $\$a$" where "$\$a$" is a parameter value. To estimate the cardinality of this selection, the optimizer must rely on the assumption of uniform distribution of A values in R and cannot make use of histograms. Since there is a runtime binding of the parameter a, the accurate selectivity of $\sigma_{A=\$a}(R)$ cannot be estimated until runtime.

Thus, it can make major estimation errors that can lead to the choice of suboptimal QEPs.

Hybrid query optimization attempts to provide the advantages of static query optimization while avoiding the issues generated by inaccurate estimates. The approach is basically static, but further optimization decisions may take place at run time. This approach was pionnered in System R by adding a conditional runtime reoptimization phase for execution plans statically optimized [Chamberlin et al., 1981]. Thus, plans that have become infeasible (e.g., because indices have been dropped) or suboptimal (e.g. because of changes in relation sizes) are reoptimized. However, detecting suboptimal plans is hard and this approach tends to perform much more reoptimization than necessary. A more general solution is to produce *dynamic QEPs* which include carefully selected optimization decisions to be made at runtime using "choose-plan" operators [Cole and Graefe, 1994]. The choose-plan operator links two or more equivalent subplans of a QEP that are incomparable at compile-time because important runtime information (e.g. parameter bindings) is missing to estimate costs. The execution of a choose-plan operator yields the comparison of the subplans based on actual costs and the selection of the best one. Choose-plan nodes can be inserted anywhere in a QEP.

Example B.13. Consider the following query expressed in relational algebra:

$$\sigma_{A \leq \$a}(R_1) \bowtie R_2 \bowtie R_3$$

Figure B.16 shows a dynamic execution plan for this query. We assume that each join is performed by nested-loop, with the left operand relation as external and the right operand relation as internal. The bottom choose-plan operator compares the cost of two alternative subplans for joining R_1 and R_2, the left subplan being better than the right one if the selection predicate has high selectivity. As stated above, since there is a runtime binding of the parameter \$a, the accurate selectivity of $\sigma_{A \le \$a}(R_1)$ cannot be estimated until runtime. The top choose-plan operator compares the cost of two alternative subplans for joining the result of the bottom choose-plan operation with R_3. Depending on the estimated size of the join of R_1 and R_2, which indirectly depends on the selectivity of the selection on R_1 it may be better to use R_3 as external or internal relation. ♦

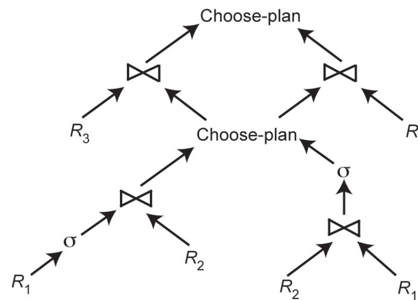


Fig. B.16: A Dynamic Execution Plan

Dynamic QEPs are produced at compile-time using any static algorithm such as the one presented in Section B.3.5. However, instead of producing a total order of operations, the optimizer must produce a partial order by introducing choose-node operators anywhere in the QEP. The main modification necessary to a static query optimizer to handle dynamic QEPs is that the cost model supports *incomparable* costs of plans in addition to the standard values “greater than”, “less than” and “equal to”. Costs may be incomparable because the costs of some subplans are unknown at compile-time. Another reason for cost incomparability is when cost is modeled as an interval of possible cost values rather than a single value [Cole and Graefe, 1994]. Therefore, if two plan costs have overlapping intervals, it is not possible to decide which one is better and they should be considered as incomparable.

Given a dynamic QEP, produced by a static query optimizer, the choose-plan decisions must be made at query startup time. The most effective solution is to simply evaluate the costs of the participating subplans and compare them. In Algorithm B.3, we describe the startup procedure (called Hybrid-QOA) which makes the optimization decisions to produce the final QEP and run it. The algorithm executes the choose-plan operators in bottom-up order and propagates cost information upward in the QEP.

Algorithm B.3: Hybrid-QOA

Input: QEP: dynamic QEP; B: Query parameter bindinds
Output: $output$: result of execution
begin
  $best_QEP \leftarrow QEP$;
  for *each choose-plan operator* CP *in bottom-up order* **do**
    for *each alternative subplan* SP **do**
      compute cost(CP) using B
    end for
    $best_QEP \leftarrow best_QEP$ without CP and SP of highest cost
  end for
  $output \leftarrow$ execute $best_QEP$
end

Experimentation with the Volcano query optimizer [Graefe, 1994] has shown that this hybrid query optimization outperforms both dynamic and static query optimization. In particular, the overhead of dynamic QEP evaluation at startup time is significantly less than that of dynamic optimization, and the reduced execution time of dynamic QEPs relative to static QEPs more than offsets the startup time overhead.