14 A TYPICAL RELATIONAL QUERY OPTIMIZER

Life is what happens while you're busy making other plans.

—John Lennon

In this chapter, we present a typical relational query optimizer in detail. We begin by discussing how SQL queries are converted into units called *blocks* and how blocks are translated into (extended) relational algebra expressions (Section 14.1). The central task of an optimizer is to find a good plan for evaluating such expressions. Optimizing a relational algebra expression involves two basic steps:

- Enumerating alternative plans for evaluating the expression. Typically, an optimizer considers a subset of all possible plans because the number of possible plans is very large.
- Estimating the cost of each enumerated plan, and choosing the plan with the least estimated cost.

To estimate the cost of a plan, we must estimate the cost of individual relational operators in the plan, using information about properties (e.g., size, sort order) of the argument relations, and we must estimate the properties of the result of an operator (in order to be able to compute the cost of any operator that uses this result as input). We discussed the cost of individual relational operators in Chapter 12. We discuss how to use system statistics to estimate the properties of the result of a relational operation, in particular result sizes, in Section 14.2.

After discussing how to estimate the cost of a given plan, we describe the space of plans considered by a typical relational query optimizer in Sections 14.3 and 14.4. Exploring all possible plans is prohibitively expensive because of the large number of alternative plans for even relatively simple queries. Thus optimizers have to somehow narrow the space of alternative plans that they consider.

We discuss how nested SQL queries are handled in Section 14.5.

This chapter concentrates on an exhaustive, dynamic-programming approach to query optimization. Although this approach is currently the most widely used, it cannot satisfactorily handle complex queries. We conclude with a short discussion of other approaches to query optimization in Section 14.6.

We will consider a number of example queries using the following schema:

Sailors(*sid:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`)
Boats(*bid:* `integer`, *bname:* `string`, *color:* `string`)
Reserves(*sid:* `integer`, *bid:* `integer`, *day:* `dates`, *rname:* `string`)

As in Chapter 12, we will assume that each tuple of Reserves is 40 bytes long, that a page can hold 100 Reserves tuples, and that we have 1,000 pages of such tuples. Similarly, we will assume that each tuple of Sailors is 50 bytes long, that a page can hold 80 Sailors tuples, and that we have 500 pages of such tuples.

14.1 TRANSLATING SQL QUERIES INTO ALGEBRA

SQL queries are optimized by decomposing them into a collection of smaller units called *blocks*. A typical relational query optimizer concentrates on optimizing a single block at a time. In this section we describe how a query is decomposed into blocks and how the optimization of a single block can be understood in terms of plans composed of relational algebra operators.

14.1.1 Decomposition of a Query into Blocks

When a user submits an SQL query, the query is parsed into a collection of query blocks and then passed on to the query optimizer. A **query block** (or simply **block**) is an SQL query with no nesting and exactly one `SELECT` clause and one `FROM` clause and at most one `WHERE` clause, `GROUP BY` clause, and `HAVING` clause. The `WHERE` clause is assumed to be in conjunctive normal form, as per the discussion in Section 12.3. We will use the following query as a running example:

For each sailor with the highest rating (over all sailors), and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

The SQL version of this query is shown in Figure 14.1. This query has two query blocks. The **nested block** is:

```
SELECT MAX (S2.rating)
FROM   Sailors S2
```

The nested block computes the highest sailor rating. The **outer block** is shown in Figure 14.2. Every SQL query can be decomposed into a collection of query blocks without nesting.

```
SELECT    S.sid, MIN (R.day)
FROM      Sailors S, Reserves R, Boats B
WHERE     S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' AND
          S.rating = ( SELECT MAX (S2.rating)
                       FROM   Sailors S2 )
GROUP BY  S.sid
HAVING    COUNT (*) > 1
```

Figure 14.1 Sailors Reserving Red Boats

```
SELECT    S.sid, MIN (R.day)
FROM      Sailors S, Reserves R, Boats B
WHERE     S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' AND
          S.rating = Reference to nested block
GROUP BY  S.sid
HAVING    COUNT (*) > 1
```

Figure 14.2 Outer Block of Red Boats Query

The optimizer examines the system catalogs to retrieve information about the types and lengths of fields, statistics about the referenced relations, and the access paths (indexes) available for them. The optimizer then considers each query block and chooses a query evaluation plan for that block. We will mostly focus on optimizing a single query block and defer a discussion of nested queries to Section 14.5.

14.1.2 A Query Block as a Relational Algebra Expression

The first step in optimizing a query block is to express it as a relational algebra expression. For uniformity, let us assume that `GROUP BY` and `HAVING` are also operators in the extended algebra used for plans, and that aggregate operations are allowed to appear in the argument list of the projection operator. The meaning of the operators should be clear from our discussion of SQL. The SQL query of Figure 14.2 can be expressed in the extended algebra as:

$$\pi_{S.sid,MIN(R.day)}($$
$$HAVING_{COUNT(*)>2}($$
$$GROUP\ \ BY_{S.sid}($$
$$\sigma_{S.sid=R.sid \wedge R.bid=B.bid \wedge B.color='red' \wedge S.rating=value\_from\_nested\_block}($$
$$Sailors \times Reserves \times Boats))))$$

For brevity, we've used S, R, and B (rather than Sailors, Reserves, and Boats) to prefix attributes. Intuitively, the selection is applied to the cross-product of the three

relations. Then the qualifying tuples are grouped by *S.sid*, and the `HAVING` clause condition is used to discard some groups. For each remaining group, a result tuple containing the attributes (and count) mentioned in the projection list is generated. This algebra expression is a faithful summary of the semantics of an SQL query, which we discussed in Chapter 5.

Every SQL query block can be expressed as an extended algebra expression having this form. The `SELECT` clause corresponds to the projection operator, the `WHERE` clause corresponds to the selection operator, the `FROM` clause corresponds to the cross-product of relations, and the remaining clauses are mapped to corresponding operators in a straightforward manner.

The alternative plans examined by a typical relational query optimizer can be understood by recognizing that *a query is essentially treated as a $\sigma\pi\times$ algebra expression*, with the remaining operations (if any, in a given query) carried out on the result of the $\sigma\pi\times$ expression. The $\sigma\pi\times$ expression for the query in Figure 14.2 is:

$$\pi_{S.sid,R.day}(\\ \sigma_{S.sid=R.sid\wedge R.bid=B.bid\wedge B.color='red'\wedge S.rating=value\_from\_nested\_block}(\\ Sailors \times Reserves \times Boats))$$

To make sure that the `GROUP BY` and `HAVING` operations in the query can be carried out, the attributes mentioned in these clauses are added to the projection list. Further, since aggregate operations in the `SELECT` clause, such as the `MIN`*(R.day)* operation in our example, are computed after first computing the $\sigma\pi\times$ part of the query, aggregate expressions in the projection list are replaced by the names of the attributes that they refer to. Thus, the optimization of the $\sigma\pi\times$ part of the query essentially ignores these aggregate operations.

The optimizer finds the best plan for the $\sigma\pi\times$ expression obtained in this manner from a query. This plan is evaluated and the resulting tuples are then sorted (alternatively, hashed) to implement the `GROUP BY` clause. The `HAVING` clause is applied to eliminate some groups, and aggregate expressions in the `SELECT` clause are computed for each remaining group. This procedure is summarized in the following extended algebra expression:

$$\pi_{S.sid,MIN(R.day)}(\\ HAVING_{COUNT(*)>2}(\\ GROUP\ \ BY_{S.sid}(\\ \pi_{S.sid,R.day}(\\ \sigma_{S.sid=R.sid\wedge R.bid=B.bid\wedge B.color='red'\wedge S.rating=value\_from\_nested\_block}(\\ Sailors \times Reserves \times Boats)))))$$

Some optimizations are possible if the FROM clause contains just one relation and the relation has some indexes that can be used to carry out the grouping operation. We discuss this situation further in Section 14.4.1.

To a first approximation therefore, the alternative plans examined by a typical optimizer can be understood in terms of the plans considered for $\sigma\pi\times$ queries. An optimizer enumerates plans by applying several equivalences between relational algebra expressions, which we present in Section 14.3. We discuss the space of plans enumerated by an optimizer in Section 14.4.

14.2 ESTIMATING THE COST OF A PLAN

For each enumerated plan, we have to estimate its cost. There are two parts to estimating the cost of an evaluation plan for a query block:

1. For each node in the tree, we must *estimate the cost* of performing the corresponding operation. Costs are affected significantly by whether pipelining is used or temporary relations are created to pass the output of an operator to its parent.

2. For each node in the tree, we must *estimate the size of the result,* and whether it is sorted. This result is the input for the operation that corresponds to the parent of the current node, and the size and sort order will in turn affect the estimation of size, cost, and sort order for the parent.

We discussed the cost of implementation techniques for relational operators in Chapter 12. As we saw there, estimating costs requires knowledge of various parameters of the input relations, such as the number of pages and available indexes. Such statistics are maintained in the DBMS's system catalogs. In this section we describe the statistics maintained by a typical DBMS and discuss how result sizes are estimated. As in Chapter 12, we will use the number of page I/Os as the metric of cost, and ignore issues such as blocked access, for the sake of simplicity.

The estimates used by a DBMS for result sizes and costs are at best approximations to actual sizes and costs. It is unrealistic to expect an optimizer to find the very best plan; it is more important to avoid the worst plans and to find a good plan.

14.2.1 Estimating Result Sizes

We now discuss how a typical optimizer estimates the size of the result computed by an operator on given inputs. Size estimation plays an important role in cost estimation as well because the output of one operator can be the input to another operator, and the cost of an operator depends on the size of its inputs.

Consider a query block of the form:

```
SELECT  attribute list
FROM    relation list
WHERE   term_1 ∧ term_2 ∧ ... ∧ term_n
```

The maximum number of tuples in the result of this query (without duplicate elimination) is the product of the cardinalities of the relations in the `FROM` clause. Every term in the `WHERE` clause, however, eliminates some of these potential result tuples. We can model the effect of the `WHERE` clause on the result size by associating a **reduction factor** with each term, which is the ratio of the (expected) result size to the input size considering only the selection represented by the term. The actual size of the result can be estimated as the maximum size times the product of the reduction factors for the terms in the `WHERE` clause. Of course, this estimate reflects the—unrealistic, but simplifying—assumption that the conditions tested by each term are statistically independent.

We now consider how reduction factors can be computed for different kinds of terms in the `WHERE` clause by using the statistics available in the catalogs:

- *column = value*: For a term of this form, the reduction factor can be approximated by $\frac{1}{NKeys(I)}$ if there is an index I on *column* for the relation in question. This formula assumes uniform distribution of tuples among the index key values; this uniform distribution assumption is frequently made in arriving at cost estimates in a typical relational query optimizer. If there is no index on *column*, the System R optimizer arbitrarily assumes that the reduction factor is $\frac{1}{10}$! Of course, it is possible to maintain statistics such as the number of distinct values present for any attribute whether or not there is an index on that attribute. If such statistics are maintained, we can do better than the arbitrary choice of $\frac{1}{10}$.
- *column1 = column2*: In this case the reduction factor can be approximated by $\frac{1}{\texttt{MAX}\ (NKeys(I1),NKeys(I2))}$ if there are indexes $I1$ and $I2$ on *column1* and *column2*, respectively. This formula assumes that each key value in the smaller index, say $I1$, has a matching value in the other index. Given a value for *column1*, we assume that each of the $NKeys(I2)$ values for *column2* is equally likely. Thus, the number of tuples that have the same value in *column2* as a given value in *column1* is $\frac{1}{NKeys(I2)}$. If only one of the two columns has an index I, we take the reduction factor to be $\frac{1}{NKeys(I)}$; if neither column has an index, we approximate it by the ubiquitous $\frac{1}{10}$. These formulas are used whether or not the two columns appear in the same relation.
- *column > value*: The reduction factor is approximated by $\frac{High(I)\ -\ value}{High(I)\ -\ Low(I)}$ if there is an index I on *column*. If the column is not of an arithmetic type or there is no index, a fraction less than half is arbitrarily chosen. Similar formulas for the reduction factor can be derived for other range selections.

- *column* `IN` *(list of values)*: The reduction factor is taken to be the reduction factor for *column* = *value* multiplied by the number of items in the list. However, it is allowed to be at most half, reflecting the heuristic belief that each selection eliminates at least half the candidate tuples.

These estimates for reduction factors are at best approximations that rely on assumptions such as uniform distribution of values and independent distribution of values in different columns. In recent years more sophisticated techniques based on storing more detailed statistics (e.g., histograms of the values in a column, which we consider later in this section) have been proposed and are finding their way into commercial systems.

Reduction factors can also be approximated for terms of the form *column* `IN` *subquery* (ratio of the estimated size of the subquery result to the number of distinct values in *column* in the outer relation); `NOT` *condition* (1−reduction factor for *condition*); *value1*<*column*<*value2*; the disjunction of two conditions; and so on, but we will not discuss such reduction factors.

To summarize, regardless of the plan chosen, we can estimate the size of the final result by taking the product of the sizes of the relations in the `FROM` clause and the reduction factors for the terms in the `WHERE` clause. We can similarly estimate the size of the result of each operator in a plan tree by using reduction factors, since the subtree rooted at that operator's node is itself a query block.

Note that the number of tuples in the result is not affected by projections if duplicate elimination is not performed. However, projections reduce the number of pages in the result because tuples in the result of a projection are smaller than the original tuples; the ratio of tuple sizes can be used as a **reduction factor for projection** to estimate the result size in pages, given the size of the input relation.

Improved Statistics: Histograms

Consider a relation with N tuples and a selection of the form *column* > *value* on a column with an index I. The reduction factor r is approximated by $\frac{High(I) - value}{High(I) - Low(I)}$, and the size of the result is estimated as rN. This estimate relies upon the assumption that the distribution of values is uniform.

Estimates can be considerably improved by maintaining more detailed statistics than just the low and high values in the index I. Intuitively, we want to approximate the distribution of key values I as accurately as possible. Consider the two distributions of values shown in Figure 14.3. The first is a nonuniform distribution D of values (say, for an attribute called *age*). The *frequency* of a value is the number of tuples with that *age* value; a distribution is represented by showing the frequency for each possible *age* value. In our example, the lowest *age* value is 0, the highest is 14, and all recorded

Estimating query characteristics: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all use histograms to estimate query characteristics such as result size and cost. As an example, Sybase ASE uses one-dimensional, equidepth histograms with some special attention paid to high frequency values, so that their count is estimated accurately. ASE also keeps the average count of duplicates for each prefix of an index in order to estimate correlations between histograms for composite keys (although it does not maintain such histograms). ASE also maintains estimates of the degree of clustering in tables and indexes. IBM DB2, Informix, and Oracle also use one-dimensional equidepth histograms; Oracle automatically switches to maintaining a count of duplicates for each value when there are few values in a column. Microsoft SQL Server uses one-dimensional equiarea histograms with some optimizations (adjacent buckets with similar distributions are sometimes combined to compress the histogram). In SQL Server, the creation and maintenance of histograms is done automatically without a need for user input.

Although sampling techniques have been studied for estimating result sizes and costs, in current systems sampling is used only by system utilities to estimate statistics or to build histograms, but not directly by the optimizer to estimate query characteristics. Sometimes, sampling is used to do load balancing in parallel implementations.

age values are integers in the range 0 to 14. The second distribution approximates D by assuming that each *age* value in the range 0 to 14 appears equally often in the underlying collection of tuples. This approximation can be stored compactly because we only need to record the low and high values for the *age* range (0 and 14 respectively) and the total count of all frequencies (which is 45 in our example).

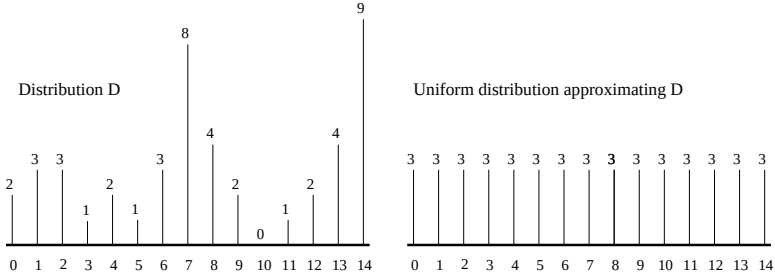

Figure 14.3 Uniform vs. Nonuniform Distributions

Consider the selection $age > 13$. From the distribution D in Figure 14.3, we see that the result has 9 tuples. Using the uniform distribution approximation, on the other

hand, we estimate the result size as $\frac{1}{15} * 45 = 3$ tuples. Clearly, the estimate is quite inaccurate.

A **histogram** is a data structure maintained by a DBMS to approximate a data distribution. In Figure 14.4, we show how the data distribution from Figure 14.3 can be approximated by dividing the range of *age* values into subranges called **buckets**, and for each bucket, counting the number of tuples with *age* values within that bucket. Figure 14.4 shows two different kinds of histograms, called *equiwidth* and *equidepth*, respectively.

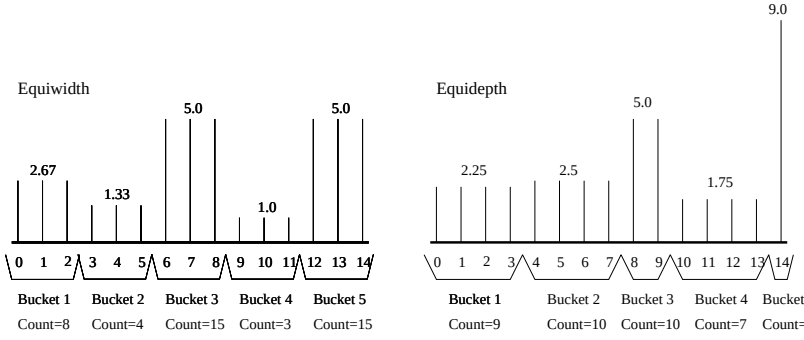

Figure 14.4 Histograms Approximating Distribution D

Consider the selection query $age > 13$ again and the first (equiwidth) histogram. We can estimate the size of the result to be 5 because the selected range includes a third of the range for Bucket 5. Since Bucket 5 represents a total of 15 tuples, the selected range corresponds to $\frac{1}{3} * 15 = 5$ tuples. As this example shows, we assume that the distribution *within* a histogram bucket is uniform. Thus, when we simply maintain the high and low values for index I, we effectively use a 'histogram' with a single bucket. Using histograms with a small number of buckets instead leads to much more accurate estimates, at the cost of a few hundred bytes per histogram. (Like all statistics in a DBMS, histograms are updated periodically, rather than whenever the data is changed.)

One important question is how to divide the value range into buckets. In an **equiwidth** histogram, we divide the range into subranges of equal size (in terms of the *age* value range). We could also choose subranges such that the number of tuples within each subrange (i.e., bucket) is equal. Such a histogram is called an **equidepth** histogram and is also illustrated in Figure 14.4. Consider the selection $age > 13$ again. Using the equidepth histogram, we are led to Bucket 5, which contains only the *age* value 15, and thus we arrive at the exact answer, 9. While the relevant bucket (or buckets) will generally contain more than one tuple, equidepth histograms provide better estimates than equiwidth histograms. Intuitively, buckets with very frequently occurring values

contain fewer values, and thus the uniform distribution assumption is applied to a smaller range of values, leading to better approximations. Conversely, buckets with mostly infrequent values are approximated less accurately in an equidepth histogram, but for good estimation, it is the frequent values that are important.

Proceeding further with the intuition about the importance of frequent values, another alternative is to separately maintain counts for a small number of very frequent values, say the *age* values 7 and 14 in our example, and to maintain an equidepth (or other) histogram to cover the remaining values. Such a histogram is called a **compressed** histogram. Most commercial DBMSs currently use equidepth histograms, and some use compressed histograms.

14.3 RELATIONAL ALGEBRA EQUIVALENCES

Two relational algebra expressions over the same set of input relations are said to be **equivalent** if they produce the same result on all instances of the input relations. In this section we present several equivalences among relational algebra expressions, and in Section 14.4 we discuss the space of alternative plans considered by a optimizer. Relational algebra equivalences play a central role in identifying alternative plans. Consider the query discussed in Section 13.3. As we saw earlier, pushing the selection in that query ahead of the join yielded a dramatically better evaluation plan; pushing selections ahead of joins is based on relational algebra equivalences involving the selection and cross-product operators.

Our discussion of equivalences is aimed at explaining the role that such equivalences play in a System R style optimizer. In essence, a basic SQL query block can be thought of as an algebra expression consisting of the cross-product of all relations in the `FROM` clause, the selections in the `WHERE` clause, and the projections in the `SELECT` clause. The optimizer can choose to evaluate any equivalent expression and still obtain the same result. Algebra equivalences allow us to convert cross-products to joins, to choose different join orders, and to push selections and projections ahead of joins. For simplicity, we will assume that naming conflicts never arise and that we do not need to consider the renaming operator ρ.

14.3.1 Selections

There are two important equivalences that involve the selection operation. The first one involves **cascading of selections**:

$$\sigma_{c_1 \wedge c_2 \wedge \ldots cn}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{cn}(R))\ldots))$$

Going from the right side to the left, this equivalence allows us to combine several selections into one selection. Intuitively, we can test whether a tuple meets each of the

conditions $c_1 \ldots cn$ at the same time. In the other direction, this equivalence allows us to take a selection condition involving several conjuncts and to replace it with several smaller selection operations. Replacing a selection with several smaller selections turns out to be very useful in combination with other equivalences, especially commutation of selections with joins or cross-products, which we will discuss shortly. Intuitively, such a replacement is useful in cases where only part of a complex selection condition can be pushed.

The second equivalence states that selections are **commutative**:

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

In other words, we can test the conditions c_1 and c_2 in either order.

14.3.2 Projections

The rule for **cascading projections** says that successively eliminating columns from a relation is equivalent to simply eliminating all but the columns retained by the final projection:

$$\pi_{a_1}(R) \equiv \pi_{a_1}(\pi_{a_2}(\ldots(\pi_{a_n}(R))\ldots))$$

Each a_i is a set of attributes of relation R, and $a_i \subseteq a_{i+1}$ for $i = 1 \ldots n-1$. This equivalence is useful in conjunction with other equivalences such as commutation of projections with joins.

14.3.3 Cross-Products and Joins

There are two important equivalences involving cross-products and joins. We present them in terms of natural joins for simplicity, but they hold for general joins as well.

First, assuming that fields are identified by name rather than position, these operations are **commutative**:

$$R \times S \equiv S \times R$$

$$R \bowtie S \equiv S \bowtie R$$

This property is very important. It allows us to choose which relation is to be the inner and which the outer in a join of two relations.

The second equivalence states that joins and cross-products are **associative**:

$$R \times (S \times T) \equiv (R \times S) \times T$$

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$

Thus we can either join R and S first and then join T to the result, or join S and T first and then join R to the result. The intuition behind associativity of cross-products is that regardless of the order in which the three relations are considered, the final result contains the same columns. Join associativity is based on the same intuition, with the additional observation that the selections specifying the join conditions can be cascaded. Thus the same rows appear in the final result, regardless of the order in which the relations are joined.

Together with commutativity, associativity essentially says that we can choose to join any pair of these relations, then join the result with the third relation, and always obtain the same final result. For example, let us verify that

$$R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$$

From commutativity, we have:

$$R \bowtie (S \bowtie T) \equiv R \bowtie (T \bowtie S)$$

From associativity, we have:

$$R \bowtie (T \bowtie S) \equiv (R \bowtie T) \bowtie S$$

Using commutativity again, we have:

$$(R \bowtie T) \bowtie S \equiv (T \bowtie R) \bowtie S$$

In other words, when joining several relations, we are free to join the relations in any order that we choose. This order-independence is fundamental to how a query optimizer generates alternative query evaluation plans.

14.3.4 Selects, Projects, and Joins

Some important equivalences involve two or more operators.

We can **commute** a selection with a projection if the selection operation involves only attributes that are retained by the projection:

$$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$$

Every attribute mentioned in the selection condition c must be included in the set of attributes a.

We can **combine** a selection with a cross-product to form a join, as per the definition of join:

$$R \bowtie_c S \equiv \sigma_c(R \times S)$$

We can **commute** a selection with a cross-product or a join if the selection condition involves only attributes of one of the arguments to the cross-product or join:

$$\sigma_c(R \times S) \equiv \sigma_c(R) \times S$$

$$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$$

The attributes mentioned in c must appear only in R, and not in S. Similar equivalences hold if c involves only attributes of S and not R, of course.

In general a selection σ_c on $R \times S$ can be replaced by a cascade of selections σ_{c_1}, σ_{c_2}, and σ_{c_3} such that c_1 involves attributes of both R and S, c_2 involves only attributes of R, and c_3 involves only attributes of S:

$$\sigma_c(R \times S) \equiv \sigma_{c_1 \wedge c_2 \wedge c_3}(R \times S)$$

Using the cascading rule for selections, this expression is equivalent to

$$\sigma_{c_1}(\sigma_{c_2}(\sigma_{c_3}(R \times S)))$$

Using the rule for commuting selections and cross-products, this expression is equivalent to

$$\sigma_{c_1}(\sigma_{c_2}(R) \times \sigma_{c_3}(S)).$$

Thus we can push part of the selection condition c ahead of the cross-product. This observation also holds for selections in combination with joins, of course.

We can **commute** a projection with a cross-product:

$$\pi_a(R \times S) \equiv \pi_{a_1}(R) \times \pi_{a_2}(S)$$

a_1 is the subset of attributes in a that appear in R, and a_2 is the subset of attributes in a that appear in S. We can also **commute** a projection with a join if the join condition involves only attributes retained by the projection:

$$\pi_a(R \bowtie_c S) \equiv \pi_{a_1}(R) \bowtie_c \pi_{a_2}(S)$$

a_1 is the subset of attributes in a that appear in R, and a_2 is the subset of attributes in a that appear in S. Further, every attribute mentioned in the join condition c must appear in a.

Intuitively, we need to retain only those attributes of R and S that are either mentioned in the join condition c or included in the set of attributes a retained by the projection. Clearly, if a includes all attributes mentioned in c, the commutation rules above hold. If a does *not* include all attributes mentioned in c, we can generalize the commutation rules by first projecting out attributes that are not mentioned in c or a, performing the join, and then projecting out all attributes that are not in a:

$$\pi_a(R \bowtie_c S) \equiv \pi_a(\pi_{a_1}(R) \bowtie_c \pi_{a_2}(S))$$

Now a_1 is the subset of attributes of R that appear in either a or c, and a_2 is the subset of attributes of S that appear in either a or c.

We can in fact derive the more general commutation rule by using the rule for cascading projections and the simple commutation rule, and we leave this as an exercise for the reader.

14.3.5 Other Equivalences

Additional equivalences hold when we consider operations such as set-difference, union, and intersection. Union and intersection are associative and commutative. Selections and projections can be commuted with each of the set operations (set-difference, union, and intersection). We will not discuss these equivalences further.

14.4 ENUMERATION OF ALTERNATIVE PLANS

We now come to an issue that is at the heart of an optimizer, namely, the space of alternative plans that is considered for a given query. Given a query, an optimizer essentially enumerates a certain set of plans and chooses the plan with the least estimated cost; the discussion in Section 13.2.1 indicated how the cost of a plan is estimated. The algebraic equivalences discussed in Section 14.3 form the basis for generating alternative plans, in conjunction with the choice of implementation technique for the relational operators (e.g., joins) present in the query. However, not all algebraically equivalent plans are considered because doing so would make the cost of optimization prohibitively expensive for all but the simplest queries. This section describes the subset of plans that are considered by a typical optimizer.

There are two important cases to consider: queries in which the `FROM` clause contains a single relation and queries in which the `FROM` clause contains two or more relations.

14.4.1 Single-Relation Queries

If the query contains a single relation in the `FROM` clause, only selection, projection, grouping, and aggregate operations are involved; there are no joins. If we have just one selection or projection or aggregate operation applied to a relation, the alternative implementation techniques and cost estimates discussed in Chapter 12 cover all the plans that must be considered. We now consider how to optimize queries that involve a combination of several such operations, using the following query as an example:

For each rating greater than 5, print the rating and the number of 20-year-old sailors with that rating, provided that there are at least two such sailors with different names.

The SQL version of this query is shown in Figure 14.5. Using the extended algebra

```
SELECT    S.rating, COUNT (*)
FROM      Sailors S
WHERE     S.rating > 5 AND S.age = 20
GROUP BY S.rating
HAVING    COUNT DISTINCT (S.sname) > 2
```

Figure 14.5 A Single-Relation Query

notation introduced in Section 14.1.2, we can write this query as:

$$\pi_{S.rating,COUNT(*)}($$
$$HAVING_{COUNTDISTINCT(S.sname)>2}($$
$$GROUP\ \ BY_{S.rating}($$
$$\pi_{S.rating,S.sname}($$
$$\sigma_{S.rating>5 \wedge S.age=20}($$
$$Sailors)))))$$

Notice that *S.sname* is added to the projection list, even though it is not in the `SELECT` clause, because it is required to test the `HAVING` clause condition.

We are now ready to discuss the plans that an optimizer would consider. The main decision to be made is which access path to use in retrieving Sailors tuples. If we considered only the selections, we would simply choose the most selective access path based on which available indexes *match* the conditions in the `WHERE` clause (as per the definition in Section 12.3.1). Given the additional operators in this query, we must also take into account the cost of subsequent sorting steps and consider whether these operations can be performed without sorting by exploiting some index. We first discuss the plans generated when there are no suitable indexes and then examine plans that utilize some index.

Plans without Indexes

The basic approach in the absence of a suitable index is to scan the Sailors relation and apply the selection and projection (without duplicate elimination) operations to each retrieved tuple, as indicated by the following algebra expression:

$$\pi_{S.rating,S.sname}($$
$$\sigma_{S.rating>5 \wedge S.age=20}($$
$$Sailors))$$

The resulting tuples are then sorted according to the `GROUP BY` clause (in the example query, on *rating*), and one answer tuple is generated for each group that meets

the condition in the HAVING clause. The computation of the aggregate functions in the SELECT and HAVING clauses is done for each group, using one of the techniques described in Section 12.7.

The cost of this approach consists of the costs of each of these steps:

1. Performing a file scan to retrieve tuples and apply the selections and projections.
2. Writing out tuples after the selections and projections.
3. Sorting these tuples to implement the GROUP BY clause.

Note that the HAVING clause does not cause additional I/O. The aggregate computations can be done on-the-fly (with respect to I/O) as we generate the tuples in each group at the end of the sorting step for the GROUP BY clause.

In the example query the cost includes the cost of a file scan on Sailors plus the cost of writing out ⟨*S.rating, S.sname*⟩ pairs plus the cost of sorting as per the GROUP BY clause. The cost of the file scan is *NPages(Sailors)*, which is 500 I/Os, and the cost of writing out ⟨*S.rating, S.sname*⟩ pairs is *NPages(Sailors)* times the ratio of the size of such a pair to the size of a Sailors tuple times the reduction factors of the two selection conditions. In our example the result tuple size ratio is about 0.8, the *rating* selection has a reduction factor of 0.5 and we use the default factor of 0.1 for the *age* selection. Thus, the cost of this step is 20 I/Os. The cost of sorting this intermediate relation (which we will call Temp) can be estimated as *3*NPages(Temp)*, which is 60 I/Os, if we assume that enough pages are available in the buffer pool to sort it in two passes. (Relational optimizers often assume that a relation can be sorted in two passes, to simplify the estimation of sorting costs. If this assumption is not met at run-time, the actual cost of sorting may be higher than the estimate!) The total cost of the example query is therefore $500 + 20 + 60 = 580$ I/Os.

Plans Utilizing an Index

Indexes can be utilized in several ways and can lead to plans that are significantly faster than any plan that does not utilize indexes.

1. **Single-index access path:** If several indexes match the selection conditions in the WHERE clause, each matching index offers an alternative access path. An optimizer can choose the access path that it estimates will result in retrieving the fewest pages, apply any projections and nonprimary selection terms (i.e., parts of the selection condition that do not match the index), and then proceed to compute the grouping and aggregation operations (by sorting on the GROUP BY attributes).
2. **Multiple-index access path:** If several indexes using Alternatives (2) or (3) for data entries match the selection condition, each such index can be used to retrieve

a set of rids. We can *intersect* these sets of rids, then sort the result by page id (assuming that the rid representation includes the page id) and retrieve tuples that satisfy the primary selection terms of all the matching indexes. Any projections and nonprimary selection terms can then be applied, followed by grouping and aggregation operations.

3. **Sorted index access path:** If the list of grouping attributes is a prefix of a tree index, the index can be used to retrieve tuples in the order required by the `GROUP BY` clause. All selection conditions can be applied on each retrieved tuple, unwanted fields can be removed, and aggregate operations computed for each group. This strategy works well for clustered indexes.

4. **Index-Only Access Path:** If all the attributes mentioned in the query (in the `SELECT`, `WHERE`, `GROUP BY`, or `HAVING` clauses) are included in the search key for some *dense* index on the relation in the `FROM` clause, an **index-only scan** can be used to compute answers. Because the data entries in the index contain all the attributes of a tuple that are needed for this query, and there is one index entry per tuple, we never need to retrieve actual tuples from the relation. Using just the data entries from the index, we can carry out the following steps as needed in a given query: apply selection conditions, remove unwanted attributes, sort the result to achieve grouping, and compute aggregate functions within each group. This *index-only* approach works even if the index does not match the selections in the `WHERE` clause. If the index matches the selection, we need only examine a subset of the index entries; otherwise, we must scan all index entries. In either case, we can avoid retrieving actual data records; therefore, the cost of this strategy does not depend on whether the index is clustered.

   In addition, if the index is a tree index and the list of attributes in the `GROUP BY` clause forms a prefix of the index key, we can retrieve data entries in the order needed for the `GROUP BY` clause and thereby avoid sorting!

We now illustrate each of these four cases, using the query shown in Figure 14.5 as a running example. We will assume that the following indexes, all using Alternative (2) for data entries, are available: a B+ tree index on *rating*, a hash index on *age*, and a B+ tree index on ⟨*rating, sname, age*⟩. For brevity, we will not present detailed cost calculations, but the reader should be able to calculate the cost of each plan. The steps in these plans are scans (a file scan, a scan retrieving tuples by using an index, or a scan of only index entries), sorting, and writing temporary relations, and we have already discussed how to estimate the costs of these operations.

As an example of the first case, we could choose to retrieve Sailors tuples such that *S.age*=20 using the hash index on *age*. The cost of this step is the cost of retrieving the index entries plus the cost of retrieving the corresponding Sailors tuples, which depends on whether the index is clustered. We can then apply the condition *S.rating*> 5 to each retrieved tuple; project out fields not mentioned in the `SELECT`, `GROUP BY`, and

Utilizing indexes: All of the main RDBMSs recognize the importance of index-only plans, and look for such plans whenever possible. In IBM DB2, when creating an index a user can specify a set of 'include' columns that are to be kept in the index but are *not* part of the index key. This allows a richer set of index-only queries to be handled because columns that are frequently accessed are included in the index even if they are not part of the key. In Microsoft SQL Server, an interesting class of index-only plans is considered: Consider a query that selects attributes *sal* and *age* from a table, given an index on *sal* and another index on *age*. SQL Server uses the indexes by joining the entries on the rid of data records to identify $\langle sal, age \rangle$ pairs that appear in the table.

`HAVING` clauses; and write the result to a temporary relation. In the example, only the *rating* and *sname* fields need to be retained. The temporary relation is then sorted on the *rating* field to identify the groups, and some groups are eliminated by applying the `HAVING` condition.

As an example of the second case, we can retrieve rids of tuples satisfying *rating*>5 using the index on *rating*, retrieve rids of tuples satisfying *age*=20 using the index on *age*, sort the retrieved rids by page number, and then retrieve the corresponding Sailors tuples. We can retain just the *rating* and *name* fields and write the result to a temporary relation, which we can sort on *rating* to implement the `GROUP BY` clause. (A good optimizer might pipeline the projected tuples to the sort operator without creating a temporary relation.) The `HAVING` clause is handled as before.

As an example of the third case, we can retrieve Sailors tuples such that *S.rating*> 5, ordered by *rating*, using the B+ tree index on *rating*. We can compute the aggregate functions in the `HAVING` and `SELECT` clauses on-the-fly because tuples are retrieved in *rating* order.

As an example of the fourth case, we can retrieve *data entries* from the ⟨*rating, sname, age*⟩ index such that *rating*> 5. These entries are sorted by *rating* (and then by *sname* and *age*, although this additional ordering is not relevant for this query). We can choose entries with *age=20* and compute the aggregate functions in the `HAVING` and `SELECT` clauses on-the-fly because the data entries are retrieved in *rating* order. In this case, in contrast to the previous case, we do not retrieve any Sailors tuples. This property of not retrieving data records makes the index-only strategy especially valuable with unclustered indexes.

14.4.2 Multiple-Relation Queries

Query blocks that contain two or more relations in the `FROM` clause require joins (or cross-products). Finding a good plan for such queries is very important because these queries can be quite expensive. Regardless of the plan chosen, the size of the final result can be estimated by taking the product of the sizes of the relations in the `FROM` clause and the reduction factors for the terms in the `WHERE` clause. But depending on the order in which relations are joined, intermediate relations of widely varying sizes can be created, leading to plans with very different costs.

In this section we consider how multiple-relation queries are optimized. We first introduce the class of plans considered by a typical optimizer, and then describe how all such plans are enumerated.

Left-Deep Plans

Consider a query of the form $A \bowtie B \bowtie C \bowtie D$, that is, the natural join of four relations. Two relational algebra operator trees that are equivalent to this query are shown in Figure 14.6.

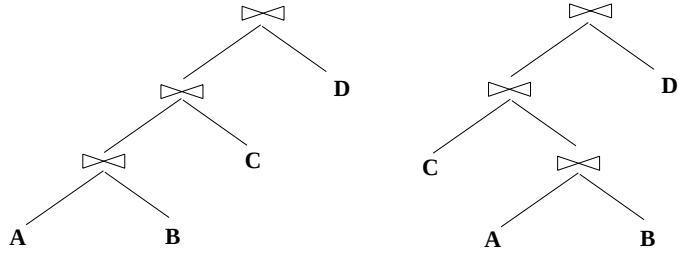

Figure 14.6 Two Linear Join Trees

We note that the left child of a join node is the outer relation and the right child is the inner relation, as per our convention. By adding details such as the join method for each join node, it is straightforward to obtain several query evaluation plans from these trees. Also, the equivalence of these trees is based on the relational algebra equivalences that we discussed earlier, particularly the associativity and commutativity of joins and cross-products.

The *form* of these trees is important in understanding the space of alternative plans explored by the System R query optimizer. Both the trees in Figure 14.6 are called **linear** trees. In a linear tree, at least one child of a join node is a base relation. The first tree is an example of a **left-deep** tree—the *right* child of each join node is a base relation. An example of a join tree that is not linear is shown in Figure 14.7; such trees are called **bushy** trees.

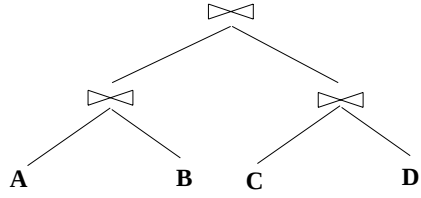

Figure 14.7 A Nonlinear Join Tree

A fundamental heuristic decision in the System R optimizer is to examine only left-deep trees in constructing alternative plans for a join query. Of course, this decision rules out many alternative plans that may cost less than the best plan using a left-deep tree; we have to live with the fact that the optimizer will never find such plans. There are two main reasons for this decision to concentrate on **left-deep plans**, or plans based on left-deep trees:

1. As the number of joins increases, the number of alternative plans increases rapidly and some pruning of the space of alternative plans becomes necessary.
2. Left-deep trees allow us to generate all **fully pipelined** plans, that is, plans in which the joins are all evaluated using pipelining. Inner relations must always be materialized fully because we must examine the entire inner relation for each tuple of the outer relation. Thus, a plan in which an inner relation is the result of a join forces us to materialize the result of that join. This observation motivates the heuristic decision to consider only left-deep trees. Of course, not all plans using left-deep trees are fully pipelined. For example, a plan that uses a sort-merge join may require the outer tuples to be retrieved in a certain sorted order, which may force us to materialize the outer relation.

Enumeration of Left-Deep Plans

Consider a query block of the form:

```
SELECT  attribute list
FROM    relation list
WHERE   term1 ∧ term2 ∧ ... ∧ termn
```

A System R style query optimizer enumerates all left-deep plans, with selections and projections considered (but not necessarily applied!) as early as possible. The enumeration of plans can be understood as a multiple-pass algorithm in which we proceed as follows:

Pass 1: We enumerate all single-relation plans (over some relation in the `FROM` clause). Intuitively, each single-relation plan is a partial left-deep plan for evaluating the query

in which the given relation is the first (in the linear join order for the left-deep plan of which it is a part). When considering plans involving a relation A, we identify those selection terms in the `WHERE` clause that mention only attributes of A. These are the selections that can be performed when first accessing A, before any joins that involve A. We also identify those attributes of A that are not mentioned in the `SELECT` clause or in terms in the `WHERE` clause involving attributes of other relations. These attributes can be projected out when first accessing A, before any joins that involve A. We choose the best access method for A to carry out these selections and projections, as per the discussion in Section 14.4.1.

For each relation, if we find plans that produce tuples in different orders, we retain the cheapest plan for each such ordering of tuples. An ordering of tuples could prove useful at a subsequent step, say for a sort-merge join or for implementing a `GROUP BY` or `ORDER BY` clause. Thus, for a single relation, we may retain a file scan (as the cheapest overall plan for fetching all tuples) and a B+ tree index (as the cheapest plan for fetching all tuples in the search key order).

Pass 2: We generate all two-relation plans by considering each single-relation plan that is retained after Pass 1 as the outer relation and (successively) every other relation as the inner relation. Suppose that A is the outer relation and B the inner relation for a particular two-relation plan. We examine the list of selections in the `WHERE` clause and identify:

1. Selections that involve only attributes of B and can be applied before the join.
2. Selections that serve to define the join (i.e., are conditions involving attributes of both A and B and no other relation).
3. Selections that involve attributes of other relations and can be applied only after the join.

The first two groups of selections can be considered while choosing an access path for the inner relation B. We also identify the attributes of B that do not appear in the `SELECT` clause or in any selection conditions in the second or third group above and can therefore be projected out before the join.

Notice that our identification of attributes that can be projected out before the join and selections that can be applied before the join is based on the relational algebra equivalences discussed earlier. In particular, we are relying on the equivalences that allow us to push selections and projections ahead of joins. As we will see, whether we actually perform these selections and projections ahead of a given join depends on cost considerations. The only selections that are really applied *before* the join are those that match the chosen access paths for A and B. The remaining selections and projections are done on-the-fly as part of the join.

An important point to note is that tuples generated by the outer plan are assumed to be *pipelined* into the join. That is, we avoid having the outer plan write its result to a file that is subsequently read by the join (to obtain outer tuples). For some join methods, the join operator might require materializing the outer tuples. For example, a hash join would partition the incoming tuples, and a sort-merge join would sort them if they are not already in the appropriate sort order. Nested loops joins, however, can use outer tuples as they are generated and avoid materializing them. Similarly, sort-merge joins can use outer tuples as they are generated if they are generated in the sorted order required for the join. We include the cost of materializing the outer, should this be necessary, in the cost of the join. The adjustments to the join costs discussed in Chapter 12 to reflect the use of pipelining or materialization of the outer are straightforward.

For each single-relation plan for A retained after Pass 1, for each join method that we consider, we must determine the best access method to use for B. The access method chosen for B will retrieve, in general, a subset of the tuples in B, possibly with some fields eliminated, as discussed below. Consider relation B. We have a collection of selections (some of which are the join conditions) and projections on a single relation, and the choice of the best access method is made as per the discussion in Section 14.4.1. The only additional consideration is that the join method might require tuples to be retrieved in some order. For example, in a sort-merge join we want the inner tuples in sorted order on the join column(s). If a given access method does not retrieve inner tuples in this order, we must add the cost of an additional sorting step to the cost of the access method.

Pass 3: We generate all three-relation plans. We proceed as in Pass 2, except that we now consider plans retained after Pass 2 as outer relations, instead of plans retained after Pass 1.

Additional passes: This process is repeated with additional passes until we produce plans that contain all the relations in the query. We now have the cheapest overall plan for the query, as well as the cheapest plan for producing the answers in some interesting order.

If a multiple-relation query contains a `GROUP BY` clause and aggregate functions such as `MIN`, `MAX`, and `SUM` in the `SELECT` clause, these are dealt with at the very end. If the query block includes a `GROUP BY` clause, a set of tuples is computed based on the rest of the query, as described above, and this set is sorted as per the `GROUP BY` clause. Of course, if there is a plan according to which the set of tuples is produced in the desired order, the cost of this plan is compared with the cost of the cheapest plan (assuming that the two are different) plus the sorting cost. Given the sorted set of tuples, partitions are identified and any aggregate functions in the `SELECT` clause are applied on a per-partition basis, as per the discussion in Chapter 12.

Optimization in commercial systems: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all search for left-deep trees using dynamic programming, as described here, with several variations. For example, Oracle always considers interchanging the two relations in a hash join, which could lead to right-deep trees or hybrids. DB2 generates some bushy trees as well. Systems often use a variety of strategies for generating plans, going beyond the systematic bottom-up enumeration that we described, in conjunction with a dynamic programming strategy for costing plans and remembering interesting plans (in order to avoid repeated analysis of the same plan). Systems also vary in the degree of control they give to users. Sybase ASE and Oracle 8 allow users to force the choice of join orders and indexes—Sybase ASE even allows users to explicitly edit the execution plan—whereas IBM DB2 does not allow users to direct the optimizer other than by setting an 'optimization level,' which influences how many alternative plans the optimizer considers.

Examples of Multiple-Relation Query Optimization

Consider the query tree shown in Figure 13.2. Figure 14.8 shows the same query, taking into account how selections and projections are considered early.

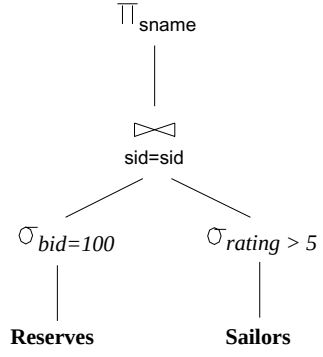

Figure 14.8 A Query Tree

In looking at this figure, it is worth emphasizing that the selections shown on the leaves are not necessarily done in a distinct step that precedes the join—rather, as we have seen, they are considered as potential matching predicates when considering the available access paths on the relations.

Suppose that we have the following indexes, all unclustered and using Alternative (2) for data entries: a B+ tree index on the *rating* field of Sailors, a hash index on the *sid* field of Sailors, and a B+ tree index on the *bid* field of Reserves. In addition, we

assume that we can do a sequential scan of both Reserves and Sailors. Let us consider how the optimizer proceeds.

In Pass 1 we consider three access methods for Sailors (B+ tree, hash index, and sequential scan), taking into account the selection $\sigma_{rating>5}$. This selection matches the B+ tree on *rating* and therefore reduces the cost for retrieving tuples that satisfy this selection. The cost of retrieving tuples using the hash index and the sequential scan is likely to be much higher than the cost of using the B+ tree. So the plan retained for Sailors is access via the B+ tree index, and it retrieves tuples in sorted order by *rating*. Similarly, we consider two access methods for Reserves taking into account the selection $\sigma_{bid=100}$. This selection matches the B+ tree index on Reserves, and the cost of retrieving matching tuples via this index is likely to be much lower than the cost of retrieving tuples using a sequential scan; access through the B+ tree index is therefore the only plan retained for Reserves after Pass 1.

In Pass 2 we consider taking the (relation computed by the) plan for Reserves and joining it (as the outer) with Sailors. In doing so, we recognize that now, we need only Sailors tuples that satisfy $\sigma_{rating>5}$ and $\sigma_{sid=value}$, where *value* is some value from an outer tuple. The selection $\sigma_{sid=value}$ matches the hash index on the *sid* field of Sailors, and the selection $\sigma_{rating>5}$ matches the B+ tree index on the *rating* field. Since the equality selection has a much lower reduction factor, the hash index is likely to be the cheaper access method. In addition to the preceding consideration of alternative access methods, we consider alternative join methods. All available join methods are considered. For example, consider a sort-merge join. The inputs must be sorted by *sid*; since neither input is sorted by *sid* or has an access method that can return tuples in this order, the cost of the sort-merge join in this case must include the cost of storing the two inputs in temporary relations and sorting them. A sort-merge join provides results in sorted order by *sid*, but this is not a useful ordering in this example because the projection π_{sname} is applied (on-the-fly) to the result of the join, thereby eliminating the *sid* field from the answer. Thus, the plan using sort-merge join will be retained after Pass 2 only if it is the least expensive plan involving Reserves and Sailors.

Similarly, we also consider taking the plan for Sailors retained after Pass 1 and joining it (as the outer) with Reserves. Now we recognize that we need only Reserves tuples that satisfy $\sigma_{bid=100}$ and $\sigma_{sid=value}$, where *value* is some value from an outer tuple. Again, we consider all available join methods.

We finally retain the cheapest plan overall.

As another example, illustrating the case when more than two relations are joined, consider the following query:

```
SELECT    S.sid, COUNT(*) AS numres
```

```
FROM      Boats B, Reserves R, Sailors S
WHERE     R.sid = S.sid AND B.bid=R.bid AND B.color = 'red'
GROUP BY S.sid
```

This query finds the number of red boats reserved by each sailor. This query is shown in the form of a tree in Figure 14.9.

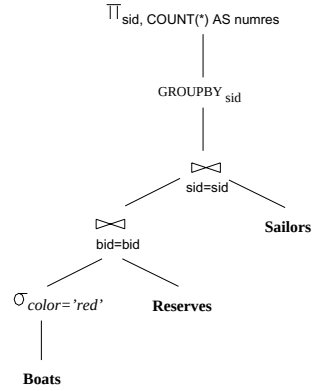

Figure 14.9 A Query Tree

Suppose that the following indexes are available: for Reserves, a B+ tree on the *sid* field and a clustered B+ tree on the *bid* field; for Sailors, a B+ tree index on the *sid* field and a hash index on the *sid* field; and for Boats, a B+ tree index on the *color* field and a hash index on the *color* field. (The list of available indexes is contrived to create a relatively simple, illustrative example.) Let us consider how this query is optimized. The initial focus is on the `SELECT`, `FROM`, and `WHERE` clauses.

Pass 1: The best plan is found for accessing each relation, regarded as the first relation in an execution plan. For Reserves and Sailors, the best plan is obviously a file scan because there are no selections that match an available index. The best plan for Boats is to use the hash index on *color*, which matches the selection *B.color = 'red'*. The B+ tree on *color* also matches this selection and is retained even though the hash index is cheaper, because it returns tuples in sorted order by *color*.

Pass 2: For each of the plans generated in Pass 1, taken as the outer, we consider joining another relation as the inner. Thus, we consider each of the following joins: file scan of Reserves (outer) with Boats (inner), file scan of Reserves (outer) with Sailors (inner), file scan of Sailors (outer) with Boats (inner), file scan of Sailors (outer) with Reserves (inner), Boats accessed via B+ tree index on *color* (outer) with Sailors (inner), Boats accessed via hash index on *color* (outer) with Sailors (inner), Boats accessed via B+ tree index on *color* (outer) with Reserves (inner), and Boats accessed via hash index on *color* (outer) with Reserves (inner).

For each such pair, we consider each join method, and for each join method, we consider every available access path for the inner relation. For each pair of relations, we retain the cheapest of the plans considered for each sorted order in which the tuples are generated. For example, with Boats accessed via the hash index on *color* as the outer relation, an index nested loops join accessing Reserves via the B+ tree index on *bid* is likely to be a good plan; observe that there is no hash index on this field of Reserves. Another plan for joining Reserves and Boats is to access Boats using the hash index on *color*, access Reserves using the B+ tree on *bid*, and use a sort-merge join; this plan, in contrast to the previous one, generates tuples in sorted order by *bid*. It is retained even if the previous plan is cheaper, unless there is an even cheaper plan that produces the tuples in sorted order by *bid*! However, the previous plan, which produces tuples in no particular order, would not be retained if this plan is cheaper.

A good heuristic is to avoid considering cross-products if possible. If we apply this heuristic, we would not consider the following 'joins' in Pass 2 of this example: file scan of Sailors (outer) with Boats (inner), Boats accessed via B+ tree index on *color* (outer) with Sailors (inner), and Boats accessed via hash index on *color* (outer) with Sailors (inner).

Pass 3: For each plan retained in Pass 2, taken as the outer, we consider how to join the remaining relation as the inner. An example of a plan generated at this step is the following: Access Boats via the hash index on *color*, access Reserves via the B+ tree index on *bid*, and join them using a sort-merge join; then take the result of this join as the outer and join with Sailors using a sort-merge join, accessing Sailors via the B+ tree index on the *sid* field. Notice that since the result of the first join is produced in sorted order by *bid*, whereas the second join requires its inputs to be sorted by *sid*, the result of the first join must be sorted by *sid* before being used in the second join. The tuples in the result of the second join are generated in sorted order by *sid*.

The `GROUP BY` clause is considered next, after all joins, and it requires sorting on the *sid* field. For each plan retained in Pass 3, if the result is not sorted on *sid*, we add the cost of sorting on the *sid* field. The sample plan generated in Pass 3 produces tuples in *sid* order; therefore it may be the cheapest plan for the query even if a cheaper plan joins all three relations but does not produce tuples in *sid* order.

14.5 NESTED SUBQUERIES

The unit of optimization in a typical system is a *query block*, and nested queries are dealt with using some form of nested loops evaluation. Consider the following nested query in SQL: *Find the names of sailors with the highest rating.*

```
SELECT S.sname
FROM   Sailors S
```

```
WHERE  S.rating = ( SELECT MAX (S2.rating)
                    FROM   Sailors S2 )
```

In this simple query the nested subquery can be evaluated just once, yielding a single value. This value is incorporated into the top-level query as if it had been part of the original statement of the query. For example, if the highest rated sailor has a rating of 8, the WHERE clause is effectively modified to WHERE *S.rating = 8*.

However, the subquery may sometimes return a relation, or more precisely, a table in the SQL sense (i.e., possibly with duplicate rows). Consider the following query: *Find the names of sailors who have reserved boat number 103.*

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN ( SELECT R.sid
                  FROM   Reserves R
                  WHERE  R.bid = 103 )
```

Again, the nested subquery can be evaluated just once, yielding a collection of *sid*s. For each tuple of Sailors, we must now check whether the *sid* value is in the computed collection of *sid*s; this check entails a join of Sailors and the computed collection of *sid*s, and in principle we have the full range of join methods to choose from. For example, if there is an index on the *sid* field of Sailors, an index nested loops join with the computed collection of *sid*s as the outer relation and Sailors as the inner might be the most efficient join method. However, in many systems, the query optimizer is not smart enough to find this strategy—a common approach is to always do a nested loops join in which the inner relation is the collection of *sid*s computed from the subquery (and this collection may not be indexed).

The motivation for this approach is that it is a simple variant of the technique used to deal with *correlated queries* such as the following version of the previous query:

```
SELECT S.sname
FROM   Sailors S
WHERE  EXISTS ( SELECT *
                FROM   Reserves R
                WHERE  R.bid = 103
                       AND S.sid = R.sid )
```

This query is *correlated*—the tuple variable S from the top-level query appears in the nested subquery. Therefore, we cannot evaluate the subquery just once. In this case the typical evaluation strategy is to evaluate the nested subquery for each tuple of Sailors.

An important point to note about nested queries is that a typical optimizer is likely to do a poor job, because of the limited approach to nested query optimization. This is highlighted below:

- In a nested query with correlation, the join method is effectively index nested loops, with the inner relation typically a subquery (and therefore potentially expensive to compute). This approach creates two distinct problems. First, the nested subquery is evaluated once per outer tuple; if the same value appears in the correlation field (*S.sid* in our example) of several outer tuples, the same subquery is evaluated many times. The second problem is that the approach to nested subqueries is not *set-oriented*. In effect, a join is seen as a scan of the outer relation with a selection on the inner subquery for each outer tuple. This precludes consideration of alternative join methods such as a sort-merge join or a hash join, which could lead to superior plans.
- Even if index nested loops is the appropriate join method, nested query evaluation may be inefficient. For example, if there is an index on the *sid* field of Reserves, a good strategy might be to do an index nested loops join with Sailors as the outer relation and Reserves as the inner relation, and to apply the selection on *bid* on-the-fly. However, this option is not considered when optimizing the version of the query that uses `IN` because the nested subquery is fully evaluated as a first step; that is, Reserves tuples that meet the *bid* selection are retrieved first.
- Opportunities for finding a good evaluation plan may also be missed because of the implicit ordering imposed by the nesting. For example, if there is an index on the *sid* field of Sailors, an index nested loops join with Reserves as the outer relation and Sailors as the inner might be the most efficient plan for our example correlated query. However, this join ordering is never considered by an optimizer.

A nested query often has an equivalent query without nesting, and a correlated query often has an equivalent query without correlation. We have already seen correlated and uncorrelated versions of the example nested query. There is also an equivalent query without nesting:

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid = R.sid AND R.bid=103
```

A typical SQL optimizer is likely to find a much better evaluation strategy if it is given the unnested or 'decorrelated' version of the example query than it would if it were given either of the nested versions of the query. Many current optimizers cannot recognize the equivalence of these queries and transform one of the nested versions to the nonnested form. This is, unfortunately, up to the educated user. From an efficiency standpoint, users are advised to consider such alternative formulations of a query.

Nested queries: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all use some version of correlated evaluation to handle nested queries, which are an important part of the TPC-D benchmark; IBM and Informix support a version in which the results of subqueries are stored in a 'memo' table and the same subquery is not executed multiple times. All these RDBMSs consider decorrelation and "flattening" of nested queries as an option. Microsoft SQL Server, Oracle 8 and IBM DB2 also use rewriting techniques, e.g., Magic Sets (see Chapter 27) or variants, in conjunction with decorrelation.

We conclude our discussion of nested queries by observing that there could be several levels of nesting. In general the approach that we have sketched is extended by evaluating such queries from the innermost to the outermost level, in order, in the absence of correlation. A correlated subquery must be evaluated for each candidate tuple of the higher-level (sub)query that refers to it. The basic idea is thus similar to the case of one-level nested queries; we omit the details.

14.6 OTHER APPROACHES TO QUERY OPTIMIZATION

We have described query optimization based on an exhaustive search of a large space of plans for a given query. The space of all possible plans grows rapidly with the size of the query expression, in particular with respect to the number of joins, because join-order optimization is a central issue. Therefore, heuristics are used to limit the space of plans considered by an optimizer. A widely used heuristic is that only left-deep plans are considered, which works well for most queries. However, once the number of joins becomes greater than about 15, the cost of optimization using this exhaustive approach becomes prohibitively high, even if we consider only left-deep plans.

Such complex queries are becoming important in decision-support environments, and other approaches to query optimization have been proposed. These include **rule-based optimizers**, which use a set of rules to guide the generation of candidate plans, and **randomized plan generation**, which uses probabilistic algorithms such as *simulated annealing* to explore a large space of plans quickly, with a reasonable likelihood of finding a good plan.

Current research in this area also involves techniques for estimating the size of intermediate relations more accurately; **parametric query optimization**, which seeks to find good plans for a given query for each of several different conditions that might be encountered at run-time; and **multiple-query optimization**, in which the optimizer takes concurrent execution of several queries into account.

14.7 POINTS TO REVIEW

- When optimizing SQL queries, they are first decomposed into small units called *blocks*. The outermost query block is often called *outer block*; the other blocks are called *nested blocks*. The first step in optimizing a query block is to translate it into an extended relational algebra expression. Extended relational algebra also contains operators for `GROUP BY` and `HAVING`. Optimizers consider the $\sigma\pi\times$ part of the query separately in a first step and then apply the remaining operations to the result of the first step. Thus, the alternative plans considered result from optimizing the $\sigma\pi\times$ part of the query. **(Section 14.1)**

- Each possible query plan has an associated estimated cost. Since the cost of each operator in the query tree is estimated from the sizes of its input relations, it is important to have good result size estimates. Consider a selection condition in conjunctive normal form. Every term has an associated *reduction factor*, which is the relative reduction in the number of result tuples due to this term. There exist heuristic reduction factor formulas for different kinds of terms that depend on the assumption of uniform distribution of values and independence of relation fields. More accurate reduction factors can be obtained by using more accurate statistics, for example histograms. A *histogram* is a data structure that approximates a data distribution by dividing the value range into buckets and maintaining summarized information about each bucket. In an *equiwidth* histogram, the value range is divided into subranges of equal size. In an *equidepth* histogram, the range is divided into subranges such that each subrange contains the same number of tuples. **(Section 14.2)**

- Two relational algebra expressions are *equivalent* if they produce the same output for all possible input instances. The existence of equivalent expressions implies a choice of evaluation strategies. Several relational algebra *equivalences* allow us to modify a relational algebra expression to obtain an expression with a cheaper plan. **(Section 14.3)**

- Several alternative query plans are constructed. When generating plans for multiple relations, heuristically only *left-deep* plans are considered. A plan is left-deep if the inner relations of all joins are base relations. For plans with a single relation, all possible access methods are considered. Possible access methods include a file scan, a single index, multiple indexes with subsequent intersection of the retrieved rids, usage of an index to retrieve tuples in sorted order, and an index-only access path. Only the cheapest plan for each ordering of tuples is maintained. Query plans for multiple relations are generated in multiple passes. In the first pass, all cheapest single-relation plans for each output order are generated. The second pass generates plans with one join. All plans that were generated during pass one are considered as outer relations and every other relation as inner. Subsequent passes proceed analogously and generate plans with more joins. This process finally generates a plan that contains all relations in the query. **(Section 14.4)**

- Nested subqueries within queries are usually evaluated using some form of nested loops join. For correlated queries, the inner block needs to be evaluated for each tuple of the outer block. Current query optimizers do not handle nested subqueries well. **(Section 14.5)**
- In some scenarios the search space of the exhaustive search algorithm we described is too large and other approaches to query optimization can be used to find a good plan. **(Section 14.6)**

EXERCISES

Exercise 14.1 Briefly answer the following questions.

1. In the context of query optimization, what is an *SQL query block*?
2. Define the term *reduction factor*.
3. Describe a situation in which projection should precede selection in processing a project-select query, and describe a situation where the opposite processing order is better. (Assume that duplicate elimination for projection is done via sorting.)
4. If there are dense, unclustered (secondary) B+ tree indexes on both $R.a$ and $S.b$, the join $R \bowtie_{a=b} S$ could be processed by doing a sort-merge type of join—without doing any sorting—by using these indexes.
   (a) Would this be a good idea if R and S each have only one tuple per page, or would it be better to ignore the indexes and sort R and S? Explain.
   (b) What if R and S each have many tuples per page? Again, explain.
5. Why does the System R optimizer consider only left-deep join trees? Give an example of a plan that would not be considered because of this restriction.
6. Explain the role of *interesting orders* in the System R optimizer.

Exercise 14.2 Consider a relation with this schema:

Employees(*eid:* `integer`, *ename:* `string`, *sal:* `integer`, *title:* `string`, *age:* `integer`)

Suppose that the following indexes, all using Alternative (2) for data entries, exist: a hash index on *eid*, a B+ tree index on *sal*, a hash index on *age*, and a clustered B+ tree index on ⟨*age, sal*⟩. Each Employees record is 100 bytes long, and you can assume that each index data entry is 20 bytes long. The Employees relation contains 10,000 pages.

1. Consider each of the following selection conditions and, assuming that the reduction factor (RF) for each term that matches an index is 0.1, compute the cost of the most selective access path for retrieving all Employees tuples that satisfy the condition:
   (a) $sal > 100$
   (b) $age = 25$

(c) $age > 20$

(d) $eid = 1,000$

(e) $sal > 200 \wedge age > 30$

(f) $sal > 200 \wedge age = 20$

(g) $sal > 200 \wedge title = 'CFO'$

(h) $sal > 200 \wedge age > 30 \wedge title = 'CFO'$

2. Suppose that for each of the preceding selection conditions, you want to retrieve the average salary of qualifying tuples. For each selection condition, describe the least expensive evaluation method and state its cost.
3. Suppose that for each of the preceding selection conditions, you want to compute the average salary for each *age* group. For each selection condition, describe the least expensive evaluation method and state its cost.
4. Suppose that for each of the preceding selection conditions, you want to compute the average age for each *sal* level (i.e., group by *sal*). For each selection condition, describe the least expensive evaluation method and state its cost.
5. For each of the following selection conditions, describe the best evaluation method:

   (a) $sal > 200 \vee age = 20$

   (b) $sal > 200 \vee title = 'CFO'$

   (c) $title = 'CFO' \wedge ename = 'Joe'$

Exercise 14.3 For each of the following SQL queries, for each relation involved, list the attributes that must be examined in order to compute the answer. All queries refer to the following relations:

Emp(*eid:* `integer`, *did:* `integer`, *sal:* `integer`, *hobby:* `char(20)`)
Dept(*did:* `integer`, *dname:* `char(20)`, *floor:* `integer`, *budget:* `real`)

1. `SELECT * FROM` Emp
2. `SELECT * FROM` Emp, Dept
3. `SELECT * FROM` Emp E, Dept D `WHERE` E.did = D.did
4. `SELECT` E.eid, D.dname `FROM` Emp E, Dept D `WHERE` E.did = D.did
5. `SELECT COUNT(*) FROM` Emp E, Dept D `WHERE` E.did = D.did
6. `SELECT MAX(E.sal) FROM` Emp E, Dept D `WHERE` E.did = D.did
7. `SELECT MAX(E.sal) FROM` Emp E, Dept D `WHERE` E.did = D.did `AND` D.floor = 5
8. `SELECT` E.did, `COUNT(*) FROM` Emp E, Dept D `WHERE` E.did = D.did `GROUP BY` D.did
9. `SELECT` D.floor, `AVG`(D.budget) `FROM` Dept D `GROUP BY` D.floor `HAVING COUNT(*) > 2`
10. `SELECT` D.floor, `AVG`(D.budget) `FROM` Dept D `GROUP BY` D.floor `ORDER BY` D.floor

Exercise 14.4 You are given the following information:

Executives has attributes *ename*, *title*, *dname*, and *address*; all are string fields of the same length.
The *ename* attribute is a candidate key.
The relation contains 10,000 pages.
There are 10 buffer pages.

1. Consider the following query:

   `SELECT` E.title, E.ename `FROM` Executives E `WHERE` E.title='CFO'

   Assume that only 10 percent of Executives tuples meet the selection condition.

   (a) Suppose that a clustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan? (In this and subsequent questions, be sure to describe the plan that you have in mind.)

   (b) Suppose that an unclustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

   (c) Suppose that a clustered B+ tree index on *ename* is (the only index) available. What is the cost of the best plan?

   (d) Suppose that a clustered B+ tree index on *address* is (the only index) available. What is the cost of the best plan?

   (e) Suppose that a clustered B+ tree index on $\langle ename, title \rangle$ is (the only index) available. What is the cost of the best plan?

2. Suppose that the query is as follows:

   `SELECT` E.ename `FROM` Executives E `WHERE` E.title='CFO' `AND` E.dname='Toy'

   Assume that only 10 percent of Executives tuples meet the condition $E.title = 'CFO'$, only 10 percent meet $E.dname = 'Toy'$, and that only 5 percent meet both conditions.

   (a) Suppose that a clustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

   (b) Suppose that a clustered B+ tree index on *dname* is (the only index) available. What is the cost of the best plan?

   (c) Suppose that a clustered B+ tree index on $\langle title, dname \rangle$ is (the only index) available. What is the cost of the best plan?

   (d) Suppose that a clustered B+ tree index on $\langle title, ename \rangle$ is (the only index) available. What is the cost of the best plan?

   (e) Suppose that a clustered B+ tree index on $\langle dname, title, ename \rangle$ is (the only index) available. What is the cost of the best plan?

   (f) Suppose that a clustered B+ tree index on $\langle ename, title, dname \rangle$ is (the only index) available. What is the cost of the best plan?

3. Suppose that the query is as follows:

   `SELECT` E.title, `COUNT`(*) `FROM` Executives E `GROUP BY` E.title

   (a) Suppose that a clustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

(b) Suppose that an unclustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

(c) Suppose that a clustered B+ tree index on *ename* is (the only index) available. What is the cost of the best plan?

(d) Suppose that a clustered B+ tree index on $\langle ename, title \rangle$ is (the only index) available. What is the cost of the best plan?

(e) Suppose that a clustered B+ tree index on $\langle title, ename \rangle$ is (the only index) available. What is the cost of the best plan?

4. Suppose that the query is as follows:

```
SELECT E.title, COUNT(*) FROM Executives E
WHERE E.dname > 'W%' GROUP BY E.title
```

Assume that only 10 percent of Executives tuples meet the selection condition.

(a) Suppose that a clustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan? If an additional index (on any search key that you want) is available, would it help to produce a better plan?

(b) Suppose that an unclustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?

(c) Suppose that a clustered B+ tree index on *dname* is (the only index) available. What is the cost of the best plan? If an additional index (on any search key that you want) is available, would it help to produce a better plan?

(d) Suppose that a clustered B+ tree index on $\langle dname, title \rangle$ is (the only index) available. What is the cost of the best plan?

(e) Suppose that a clustered B+ tree index on $\langle title, dname \rangle$ is (the only index) available. What is the cost of the best plan?

Exercise 14.5 Consider the query $\pi_{A,B,C,D}(R \bowtie_{A=C} S)$. Suppose that the projection routine is based on sorting and is smart enough to eliminate all but the desired attributes during the initial pass of the sort, and also to toss out duplicate tuples on-the-fly while sorting, thus eliminating two potential extra passes. Finally, assume that you know the following:

R is 10 pages long, and R tuples are 300 bytes long.
S is 100 pages long, and S tuples are 500 bytes long.
C is a key for S, and A is a key for R.
The page size is 1,024 bytes.
Each S tuple joins with exactly one R tuple.
The combined size of attributes A, B, C, and D is 450 bytes.
A and B are in R and have a combined size of 200 bytes; C and D are in S.

1. What is the cost of writing out the final result? (As usual, you should ignore this cost in answering subsequent questions.)
2. Suppose that three buffer pages are available, and the only join method that is implemented is simple (page-oriented) nested loops.

(a) Compute the cost of doing the projection followed by the join.

(b) Compute the cost of doing the join followed by the projection.

(c) Compute the cost of doing the join first and then the projection on-the-fly.

(d) Would your answers change if 11 buffer pages were available?

3. Suppose that there are three buffer pages available, and the only join method that is implemented is block nested loops.

(a) Compute the cost of doing the projection followed by the join.

(b) Compute the cost of doing the join followed by the projection.

(c) Compute the cost of doing the join first and then the projection on-the-fly.

(d) Would your answers change if 11 buffer pages were available?

Exercise 14.6 Briefly answer the following questions.

1. Explain the role of relational algebra equivalences in the System R optimizer.
2. Consider a relational algebra expression of the form $\sigma_c(\pi_l(R \times S))$. Suppose that the equivalent expression with selections and projections pushed as much as possible, taking into account only relational algebra equivalences, is in one of the following forms. In each case give an illustrative example of the selection conditions and the projection lists (c, l, $c1$, $l1$, etc.).

(a) *Equivalent maximally pushed form:* $\pi_{l1}(\sigma_{c1}(R) \times S)$.

(b) *Equivalent maximally pushed form:* $\pi_{l1}(\sigma_{c1}(R) \times \sigma_{c2}(S))$.

(c) *Equivalent maximally pushed form:* $\sigma_c(\pi_{l1}(\pi_{l2}(R) \times S))$.

(d) *Equivalent maximally pushed form:* $\sigma_{c1}(\pi_{l1}(\sigma_{c2}(\pi_{l2}(R)) \times S))$.

(e) *Equivalent maximally pushed form:* $\sigma_{c1}(\pi_{l1}(\pi_{l2}(\sigma_{c2}(R)) \times S))$.

(f) *Equivalent maximally pushed form:* $\pi_l(\sigma_{c1}(\pi_{l1}(\pi_{l2}(\sigma_{c2}(R)) \times S)))$.

Exercise 14.7 Consider the following relational schema and SQL query. The schema captures information about employees, departments, and company finances (organized on a per department basis).

Emp(*eid:* `integer`, *did:* `integer`, *sal:* `integer`, *hobby:* `char(20)`)
Dept(*did:* `integer`, *dname:* `char(20)`, *floor:* `integer`, *phone:* `char(10)`)
Finance(*did:* `integer`, *budget:* `real`, *sales:* `real`, *expenses:* `real`)

Consider the following query:

```
SELECT D.dname, F.budget
FROM   Emp E, Dept D, Finance F
WHERE  E.did=D.did AND D.did=F.did AND D.floor=1
       AND E.sal ≥ 59000 AND E.hobby = 'yodeling'
```

1. Identify a relational algebra tree (or a relational algebra expression if you prefer) that reflects the order of operations that a decent query optimizer would choose.

2. List the join orders (i.e., orders in which pairs of relations can be joined together to compute the query result) that a relational query optimizer will consider. (Assume that the optimizer follows the heuristic of never considering plans that require the computation of cross-products.) Briefly explain how you arrived at your list.
3. Suppose that the following additional information is available: Unclustered B+ tree indexes exist on *Emp.did*, *Emp.sal*, *Dept.floor*, *Dept.did*, and *Finance.did*. The system's statistics indicate that employee salaries range from 10,000 to 60,000, employees enjoy 200 different hobbies, and the company owns two floors in the building. There are a total of 50,000 employees and 5,000 departments (each with corresponding financial information) in the database. The DBMS used by the company has just one join method available, namely, index nested loops.
   (a) For each of the query's base relations (Emp, Dept and Finance) estimate the number of tuples that would be initially selected from that relation if all of the non-join predicates on that relation were applied to it before any join processing begins.
   (b) Given your answer to the preceding question, which of the join orders that are considered by the optimizer has the least estimated cost?

Exercise 14.8 Consider the following relational schema and SQL query:

Suppliers(*sid:* `integer`, *sname:* `char(20)`, *city:* `char(20)`)
Supply(*sid:* `integer`, *pid:* `integer`)
Parts(*pid:* `integer`, *pname:* `char(20)`, *price:* `real`)

```
SELECT S.sname, P.pname
FROM   Suppliers S, Parts P, Supply Y
WHERE  S.sid = Y.sid AND Y.pid = P.pid AND
       S.city = 'Madison' AND P.price ≤ 1,000
```

1. What information about these relations will the query optimizer need to select a good query execution plan for the given query?
2. How many different join orders, assuming that cross-products are disallowed, will a System R style query optimizer consider when deciding how to process the given query? List each of these join orders.
3. What indexes might be of help in processing this query? Explain briefly.
4. How does adding `DISTINCT` to the `SELECT` clause affect the plans produced?
5. How does adding `ORDER BY` *sname* to the query affect the plans produced?
6. How does adding `GROUP BY` *sname* to the query affect the plans produced?

Exercise 14.9 Consider the following scenario:

Emp(*eid:* `integer`, *sal:* `integer`, *age:* `real`, *did:* `integer`)
Dept(*did:* `integer`, *projid:* `integer`, *budget:* `real`, *status:* `char(10)`)
Proj(*projid:* `integer`, *code:* `integer`, *report:* `varchar`)

Assume that each Emp record is 20 bytes long, each Dept record is 40 bytes long, and each Proj record is 2,000 bytes long on average. There are 20,000 tuples in Emp, 5,000 tuples in Dept (note that *did* is not a key), and 1,000 tuples in Proj. Each department, identified by *did*, has 10 projects on average. The file system supports 4,000 byte pages, and 12 buffer pages are available. The following questions are all based on this information. You can assume uniform distribution of values. State any additional assumptions. The cost metric to use is *the number of page I/Os*. Ignore the cost of writing out the final result.

1. Consider the following two queries: "Find all employees with $age = 30$" and "Find all projects with $code = 20$." Assume that the number of qualifying tuples is the same in each case. If you are building indexes on the selected attributes to speed up these queries, for which query is a *clustered* index (in comparison to an *unclustered* index) more important?
2. Consider the following query: "Find all employees with $age > 30$." Assume that there is an unclustered index on *age*. Let the number of qualifying tuples be N. For what values of N is a sequential scan cheaper than using the index?
3. Consider the following query:

```
SELECT *
FROM   Emp E, Dept D
WHERE  E.did=D.did
```

   (a) Suppose that there is a clustered hash index on *did* on Emp. List all the plans that are considered and identify the plan with the least estimated cost.

   (b) Assume that both relations are sorted on the join column. List all the plans that are considered and show the plan with the least estimated cost.

   (c) Suppose that there is a clustered B+ tree index on *did* on Emp and that Dept is sorted on *did*. List all the plans that are considered and identify the plan with the least estimated cost.

4. Consider the following query:

```
SELECT      D.did, COUNT(*)
FROM        Dept D, Proj P
WHERE       D.projid=P.projid
GROUP BY    D.did
```

   (a) Suppose that no indexes are available. Show the plan with the least estimated cost.

   (b) If there is a hash index on *P.projid* what is the plan with least estimated cost?

   (c) If there is a hash index on *D.projid* what is the plan with least estimated cost?

   (d) If there is a hash index on *D.projid* and *P.projid* what is the plan with least estimated cost?

   (e) Suppose that there is a clustered B+ tree index on *D.did* and a hash index on *P.projid*. Show the plan with the least estimated cost.

   (f) Suppose that there is a clustered B+ tree index on *D.did*, a hash index on *D.projid*, and a hash index on *P.projid*. Show the plan with the least estimated cost.

   (g) Suppose that there is a clustered B+ tree index on $\langle D.did,\ D.projid \rangle$ and a hash index on *P.projid*. Show the plan with the least estimated cost.

(h) Suppose that there is a clustered B+ tree index on ⟨*D.projid, D.did*⟩ and a hash index on *P.projid*. Show the plan with the least estimated cost.

5. Consider the following query:

```
SELECT      D.did, COUNT(*)
FROM        Dept D, Proj P
WHERE       D.projid=P.projid AND D.budget>99000
GROUP BY    D.did
```

Assume that department budgets are uniformly distributed in the range 0 to 100,000.

(a) Show the plan with least estimated cost if no indexes are available.

(b) If there is a hash index on *P.projid* show the plan with least estimated cost.

(c) If there is a hash index on *D.budget* show the plan with least estimated cost.

(d) If there is a hash index on *D.projid* and *D.budget* show the plan with least estimated cost.

(e) Suppose that there is a clustered B+ tree index on ⟨*D.did,D.budget*⟩ and a hash index on *P.projid*. Show the plan with the least estimated cost.

(f) Suppose that there is a clustered B+ tree index on *D.did*, a hash index on *D.budget*, and a hash index on *P.projid*. Show the plan with the least estimated cost.

(g) Suppose that there is a clustered B+ tree index on ⟨*D.did, D.budget, D.projid*⟩ and a hash index on *P.projid*. Show the plan with the least estimated cost.

(h) Suppose that there is a clustered B+ tree index on ⟨*D.did, D.projid, D.budget*⟩ and a hash index on *P.projid*. Show the plan with the least estimated cost.

6. Consider the following query:

```
SELECT E.eid, D.did, P.projid
FROM   Emp E, Dept D, Proj P
WHERE  E.sal=50,000 AND D.budget>20,000
       E.did=D.did AND D.projid=P.projid
```

Assume that employee salaries are uniformly distributed in the range 10,009 to 110,008 and that project budgets are uniformly distributed in the range 10,000 to 30,000. There is a clustered index on *sal* for Emp, a clustered index on *did* for Dept, and a clustered index on *projid* for Proj.

(a) List all the one-relation, two-relation, and three-relation subplans considered in optimizing this query.

(b) Show the plan with the least estimated cost for this query.

(c) If the index on Proj were unclustered, would the cost of the preceding plan change substantially? What if the index on Emp or on Dept were unclustered?

PROJECT-BASED EXERCISES

Exercise 14.10 (*Note to instructors: This exercise can be made more specific by providing additional details about the queries and the catalogs. See Appendix B.*) Minibase has a nice query optimizer visualization tool that lets you see how a query is optimized. Try initializing the catalogs to reflect various scenarios (perhaps taken from the chapter or the other exercises) and optimizing different queries. Using the graphical interface, you can look at each enumerated plan at several levels of detail, toggle (i.e., turn on/off) the availability of indexes, join methods, and so on.

BIBLIOGRAPHIC NOTES

Query optimization is critical in a relational DBMS, and it has therefore been extensively studied. We have concentrated in this chapter on the approach taken in System R, as described in [581], although our discussion incorporated subsequent refinements to the approach. [688] describes query optimization in Ingres. Good surveys can be found in [349] and [338]. [372] contains several articles on query processing and optimization.

From a theoretical standpoint, [132] showed that determining whether two *conjunctive queries* (queries involving only selections, projections, and cross-products) are equivalent is an NP-complete problem; if relations are *multisets*, rather than sets of tuples, it is not known whether the problem is decidable, although it is Π_2^p hard. The equivalence problem was shown to be decidable for queries involving selections, projections, cross-products, and unions in [560]; surprisingly, this problem is undecidable if relations are multisets [343]. Equivalence of conjunctive queries in the presence of integrity constraints is studied in [26], and equivalence of conjunctive queries with inequality selections is studied in [379].

An important problem in query optimization is estimating the size of the result of a query expression. Approaches based on sampling are explored in [298, 299, 324, 416, 497]. The use of detailed statistics, in the form of histograms, to estimate size is studied in [344, 487, 521]. Unless care is exercised, errors in size estimation can quickly propagate and make cost estimates worthless for expressions with several operators. This problem is examined in [339]. [445] surveys several techniques for estimating result sizes and correlations between values in relations. There are a number of other papers in this area, for example, [22, 143, 517, 636], and our list is far from complete.

Semantic query optimization is based on transformations that preserve equivalence only when certain integrity constraints hold. The idea was introduced in [375] and developed further in [594, 127, 599].

In recent years, there has been increasing interest in complex queries for decision support applications. Optimization of nested SQL queries is discussed in [667, 256, 364, 368, 486]. The use of the Magic Sets technique for optimizing SQL queries is studied in [482, 484, 483, 586, 583]. Rule-based query optimizers are studied in [246, 277, 425, 468, 519]. Finding a good join order for queries with a large number of joins is studied in [391, 340, 341, 637]. Optimization of multiple queries for simultaneous execution is considered in [510, 551, 582].

Determining query plans at run-time is discussed in [278, 342]. Re-optimization of running queries based on statistics gathered during query execution is considered by Kabra and DeWitt [353]. Probabilistic optimization of queries is proposed in [152, 192].