by the names of the attributes to which they refer. Thus, the optimization of the $\sigma\pi$x part of the query essentially ignores these aggregate operations.

The optimizer finds the best plan for the $\sigma\pi$x 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 x *Reserves* x *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 15.4.1.

To a first approximation therefore, the alternative plans examined by a typical optimizer can be understood in terms of the plans considered for $\sigma\pi$x queries. An optimizer enumerates plans by applying several equivalences between relational algebra expressions, which we present in Section 15.3. We discuss the space of plans enumerated by an optimizer in Section 15.4.

15.2 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 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 14. 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 14, we 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 find a good plan.

15.2.1 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 *attTibute list*
FROM *Telation list*
WHERE *teTml 1\ teTm2 1\ . . . 1\ teTm$_n$*

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 $\frac{1}{Keys(I)}$ 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 *col'umn,* 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 $\frac{1}{MAX(NKeys(I1),NKeys(I2))}$ if there are indexes I1 and 12 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 *columnl,* we assume that each of the $NJ(eys(I2)$ values for *column2* is equally likely. Therefore, the number of tuples that have the same value in *column2* as a given value in *columni* 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* (I-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

Estimating Query Characteristics: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all usehistograms 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 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 with no 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 build histograms but not directly by the optimizer to estimate query characteristics. Sometimes, sampling is used to do load balancing in parallel implementations.

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 $colu:rmn > value$ 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 on the assumption that the distribution of values is uniform.

Estimates can be improved considerably 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 15.3. 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 *age* values are integers in the range 0 to 14. The second distribution approximates *D* by assuming that each *age* value in the range a to 14 appears equally often in the underlying collection of tuples. This approximation can be stored compactly because we need to record only 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).

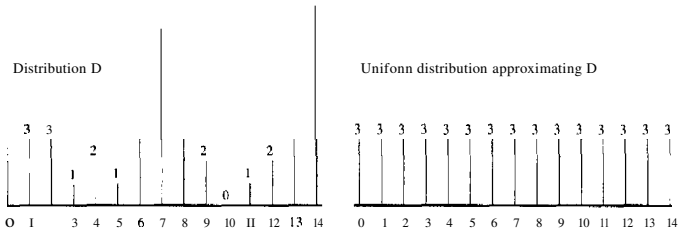

Figure 15.3 Uniform vs. Nonuniform Distributions

Consider the selection $age > 13$. From the distribution *D* in Figure 15.3, we see that the result has 9 tuples. Using the uniform distribution approximation, on the other hand, we estimate the result size as $\frac{1}{15} \cdot 45 = 3$ tuples. Clearly, the estimate is quite inaccurate.

A histogram is a data structure maintained by a DBMS to approximate a data distribution. In Figure 15.4, we show how the data distribution from Figure 15.3 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 15.4 shows two different kinds of histograms, called *equiwidth* and *equidepth,* respectively.

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 $\frac{1}{3} \cdot 15 = 5$ tuples. As this example shows, we assume that the distribution *within* a histogram bucket is uniform. Therefore, when we simply maintain the high and low values for index

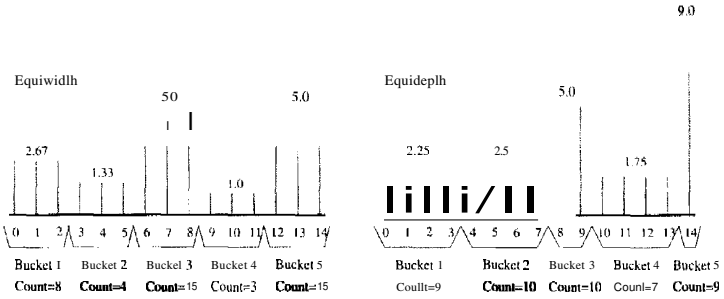

Figure 15.4 Histograms Approximating Distribution *D*

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, called an equidepth histogram, is also illustrated in Figure 15.4. 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) generally contains 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, the frequent values are important.

Proceeding further with the intuition about the importance of frequent values, another alternative is to maintain separate counts for a small number of very frequent values, say the *age* values 7 and 14 in our example, and maintain an equidepth (or other) histogram to cover the remaining values. Such a histogram is called a compressed histogram. Most commercial DB1\1Ss currently use equidepth histograms, and some use compressed histograms.

```
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 15.5 A Single-Relation Query

15.4 ENUMERATION OF ALTERNATIVE PLANS

We now come to an issue that is at the heart of an optimizer, namely, the space of alternative plans 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 12.1.1 indicated how the cost of a plan is estimated. The algebraic equivalences discussed in Section 15.3 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 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.

15.4.1 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 14 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 15.5. Using the extended algebra notation introduced in Section 15.1.2, we can write this query as:

$$\pi_{S.rating,COUNT(*)}($$

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

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 14.2.1). 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 14.6.

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

1. Perfonning a file scan to retrieve tuples and apply the selections and pro-· jections.
2. 'Writing out tuples after the selections and projectiolls.
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-fiy (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 conditiolls. 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. Therefore, the cost of this step is 20 l/Os. The cost of sorting this intermediate relation (which we 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 l/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 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 trec 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 gTOUp. 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 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 examine only 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 15.5 as a running example. We 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 do 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.mting* > 5 to each retrieved tuple; project out fields not mentioned in the SELECT, GROUP BY, and 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 conclitioIl.

Utilizing Indexes: All of the main RDBMSs recognize the importance of index-only plans and look for such plans whenever possible. In IBM DD2, 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 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 *(sal, age)* pairs that appear in the table.

As an example of the second case, we can retrieve rids of tuples satisfying *mting>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 in which *S.mting* > 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 *(mting, sname, age)* index in which *mting* > 5. These entries are sorted by *rating* (and then by *snarne* 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 unclusterecl indexes.

15.4.2 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.

Enumeration of Left-Deep Plans

As we saw in Chapter 12, current relational systems, following the lead of the System R optimizer, only consider left-deep plans. We now discuss how this dass of plans is efficiently searched using dynamic programming.

Consider a query block of the form:

SELECT *attribute list*
FROM *relation list*
WHERE $term_1 \wedge term2 \wedge \ldots \wedge term_n$

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* 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 15.4.1.

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 implementing a GROUP BY or ORDER BY clause. Hence, 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 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 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 and can therefore be projected out before the join.

Note 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 rely 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 *befor"e* 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 SOlne join methods, the join operator rnight 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 relation, should this be necessary, in the cost of the join. The adjustments to the join costs discussed in Chapter 14 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 lnethod to llse for B. The access method chosen for B retrieves, in general, a subset of the tuples in B, possibly with some fields eliminated, as discllssed later. 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 15.4.1. 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 14.

Examples of Multiple-Relation Query Optimization

Consider the query tree shown in Figure 12.3. Figure 15.6 shows the same query, taking into account how selections and projections are considered early.

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

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 (to avoid repeated analysis of the same plan). Systems also vary in the degree of control they give 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.

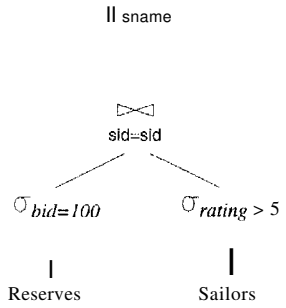

Figure 15.6 A Query Tree

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. Therefore, the plan using sort-merge join is 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 relation) 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 Rcolor = '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 15.7.

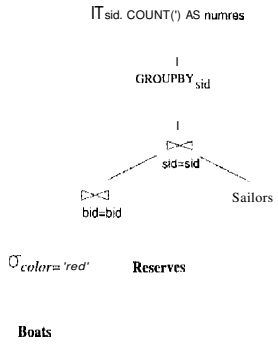

Figure 15.7 A Query Tree

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.

In 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 no selections match an available index. The best plan for Boats is to use the hash index on *color,* which matches the selection *B.coloT = '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.*

In Pass 2, for each of the plans generated in Pass 1, taken as the outer relation, we consider joining another relation as the inner one. Hence, 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 every 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 every sorted order in which the tuples are generated. For example, with Boats accessed via the hash index on *coloT* 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 *coloT,* 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 an even cheaper plan 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).

In Pass 3, for each plan retained in Pass 2, taken as the outer relation, we consider how to join the remaining relation as the inner one. An example of a plan generated at this step is the following: Access Boats via the hash index on *coloT,* 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. Note 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 *s'id,* 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 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.