13 INTRODUCTION TO QUERY OPTIMIZATION

This very remarkable man
Commends a most practical plan:
You can do what you want
If you don't think you can't,
So don't think you can't if you can.

—Charles Inge

Consider a simple selection query asking for all reservations made by sailor Joe. As we saw in the previous chapter, there are many ways to evaluate even this simple query, each of which is superior in certain situations, and the DBMS must consider these alternatives and choose the one with the least estimated cost. Queries that consist of several operations have many more evaluation options, and finding a good plan represents a significant challenge.

A more detailed view of the query optimization and execution layer in the DBMS architecture presented in Section 1.8 is shown in Figure 13.1. Queries are parsed and then presented to a **query optimizer**, which is responsible for identifying an efficient execution plan for evaluating the query. The optimizer generates alternative plans and chooses the plan with the least estimated cost. To estimate the cost of a plan, the optimizer uses information in the system catalogs.

This chapter presents an overview of query optimization, some relevant background information, and a case study that illustrates and motivates query optimization. We discuss relational query optimizers in detail in Chapter 14.

Section 13.1 lays the foundation for our discussion. It introduces query evaluation plans, which are composed of relational operators; considers alternative techniques for passing results between relational operators in a plan; and describes an *iterator* interface that makes it easy to combine code for individual relational operators into an executable plan. In Section 13.2, we describe the system catalogs for a relational DBMS. The catalogs contain the information needed by the optimizer to choose between alternate plans for a given query. Since the costs of alternative plans for a given query can vary by orders of magnitude, the choice of query evaluation plan can have a dramatic impact on execution time. We illustrate the differences in cost between alternative plans through a detailed motivating example in Section 13.3.

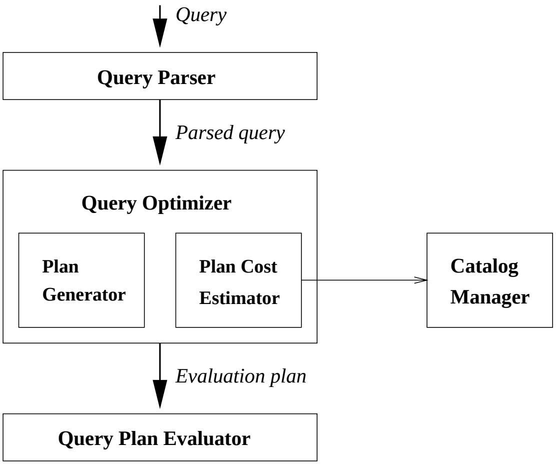

Figure 13.1 Query Parsing, Optimization, and Execution

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

Sailors(*sid:* `integer`, *sname:* `string`, *rating:* `integer`, *age:* `real`)
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.

13.1 OVERVIEW OF RELATIONAL QUERY OPTIMIZATION

The goal of a query optimizer is to find a good evaluation plan for a given query. The space of plans considered 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. Optimizing such 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.

Commercial optimizers: Current RDBMS optimizers are complex pieces of software with many closely guarded details and typically represent 40 to 50 man-years of development effort!

In this section we lay the foundation for our discussion of query optimization by introducing evaluation plans. We conclude this section by highlighting IBM's System R optimizer, which influenced subsequent relational optimizers.

13.1.1 Query Evaluation Plans

A **query evaluation plan** (or simply **plan**) consists of an extended relational algebra tree, with additional annotations at each node indicating the access methods to use for each relation and the implementation method to use for each relational operator.

Consider the following SQL query:

```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
       AND R.bid = 100 AND S.rating > 5
```

This query can be expressed in relational algebra as follows:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$

This expression is shown in the form of a tree in Figure 13.2. The algebra expression partially specifies how to evaluate the query—we first compute the natural join of Reserves and Sailors, then perform the selections, and finally project the *sname* field.

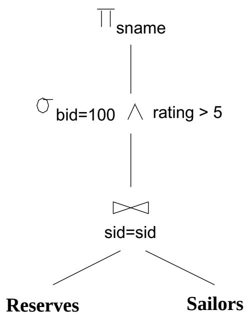

Figure 13.2 Query Expressed as a Relational Algebra Tree

To obtain a fully specified evaluation plan, we must decide on an implementation for each of the algebra operations involved. For example, we can use a page-oriented simple nested loops join with Reserves as the outer relation and apply selections and projections to each tuple in the result of the join as it is produced; the result of the join before the selections and projections is never stored in its entirety. This query evaluation plan is shown in Figure 13.3.

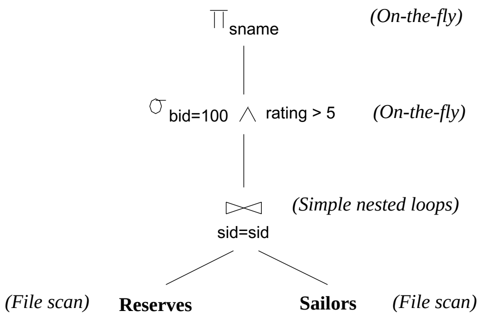

Figure 13.3 Query Evaluation Plan for Sample Query

In drawing the query evaluation plan, we have used the convention that the *outer relation* is the *left child* of the join operator. We will adopt this convention henceforth.

13.1.2 Pipelined Evaluation

When a query is composed of several operators, the result of one operator is sometimes **pipelined** to another operator without creating a temporary relation to hold the intermediate result. The plan in Figure 13.3 pipelines the output of the join of Sailors and Reserves into the selections and projections that follow. Pipelining the output of an operator into the next operator saves the cost of writing out the intermediate result and reading it back in, and the cost savings can be significant. If the output of an operator is saved in a temporary relation for processing by the next operator, we say that the tuples are **materialized**. Pipelined evaluation has lower overhead costs than materialization and is chosen whenever the algorithm for the operator evaluation permits it.

There are many opportunities for pipelining in typical query plans, even simple plans that involve only selections. Consider a selection query in which only part of the selection condition matches an index. We can think of such a query as containing *two* instances of the selection operator: The first contains the primary, or matching, part of the original selection condition, and the second contains the rest of the selection condition. We can evaluate such a query by applying the primary selection and writing the result to a temporary relation and then applying the second selection to the temporary relation. In contrast, a pipelined evaluation consists of applying the second selection to each tuple in the result of the primary selection as it is produced and adding tuples that qualify to the final result. When the input relation to a unary operator (e.g., selection or projection) is pipelined into it, we sometimes say that the operator is applied **on-the-fly**.

As a second and more general example, consider a join of the form $(A \bowtie B) \bowtie C$, shown in Figure 13.4 as a tree of join operations.

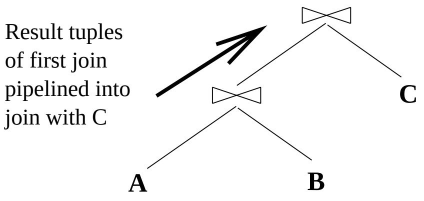

Figure 13.4 A Query Tree Illustrating Pipelining

Both joins can be evaluated in pipelined fashion using some version of a nested loops join. Conceptually, the evaluation is initiated from the root, and the node joining A and B produces tuples as and when they are requested by its parent node. When the root node gets a page of tuples from its left child (the outer relation), all the matching inner tuples are retrieved (using either an index or a scan) and joined with matching outer tuples; the current page of outer tuples is then discarded. A request is then made to the left child for the next page of tuples, and the process is repeated. Pipelined evaluation is thus a *control strategy* governing the rate at which different joins in the plan proceed. It has the great virtue of not writing the result of intermediate joins to a temporary file because the results are produced, consumed, and discarded one page at a time.

13.1.3 The Iterator Interface for Operators and Access Methods

A query evaluation plan is a tree of relational operators and is executed by calling the operators in some (possibly interleaved) order. Each operator has one or more inputs and an output, which are also nodes in the plan, and tuples must be passed between operators according to the plan's tree structure.

In order to simplify the code that is responsible for coordinating the execution of a plan, the relational operators that form the nodes of a plan tree (which is to be evaluated using pipelining) typically support a uniform **iterator** interface, hiding the internal implementation details of each operator. The iterator interface for an operator includes the functions **open**, **get_next**, and **close**. The *open* function initializes the state of the iterator by allocating buffers for its inputs and output, and is also used to pass in arguments such as selection conditions that modify the behavior of the operator. The code for the *get_next* function calls the *get_next* function on each input node and calls operator-specific code to process the input tuples. The output tuples generated by the processing are placed in the output buffer of the operator, and the state of the iterator is updated to keep track of how much input has been consumed. When all output tuples have been produced through repeated calls to *get_next*, the *close* function is called (by the code that initiated execution of this operator) to deallocate state information.

The iterator interface supports pipelining of results naturally; the decision to pipeline or materialize input tuples is encapsulated in the operator-specific code that processes input tuples. If the algorithm implemented for the operator allows input tuples to be processed completely when they are received, input tuples are not materialized and the evaluation is pipelined. If the algorithm examines the same input tuples several times, they are materialized. This decision, like other details of the operator's implementation, is hidden by the iterator interface for the operator.

The iterator interface is also used to encapsulate access methods such as B+ trees and hash-based indexes. Externally, access methods can be viewed simply as operators that produce a stream of output tuples. In this case, the *open* function can be used to pass the selection conditions that match the access path.

13.1.4 The System R Optimizer

Current relational query optimizers have been greatly influenced by choices made in the design of IBM's System R query optimizer. Important design choices in the System R optimizer include:

1. The use of *statistics* about the database instance to estimate the cost of a query evaluation plan.
2. A decision to consider only plans with binary joins in which the inner relation is a base relation (i.e., not a temporary relation). This heuristic reduces the (potentially very large) number of alternative plans that must be considered.
3. A decision to focus optimization on the class of SQL queries without nesting and to treat nested queries in a relatively ad hoc way.
4. A decision not to perform duplicate elimination for projections (except as a final step in the query evaluation when required by a `DISTINCT` clause).
5. A model of cost that accounted for CPU costs as well as I/O costs.

Our discussion of optimization reflects these design choices, except for the last point in the preceding list, which we ignore in order to retain our simple cost model based on the number of page I/Os.

13.2 SYSTEM CATALOG IN A RELATIONAL DBMS

We can store a relation using one of several alternative file structures, and we can create one or more indexes—each stored as a file—on every relation. Conversely, in a relational DBMS, every file contains either the tuples in a relation or the entries in an index. The collection of files corresponding to users' relations and indexes represents the *data* in the database.

A fundamental property of a database system is that it maintains a description of all the data that it contains. A relational DBMS maintains information about every relation and index that it contains. The DBMS also maintains information about views, for which no tuples are stored explicitly; rather, a definition of the view is stored and used to compute the tuples that belong in the view when the view is queried. This information is stored in a collection of relations, maintained by the system, called the **catalog relations**; an example of a catalog relation is shown in Figure 13.5. The catalog relations are also called the **system catalog**, the *catalog*, or the **data dictionary**. The system catalog is sometimes referred to as **metadata**; that is, not data, but descriptive information about the data. The information in the system catalog is used extensively for query optimization.

13.2.1 Information Stored in the System Catalog

Let us consider what is stored in the system catalog. At a minimum we have system-wide information, such as the size of the buffer pool and the page size, and the following information about individual relations, indexes, and views:

- For each relation:
  - Its *relation name*, the *file name* (or some identifier), and the *file structure* (e.g., heap file) of the file in which it is stored.
  - The *attribute name* and *type* of each of its attributes.
  - The *index name* of each index on the relation.
  - The *integrity constraints* (e.g., primary key and foreign key constraints) on the relation.
- For each index:
  - The *index name* and the *structure* (e.g., B+ tree) of the index.
  - The *search key* attributes.
- For each view:
  - Its *view name* and *definition*.

In addition, statistics about relations and indexes are stored in the system catalogs and updated periodically (*not* every time the underlying relations are modified). The following information is commonly stored:

- **Cardinality:** The number of tuples *NTuples(R)* for each relation R.
- **Size:** The number of pages *NPages(R)* for each relation R.
- **Index Cardinality:** Number of distinct key values *NKeys(I)* for each index I.
- **Index Size:** The number of pages *INPages(I)* for each index I. (For a B+ tree index I, we will take *INPages* to be the number of leaf pages.)
- **Index Height:** The number of nonleaf levels *IHeight(I)* for each tree index I.
- **Index Range:** The minimum present key value *ILow(I)* and the maximum present key value *IHigh(I)* for each index I.

We will assume that the database architecture presented in Chapter 1 is used. Further, we assume that each file of records is implemented as a separate file of pages. Other file organizations are possible, of course. For example, in System R a page file can contain pages that store records from more than one record file. (System R uses different names for these abstractions and in fact uses somewhat different abstractions.) If such a file organization is used, additional statistics must be maintained, such as the fraction of pages in a file that contain records from a given collection of records.

The catalogs also contain information about *users*, such as accounting information and *authorization* information (e.g., Joe User can modify the Enrolled relation, but only read the Faculty relation).

How Catalogs are Stored

A very elegant aspect of a relational DBMS is that the system catalog is itself a collection of relations. For example, we might store information about the attributes of relations in a catalog relation called Attribute_Cat:

Attribute_Cat(*attr_name:* `string`, *rel_name:* `string`,
type: `string`, *position:* `integer`)

Suppose that the database contains two relations:

Students(*sid:* `string`, *name:* `string`, *login:* `string`,
age: `integer`, *gpa:* `real`)
Faculty(*fid:* `string`, *fname:* `string`, *sal:* `real`)

Figure 13.5 shows the tuples in the Attribute_Cat relation that describe the attributes of these two relations. Notice that in addition to the tuples describing Students and Faculty, other tuples (the first four listed) describe the four attributes of the Attribute_Cat relation itself! These other tuples illustrate an important point: the catalog relations describe all the relations in the database, *including* the catalog relations themselves. When information about a relation is needed, it is obtained from the system catalog. Of course, at the implementation level, whenever the DBMS needs to find the schema of a *catalog* relation, the code that retrieves this information must be handled specially. (Otherwise, this code would have to retrieve this information from the catalog relations without, presumably, knowing the schema of the catalog relations!)

attr_name	*rel_name*	*type*	*position*
attr_name	Attribute_cat	string	1
rel_name	Attribute_cat	string	2
type	Attribute_cat	string	3
position	Attribute_cat	integer	4
sid	Students	string	1
name	Students	string	2
login	Students	string	3
age	Students	integer	4
gpa	Students	real	5
fid	Faculty	string	1
fname	Faculty	string	2
sal	Faculty	real	3

Figure 13.5 An Instance of the Attribute_Cat Relation

The fact that the system catalog is also a collection of relations is very useful. For example, catalog relations can be queried just like any other relation, using the query language of the DBMS! Further, all the techniques available for implementing and managing relations apply directly to catalog relations. The choice of catalog relations and their schemas is not unique and is made by the implementor of the DBMS. Real systems vary in their catalog schema design, but the catalog is always implemented as a collection of relations, and it essentially describes all the data stored in the database.[1]

[1]Some systems may store additional information in a non-relational form. For example, a system with a sophisticated query optimizer may maintain histograms or other statistical information about the distribution of values in certain attributes of a relation. We can think of such information, when it is maintained, as a supplement to the catalog relations.

13.3 ALTERNATIVE PLANS: A MOTIVATING EXAMPLE

Consider the example query from Section 13.1. Let us consider the cost of evaluating the plan shown in Figure 13.3. The cost of the join is $1,000 + 1,000 * 500 = 501,000$ page I/Os. The selections and the projection are done on-the-fly and do not incur additional I/Os. Following the cost convention described in Section 12.1.2, we ignore the cost of writing out the final result. The total cost of this plan is therefore 501,000 page I/Os. This plan is admittedly naive; however, it is possible to be even more naive by treating the join as a cross-product followed by a selection!

We now consider several alternative plans for evaluating this query. Each alternative improves on the original plan in a different way and introduces some optimization ideas that are examined in more detail in the rest of this chapter.

13.3.1 Pushing Selections

A join is a relatively expensive operation, and a good heuristic is to reduce the sizes of the relations to be joined as much as possible. One approach is to apply selections early; if a selection operator appears after a join operator, it is worth examining whether the selection can be 'pushed' ahead of the join. As an example, the selection *bid=100* involves only the attributes of Reserves and can be applied to Reserves *before* the join. Similarly, the selection *rating*> 5 involves only attributes of Sailors and can be applied to Sailors before the join. Let us suppose that the selections are performed using a simple file scan, that the result of each selection is written to a temporary relation on disk, and that the temporary relations are then joined using a sort-merge join. The resulting query evaluation plan is shown in Figure 13.6.

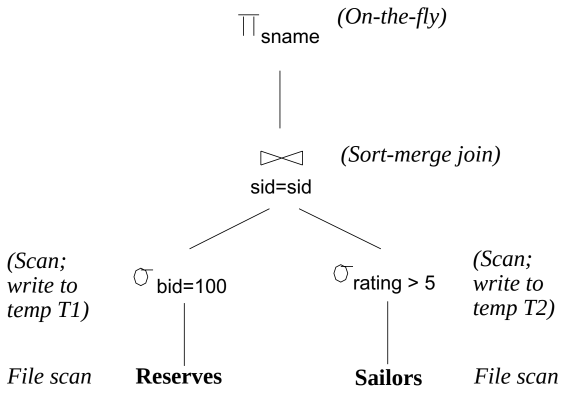

Figure 13.6 A Second Query Evaluation Plan

Let us assume that five buffer pages are available and estimate the cost of this query evaluation plan. (It is likely that more buffer pages will be available in practice. We have chosen a small number simply for illustration purposes in this example.) The cost of applying *bid=100* to Reserves is the cost of scanning Reserves (1,000 pages) plus the cost of writing the result to a temporary relation, say T1. Note that the cost of writing the temporary relation cannot be ignored—we can only ignore the cost of writing out the *final* result of the query, which is the only component of the cost that is the same for all plans, according to the convention described in Section 12.1.2. To estimate the size of T1, we require some additional information. For example, if we assume that the maximum number of reservations of a given boat is one, just one tuple appears in the result. Alternatively, if we know that there are 100 boats, we can assume that reservations are spread out uniformly across all boats and estimate the number of pages in T1 to be 10. For concreteness, let us assume that the number of pages in T1 is indeed 10.

The cost of applying *rating*> 5 to Sailors is the cost of scanning Sailors (500 pages) plus the cost of writing out the result to a temporary relation, say T2. If we assume that ratings are uniformly distributed over the range 1 to 10, we can approximately estimate the size of T2 as 250 pages.

To do a sort-merge join of T1 and T2, let us assume that a straightforward implementation is used in which the two relations are first completely sorted and then merged. Since five buffer pages are available, we can sort T1 (which has 10 pages) in two passes. Two runs of five pages each are produced in the first pass and these are merged in the second pass. In each pass, we read and write 10 pages; thus, the cost of sorting T1 is $2 * 2 * 10 = 40$ page I/Os. We need four passes to sort T2, which has 250 pages. The cost is $2 * 4 * 250 = 2,000$ page I/Os. To merge the sorted versions of T1 and T2, we need to scan these relations, and the cost of this step is $10 + 250 = 260$. The final projection is done on-the-fly, and by convention we ignore the cost of writing the final result.

The total cost of the plan shown in Figure 13.6 is the sum of the cost of the selection $(1,000 + 10 + 500 + 250 = 1,760)$ and the cost of the join $(40 + 2,000 + 260 = 2,300)$, that is, 4,060 page I/Os.

Sort-merge join is one of several join methods. We may be able to reduce the cost of this plan by choosing a different join method. As an alternative, suppose that we used block nested loops join instead of sort-merge join. Using T1 as the outer relation, for every three-page block of T1, we scan all of T2; thus, we scan T2 four times. The cost of the join is therefore the cost of scanning T1 (10) plus the cost of scanning T2 $(4 * 250 = 1,000)$. The cost of the plan is now $1,760 + 1,010 = 2,770$ page I/Os.

A further refinement is to push the projection, just like we pushed the selections past the join. Observe that only the *sid* attribute of T1 and the *sid* and *sname* attributes of T2 are really required. As we scan Reserves and Sailors to do the selections, we could also eliminate unwanted columns. This on-the-fly projection reduces the sizes of the temporary relations T1 and T2. The reduction in the size of T1 is substantial because only an integer field is retained. In fact, T1 will now fit within three buffer pages, and we can perform a block nested loops join with a single scan of T2. The cost of the join step thus drops to under 250 page I/Os, and the total cost of the plan drops to about 2,000 I/Os.

13.3.2 Using Indexes

If indexes are available on the Reserves and Sailors relations, even better query evaluation plans may be available. For example, suppose that we have a clustered static hash index on the *bid* field of Reserves and another hash index on the *sid* field of Sailors. We can then use the query evaluation plan shown in Figure 13.7.

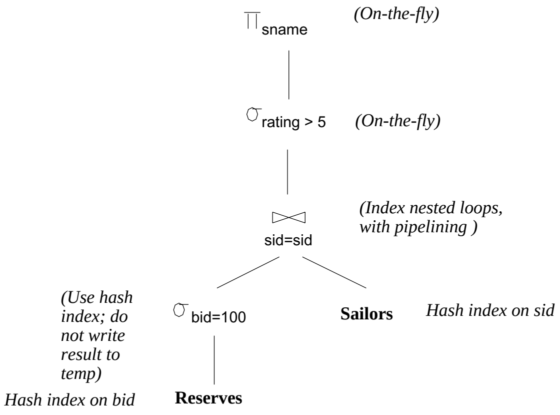

Figure 13.7 A Query Evaluation Plan Using Indexes

The selection *bid*=100 is performed on Reserves by using the hash index on *bid* to retrieve only matching tuples. As before, if we know that 100 boats are available and assume that reservations are spread out uniformly across all boats, we can estimate the number of selected tuples to be $100,000/100 = 1,000$. Since the index on *bid* is clustered, these 1,000 tuples appear consecutively within the same bucket; thus, the cost is 10 page I/Os.

For each selected tuple, we retrieve matching Sailors tuples using the hash index on the *sid* field; selected Reserves tuples are not materialized and the join is pipelined. For each tuple in the result of the join, we perform the selection *rating*>5 and the projection of *sname* on-the-fly. There are several important points to note here:

1. Since the result of the selection on Reserves is not materialized, the optimization of projecting out fields that are not needed subsequently is unnecessary (and is not used in the plan shown in Figure 13.7).

2. The join field *sid* is a key for Sailors. Therefore, at most one Sailors tuple matches a given Reserves tuple. The cost of retrieving this matching tuple depends on whether the directory of the hash index on the *sid* column of Sailors fits in memory and on the presence of overflow pages (if any). However, the cost does *not* depend on whether this index is clustered because there is at most one matching Sailors tuple and requests for Sailors tuples are made in random order by *sid* (because Reserves tuples are retrieved by *bid* and are therefore considered in random order by *sid*). For a hash index, 1.2 page I/Os (on average) is a good estimate of the cost for retrieving a data entry. Assuming that the *sid* hash index on Sailors uses Alternative (1) for data entries, 1.2 I/Os is the cost to retrieve a matching Sailors tuple (and if one of the other two alternatives is used, the cost would be 2.2 I/Os).

3. We have chosen not to push the selection *rating*>5 ahead of the join, and there is an important reason for this decision. If we performed the selection before the join, the selection would involve scanning Sailors, assuming that no index is available on the *rating* field of Sailors. Further, whether or not such an index is available, once we apply such a selection, we do not have an index on the *sid* field of the result of the selection (unless we choose to build such an index solely for the sake of the subsequent join). Thus, pushing selections ahead of joins is a good heuristic, but not always the best strategy. Typically, as in this example, the existence of useful indexes is the reason that a selection is *not* pushed. (Otherwise, selections are pushed.)

Let us estimate the cost of the plan shown in Figure 13.7. The selection of Reserves tuples costs 10 I/Os, as we saw earlier. There are 1,000 such tuples, and for each the cost of finding the matching Sailors tuple is 1.2 I/Os, on average. The cost of this step (the join) is therefore 1,200 I/Os. All remaining selections and projections are performed on-the-fly. The total cost of the plan is 1,210 I/Os.

As noted earlier, this plan does not utilize clustering of the Sailors index. The plan can be further refined if the index on the *sid* field of Sailors is clustered. Suppose we materialize the result of performing the selection *bid*=100 on Reserves and sort this temporary relation. This relation contains 10 pages. Selecting the tuples costs 10 page I/Os (as before), writing out the result to a temporary relation costs another 10 I/Os, and with five buffer pages, sorting this temporary costs $2 * 2 * 10 = 40$ I/Os. (The cost of this step is reduced if we push the projection on *sid*. The *sid* column of materialized Reserves tuples requires only three pages and can be sorted in memory with five buffer pages.) The selected Reserves tuples can now be retrieved in order by *sid*.

If a sailor has reserved the same boat many times, all corresponding Reserves tuples are now retrieved consecutively; the matching Sailors tuple will be found in the buffer pool on all but the first request for it. This improved plan also demonstrates that pipelining is not always the best strategy.

The combination of pushing selections and using indexes that is illustrated by this plan is very powerful. If the selected tuples from the outer relation join with a single inner tuple, the join operation may become trivial, and the performance gains with respect to the naive plan shown in Figure 13.6 are even more dramatic. The following variant of our example query illustrates this situation:

```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
       AND R.bid = 100 AND S.rating > 5
       AND R.day = '8/9/94'
```

A slight variant of the plan shown in Figure 13.7, designed to answer this query, is shown in Figure 13.8. The selection *day='8/9/94'* is applied on-the-fly to the result of the selection *bid=100* on the Reserves relation.

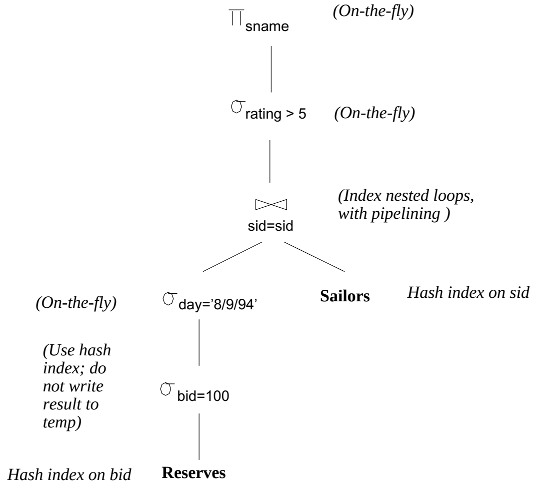

Figure 13.8 A Query Evaluation Plan for the Second Example

Suppose that *bid* and *day* form a key for Reserves. (Note that this assumption differs from the schema presented earlier in this chapter.) Let us estimate the cost of the plan shown in Figure 13.8. The selection *bid=100* costs 10 page I/Os, as before, and the additional selection *day='8/9/94'* is applied on-the-fly, eliminating all but (at most) one Reserves tuple. There is at most one matching Sailors tuple, and this is retrieved in 1.2 I/Os (an average number!). The selection on *rating* and the projection on *sname* are then applied on-the-fly at no additional cost. The total cost of the plan in Figure 13.8 is thus about 11 I/Os. In contrast, if we modify the naive plan in Figure 13.6 to perform the additional selection on *day* together with the selection *bid=100*, the cost remains at 501,000 I/Os.

13.4 POINTS TO REVIEW

- The goal of query optimization is usually to avoid the worst evaluation plans and find a good plan, rather than to find the best plan. To optimize an SQL query, we first express it in relational algebra, consider several query evaluation plans for the algebra expression, and choose the plan with the least estimated cost. A *query evaluation plan* is a tree with relational operators at the intermediate nodes and relations at the leaf nodes. Intermediate nodes are annotated with the algorithm chosen to execute the relational operator and leaf nodes are annotated with the access method used to retrieve tuples from the relation. Results of one operator can be *pipelined* into another operator without *materializing* the intermediate result. If the input tuples to a unary operator are pipelined, this operator is said to be applied *on-the-fly*. Operators have a uniform *iterator* interface with functions *open*, *get_next*, and *close*. **(Section 13.1)**

- A DBMS maintains information (called *metadata*) about the data in a special set of relations called the *catalog* (also called the *system catalog* or *data dictionary*). The system catalog contains information about each relation, index, and view. In addition, it contains statistics about relations and indexes. Since the system catalog itself is stored in a set of relations, we can use the full power of SQL to query it and manipulate it. **(Section 13.2)**

- Alternative plans can differ substantially in their overall cost. One heuristic is to apply selections as early as possible to reduce the size of intermediate relations. Existing indexes can be used as matching access paths for a selection condition. In addition, when considering the choice of a join algorithm the existence of indexes on the inner relation impacts the cost of the join. **(Section 13.3)**

EXERCISES

Exercise 13.1 Briefly answer the following questions.

1. What is the goal of query optimization? Why is it important?
2. Describe the advantages of *pipelining*.
3. Give an example in which pipelining *cannot* be used.
4. Describe the *iterator* interface and explain its advantages.
5. What role do statistics gathered from the database play in query optimization?
6. What information is stored in the system catalogs?
7. What are the benefits of making the system catalogs be relations?
8. What were the important design decisions made in the System R optimizer?

Additional exercises and bibliographic notes can be found at the end of Chapter 14.