12 EVALUATION OF RELATIONAL OPERATORS

Now, *here*, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!

—Lewis Carroll, *Through the Looking Glass*

The relational operators serve as the building blocks for query evaluation. Queries, written in a language such as SQL, are presented to a *query optimizer*, which uses information about how the data is stored (available in the system catalogs) to produce an efficient *execution plan* for evaluating the query. Finding a good execution plan for a query consists of more than just choosing an implementation for each of the relational operators that appear in the query. For example, the order in which operators are applied can influence the cost. Issues in finding a good plan that go beyond implementation of individual operators are discussed in Chapter 13.

This chapter considers the implementation of individual relational operators. Section 12.1 provides an introduction to query processing, highlighting some common themes that recur throughout this chapter, and discusses how tuples are retrieved from relations while evaluating various relational operators. We present implementation alternatives for the selection operator in Sections 12.2 and 12.3. It is instructive to see the variety of alternatives, and the wide variation in performance of these alternatives, for even such a simple operator. In Section 12.4 we consider the other unary operator in relational algebra, namely, projection.

We then discuss the implementation of binary operators, beginning with joins in Section 12.5. Joins are among the most expensive operators in a relational database system, and their implementation has a big impact on performance. After discussing the join operator, we consider implementation of the binary operators cross-product, intersection, union, and set-difference in Section 12.6. We discuss the implementation of grouping and aggregate operators, which are extensions of relational algebra, in Section 12.7. We conclude with a discussion of how buffer management affects operator evaluation costs in Section 12.8.

The discussion of each operator is largely independent of the discussion of other operators. Several alternative implementation techniques are presented for each operator; the reader who wishes to cover this material in less depth can skip some of these alternatives without loss of continuity.

12.1 INTRODUCTION TO QUERY PROCESSING

One virtue of a relational DBMS is that queries are composed of a few basic operators, and the implementation of these operators can (and should!) be carefully optimized for good performance. There are several alternative algorithms for implementing each relational operator, and for most operators there is no universally superior technique. Which algorithm is best depends on several factors, including the sizes of the relations involved, existing indexes and sort orders, the size of the available buffer pool, and the buffer replacement policy.

The algorithms for various relational operators actually have a lot in common. As this chapter will demonstrate, a few simple techniques are used to develop algorithms for each operator:

- **Iteration:** Examine all tuples in input relations iteratively. Sometimes, instead of examining tuples, we can examine index data entries (which are smaller) that contain all necessary fields.
- **Indexing:** If a selection or join condition is specified, use an index to examine just the tuples that satisfy the condition.
- **Partitioning:** By partitioning tuples on a sort key, we can often decompose an operation into a less expensive collection of operations on partitions. *Sorting* and *hashing* are two commonly used partitioning techniques.

12.1.1 Access Paths

All the algorithms discussed in this chapter have to retrieve tuples from one or more input relations. There is typically more than one way to retrieve tuples from a relation because of the availability of indexes and the (possible) presence of a selection condition in the query that restricts the subset of the relation we need. (The selection condition can come from a selection operator or from a join.) The alternative ways to retrieve tuples from a relation are called *access paths*.

An **access path** is either (1) a file scan or (2) an index plus a **matching** selection condition. Intuitively, an index *matches* a selection condition if the index can be used to retrieve just the tuples that satisfy the condition. Consider a simple selection of the form *attr* ***op*** *value*, where ***op*** is one of the comparison operators $<$, $\leq$, $=$, $\neq$, $\geq$, or $>$. An index matches such a selection if the index search key is *attr* and either (1) the index is a tree index or (2) the index is a hash index and ***op*** is equality. We consider when more complex selection conditions match an index in Section 12.3.

The **selectivity** of an access path is the number of pages retrieved (index pages plus data pages) if we use this access path to retrieve all desired tuples. If a relation contains an index that matches a given selection, there are at least two access paths, namely, the index and a scan of the data file. The **most selective** access path is the one that retrieves the fewest pages; using the most selective access path minimizes the cost of data retrieval.

12.1.2 Preliminaries: Examples and Cost Calculations

We will present 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`)

This schema is a variant of the one that we used in Chapter 5; we have added a string field *rname* to Reserves. Intuitively, this field is the name of the person who has made the reservation (and may be different from the name of the sailor *sid* for whom the reservation was made; a reservation may be made by a person who is not a sailor on behalf of a sailor). The addition of this field gives us more flexibility in choosing illustrative examples. 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.

Two points must be kept in mind to understand our discussion of costs:

- As discussed in Chapter 8, we consider only I/O costs and measure I/O cost in terms of the number of page I/Os. We also use big-O notation to express the complexity of an algorithm in terms of an input parameter and assume that the reader is familiar with this notation. For example, the cost of a file scan is $O(M)$, where M is the size of the file.

- We discuss several alternate algorithms for each operation. Since each alternative incurs the same cost in writing out the result, should this be necessary, we will uniformly ignore this cost in comparing alternatives.

12.2 THE SELECTION OPERATION

In this section we describe various algorithms to evaluate the selection operator. To motivate the discussion, consider the selection query shown in Figure 12.1, which has the selection condition *rname='Joe'*.

We can evaluate this query by scanning the entire relation, checking the condition on each tuple, and adding the tuple to the result if the condition is satisfied. The cost of this approach is 1,000 I/Os, since Reserves contains 1,000 pages. If there are only a

```
SELECT *
FROM   Reserves R
WHERE  R.rname='Joe'
```

Figure 12.1 Simple Selection Query

few tuples with *rname='Joe'*, this approach is expensive because it does not utilize the selection to reduce the number of tuples retrieved in any way. How can we improve on this approach? The key is to utilize information in the selection condition and to use an index if a suitable index is available. For example, a B+ tree index on *rname* could be used to answer this query considerably faster, but an index on *bid* would not be useful.

In the rest of this section we consider various situations with respect to the file organization used for the relation and the availability of indexes and discuss appropriate algorithms for the selection operation. We discuss only simple selection operations of the form $\sigma_{R.attr\ \mathbf{op}\ value}(R)$ until Section 12.3, where we consider general selections. In terms of the general techniques listed in Section 12.1, the algorithms for selection use either iteration or indexing.

12.2.1 No Index, Unsorted Data

Given a selection of the form $\sigma_{R.attr\ \mathbf{op}\ value}(R)$, if there is no index on *R.attr* and R is not sorted on *R.attr*, we have to scan the entire relation. Thus, the most selective access path is a file scan. For each tuple, we must test the condition *R.attr* **op** *value* and add the tuple to the result if the condition is satisfied.

The cost of this approach is M I/Os, where M is the number of pages in R. In the example selection from Reserves (Figure 12.1), the cost is 1,000 I/Os.

12.2.2 No Index, Sorted Data

Given a selection of the form $\sigma_{R.attr\ \mathbf{op}\ value}(R)$, if there is no index on *R.attr*, but R is physically sorted on *R.attr*, we can utilize the sort order by doing a binary search to locate the first tuple that satisfies the selection condition. Further, we can then retrieve all tuples that satisfy the selection condition by starting at this location and scanning R until the selection condition is no longer satisfied. The access method in this case is a sorted-file scan with selection condition $\sigma_{R.attr\ \mathbf{op}\ value}(R)$.

For example, suppose that the selection condition is $R.attr1 > 5$, and that R is sorted on $attr1$ in ascending order. After a binary search to locate the position in R corresponding to 5, we simply scan all remaining records.

The cost of the binary search is O(log_2M). In addition, we have the cost of the scan to retrieve qualifying tuples. The cost of the scan depends on the number of such tuples and can vary from zero to M. In our selection from Reserves (Figure 12.1), the cost of the binary search is $log_2 1,000 \approx 10$ I/Os.

In practice, it is unlikely that a relation will be kept sorted if the DBMS supports Alternative (1) for index data entries, that is, allows data records to be stored as index data entries. If the ordering of data records is important, a better way to maintain it is through a B+ tree index that uses Alternative (1).

12.2.3 B+ Tree Index

If a clustered B+ tree index is available on *R.attr*, the best strategy for selection conditions $\sigma_{R.attr\ \textbf{op}\ value}(R)$ in which ***op*** is not equality is to use the index. This strategy is also a good access path for equality selections, although a hash index on *R.attr* would be a little better. If the B+ tree index is not clustered, the cost of using the index depends on the number of tuples that satisfy the selection, as discussed below.

We can use the index as follows: We search the tree to find the first index entry that points to a qualifying tuple of R. Then we scan the leaf pages of the index to retrieve all entries in which the key value satisfies the selection condition. For each of these entries, we retrieve the corresponding tuple of R. (For concreteness in this discussion, we will assume that data entries use Alternatives (2) or (3); if Alternative (1) is used, the data entry contains the actual tuple and there is no additional cost—beyond the cost of retrieving data entries—for retrieving tuples.)

The cost of identifying the starting leaf page for the scan is typically two or three I/Os. The cost of scanning the leaf level page for qualifying data entries depends on the number of such entries. The cost of retrieving qualifying tuples from R depends on two factors:

- The number of qualifying tuples.
- Whether the index is clustered. (Clustered and unclustered B+ tree indexes are illustrated in Figures 11.11 and 11.12. The figures should give the reader a feel for the impact of clustering, regardless of the type of index involved.)

If the index is clustered, the cost of retrieving qualifying tuples is probably just one page I/O (since it is likely that all such tuples are contained in a single page). If the index is not clustered, each index entry could point to a qualifying tuple on a different page, and the cost of retrieving qualifying tuples in a straightforward way could be one page I/O per qualifying tuple (unless we get lucky with buffering). We can significantly reduce the number of I/Os to retrieve qualifying tuples from R by first sorting the rids (in the index's data entries) by their *page-id* component. This sort ensures that when we bring in a page of R, all qualifying tuples on this page are retrieved one after the other. The cost of retrieving qualifying tuples is now the number of pages of R that contain qualifying tuples.

Consider a selection of the form *rname* < *'C%'* on the Reserves relation. Assuming that names are uniformly distributed with respect to the initial letter, for simplicity, we estimate that roughly 10 percent of Reserves tuples are in the result. This is a total of 10,000 tuples, or 100 pages. If we have a clustered B+ tree index on the *rname* field of Reserves, we can retrieve the qualifying tuples with 100 I/Os (plus a few I/Os to traverse from the root to the appropriate leaf page to start the scan). However, if the index is unclustered, we could have up to 10,000 I/Os in the worst case, since each tuple could cause us to read a page. If we sort the rids of Reserves tuples by the page number and then retrieve pages of Reserves, we will avoid retrieving the same page multiple times; nonetheless, the tuples to be retrieved are likely to be scattered across many more than 100 pages. Therefore, the use of an unclustered index for a range selection could be expensive; it might be cheaper to simply scan the entire relation (which is 1,000 pages in our example).

12.2.4 Hash Index, Equality Selection

If a hash index is available on *R.attr* and ***op*** is equality, the best way to implement the selection $\sigma_{R.attr\ \textbf{\textit{op}}\ value}(R)$ is obviously to use the index to retrieve qualifying tuples.

The cost includes a few (typically one or two) I/Os to retrieve the appropriate bucket page in the index, plus the cost of retrieving qualifying tuples from R. The cost of retrieving qualifying tuples from R depends on the number of such tuples and on whether the index is clustered. Since ***op*** is equality, there is exactly one qualifying tuple if *R.attr* is a (candidate) key for the relation. Otherwise, we could have several tuples with the same value in this attribute.

Consider the selection in Figure 12.1. Suppose that there is an unclustered hash index on the *rname* attribute, that we have 10 buffer pages, and that there are 100 reservations made by people named Joe. The cost of retrieving the index page containing the rids of such reservations is one or two I/Os. The cost of retrieving the 100 Reserves tuples can vary between 1 and 100, depending on how these records are distributed across pages of Reserves and the order in which we retrieve these records. If these 100 records are contained in, say, some five pages of Reserves, we have just five additional I/Os if we sort the rids by their page component. Otherwise, it is possible that we bring in one of these five pages, then look at some of the other pages, and find that the first page has been paged out when we need it again. (Remember that several users and DBMS operations share the buffer pool.) This situation could cause us to retrieve the same page several times.

12.3 GENERAL SELECTION CONDITIONS *

In our discussion of the selection operation thus far, we have considered selection conditions of the form $\sigma_{R.attr\ \mathbf{op}\ value}(R)$. In general a selection condition is a boolean combination (i.e., an expression using the logical connectives $\wedge$ and $\vee$) of **terms** that have the form *attribute* **op** *constant* or *attribute1* **op** *attribute2*. For example, if the `WHERE` clause in the query shown in Figure 12.1 contained the condition *R.rname='Joe'* `AND` *R.bid=r*, the equivalent algebra expression would be $\sigma_{R.rname='Joe' \wedge R.bid=r}(R)$.

In Section 12.3.1 we introduce a standard form for general selection conditions and define when an index matches such a condition. We consider algorithms for applying selection conditions without disjunction in Section 12.3.2 and then discuss conditions with disjunction in Section 12.3.3.

12.3.1 CNF and Index Matching

To process a selection operation with a general selection condition, we first express the condition in **conjunctive normal form (CNF)**, that is, as a collection of *conjuncts* that are connected through the use of the $\wedge$ operator. Each **conjunct** consists of one or more *terms* (of the form described above) connected by $\vee$.[1] Conjuncts that contain $\vee$ are said to be **disjunctive**, or to **contain disjunction**.

As an example, suppose that we have a selection on Reserves with the condition *(day* $<$ *8/9/94* $\wedge$ *rname = 'Joe')* $\vee$ *bid=5* $\vee$ *sid=3*. We can rewrite this in conjunctive normal form as *(day* $<$ *8/9/94* $\vee$ *bid=5* $\vee$ *sid=3)* $\wedge$ *(rname = 'Joe'* $\vee$ *bid=5* $\vee$ *sid=3)*.

We now turn to the issue of when a general selection condition, represented in CNF, matches an index. The following examples provide some intuition:

- If we have a hash index on the search key ⟨*rname,bid,sid*⟩, we can use the index to retrieve just the tuples that satisfy the condition *rname='Joe'* $\wedge$ *bid=5* $\wedge$ *sid=3*. The index matches the entire condition *rname='Joe'* $\wedge$ *bid=5* $\wedge$ *sid=3*. On the other hand, if the selection condition is *rname='Joe'* $\wedge$ *bid=5*, or some condition on *date*, this index does not match. That is, it cannot be used to retrieve just the tuples that satisfy these conditions.

  In contrast, if the index were a B+ tree, it would match both *rname='Joe'* $\wedge$ *bid=5* $\wedge$ *sid=3* and *rname='Joe'* $\wedge$ *bid=5*. However, it would not match *bid=5* $\wedge$ *sid=3* (since tuples are sorted primarily by *rname*).

- If we have an index (hash or tree) on the search key ⟨*bid,sid*⟩ and the selection condition *rname='Joe'* $\wedge$ *bid=5* $\wedge$ *sid=3*, we can use the index to retrieve tuples

[1]Every selection condition can be expressed in CNF. We refer the reader to any standard text on mathematical logic for the details.

that satisfy *bid=5* $\wedge$ *sid=3*, but the additional condition on *rname* must then be applied to each retrieved tuple and will eliminate some of the retrieved tuples from the result. In this case the index only matches a part of the selection condition (the part *bid=5* $\wedge$ *sid=3*).

- If we have an index on the search key $\langle$*bid, sid*$\rangle$ and we also have a B+ tree index on *day*, the selection condition *day* < *8/9/94* $\wedge$ *bid=5* $\wedge$ *sid=3* offers us a choice. Both indexes match (part of) the selection condition, and we can use either to retrieve Reserves tuples. Whichever index we use, the conjuncts in the selection condition that are not matched by the index (e.g., *bid=5* $\wedge$ *sid=3* if we use the B+ tree index on *day*) must be checked for each retrieved tuple.

Generalizing the intuition behind these examples, the following rules define when an index **matches** a selection condition that is in CNF:

- A hash index matches a selection condition containing no disjunctions if there is a term of the form *attribute=value* for each attribute in the index's search key.
- A tree index matches a selection condition containing no disjunctions if there is a term of the form *attribute* ***op*** *value* for each attribute in a *prefix* of the index's search key. ($\langle a \rangle$ and $\langle a, b \rangle$ are prefixes of key $\langle a, b, c \rangle$, but $\langle a, c \rangle$ and $\langle b, c \rangle$ are not.) Note that ***op*** can be any comparison; it is not restricted to be equality as it is for matching selections on a hash index.

The above definition does not address when an index matches a selection with disjunctions; we discuss this briefly in Section 12.3.3. As we observed in the examples, an index could match some subset of the conjuncts in a selection condition (in CNF), even though it does not match the entire condition. We will refer to the conjuncts that the index matches as the **primary conjuncts** in the selection.

The selectivity of an access path obviously depends on the selectivity of the primary conjuncts in the selection condition (with respect to the index involved).

12.3.2 Evaluating Selections without Disjunction

When the selection does not contain disjunction, that is, it is a conjunction of terms, we have two evaluation options to consider:

- We can retrieve tuples using a file scan or a single index that matches some conjuncts (and which we estimate to be the most selective access path) and apply all nonprimary conjuncts in the selection to each retrieved tuple. This approach is very similar to how we use indexes for simple selection conditions, and we will not discuss it further. (We emphasize that the number of tuples retrieved depends on the selectivity of the primary conjuncts in the selection, and the remaining conjuncts only serve to reduce the cardinality of the result of the selection.)

Intersecting rid sets: Oracle 8 uses several techniques to do rid set intersection for selections with `AND`. One is to `AND` bitmaps. Another is to do a hash join of indexes. For example, given $sal < 5 \wedge price > 30$ and indexes on *sal* and *price*, we can join the indexes on the rid column, considering only entries that satisfy the given selection conditions. Microsoft SQL Server implements rid set intersection through index joins. IBM DB2 implements intersection of rid sets using Bloom filters (which are discussed in Section 21.9.2). Sybase ASE does not do rid set intersection for `AND` selections; Sybase ASIQ does it using bitmap operations. Informix also does rid set intersection.

- We can try to utilize several indexes. We examine this approach in the rest of this section.

If several indexes containing data entries with rids (i.e., Alternatives (2) or (3)) match conjuncts in the selection, we can use these indexes to compute sets of rids of candidate tuples. We can then intersect these sets of rids, typically by first sorting them, and then retrieve those records whose rids are in the intersection. If additional conjuncts are present in the selection, we can then apply these conjuncts to discard some of the candidate tuples from the result.

As an example, given the condition $day < 8/9/94 \wedge bid{=}5 \wedge sid{=}3$, we can retrieve the rids of records that meet the condition $day < 8/9/94$ by using a B+ tree index on *day*, retrieve the rids of records that meet the condition *sid=3* by using a hash index on *sid*, and intersect these two sets of rids. (If we sort these sets by the page id component to do the intersection, a side benefit is that the rids in the intersection are obtained in sorted order by the pages that contain the corresponding tuples, which ensures that we do not fetch the same page twice while retrieving tuples using their rids.) We can now retrieve the necessary pages of Reserves to retrieve tuples, and check *bid=5* to obtain tuples that meet the condition $day < 8/9/94 \wedge bid{=}5 \wedge sid{=}3$.

12.3.3 Selections with Disjunction

Now let us consider the case that one of the conjuncts in the selection condition is a *disjunction of terms*. If even one of these terms requires a file scan because suitable indexes or sort orders are unavailable, testing this conjunct by itself (i.e., without taking advantage of other conjuncts) requires a file scan. For example, suppose that the only available indexes are a hash index on *rname* and a hash index on *sid*, and that the selection condition contains just the (disjunctive) conjunct $(day < 8/9/94 \vee rname{=}\text{'Joe'})$. We can retrieve tuples satisfying the condition *rname='Joe'* by using the index on *rname*. However, $day < 8/9/94$ requires a file scan. So we might as well

Disjunctions: Microsoft SQL Server considers the use of unions and bitmaps for dealing with disjunctive conditions. Oracle 8 considers four ways to handle disjunctive conditions: (1) Convert the query into a union of queries without `OR`. (2) If the conditions involve the same attribute, e.g., $sal < 5 \vee sal > 30$, use a nested query with an `IN` list and an index on the attribute to retrieve tuples matching a value in the list. (3) Use bitmap operations, e.g., evaluate $sal < 5 \vee sal > 30$ by generating bitmaps for the values 5 and 30 and `OR` the bitmaps to find the tuples that satisfy one of the conditions. (We discuss bitmaps in Chapter 23.) (4) Simply apply the disjunctive condition as a filter on the set of retrieved tuples. Sybase ASE considers the use of unions for dealing with disjunctive queries and Sybase ASIQ uses bitmap operations.

do a file scan and check the condition *rname='Joe'* for each retrieved tuple. Thus, the most selective access path in this example is a file scan.

On the other hand, if the selection condition is *(day < 8/9/94 ∨ rname='Joe') ∧ sid=3*, the index on *sid* matches the conjunct *sid=3*. We can use this index to find qualifying tuples and apply *day < 8/9/94 ∨ rname='Joe'* to just these tuples. The best access path in this example is the index on *sid* with the primary conjunct *sid=3*.

Finally, if every term in a disjunction has a matching index, we can retrieve candidate tuples using the indexes and then take the union. For example, if the selection condition is the conjunct *(day < 8/9/94 ∨ rname='Joe')* and we have B+ tree indexes on *day* and *rname*, we can retrieve all tuples such that *day < 8/9/94* using the index on *day*, retrieve all tuples such that *rname='Joe'* using the index on *rname*, and then take the union of the retrieved tuples. If all the matching indexes use Alternative (2) or (3) for data entries, a better approach is to take the union of rids and sort them before retrieving the qualifying data records. Thus, in the example, we can find rids of tuples such that *day < 8/9/94* using the index on *day*, find rids of tuples such that *rname='Joe'* using the index on *rname*, take the union of these sets of rids and sort them by page number, and then retrieve the actual tuples from Reserves. This strategy can be thought of as a (complex) access path that matches the selection condition *(day < 8/9/94 ∨ rname='Joe')*.

Most current systems do not handle selection conditions with disjunction efficiently, and concentrate on optimizing selections without disjunction.

12.4 THE PROJECTION OPERATION

Consider the query shown in Figure 12.2. The optimizer translates this query into the relational algebra expression $\pi_{sid,bid}Reserves$. In general the projection operator is of the form $\pi_{attr1,attr2,\ldots,attrm}(R)$.

```
SELECT DISTINCT R.sid, R.bid
FROM   Reserves R
```

Figure 12.2 Simple Projection Query

To implement projection, we have to do the following:

1. Remove unwanted attributes (i.e., those not specified in the projection).
2. Eliminate any duplicate tuples that are produced.

The second step is the difficult one. There are two basic algorithms, one based on sorting and one based on hashing. In terms of the general techniques listed in Section 12.1, both algorithms are instances of partitioning. While the technique of using an index to identify a subset of useful tuples is not applicable for projection, the sorting or hashing algorithms can be applied to data entries in an index, instead of to data records, under certain conditions described in Section 12.4.4.

12.4.1 Projection Based on Sorting

The algorithm based on sorting has the following steps (at least conceptually):

1. Scan R and produce a set of tuples that contain only the desired attributes.
2. Sort this set of tuples using the combination of all its attributes as the key for sorting.
3. Scan the sorted result, comparing adjacent tuples, and discard duplicates.

If we use temporary relations at each step, the first step costs M I/Os to scan R, where M is the number of pages of R, and T I/Os to write the temporary relation, where T is the number of pages of the temporary; T is O(M). (The exact value of T depends on the number of fields that are retained and the sizes of these fields.) The second step costs O($TlogT$) (which is also O($MlogM$), of course). The final step costs T. The total cost is O($MlogM$). The first and third steps are straightforward and relatively inexpensive. (As noted in the chapter on sorting, the cost of sorting grows linearly with dataset size in practice, given typical dataset sizes and main memory sizes.)

Consider the projection on Reserves shown in Figure 12.2. We can scan Reserves at a cost of 1,000 I/Os. If we assume that each tuple in the temporary relation created in the first step is 10 bytes long, the cost of writing this temporary relation is 250 I/Os. Suppose that we have 20 buffer pages. We can sort the temporary relation in two passes at a cost of $2 * 2 * 250 = 1,000$ I/Os. The scan required in the third step costs an additional 250 I/Os. The total cost is 2,500 I/Os.

This approach can be improved on by modifying the sorting algorithm to do projection with duplicate elimination. Recall the structure of the external sorting algorithm that we presented in Chapter 11. The very first pass (Pass 0) involves a scan of the records that are to be sorted to produce the initial set of (internally) sorted runs. Subsequently one or more passes merge runs. Two important modifications to the sorting algorithm adapt it for projection:

- We can project out unwanted attributes during the first pass (Pass 0) of sorting. If B buffer pages are available, we can read in B pages of R and write out $(T/M) * B$ *internally sorted* pages of the temporary relation. In fact, with a more aggressive implementation, we can write out approximately $2 * B$ internally sorted pages of the temporary relation on average. (The idea is similar to the refinement of external sorting that is discussed in Section 11.2.1.)
- We can eliminate duplicates during the merging passes. In fact, this modification will reduce the cost of the merging passes since fewer tuples are written out in each pass. (Most of the duplicates will be eliminated in the very first merging pass.)

Let us consider our example again. In the first pass we scan Reserves, at a cost of 1,000 I/Os and write out 250 pages. With 20 buffer pages, the 250 pages are written out as seven internally sorted runs, each (except the last) about 40 pages long. In the second pass we read the runs, at a cost of 250 I/Os, and merge them. The total cost is 1,500 I/Os, which is much lower than the cost of the first approach used to implement projection.

12.4.2 Projection Based on Hashing *

If we have a fairly large number (say, B) of buffer pages relative to the number of pages of R, a hash-based approach is worth considering. There are two phases: *partitioning* and *duplicate elimination.*

In the *partitioning* phase we have one *input* buffer page and $B - 1$ *output* buffer pages. The relation R is read into the input buffer page, one page at a time. The input page is processed as follows: For each tuple, we project out the unwanted attributes and then apply a hash function h to the combination of all remaining attributes. The function h is chosen so that tuples are distributed uniformly to one of $B - 1$ partitions; there is one output page per partition. After the projection the tuple is written to the output buffer page that it is hashed to by h.

At the end of the partitioning phase, we have $B - 1$ partitions, each of which contains a collection of tuples that share a common hash value (computed by applying h to all fields), and have only the desired fields. The partitioning phase is illustrated in Figure 12.3.

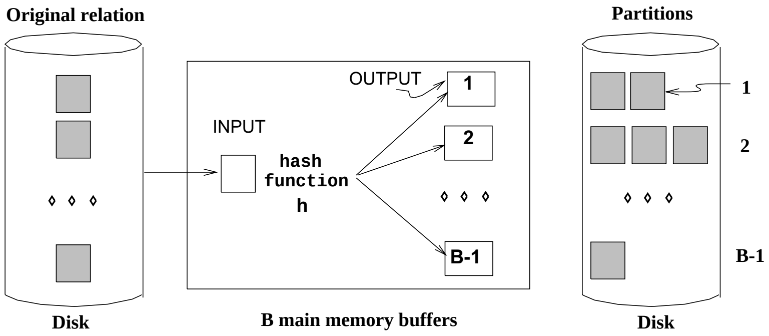

Figure 12.3 Partitioning Phase of Hash-Based Projection

Two tuples that belong to different partitions are guaranteed not to be duplicates because they have different hash values. Thus, if two tuples are duplicates, they are in the same partition. In the *duplicate elimination* phase, we read in the $B - 1$ partitions one at a time to eliminate duplicates. The basic idea is to build an in-memory hash table as we process tuples in order to detect duplicates.

For each partition produced in the first phase:

1. Read in the partition one page at a time. Hash each tuple by applying hash function $h2$ ($\neq h$!) to the combination of all fields and then insert it into an in-memory hash table. If a new tuple hashes to the same value as some existing tuple, compare the two to check whether the new tuple is a duplicate. Discard duplicates as they are detected.

2. After the entire partition has been read in, write the tuples in the hash table (which is free of duplicates) to the result file. Then clear the in-memory hash table to prepare for the next partition.

Note that $h2$ is intended to distribute the tuples in a partition across many buckets, in order to minimize *collisions* (two tuples having the same $h2$ values). Since all tuples in a given partition have the same h value, $h2$ cannot be the same as h!

This hash-based projection strategy will not work well if the size of the hash table for a partition (produced in the partitioning phase) is greater than the number of available buffer pages B. One way to handle this *partition overflow* problem is to recursively apply the hash-based projection technique to eliminate the duplicates in each partition that overflows. That is, we divide an overflowing partition into subpartitions, then read each subpartition into memory to eliminate duplicates.

If we assume that h distributes the tuples with perfect uniformity and that the number of pages of tuples *after* the projection (but before duplicate elimination) is T, each partition contains $\frac{T}{B-1}$ pages. (Note that the number of partitions is $B-1$ because one of the buffer pages is used to read in the relation during the partitioning phase.) The size of a partition is therefore $\frac{T}{B-1}$, and the size of a hash table for a partition is $\frac{T}{B-1} * f$; where f is a *fudge factor* used to capture the (small) increase in size between the partition and a hash table for the partition. The number of buffer pages B must be greater than the partition size $\frac{T}{B-1} * f$, in order to avoid partition overflow. This observation implies that we require approximately $B > \sqrt{f * T}$ buffer pages.

Now let us consider the cost of hash-based projection. In the partitioning phase, we read R, at a cost of M I/Os. We also write out the projected tuples, a total of T pages, where T is some fraction of M, depending on the fields that are projected out. The cost of this phase is therefore $M + T$ I/Os; the cost of hashing is a CPU cost, and we do not take it into account. In the duplicate elimination phase, we have to read in every partition. The total number of pages in all partitions is T. We also write out the in-memory hash table for each partition after duplicate elimination; this hash table is part of the result of the projection, and we ignore the cost of writing out result tuples, as usual. Thus, the total cost of both phases is $M + 2T$. In our projection on Reserves (Figure 12.2), this cost is $1,000 + 2 * 250 = 1,500$ I/Os.

12.4.3 Sorting versus Hashing for Projections *

The sorting-based approach is superior to hashing if we have many duplicates or if the distribution of (hash) values is very nonuniform. In this case, some partitions could be much larger than average, and a hash table for such a partition would not fit in memory during the duplicate elimination phase. Also, a useful side effect of using the sorting-based approach is that the result is sorted. Further, since external sorting is required for a variety of reasons, most database systems have a sorting utility, which can be used to implement projection relatively easily. For these reasons, sorting is the standard approach for projection. And perhaps due to a simplistic use of the sorting utility, unwanted attribute removal and duplicate elimination are separate steps in many systems (i.e., the basic sorting algorithm is often used without the refinements that we outlined).

We observe that if we have $B > \sqrt{T}$ buffer pages, where T is the size of the projected relation before duplicate elimination, both approaches have the same I/O cost. Sorting takes two passes. In the first pass we read M pages of the original relation and write

Projection in commercial systems: Informix uses hashing. IBM DB2, Oracle 8, and Sybase ASE use sorting. Microsoft SQL Server and Sybase ASIQ implement both hash-based and sort-based algorithms.

out T pages. In the second pass we read the T pages and output the result of the projection. Using hashing, in the partitioning phase we read M pages and write T pages' worth of partitions. In the second phase, we read T pages and output the result of the projection. Thus, considerations such as CPU costs, desirability of sorted order in the result, and skew in the distribution of values drive the choice of projection method.

12.4.4 Use of Indexes for Projections *

Neither the hashing nor the sorting approach utilizes any existing indexes. An existing index is useful if the key includes all the attributes that we wish to retain in the projection. In this case, we can simply retrieve the key values from the index—without ever accessing the actual relation—and apply our projection techniques to this (much smaller) set of pages. This technique is called an **index-only scan**. If we have an ordered (i.e., a tree) index whose search key includes the wanted attributes as a *prefix*, we can do even better: Just retrieve the data entries in order, discarding unwanted fields, and compare adjacent entries to check for duplicates. The index-only scan technique is discussed further in Section 14.4.1.

12.5 THE JOIN OPERATION

Consider the following query:

```
SELECT *
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
```

This query can be expressed in relational algebra using the join operation: $R \bowtie S$. The *join* operation is one of the most useful operations in relational algebra and is the primary means of combining information from two or more relations.

Although a join can be defined as a cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross-products. Further, the result of a cross-product is typically much larger than the result of a join, so it is very important to recognize joins and implement them without materializing the underlying cross-product. Joins have therefore received a lot of attention.

Joins in commercial systems: Sybase ASE supports index nested loop and sort-merge join. Sybase ASIQ supports page-oriented nested loop, index nested loop, simple hash, and sort merge join, in addition to join indexes (which we discuss in Chapter 23). Oracle 8 supports page-oriented nested loops join, sort-merge join, and a variant of hybrid hash join. IBM DB2 supports block nested loop, sort-merge, and hybrid hash join. Microsoft SQL Server supports block nested loops, index nested loops, sort-merge, hash join, and a technique called *hash teams*. Informix supports block nested loops, index nested loops, and hybrid hash join.

We will consider several alternative techniques for implementing joins. We begin by discussing two algorithms (simple nested loops and block nested loops) that essentially enumerate all tuples in the cross-product and discard tuples that do not meet the join conditions. These algorithms are instances of the simple iteration technique mentioned in Section 12.1.

The remaining join algorithms avoid enumerating the cross-product. They are instances of the indexing and partitioning techniques mentioned in Section 12.1. Intuitively, if the join condition consists of equalities, tuples in the two relations can be thought of as belonging to *partitions* such that only tuples in the same partition can join with each other; the tuples in a partition contain the same values in the join columns. Index nested loops join scans one of the relations and, for each tuple in it, uses an index on the (join columns of the) second relation to locate tuples in the same partition. Thus, only a subset of the second relation is compared with a given tuple of the first relation, and the entire cross-product is not enumerated. The last two algorithms (sort-merge join and hash join) also take advantage of join conditions to partition tuples in the relations to be joined and compare only tuples in the same partition while computing the join, but they do not rely on a pre-existing index. Instead, they either sort or hash the relations to be joined to achieve the partitioning.

We discuss the join of two relations R and S, with the join condition $R_i = S_j$, using positional notation. (If we have more complex join conditions, the basic idea behind each algorithm remains essentially the same. We discuss the details in Section 12.5.4.) We assume that there are M pages in R with p_R tuples per page, and N pages in S with p_S tuples per page. We will use R and S in our presentation of the algorithms, and the Reserves and Sailors relations for specific examples.

12.5.1 Nested Loops Join

The simplest join algorithm is a tuple-at-a-time nested loops evaluation.

```
foreach tuple $r \in R$ do
    foreach tuple $s \in S$ do
        if $r_i$==$s_j$ then add $\langle r, s \rangle$ to result
```

Figure 12.4 Simple Nested Loops Join

We scan the *outer* relation R, and for each tuple $r \in R$, we scan the entire *inner* relation S. The cost of scanning R is M I/Os. We scan S a total of $p_R * M$ times, and each scan costs N I/Os. Thus, the total cost is $M + p_R * M * N$.

Suppose that we choose R to be Reserves and S to be Sailors. The value of M is then 1,000, p_R is 100, and N is 500. The cost of simple nested loops join is $1,000 + 100 * 1,000 * 500$ page I/Os (plus the cost of writing out the result; we remind the reader again that we will uniformly ignore this component of the cost). The cost is staggering: $1,000 + (5 * 10^7)$ I/Os. Note that each I/O costs about 10ms on current hardware, which means that this join will take about 140 hours!

A simple refinement is to do this join *page-at-a-time*: For each page of R, we can retrieve each page of S and write out tuples $\langle r, s \rangle$ for all qualifying tuples $r \in$ *R-page* and $s \in$ *S-page*. This way, the cost is M to scan R, as before. However, S is scanned only M times, and so the total cost is $M + M * N$. Thus, the page-at-a-time refinement gives us an improvement of a factor of p_R. In the example join of the Reserves and Sailors relations, the cost is reduced to $1,000 + 1,000 * 500 = 501,000$ I/Os and would take about 1.4 hours. This dramatic improvement underscores the importance of page-oriented operations for minimizing disk I/O.

From these cost formulas a straightforward observation is that we should choose the outer relation R to be the smaller of the two relations ($R \bowtie B = B \bowtie R$, as long as we keep track of field names). This choice does not change the costs significantly, however. If we choose the smaller relation, Sailors, as the outer relation, the cost of the page-at-a-time algorithm is $500 + 500 * 1,000 = 500,500$ I/Os, which is only marginally better than the cost of page-oriented simple nested loops join with Reserves as the outer relation.

Block Nested Loops Join

The simple nested loops join algorithm does not effectively utilize buffer pages. Suppose that we have enough memory to hold the smaller relation, say R, with at least two extra buffer pages left over. We can read in the smaller relation and use one of the extra buffer pages to scan the larger relation S. For each tuple $s \in S$, we check R and output a tuple $\langle r, s \rangle$ for qualifying tuples s (i.e., $r_i = s_j$). The second extra buffer page is used as an output buffer. Each relation is scanned just once, for a total I/O cost of $M + N$, which is optimal.

If enough memory is available, an important refinement is to build an in-memory *hash table* for the smaller relation R. The I/O cost is still $M + N$, but the CPU cost is typically much lower with the hash table refinement.

What if we do not have enough memory to hold the entire smaller relation? We can generalize the preceding idea by breaking the relation R into *blocks* that can fit into the available buffer pages and scanning all of S for each block of R. R is the *outer* relation, since it is scanned only once, and S is the *inner* relation, since it is scanned multiple times. If we have B buffer pages, we can read in $B - 2$ pages of the outer relation R and scan the inner relation S using one of the two remaining pages. We can write out tuples $\langle r, s\rangle$, where $r \in$ *R-block* and $s \in$ *S-page* and $r_i = s_j$, using the last buffer page for output.

An efficient way to find **matching pairs** of tuples (i.e., tuples satisfying the join condition $r_i = s_j$) is to build a main-memory hash table for the block of R. Because a hash table for a set of tuples takes a little more space than just the tuples themselves, building a hash table involves a trade-off: the effective block size of R, in terms of the number of tuples per block, is reduced. Building a hash table is well worth the effort. The block nested loops algorithm is described in Figure 12.5. Buffer usage in this algorithm is illustrated in Figure 12.6.

```
foreach block of $B - 2$ pages of $R$ do
    foreach page of $S$ do {
        for all matching in-memory tuples $r \in$ R-block and $s \in$ S-page,
        add $\langle r, s\rangle$ to result
    }
```

Figure 12.5 Block Nested Loops Join

The cost of this strategy is M I/Os for reading in R (which is scanned only once). S is scanned a total of $\lceil \frac{M}{B-2} \rceil$ times—ignoring the extra space required per page due to the in-memory hash table—and each scan costs N I/Os. The total cost is thus $M + N * \lceil \frac{M}{B-2} \rceil$.

Consider the join of the Reserves and Sailors relations. Let us choose Reserves to be the outer relation R and assume that we have enough buffers to hold an in-memory hash table for 100 pages of Reserves (with at least two additional buffers, of course). We have to scan Reserves, at a cost of 1,000 I/Os. For each 100-page block of Reserves, we have to scan Sailors. Thus we perform 10 scans of Sailors, each costing 500 I/Os. The total cost is $1,000 + 10 * 500 = 6,000$ I/Os. If we had only enough buffers to hold

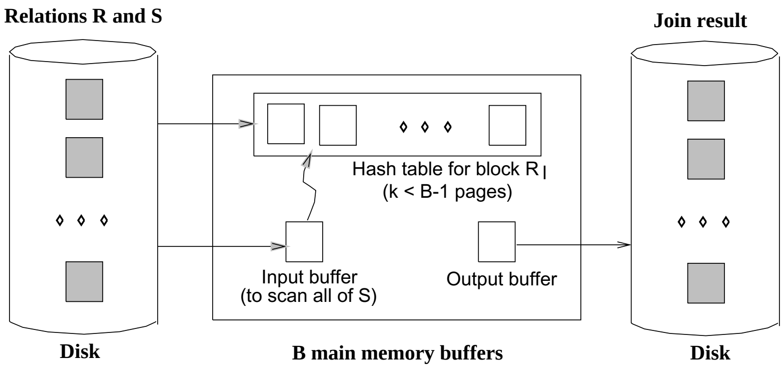

Figure 12.6 Buffer Usage in Block Nested Loops Join

90 pages of Reserves, we would have to scan Sailors $\lceil 1,000/90 \rceil = 12$ times, and the total cost would be $1,000 + 12 * 500 = 7,000$ I/Os.

Suppose we choose Sailors to be the outer relation R instead. Scanning Sailors costs 500 I/Os. We would scan Reserves $\lceil 500/100 \rceil = 5$ times. The total cost is $500 + 5 * 1,000 = 5,500$ I/Os. If instead we have only enough buffers for 90 pages of Sailors, we would scan Reserves a total of $\lceil 500/90 \rceil = 6$ times. The total cost in this case is $500+6*1,000 = 6,500$ I/Os. We note that the block nested loops join algorithm takes a little over a minute on our running example, assuming 10ms per I/O as before.

Impact of Blocked Access

If we consider the effect of blocked access to several pages, there is a fundamental change in the way we allocate buffers for block nested loops. Rather than using just one buffer page for the inner relation, the best approach is to split the buffer pool evenly between the two relations. This allocation results in more passes over the inner relation, leading to more page fetches. However, the time spent on *seeking* for pages is dramatically reduced.

The technique of double buffering (discussed in Chapter 11 in the context of sorting) can also be used, but we will not discuss it further.

Index Nested Loops Join

If there is an index on one of the relations on the join attribute(s), we can take advantage of the index by making the indexed relation be the inner relation. Suppose that we have a suitable index on S; Figure 12.7 describes the index nested loops join algorithm.

```
foreach tuple $r \in R$ do
    foreach tuple $s \in S$ where $r_i == s_j$
        add $\langle r, s \rangle$ to result
```

Figure 12.7 Index Nested Loops Join

For each tuple $r \in R$, we use the index to retrieve matching tuples of S. Intuitively, we compare r only with tuples of S that are in the same *partition*, in that they have the same value in the join column. Unlike the other nested loops join algorithms, therefore, the index nested loops join algorithm does not enumerate the cross-product of R and S. The cost of scanning R is M, as before. The cost of retrieving matching S tuples depends on the kind of index and the number of matching tuples; for each R tuple, the cost is as follows:

1. If the index on S is a B+ tree index, the cost to find the appropriate leaf is typically 2 to 4 I/Os. If the index is a hash index, the cost to find the appropriate bucket is 1 or 2 I/Os.

2. Once we find the appropriate leaf or bucket, the cost of retrieving matching S tuples depends on whether the index is clustered. If it is, the cost per outer tuple $r \in R$ is typically just one more I/O. If it is not clustered, the cost could be one I/O per matching S-tuple (since each of these could be on a different page in the worst case).

As an example, suppose that we have a hash-based index using Alternative (2) on the *sid* attribute of Sailors and that it takes about 1.2 I/Os on average[2] to retrieve the appropriate page of the index. Since *sid* is a key for Sailors, we have at most one matching tuple. Indeed, *sid* in Reserves is a foreign key referring to Sailors, and therefore we have *exactly* one matching Sailors tuple for each Reserves tuple. Let us consider the cost of scanning Reserves and using the index to retrieve the matching Sailors tuple for each Reserves tuple. The cost of scanning Reserves is 1,000. There are $100 * 1,000$ tuples in Reserves. For each of these tuples, retrieving the index page containing the rid of the matching Sailors tuple costs 1.2 I/Os (on average); in addition, we have to retrieve the Sailors page containing the qualifying tuple. Thus we have $100,000 * (1 + 1.2)$ I/Os to retrieve matching Sailors tuples. The total cost is 221,000 I/Os.

As another example, suppose that we have a hash-based index using Alternative (2) on the *sid* attribute of Reserves. Now we can scan Sailors (500 I/Os) and for each tuple, use the index to retrieve matching Reserves tuples. We have a total of $80 * 500$ Sailors tuples, and each tuple could match with either zero or more Reserves tuples; a sailor

[2]This is a typical cost for hash-based indexes.

may have no reservations, or have several. For each Sailors tuple, we can retrieve the index page containing the rids of matching Reserves tuples (assuming that we have at most one such index page, which is a reasonable guess) in 1.2 I/Os on average. The total cost thus far is $500 + 40,000 * 1.2 = 48,500$ I/Os.

In addition, we have the cost of retrieving matching Reserves tuples. Since we have 100,000 reservations for 40,000 Sailors, assuming a uniform distribution we can estimate that each Sailors tuple matches with 2.5 Reserves tuples on average. If the index on Reserves is clustered, and these matching tuples are typically on the same page of Reserves for a given sailor, the cost of retrieving them is just one I/O per Sailor tuple, which adds up to 40,000 extra I/Os. If the index is not clustered, each matching Reserves tuple may well be on a different page, leading to a total of $2.5 * 40,000$ I/Os for retrieving qualifying tuples. Thus, the total cost can vary from $48,500 + 40,000 = 88,500$ to $48,500 + 100,000 = 148,500$ I/Os. Assuming 10ms per I/O, this would take about 15 to 25 minutes.

Thus, even with an unclustered index, if the number of matching inner tuples for each outer tuple is small (on average), the cost of the index nested loops join algorithm is likely to be much less than the cost of a simple nested loops join. The cost difference can be so great that some systems build an index on the inner relation at run-time if one does not already exist and do an index nested loops join using the newly created index.

12.5.2 Sort-Merge Join *

The basic idea behind the **sort-merge join** algorithm is to *sort* both relations on the join attribute and to then look for qualifying tuples $r \in R$ and $s \in S$ by essentially *merging* the two relations. The sorting step groups all tuples with the same value in the join column together and thus makes it easy to identify partitions, or groups of tuples with the same value in the join column. We exploit this partitioning by comparing the R tuples in a partition with only the S tuples in the same partition (rather than with all S tuples), thereby avoiding enumeration of the cross-product of R and S. (This partition-based approach works only for equality join conditions.)

The external sorting algorithm discussed in Chapter 11 can be used to do the sorting, and of course, if a relation is already sorted on the join attribute, we need not sort it again. We now consider the merging step in detail: We scan the relations R and S, looking for qualifying tuples (i.e., tuples Tr in R and Ts in S such that $Tr_i = Ts_j$). The two scans start at the first tuple in each relation. We advance the scan of R as long as the current R tuple is less than the current S tuple (with respect to the values in the join attribute). Similarly, we then advance the scan of S as long as the current S tuple is less than the current R tuple. We alternate between such advances until we find an R tuple Tr and a S tuple Ts with $Tr_i = Ts_j$.

When we find tuples Tr and Ts such that $Tr_i = Ts_j$, we need to output the joined tuple. In fact, we could have several R tuples and several S tuples with the same value in the join attributes as the current tuples Tr and Ts. We refer to these tuples as the *current R partition* and the *current S partition.* For each tuple r in the current R partition, we scan all tuples s in the current S partition and output the joined tuple $\langle r, s \rangle$. We then resume scanning R and S, beginning with the first tuples that follow the partitions of tuples that we just processed.

The sort-merge join algorithm is shown in Figure 12.8. We assign only tuple values to the variables Tr, Ts, and Gs and use the special value *eof* to denote that there are no more tuples in the relation being scanned. Subscripts identify fields, for example, Tr_i denotes the ith field of tuple Tr. If Tr has the value *eof*, any comparison involving Tr_i is defined to evaluate to `false`.

We illustrate sort-merge join on the Sailors and Reserves instances shown in Figures 12.9 and 12.10, with the join condition being equality on the *sid* attributes.

These two relations are already sorted on *sid*, and the merging phase of the sort-merge join algorithm begins with the scans positioned at the first tuple of each relation instance. We advance the scan of Sailors, since its *sid* value, now 22, is less than the *sid* value of Reserves, which is now 28. The second Sailors tuple has $sid = 28$, which is equal to the *sid* value of the current Reserves tuple. Therefore, we now output a result tuple for each pair of tuples, one from Sailors and one from Reserves, in the current partition (i.e., with $sid = 28$). Since we have just one Sailors tuple with $sid = 28$, and two such Reserves tuples, we write two result tuples. After this step, we position the scan of Sailors at the first tuple after the partition with $sid = 28$, which has $sid = 31$. Similarly, we position the scan of Reserves at the first tuple with $sid = 31$. Since these two tuples have the same *sid* values, we have found the next matching partition, and we must write out the result tuples generated from this partition (there are three such tuples). After this, the Sailors scan is positioned at the tuple with $sid = 36$, and the Reserves scan is positioned at the tuple with $sid = 58$. The rest of the merge phase proceeds similarly.

In general, we have to scan a partition of tuples in the second relation as often as the number of tuples in the corresponding partition in the first relation. The first relation in the example, Sailors, has just one tuple in each partition. (This is not happenstance, but a consequence of the fact that *sid* is a key—this example is a key–foreign key join.) In contrast, suppose that the join condition is changed to be *sname=rname*. Now, both relations contain more than one tuple in the partition with *sname=rname='lubber'*. The tuples with *rname='lubber'* in Reserves have to be scanned for each Sailors tuple with *sname='lubber'*.

```
proc $smjoin(R, S, `R_i = S'_j)$

if $R$ not sorted on attribute $i$, sort it;
if $S$ not sorted on attribute $j$, sort it;

$Tr$ = first tuple in $R$;                                   // ranges over $R$
$Ts$ = first tuple in $S$;                                   // ranges over $S$
$Gs$ = first tuple in $S$;                                   // start of current $S$-partition

while $Tr \neq eof$ and $Gs \neq eof$ do {

    while $Tr_i < Gs_j$ do
        $Tr$ = next tuple in $R$ after $Tr$;                 // continue scan of $R$

    while $Tr_i > Gs_j$ do
        $Gs$ = next tuple in $S$ after $Gs$                  // continue scan of $S$

    $Ts = Gs$;                                               // Needed in case $Tr_i \neq Gs_j$
    while $Tr_i == Gs_j$ do {                                // process current $R$ partition
        $Ts = Gs$;                                           // reset $S$ partition scan
        while $Ts_j == Tr_i$ do {                            // process current $R$ tuple
            add $\langle Tr, Ts \rangle$ to result;          // output joined tuples
            $Ts$ = next tuple in $S$ after $Ts$;}            // advance $S$ partition scan
        $Tr$ = next tuple in $R$ after $Tr$;                 // advance scan of $R$
        }                                                    // done with current $R$ partition

    $Gs = Ts$;                                               // initialize search for next $S$ partition

    }
```

Figure 12.8 Sort-Merge Join

sid	*sname*	*rating*	*age*
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
36	lubber	6	36.0
44	guppy	5	35.0
58	rusty	10	35.0

Figure 12.9 An Instance of Sailors

sid	*bid*	*day*	*rname*
28	103	12/04/96	guppy
28	103	11/03/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Figure 12.10 An Instance of Reserves

Cost of Sort-Merge Join

The cost of sorting R is O($MlogM$) and the cost of sorting S is O($NlogN$). The cost of the merging phase is $M + N$ if no S partition is scanned multiple times (or the necessary pages are found in the buffer after the first pass). This approach is especially attractive if at least one relation is already sorted on the join attribute or has a clustered index on the join attribute.

Consider the join of the relations Reserves and Sailors. Assuming that we have 100 buffer pages (roughly the same number that we assumed were available in our discussion of block nested loops join), we can sort Reserves in just two passes. The first pass produces 10 internally sorted runs of 100 pages each. The second pass merges these 10 runs to produce the sorted relation. Because we read and write Reserves in each pass, the sorting cost is $2 * 2 * 1,000 = 4,000$ I/Os. Similarly, we can sort Sailors in two passes, at a cost of $2 * 2 * 500 = 2,000$ I/Os. In addition, the second phase of the sort-merge join algorithm requires an additional scan of both relations. Thus the total cost is $4,000 + 2,000 + 1,000 + 500 = 7,500$ I/Os, which is similar to the cost of the block nested loops algorithm.

Suppose that we have only 35 buffer pages. We can still sort both Reserves and Sailors in two passes, and the cost of the sort-merge join algorithm remains at 7,500 I/Os. However, the cost of the block nested loops join algorithm is more than 15,000 I/Os. On the other hand, if we have 300 buffer pages, the cost of the sort-merge join remains at 7,500 I/Os, whereas the cost of the block nested loops join drops to 2,500 I/Os. (We leave it to the reader to verify these numbers.)

We note that multiple scans of a partition of the second relation are potentially expensive. In our example, if the number of Reserves tuples in a repeatedly scanned partition is small (say, just a few pages), the likelihood of finding the entire partition in the buffer pool on repeated scans is very high, and the I/O cost remains essentially the same as for a single scan. However, if there are many pages of Reserves tuples in a given partition, the first page of such a partition may no longer be in the buffer pool when we request it a second time (after first scanning all pages in the partition; remember that each page is unpinned as the scan moves past it). In this case, the I/O cost could be as high as the number of pages in the Reserves partition times the number of tuples in the corresponding Sailors partition!

In the worst-case scenario, the merging phase could require us to read all of the second relation for each *tuple* in the first relation, and the number of I/Os is $O(M * N)$ I/Os! (This scenario occurs when all tuples in both relations contain the same value in the join attribute; it is extremely unlikely.)

In practice the I/O cost of the merge phase is typically just a single scan of each relation. A single scan can be guaranteed if at least one of the relations involved has no duplicates in the join attribute; this is the case, fortunately, for key–foreign key joins, which are very common.

A Refinement

We have assumed that the two relations are sorted first and then merged in a distinct pass. It is possible to improve the sort-merge join algorithm by combining the merging phase of sorting with the merging phase of the join. First we produce sorted runs of size B for both R and S. If $B > \sqrt{L}$, where L is the size of the larger relation, the number of runs per relation is less than $\sqrt{L}$. Suppose that the number of buffers available for the merging phase is at least $2\sqrt{L}$, that is, more than the total number of runs for R and S. We allocate one buffer page for each run of R *and* one for each run of S. We then merge the runs of R (to generate the sorted version of R), merge the runs of S, and merge the resulting R and S streams as they are generated; we apply the join condition as we merge the R and S streams and discard tuples in the cross-product that do not meet the join condition.

Unfortunately, this idea increases the number of buffers required to $2\sqrt{L}$. However, by using the technique discussed in Section 11.2.1 we can produce sorted runs of size approximately $2 * B$ for both R and S. Consequently we have fewer than $\sqrt{L}/2$ runs of each relation, given the assumption that $B > \sqrt{L}$. Thus, the total number of runs is less than $\sqrt{L}$, that is, less than B, and we can combine the merging phases with no need for additional buffers.

This approach allows us to perform a sort-merge join at the cost of reading and writing R and S in the first pass and of reading R and S in the second pass. The total cost is thus $3 * (M + N)$. In our example the cost goes down from 7,500 to 4,500 I/Os.

Blocked Access and Double-Buffering

The blocked I/O and double-buffering optimizations, discussed in Chapter 11 in the context of sorting, can be used to speed up the merging pass, as well as the sorting of the relations to be joined; we will not discuss these refinements.

12.5.3 Hash Join *

The **hash join** algorithm, like the sort-merge join algorithm, identifies partitions in R and S in a **partitioning phase**, and in a subsequent **probing phase** compares tuples in an R partition only with tuples in the corresponding S partition for testing equality join conditions. Unlike sort-merge join, hash join uses hashing to identify partitions, rather than sorting. The partitioning (also called **building**) phase of hash join is similar to the partitioning in hash-based projection and is illustrated in Figure 12.3. The probing (sometimes called **matching**) phase is illustrated in Figure 12.11.

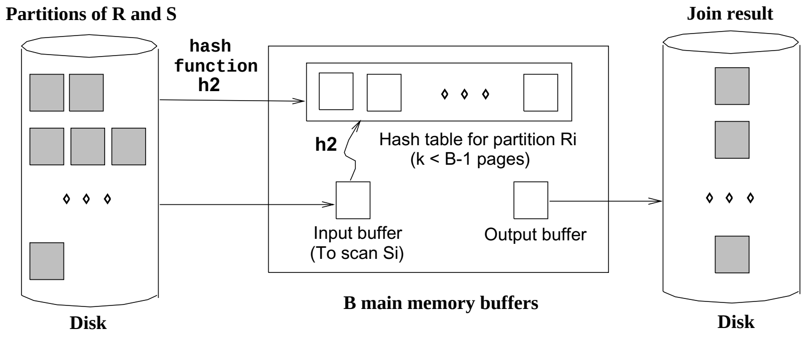

Figure 12.11 Probing Phase of Hash Join

The idea is to hash *both* relations on the join attribute, using the *same* hash function h. If we hash each relation (hopefully uniformly) into k partitions, we are assured that R tuples in partition i can join only with S tuples in the same partition i. This observation can be used to good effect: We can read in a (complete) partition of the smaller relation R and scan just the corresponding partition of S for matches. We never need to consider these R and S tuples again. Thus, once R and S are partitioned, we can perform the join by reading in R and S just once, provided that enough memory is available to hold all the tuples in any given partition of R.

In practice we build an in-memory hash table for the R partition, using a hash function $h2$ that is different from h (since $h2$ is intended to distribute tuples in a partition based on h!), in order to reduce CPU costs. We need enough memory to hold this hash table, which is a little larger than the R partition itself.

The hash join algorithm is presented in Figure 12.12. (There are several variants on this idea; the version that we present is called *Grace hash join* in the literature.) Consider the cost of the hash join algorithm. In the partitioning phase we have to scan both R and S once and write them both out once. The cost of this phase is therefore $2(M + N)$. In the second phase we scan each partition once, assuming no partition overflows, at a cost of $M + N$ I/Os. The total cost is therefore $3(M + N)$, given our assumption that each partition fits into memory in the second phase. On our example join of Reserves and Sailors, the total cost is $3 * (500 + 1,000) = 4,500$ I/Os, and assuming 10ms per I/O, hash join takes under a minute. Compare this with simple nested loops join, which took about 140 *hours*—this difference underscores the importance of using a good join algorithm.

```
// Partition $R$ into $k$ partitions
foreach tuple $r \in R$ do
    read $r$ and add it to buffer page $h(r_i)$;                // flushed as page fills

// Partition $S$ into $k$ partitions
foreach tuple $s \in S$ do
    read $s$ and add it to buffer page $h(s_j)$;                // flushed as page fills

// Probing Phase
for $l = 1, \ldots, k$ do {

    // Build in-memory hash table for $R_l$, using $h2$
    foreach tuple $r \in$ partition $R_l$ do
        read $r$ and insert into hash table using $h2(r_i)$ ;

    // Scan $S_l$ and probe for matching $R_l$ tuples
    foreach tuple $s \in$ partition $S_l$ do {
        read $s$ and probe table using $h2(s_j)$;
        for matching $R$ tuples $r$, output $\langle r, s \rangle$ };

    clear hash table to prepare for next partition;
    }
```

Figure 12.12 Hash Join

Memory Requirements and Overflow Handling

To increase the chances of a given partition fitting into available memory in the probing phase, we must minimize the size of a partition by maximizing the number of partitions. In the partitioning phase, to partition R (similarly, S) into k partitions, we need at least k output buffers and one input buffer. Thus, given B buffer pages, the maximum number of partitions is $k = B - 1$. Assuming that partitions are equal in size, this means that the size of each R partition is $\frac{M}{B-1}$ (as usual, M is the number of pages of R). The number of pages in the (in-memory) hash table built during the probing phase for a partition is thus $\frac{f*M}{B-1}$, where f is a *fudge factor* used to capture the (small) increase in size between the partition and a hash table for the partition.

During the probing phase, in addition to the hash table for the R partition, we require a buffer page for scanning the S partition, and an output buffer. Therefore, we require $B > \frac{f*M}{B-1} + 2$. *We need approximately* $B > \sqrt{f * M}$ for the hash join algorithm to perform well.

Since the partitions of R are likely to be close in size, but not identical, the largest partition will be somewhat larger than $\frac{M}{B-1}$, and the number of buffer pages required is a little more than $B > \sqrt{f * M}$. There is also the risk that if the hash function h does not partition R uniformly, the hash table for one or more R partitions may not fit in memory during the probing phase. This situation can significantly degrade performance.

As we observed in the context of hash-based projection, one way to handle this *partition overflow* problem is to recursively apply the hash join technique to the join of the overflowing R partition with the corresponding S partition. That is, we first divide the R and S partitions into subpartitions. Then we join the subpartitions pairwise. All subpartitions of R will probably fit into memory; if not, we apply the hash join technique recursively.

Utilizing Extra Memory: Hybrid Hash Join

The minimum amount of memory required for hash join is $B > \sqrt{f * M}$. If more memory is available, a variant of hash join called **hybrid hash join** offers better performance. Suppose that $B > f * (M/k)$, for some integer k. This means that if we divide R into k partitions of size M/k, an in-memory hash table can be built for each partition. To partition R (similarly, S) into k partitions, we need k output buffers and one input buffer, that is, $k + 1$ pages. This leaves us with $B - (k + 1)$ extra pages during the partitioning phase.

Suppose that $B - (k+1) > f * (M/k)$. That is, we have enough extra memory during the partitioning phase to hold an in-memory hash table for a partition of R. The idea behind hybrid hash join is to build an in-memory hash table for the first partition of R during the partitioning phase, which means that we don't write this partition to disk. Similarly, while partitioning S, rather than write out the tuples in the first partition of S, we can directly probe the in-memory table for the first R partition and write out the results. At the end of the partitioning phase, we have completed the join of the first partitions of R and S, in addition to partitioning the two relations; in the probing phase, we join the remaining partitions as in hash join.

The savings realized through hybrid hash join is that we avoid writing the first partitions of R and S to disk during the partitioning phase and reading them in again during the probing phase. Consider our example, with 500 pages in the smaller relation R and 1,000 pages in S.[3] If we have $B = 300$ pages, we can easily build an in-memory hash table for the first R partition while partitioning R into two partitions. During the partitioning phase of R, we scan R and write out one partition; the cost is $500 + 250$

[3]It is unfortunate that in our running example, the smaller relation, which we have denoted by the variable R in our discussion of hash join, is in fact the Sailors relation, which is more naturally denoted by S!

if we assume that the partitions are of equal size. We then scan S and write out one partition; the cost is $1,000 + 500$. In the probing phase, we scan the second partition of R and of S; the cost is $250 + 500$. The total cost is $750 + 1,500 + 750 = 3,000$. In contrast, the cost of hash join is $4,500$.

If we have enough memory to hold an in-memory hash table for all of R, the savings are even greater. For example, if $B > f * N + 2$, that is, $k = 1$, we can build an in-memory hash table for all of R. This means that we only read R once, to build this hash table, and read S once, to probe the R hash table. The cost is $500 + 1,000 = 1,500$.

Hash Join versus Block Nested Loops Join

While presenting the block nested loops join algorithm, we briefly discussed the idea of building an in-memory hash table for the inner relation. We now compare this (more CPU-efficient) version of block nested loops join with hybrid hash join.

If a hash table for the entire smaller relation fits in memory, the two algorithms are identical. If both relations are large relative to the available buffer size, we require several passes over one of the relations in block nested loops join; hash join is a more effective application of hashing techniques in this case. The I/O that is saved in this case by using the hash join algorithm in comparison to a block nested loops join is illustrated in Figure 12.13. In the latter, we read in all of S for each block of R; the I/O cost corresponds to the whole rectangle. In the hash join algorithm, for each block of R, we read only the corresponding block of S; the I/O cost corresponds to the shaded areas in the figure. This difference in I/O due to scans of S is highlighted in the figure.

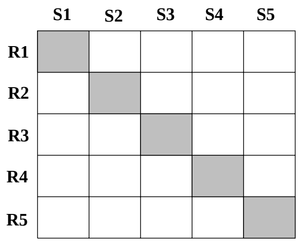

Figure 12.13 Hash Join versus Block Nested Loops for Large Relations

We note that this picture is rather simplistic. It does not capture the cost of scanning R in block nested loops join and the cost of the partitioning phase in hash join, and it focuses on the cost of the probing phase.

Hash Join versus Sort-Merge Join

Let us compare hash join with sort-merge join. If we have $B > \sqrt{M}$ buffer pages, where M is the number of pages in the *smaller* relation, and we assume uniform partitioning, the cost of hash join is $3(M + N)$ I/Os. If we have $B > \sqrt{N}$ buffer pages, where N is the number of pages in the *larger* relation, the cost of sort-merge join is also $3(M + N)$, as discussed in Section 12.5.2. A choice between these techniques is therefore governed by other factors, notably:

- If the partitions in hash join are not uniformly sized, hash join could cost more. Sort-merge join is less sensitive to such data skew.
- If the available number of buffers falls between $\sqrt{M}$ and $\sqrt{N}$, hash join costs less than sort-merge join, since we need only enough memory to hold partitions of the smaller relation, whereas in sort-merge join the memory requirements depend on the size of the larger relation. The larger the difference in size between the two relations, the more important this factor becomes.
- Additional considerations include the fact that the result is sorted in sort-merge join.

12.5.4 General Join Conditions *

We have discussed several join algorithms for the case of a simple equality join condition. Other important cases include a join condition that involves equalities over several attributes and inequality conditions. To illustrate the case of several equalities, we consider the join of Reserves R and Sailors S with the join condition $R.sid{=}S.sid \wedge R.rname{=}S.sname$:

- For index nested loops join, we can build an index on Reserves on the combination of fields $\langle R.sid, R.rname \rangle$ and treat Reserves as the inner relation. We can also use an existing index on this combination of fields, or on *R.sid*, or on *R.rname*. (Similar remarks hold for the choice of Sailors as the inner relation, of course.)
- For sort-merge join, we sort Reserves on the combination of fields $\langle sid, rname \rangle$ and Sailors on the combination of fields $\langle sid, sname \rangle$. Similarly, for hash join, we partition on these combinations of fields.
- The other join algorithms that we discussed are essentially unaffected.

If we have an inequality comparison, for example, a join of Reserves R and Sailors S with the join condition $R.rname < S.sname$:

- We require a B+ tree index for index nested loops join.

- Hash join and sort-merge join are not applicable.
- The other join algorithms that we discussed are essentially unaffected.

Of course, regardless of the algorithm, the number of qualifying tuples in an inequality join is likely to be much higher than in an equality join.

We conclude our presentation of joins with the observation that there is no join algorithm that is uniformly superior to the others. The choice of a good algorithm depends on the sizes of the relations being joined, available access methods, and the size of the buffer pool. This choice can have a considerable impact on performance because the difference between a good and a bad algorithm for a given join can be enormous.

12.6 THE SET OPERATIONS *

We now briefly consider the implementation of the set operations $R \cap S$, $R \times S$, $R \cup S$, and $R - S$. From an implementation standpoint, intersection and cross-product can be seen as special cases of join (with equality on all fields as the join condition for intersection, and with no join condition for cross-product). Therefore, we will not discuss them further.

The main point to address in the implementation of union is the elimination of duplicates. Set-difference can also be implemented using a variation of the techniques for duplicate elimination. (Union and difference queries on a single relation can be thought of as a selection query with a complex selection condition. The techniques discussed in Section 12.3 are applicable for such queries.)

There are two implementation algorithms for union and set-difference, again based on sorting and hashing. Both algorithms are instances of the partitioning technique mentioned in Section 12.1.

12.6.1 Sorting for Union and Difference

To implement $R \cup S$:

1. Sort R using the combination of all fields; similarly, sort S.
2. Scan the sorted R and S in parallel and merge them, eliminating duplicates.

As a refinement, we can produce sorted runs of R and S and merge these runs in parallel. (This refinement is similar to the one discussed in detail for projection.) The implementation of $R - S$ is similar. During the merging pass, we write only tuples of R to the result, after checking that they do not appear in S.

12.6.2 Hashing for Union and Difference

To implement $R \cup S$:

1. Partition R and S using a hash function h.
2. Process each partition l as follows:
   - Build an in-memory hash table (using hash function $h2 \neq h$) for S_l.
   - Scan R_l. For each tuple, probe the hash table for S_l. If the tuple is in the hash table, discard it; otherwise, add it to the table.
   - Write out the hash table and then clear it to prepare for the next partition.

To implement $R - S$, we proceed similarly. The difference is in the processing of a partition. After building an in-memory hash table for S_l, we scan R_l. For each R_l tuple, we probe the hash table; if the tuple is not in the table, we write it to the result.

12.7 AGGREGATE OPERATIONS *

The SQL query shown in Figure 12.14 involves an *aggregate operation*, `AVG`. The other aggregate operations supported in SQL-92 are `MIN`, `MAX`, `SUM`, and `COUNT`.

```
SELECT AVG(S.age)
FROM   Sailors S
```

Figure 12.14 Simple Aggregation Query

The basic algorithm for aggregate operators consists of scanning the entire Sailors relation and maintaining some **running information** about the scanned tuples; the details are straightforward. The running information for each aggregate operation is shown in Figure 12.15. The cost of this operation is the cost of scanning all Sailors tuples.

Aggregate Operation	*Running Information*
`SUM`	*Total* of the values retrieved
`AVG`	⟨*Total, Count*⟩ of the values retrieved
`COUNT`	*Count* of values retrieved
`MIN`	Smallest value retrieved
`MAX`	Largest value retrieved

Figure 12.15 Running Information for Aggregate Operations

Aggregate operators can also be used in combination with a `GROUP BY` clause. If we add `GROUP BY` *rating* to the query in Figure 12.14, we would have to compute the average age of sailors for each *rating* group. For queries with grouping, there are two good evaluation algorithms that do not rely on an existing index; one algorithm is based on sorting and the other is based on hashing. Both algorithms are instances of the partitioning technique mentioned in Section 12.1.

The *sorting* approach is simple—we sort the relation on the grouping attribute (*rating*) and then scan it again to compute the result of the aggregate operation for each group. The second step is similar to the way we implement aggregate operations without grouping, with the only additional point being that we have to watch for group boundaries. (It is possible to refine the approach by doing aggregation as part of the sorting step; we leave this as an exercise for the reader.) The I/O cost of this approach is just the cost of the sorting algorithm.

In the *hashing* approach we build a hash table (in main memory if possible) on the grouping attribute. The entries have the form ⟨*grouping-value, running-info*⟩. The running information depends on the aggregate operation, as per the discussion of aggregate operations without grouping. As we scan the relation, for each tuple, we probe the hash table to find the entry for the group to which the tuple belongs and update the running information. When the hash table is complete, the entry for a grouping value can be used to compute the answer tuple for the corresponding group in the obvious way. If the hash table fits in memory, which is likely because each entry is quite small and there is only one entry per grouping value, the cost of the hashing approach is $O(M)$, where M is the size of the relation.

If the relation is so large that the hash table does not fit in memory, we can partition the relation using a hash function h on *grouping-value.* Since all tuples with a given grouping-value are in the same partition, we can then process each partition independently by building an in-memory hash table for the tuples in it.

12.7.1 Implementing Aggregation by Using an Index

The technique of using an index to select a subset of useful tuples is not applicable for aggregation. However, under certain conditions we can evaluate aggregate operations efficiently by using the data entries in an index instead of the data records:

- If the search key for the index includes all the attributes needed for the aggregation query, we can apply the techniques described earlier in this section to the set of data entries in the index, rather than to the collection of data records, and thereby avoid fetching data records.
- If the `GROUP BY` clause attribute list forms a prefix of the index search key and the index is a tree index, we can retrieve data entries (and data records, if necessary) in the order required for the grouping operation, and thereby avoid a sorting step.

A given index may support one or both of these techniques; both are examples of *index-only* plans. We discuss the use of indexes for queries with grouping and aggregation in the context of queries that also include selections and projections in Section 14.4.1.

12.8 THE IMPACT OF BUFFERING *

In implementations of relational operators, effective use of the buffer pool is very important, and we explicitly considered the size of the buffer pool in determining algorithm parameters for several of the algorithms that we discussed. There are three main points to note:

1. If several operations execute concurrently, they share the buffer pool. This effectively reduces the number of buffer pages available for each operation.

2. If tuples are accessed using an index, especially an unclustered index, the likelihood of finding a page in the buffer pool if it is requested multiple times depends (in a rather unpredictable way, unfortunately) on the size of the buffer pool and the replacement policy. Further, if tuples are accessed using an unclustered index, each tuple retrieved is likely to require us to bring in a new page; thus, the buffer pool fills up quickly, leading to a high level of paging activity.

3. If an operation has a *pattern* of repeated page accesses, we can increase the likelihood of finding a page in memory by a good choice of replacement policy or by *reserving* a sufficient number of buffers for the operation (if the buffer manager provides this capability). Several examples of such patterns of repeated access follow:

   - Consider a simple nested loops join. For each tuple of the outer relation, we repeatedly scan all pages in the inner relation. If we have enough buffer pages to hold the entire inner relation, the replacement policy is irrelevant. Otherwise, the replacement policy becomes critical. With LRU we will *never* find a page when it is requested, because it is paged out. This is the *sequential flooding* problem that we discussed in Section 7.4.1. With MRU we obtain the best buffer utilization—the first $B - 2$ pages of the inner relation always remain in the buffer pool. (B is the number of buffer pages; we use one page for scanning the outer relation,[4] and always replace the last page used for scanning the inner relation.)

   - In a block nested loops join, for each block of the outer relation, we scan the entire inner relation. However, since only one unpinned page is available for the scan of the inner relation, the replacement policy makes no difference.

   - In an index nested loops join, for each tuple of the outer relation, we use the index to find matching inner tuples. If several tuples of the outer relation

[4] Think about the sequence of pins and unpins used to achieve this.

have the same value in the join attribute, there is a repeated pattern of access on the inner relation; we can maximize the repetition by sorting the outer relation on the join attributes.

12.9 POINTS TO REVIEW

- Queries are composed of a few basic operators whose implementation impacts performance. All queries need to retrieve tuples from one or more input relations. The alternative ways of retrieving tuples from a relation are called *access paths*. An index *matches* selection conditions in a query if the index can be used to only retrieve tuples that satisfy the selection conditions. The *selectivity* of an access path with respect to a query is the total number of pages retrieved using the access path for this query. **(Section 12.1)**
- Consider a simple selection query of the form $\sigma_{R.attr \ \mathbf{op} \ value}(R)$. If there is no index and the file is not sorted, the only access path is a file scan. If there is no index but the file is sorted, a binary search can find the first occurrence of a tuple in the query. If a B+ tree index matches the selection condition, the selectivity depends on whether the index is clustered or unclustered and the number of result tuples. Hash indexes can be used only for equality selections. **(Section 12.2)**
- General selection conditions can be expressed in *conjunctive normal form*, where each *conjunct* consists of one or more *terms*. Conjuncts that contain $\vee$ are called *disjunctive*. A more complicated rule can be used to determine whether a general selection condition matches an index. There are several implementation options for general selections. **(Section 12.3)**
- The projection operation can be implemented by sorting and duplicate elimination during the sorting step. Another, hash-based implementation first partitions the file according to a hash function on the output attributes. Two tuples that belong to different partitions are guaranteed not to be duplicates because they have different hash values. In a subsequent step each partition is read into main memory and within-partition duplicates are eliminated. If an index contains all output attributes, tuples can be retrieved solely from the index. This technique is called an *index-only scan*. **(Section 12.4)**
- Assume that we join relations R and S. In a *nested loops join*, the join condition is evaluated between each pair of tuples from R and S. A *block nested loops join* performs the pairing in a way that minimizes the number of disk accesses. An *index nested loops join* fetches only matching tuples from S for each tuple of R by using an index. A *sort-merge join* sorts R and S on the join attributes using an external merge sort and performs the pairing during the final merge step. A *hash join* first partitions R and S using a hash function on the join attributes. Only partitions with the same hash values need to be joined in a subsequent step. A *hybrid hash join* extends the basic hash join algorithm by making more efficient

use of main memory if more buffer pages are available. Since a join is a very expensive, but common operation, its implementation can have great impact on overall system performance. The choice of the join implementation depends on the number of buffer pages available and the sizes of R and S. **(Section 12.5)**

- The set operations $R \cap S$, $R \times S$, $R \cup S$, and $R - S$ can be implemented using sorting or hashing. In sorting, R and S are first sorted and the set operation is performed during a subsequent merge step. In a hash-based implementation, R and S are first partitioned according to a hash function. The set operation is performed when processing corresponding partitions. **(Section 12.6)**

- Aggregation can be performed by maintaining *running information* about the tuples. Aggregation with grouping can be implemented using either sorting or hashing with the grouping attribute determining the partitions. If an index contains sufficient information for either simple aggregation or aggregation with grouping, *index-only plans* that do not access the actual tuples are possible. **(Section 12.7)**

- The number of buffer pool pages available —influenced by the number of operators being evaluated concurrently—and their effective use has great impact on the performance of implementations of relational operators. If an operation has a regular pattern of page accesses, choice of a good buffer pool replacement policy can influence overall performance. **(Section 12.8)**

EXERCISES

Exercise 12.1 Briefly answer the following questions:

1. Consider the three basic techniques, *iteration*, *indexing*, and *partitioning*, and the relational algebra operators *selection*, *projection*, and *join*. For each technique–operator pair, describe an algorithm based on the technique for evaluating the operator.
2. Define the term *most selective access path for a query.*
3. Describe *conjunctive normal form*, and explain why it is important in the context of relational query evaluation.
4. When does a general selection condition *match* an index? What is a *primary term* in a selection condition with respect to a given index?
5. How does hybrid hash join improve upon the basic hash join algorithm?
6. Discuss the pros and cons of hash join, sort-merge join, and block nested loops join.
7. If the join condition is not equality, can you use sort-merge join? Can you use hash join? Can you use index nested loops join? Can you use block nested loops join?
8. Describe how to evaluate a grouping query with aggregation operator `MAX` using a sorting-based approach.
9. Suppose that you are building a DBMS and want to add a new aggregate operator called `SECOND LARGEST`, which is a variation of the `MAX` operator. Describe how you would implement it.

10. Give an example of how buffer replacement policies can affect the performance of a join algorithm.

Exercise 12.2 Consider a relation R(a,b,c,d,e) containing 5,000,000 records, where each data page of the relation holds 10 records. R is organized as a sorted file with dense secondary indexes. Assume that *R.a* is a candidate key for R, with values lying in the range 0 to 4,999,999, and that R is stored in *R.a* order. For each of the following relational algebra queries, state which of the following three approaches is most likely to be the cheapest:

- Access the sorted file for R directly.
- Use a (clustered) B+ tree index on attribute *R.a*.
- Use a linear hashed index on attribute *R.a*.

1. $\sigma_{a<50,000}(R)$
2. $\sigma_{a=50,000}(R)$
3. $\sigma_{a>50,000 \wedge a<50,010}(R)$
4. $\sigma_{a\neq 50,000}(R)$

Exercise 12.3 Consider processing the following SQL projection query:

```
SELECT DISTINCT E.title, E.ename FROM Executives E
```

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.

Consider the optimized version of the sorting-based projection algorithm: The initial sorting pass reads the input relation and creates sorted runs of tuples containing only attributes *ename* and *title*. Subsequent merging passes eliminate duplicates while merging the initial runs to obtain a single sorted result (as opposed to doing a separate pass to eliminate duplicates from a sorted result containing duplicates).

1. How many sorted runs are produced in the first pass? What is the average length of these runs? (Assume that memory is utilized well and that any available optimization to increase run size is used.) What is the I/O cost of this sorting pass?
2. How many additional merge passes will be required to compute the final result of the projection query? What is the I/O cost of these additional passes?
3. (a) Suppose that a clustered B+ tree index on *title* is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered? Would your answer change if the index were a hash index?

   (b) Suppose that a clustered B+ tree index on *ename* is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered? Would your answer change if the index were a hash index?

   (c) Suppose that a clustered B+ tree index on ⟨*ename, title*⟩ is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered? Would your answer change if the index were a hash index?

4. Suppose that the query is as follows:

   `SELECT` E.title, E.ename `FROM` Executives E

   That is, you are not required to do duplicate elimination. How would your answers to the previous questions change?

Exercise 12.4 Consider the join $R \bowtie_{R.a=S.b} S$, given the following information about the relations to be joined. The cost metric is the number of page I/Os unless otherwise noted, and the cost of writing out the result should be uniformly ignored.

Relation R contains 10,000 tuples and has 10 tuples per page.
Relation S contains 2,000 tuples and also has 10 tuples per page.
Attribute *b* of relation S is the primary key for S.
Both relations are stored as simple heap files.
Neither relation has any indexes built on it.
52 buffer pages are available.

1. What is the cost of joining R and S using a page-oriented simple nested loops join? What is the minimum number of buffer pages required for this cost to remain unchanged?
2. What is the cost of joining R and S using a block nested loops join? What is the minimum number of buffer pages required for this cost to remain unchanged?
3. What is the cost of joining R and S using a sort-merge join? What is the minimum number of buffer pages required for this cost to remain unchanged?
4. What is the cost of joining R and S using a hash join? What is the minimum number of buffer pages required for this cost to remain unchanged?
5. What would be the lowest possible I/O cost for joining R and S using *any* join algorithm, and how much buffer space would be needed to achieve this cost? Explain briefly.
6. How many tuples will the join of R and S produce, at most, and how many pages would be required to store the result of the join back on disk?
7. Would your answers to any of the previous questions in this exercise change if you are told that *R.a* is a foreign key that refers to *S.b*?

Exercise 12.5 Consider the join of R and S described in Exercise 12.4.

1. With 52 buffer pages, if unclustered B+ indexes existed on *R.a* and *S.b*, would either provide a cheaper alternative for performing the join (using an index nested loops join) than a block nested loops join? Explain.

   (a) Would your answer change if only five buffer pages were available?

   (b) Would your answer change if S contained only 10 tuples instead of 2,000 tuples?

2. With 52 buffer pages, if *clustered* B+ indexes existed on *R.a* and *S.b*, would either provide a cheaper alternative for performing the join (using the *index nested loops* algorithm) than a block nested loops join? Explain.

   (a) Would your answer change if only five buffer pages were available?
   (b) Would your answer change if S contained only 10 tuples instead of 2,000 tuples?

3. If only 15 buffers were available, what would be the cost of a sort-merge join? What would be the cost of a hash join?
4. If the size of S were increased to also be 10,000 tuples, but only 15 buffer pages were available, what would be the cost of a sort-merge join? What would be the cost of a hash join?
5. If the size of S were increased to also be 10,000 tuples, and 52 buffer pages were available, what would be the cost of sort-merge join? What would be the cost of hash join?

Exercise 12.6 Answer each of the questions—if some question is inapplicable, explain why—in Exercise 12.4 again, but using the following information about R and S:

Relation R contains 200,000 tuples and has 20 tuples per page.
Relation S contains 4,000,000 tuples and also has 20 tuples per page.
Attribute *a* of relation R is the primary key for R.
Each tuple of R joins with exactly 20 tuples of S.
1,002 buffer pages are available.

Exercise 12.7 We described variations of the join operation called *outer joins* in Section 5.6.4. One approach to implementing an outer join operation is to first evaluate the corresponding (inner) join and then add additional tuples padded with *null* values to the result in accordance with the semantics of the given outer join operator. However, this requires us to compare the result of the inner join with the input relations to determine the additional tuples to be added. The cost of this comparison can be avoided by modifying the join algorithm to add these extra tuples to the result while input tuples are processed during the join. Consider the following join algorithms: *block nested loops join, index nested loops join, sort-merge join,* and *hash join.* Describe how you would modify each of these algorithms to compute the following operations on the Sailors and Reserves tables discussed in this chapter:

1. Sailors `NATURAL LEFT OUTER JOIN` Reserves
2. Sailors `NATURAL RIGHT OUTER JOIN` Reserves
3. Sailors `NATURAL FULL OUTER JOIN` Reserves

PROJECT-BASED EXERCISES

Exercise 12.8 (*Note to instructors: Additional details must be provided if this exercise is assigned; see Appendix B.*) Implement the various join algorithms described in this chapter in Minibase. (As additional exercises, you may want to implement selected algorithms for the other operators as well.)

BIBLIOGRAPHIC NOTES

The implementation techniques used for relational operators in System R are discussed in [88]. The implementation techniques used in PRTV, which utilized relational algebra transformations and a form of multiple-query optimization, are discussed in [303]. The techniques used for aggregate operations in Ingres are described in [209]. [275] is an excellent survey of algorithms for implementing relational operators and is recommended for further reading.

Hash-based techniques are investigated (and compared with sort-based techniques) in [93], [187], [276], and [588]. Duplicate elimination was discussed in [86]. [238] discusses secondary storage access patterns arising in join implementations. Parallel algorithms for implementing relational operations are discussed in [86, 141, 185, 189, 196, 251, 464].