the optimizer will never find such plans. There are two main reasons for this decision to concentrate on **left-deep plans,** or plans based on left-deep trees:

1. As the number of joins increases, the number of alternative plans increases rapidly and it becomes necessary to prune the space of alternative plans.
2. Left-deep trees allow us to generate all **fully pipelined** plans; that is, plans in which all joins are evaluated using pipelining. (Inner tables must always be materialized because we must examine the entire inner table for each tuple of the outer table. So, a plan in which an inner table is the result of a join forces us to materialize the result of that join.)

12.6.2 Estimating the Cost of a Plan

The cost of a plan is the sum of costs for the operators it contains. The cost of individual relational operators in the plan is estimated using information, obtained from the system catalog, about properties (e.g., size, sort order) of their input tables. We illustrated how to estimate the cost of single-operator plans in Sections 12.2 and 12.3, and how to estimate the cost of multi-operator plans in Section 12.5.

If we focus on the metric of I/O costs, the cost of a plan can be broken down into three parts: (1) reading the input tables (possibly rnultiple times in the case of some join and sorting algorithms), (2) writing intermediate tables, and (possibly) (3) sorting the final result (if the query specifies duplicate elimination or an output order). The third part is common to all plans (unless one of the plans happens to produce output in the required order), and, in the common case that a fully-pipelined plan is chosen, no intermediate tables are written.

Thus, the cost for a fully-pipelined plan is dominated by part (1). This cost depends greatly on the access paths used to read input tables; of course, access paths that are used repeatedly to retrieve matching tuples in a join algorithm are especially important.

For plans that are not fully pipelined, the cost of rnaterializing temporary tables can be significant. The cost of materializing an intermediate result depends on its size, and the size also infiuences the cost of the operator for which the temporary is an input table. The number of tuples in the result of a selection is estimated by multiplying the input size by the reduction factor for the selection conditions. The number of tuples in the result of a projection is the same as the input, assuming that duplicates are not eliminated; of course, each result tuple is smaller since it contains fewer fields.

The result size for a join can be estimated by multiplying the maximum result size, which is the product of the input table sizes, by the reduction factor of the join condition. The reduction factor for join condition *columni = column2* can be approximated by the formula $\frac{1}{MAX\ (NKeys(I1), NKeys(I2))}$ if there are indexes 11 and 12 on *columni* and *colwnn2,* respectively. This formula assumes that each key value in the smaller index, say 11, has a matching value in the other index. Given a value for *columni,* we assume that each of the *NKeys(I2)* values for *column2* is equally likely. Thus, the number of tuples that have the same value in *column2* as a given value in *columni* is $\frac{1}{NKeys(I2)}$.

12.7 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What is *metadata?* What metadata is stored in the *system catalog?* Describe the information stored per relation, and per index. (Section 12.1)
- The catalog is itself stored as a collection of relations. Explain why. (Section 12.1)
- What three techniques are commonly used in algorithms to evaluate relational operators? (Section 12.2)
- What is an access path? When does an index *match* a search condition? (Section 12.2.2)
- What are the main approaches to evaluating selections? Discuss the use of indexes, in particular. (Section 12.3.1)
- What are the main approaches to evaluating projections? What makes projections potentially expensive? (Section 12.3.2)
- What are the main approaches to evaluating joins? Why are joins expensive? (Section 12.3.3)
- What is the goal of query optimization? Is it to find the best plan? (Section 12.4)
- How does a DBMS represent a relational query evaluation plan? (Section 12.4.1)
- What is *pipelined evaluation?* What is its benefit? (Section 12.4.2)
- Describe the iterator interface for operators and access methods. 'What is its purpose? (Section 12.4.3)

- Discuss why the difference in cost between alternative plans for a query can be very large. Give specific examples to illustrate the impact of pushing selections, the choice of join methods, and the availability of appropriate indexes. (Section 12.5)
- What is the role of relational algebra equivalences in query optimization? (Section 12.6)
- What is the space of plans considered by a typical relational query optimizer? Justify the choice of this space of plans. (Section 12.6.1)
- How is the cost of a plan estimated? What is the role of the system catalog? What is the *selectivity* of an access path, and how does it influence the cost of a plan? Why is it important to be able to estimate the size of the result of a plan? (Section 12.6.2)

EXERCISES

Exercise 12.1 Briefly answer the following questions:

1. Describe three techniques commonly used when developing algorithms for relational operators. Explain how these techniques can be used to design algorithms for the selection, projection, and join operators.
2. What is an access path? When does an index *match* an access path? What is a *primary conjunct*, and why is it important?
3. What information is stored in the system catalogs?
4. What are the benefits of making the system catalogs be relations?
5. What is the goal of query optimization? Why is optimization important?
6. Describe *pipelining* and its advantages.
7. Give an example query and plan in which pipelining *cannot* be used.
8. Describe the *iterator* interface and explain its advantages.
9. What role do statistics gathered from the database play in query optimization?
10. What were the important design decisions made in the System R optimizer?
11. Why do query optimizers consider only left-deep join trees? Give an example of a query and a plan that would not be considered because of this restriction.

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 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.o*, 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 $R.a$.
- Usc 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** For each of the following SQL queries, for each relation involved, list the attributes that must be examined to compute the answer. All queries refer to the following relations:

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

1. SELECT * FROM Emp
2. SELECT * FROM Emp, Dept
3. SELECT * FROM Emp E, Dept D WHERE E.did = D.did
4. SELECT E.eid, D,dname FROM Emp E, Dept D WHERE E.did = D.did

Exercise **12.4** Consider the following schema with the Sailors relation:

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

For each of the following indexes, list whether the index matches the given selection conditions. If there is a match, list the primary conjuncts.

1. A B+-tree index on the search key ⟨ Sailors.sid ⟩.
   (a) $\sigma_{Sailors.sid<50,000}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000}(Sailors)$
2. A hash index on the search key ⟨ Sailors.sid ⟩.
   (a) $\sigma_{Sailors.sid<50,000}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000}(Sailors)$
3. A B+-tree index on the search key ⟨ Sailors.sid, Sailors.age ⟩.
   (a) $\sigma_{Sailors.sid<50,000 \wedge Sailors.age=21}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000 \wedge Sailors.age>21}(Sailors)$
   (c) $\sigma_{Sailors.sid=50,000}(Sailors)$
   (d) $\sigma_{Sailors.age=21}(Sailors)$
4. A hash-tree index on the search key ⟨ Sailors.sid, Sailors.age ⟩.
   (a) $\sigma_{Sailors.sid=50,000 \wedge Sailors.age=21}(Sailors)$
   (b) $\sigma_{Sailors.sid=50,000 \wedge Sailors.age>21}(Sailors)$
   (c) $\sigma_{Sailors.sid=50,000}(Sailors)$
   (d) $\sigma_{Sailors.age=21}(Sailors)$

Exercise **12.5** Consider again the schema with the Sailors relation:

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

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. For each of the following selection conditions, estimate the number of pages retrieved, given the catalog information in the question.

1. Assume that we have a B+-tree index T on the search key (Sailors.sid), and assume that *IHeight(T)* = 4, *INPages(T)* = 50, *Low(T)* = 1, and *High(T)* = 100,000.
   (a) *aSailors.'id<5o,ooo(S'a'ilors)*
   (b) $\sigma_{Sailors.sid=50,000}(Sailors)$
2. Assume that we have a hash index T on the search key (Sailors.sid), and assume that *IHeight(7')* = 2, *INPages(7')* = 50, *Low(7')* = 1, and *High(T)* = 100,000.
   (a) *aSa'lor's.sid<50,OOo(Sailors)*
   (b) *aSailor·s.sid=5o,ooo(5ailors)*

Exercise **12.6** Consider the two join methods described in Section 12.3.3. Assume that we join two relations *Rand* 5, and that the systems catalog contains appropriate statistics about Rand *S.* Write formulas for the cost estimates of the index nested loops join and sort-merge join using the appropriate variables from the systems catalog in Section 12.1. For index nested loops join, consider both a B+ tree index and a hash index. (For the hash index, you can assume that you can retrieve the index page containing the rid of the matching tuple with 1.2 l/Os on average.)

Note.' Additional exercises on the material covered in this chapter can be found in the exercises for Chapters 14 and 15.

BIBLIOGRAPHIC NOTES

See the bibliograpic notes for Chapters 14 and 15.

13

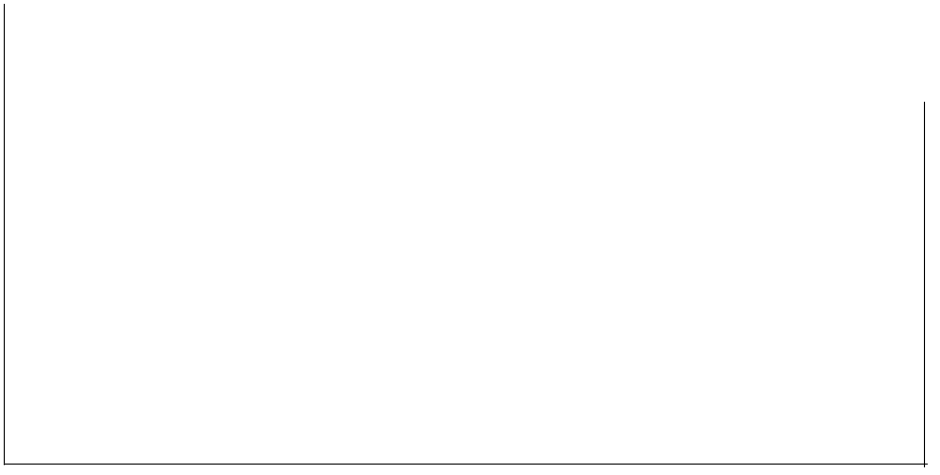
EXTERNAL SORTING

- Why is sorting important in a DBMS?
- Why is sorting data on disk different from sorting in-memory data?
- How does external merge-sort work?
- How do techniques like blockecl I/O and overlapped I/O affect the design of external sorting algorithms?
- When can we use a B+ tree to retrieve records in sorted order?
- Key concepts: motivation, bulk-loading, duplicate elimination, sort-merge joins; external merge sort, sorted runs, merging runs; replacement sorting, increasing run length; I/O cost versus number of I/Os, blocked I/Os, double buffering; B+ trees for sorting, impact of clustering.

Good order is the foundation of all things.

Edmund Burke

In this chapter, we consider a widely used and relatively expensive operation, sorting records according to a search key. We begin by considering the lnany uses of sorting in a database system in Section 13.1. We introduce the idea of external sorting by considering a very simple algorithm in Section 13.2; using repeated passes over the data, even very large datasets can be sorted with a small amount of rnemory. This algol'ithrn is generalized to develop a realistic external sorting algorithrn in Section 13.3. Three important refinements are



discussed. The first, discussed in Section 13.3.1, enables us to reduce the number of passes. The next two refinements, covered in Section 13.4, require us to consider a more detailed model of I/O costs than the number of page I/Os. Section 13.4.1 discusses the effect of *blocked* I/O, that is, reading and writing several pages at a time; and Section 13.4.2 considers how to use a technique called *double buffering* to minimize the time spent waiting for an I/O operation to complete. Section 13.5 discusses the use of B+ trees for sorting.

With the exception of Section 13.4, we consider only I/O costs, which we approximate by counting the number of pages read or written, as per the cost model discussed in Chapter 8. Our goal is to use a simple cost model to convey the main ideas, rather than to provide a detailed analysis.

13.1 WHEN DOES A DBMS SORT DATA?

Sorting a collection of records on some (search) key is a very useful operation. The key can be a single attribute or an ordered list of attributes, of course. Sorting is required in a variety of situations, including the following important ones:

- Users may want answers in some order; for example, by increasing age (Section 5.2).
- Sorting records is the first step in *bulk loading* a tree index (Section 10.8.2).
- Sorting is useful for eliminating *duplicate* copies in a collection of records (Section 14.3).

- A widely used algorithm for performing a very important relational algebra operation, called *jo'in,* requires a sorting step (Section 14.4.2).

Although main memory sizes are growing rapidly the ubiquity of database systems has lead to increasingly larger datasets as well. When the data to be sorted is too large to fit into available main memory, we need an *external sorting* algorithm. Such algorithms seek to minimize the cost of disk accesses.

13.2 A SIMPLE TWO-WAY MERGE SORT

We begin by presenting a simple algorithm to illustrate the idea behind external sorting. This algorithm utilizes only three pages of main memory, and it is presented only for pedagogical purposes. In practice, many more pages of memory are available, and we want our sorting algorithm to use the additional memory effectively; such an algorithm is presented in Section 13.3. When sorting a file, several sorted subfiles are typically generated in intermediate steps. In this chapter, we refer to each sorted subfile as a **run.**

Even if the entire file does not fit into the available main memory, we can sort it by breaking it into smaller subfiles, sorting these subfiles, and then merging them using a minimal amount of main memory at any given time. In the first pass, the pages in the file are read in one at a time. After a page is read in, the records on it are sorted and the sorted page (a sorted run one page long) is written out. Quicksort or any other in-memory sorting technique can be used to sort the records on a page. In subsequent passes, pairs of runs from the output of the previous pass are read in and *merged* to produce runs that are twice as long. This algorithm is shown in Figure 13.1.

If the number of pages in the input file is 2^k, for some k, then:

Pass 0 produces 2^k sorted runs of one page each,
Pass 1 produces 2^{k-1} sortecl runs of two pages each,
Pass 2 produces 2^{k-2} sortecl runs of four pages each,
and so on, until
Pass k produces one sorted run of 2^k pages.

In each pass, we read every page in the file, process it, and write it out. Therefore we have two disk I/Os per page, per pass. The number of passes is $\lceil log_2 N \rceil + 1$, where N is the number of pages in the file. The overall cost is $2N(\lceil log_2 N \rceil + 1)$ I/Os.

The algorithm is illustrated on all example input file containing seven pages in Figure 13.2. The sort takes four passes, and in each pass, we read and

```
proc 2-way_extsort (file)
// Given a file on disk, sorts it using three buffer pages
// Produce runs that are one page long: Pass 0
Read each page into memory, sort it, write it out.
// Merge pairs of runs to produce longer runs until only
// one run (containing all records of input file) is left
While the number of runs at end of previous pass is > 1:
    // Pass i = 1, 2, ...
    While there are runs to be merged from previous pass:
        Choose next two runs (from previous pass).
        Read each run into an input buffer; page at a time.
        Merge the runs and write to the output buffer;
        force output buffer to disk one page at a time.

endproc
```

Figure 13.1 Two-Way Merge Sort

write seven pages, for a total of 56 I/Os. This result agrees with the preceding analysis because $2 \cdot 7(\lceil \log_2 7 \rceil + 1) = 56$. The dark pages in the figure illustrate what would happen on a file of eight pages; the number of passes remains at four ($\lceil \log_2 8 \rceil + 1 = 4$), but we read and write an additional page in each pass for a total of 64 I/Os. (Try to work out what would happen on a file with, say, five pages.)

This algorithm requires just three buffer pages in main memory, as Figure 13.3 illustrates. This observation raises an important point: Even if we have more buffer space available, this simple algorithm does not utilize it effectively. The external merge sort algorithm that we discuss next addresses this problem.

13.3 EXTERNAL MERGE SORT

Suppose that B buffer pages are available in memory and that we need to sort a large file with N pages. How can we improve on the two-way merge sort presented in the previous section? The intuition behind the generalized algorithm that we now present is to retain the basic structure of making multiple passes while trying to minimize the number of passes. There are two important modifications to the two-way merge sort algorithm:

1. In Pass 0, read in B pages at a time and sort internally to produce $\lceil N/B \rceil$ runs of B pages each (except for the last run, which may contain fewer

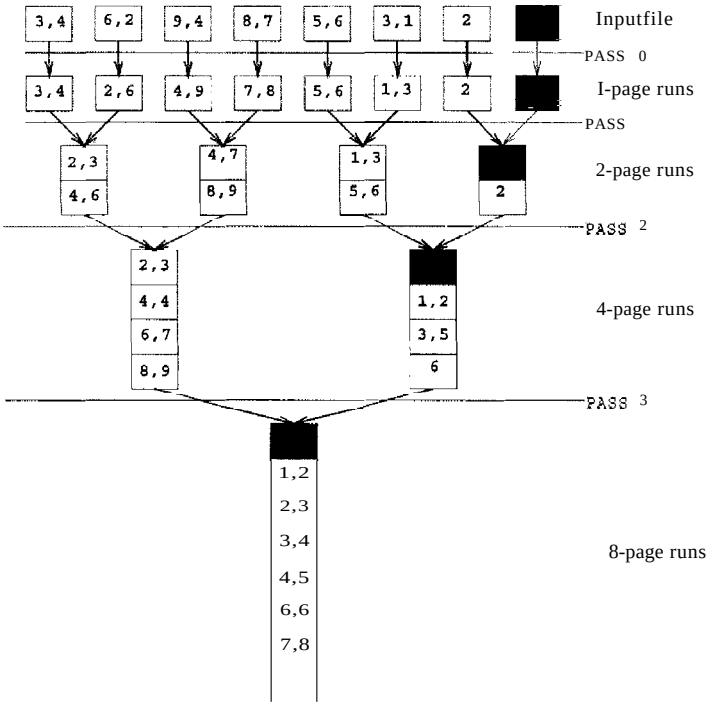


Figure 13.2 Two-Way Merge Sort of a Seven-Page File

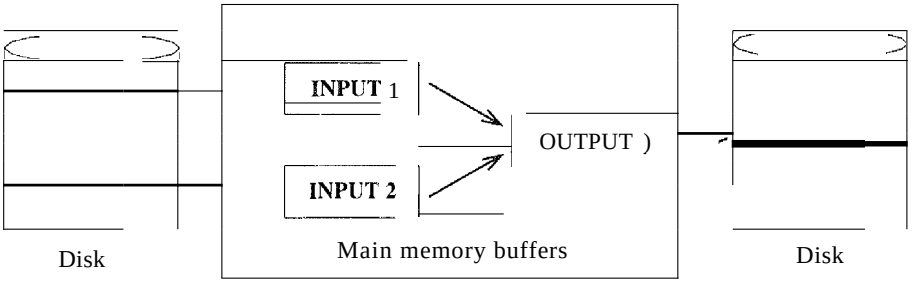


Figure 13.3 Two-'Way Merge Sort with Three Buffer Pages

pages). This modification is illustrated in Figure 13.4, using the input file from Figure 13.2 and a buffer pool with four pages.

2. In passes i = 1,2, ... use $B-1$ buffer pages for input and use the remaining page for output; hence, you do a $(B$ - I)-way merge in each pass. The utilization of buffer pages in the merging passes is illustrated in Figure 13.5.

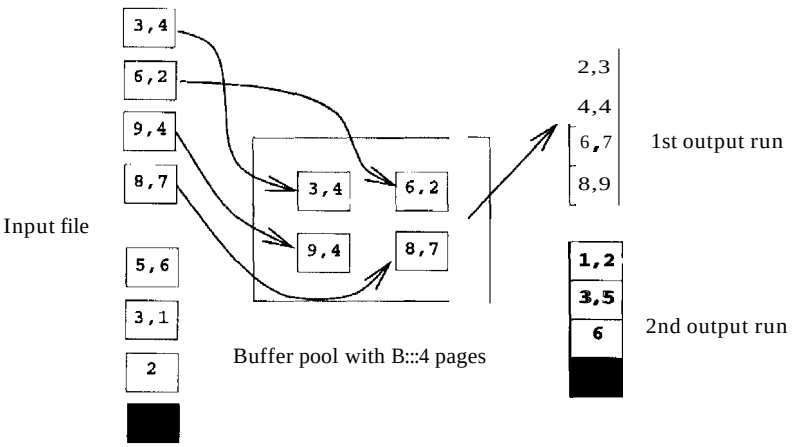


Figure 13.4 External Merge Sort with B Buffer Pages: Pass 0

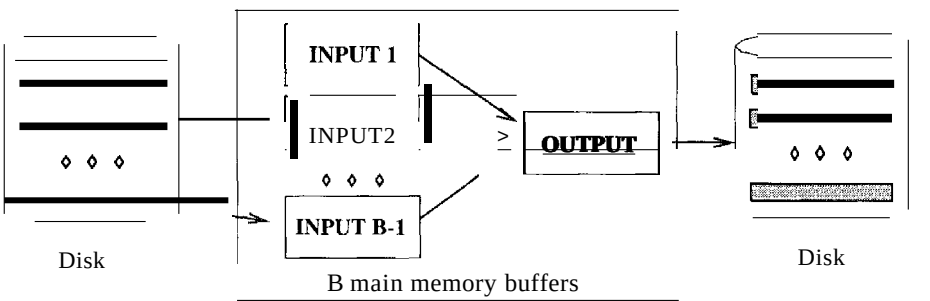


Figure 13.5 External IVlerge Sort with B Buffer Pages: Pass $i > 0$

The first refinement reduces the number of runs produced by Pass 0 to NI $\lceil N/B \rceil$, versus N for the two-way merge.[1] The second refinement is even more important. By doing a $(B-$ I)-way merge, the number of passes is reduced dramatically including the initial pass, it becomes rZ0.9B-$_1$NIl + 1 versus $\lceil log_2 N \rceil + 1$ for the two-way merge algorithm presented earlier. Because B is

[1] Note that the technique used for sorting data in buffer pages is orthogonal to external sorting. You could use, say, Quicksort for sorting data in buffer pages.

typically quite large, the savings can be substantial. The external merge sort algorithm is shown is Figure 13.6.

```
proc extsort (file)
// Given a file on disk, sorts it using three buffer pages
// Produce runs that are B pages long: Pass 0
Read B pages into memory, sort them, write out a run.
// Merge B-1 runs at a time to produce longer runs until only
// one run (containing all records of input file) is left
While the number of runs at end of previous pass is > 1:
    // Pass i = 1,2, ...
    While there are runs to be merged from previous pass:
        Choose next B -- 1 runs (from previous pass).
        Read each rUll into an input buffer; page at a time.
        Merge the rUllS and write to the output buffer;
        force output buffer to disk one page at a time.

endproc
```

Figure 13.6 External Merge Sort

As an example, suppose that we have five buffer pages available and want to sort a file with lOS pages.

Pass 0 produces $\lceil 108/5 \rceil$ = 22 sorted runs of five pages each, except for the last run, which is only three pages long.
Pass 1 does a four-way merge to produce 122/41 = six sorted runs of 20 pages each, except for the iast run, which is only eight pages long.
Pass 2 produces $\lceil 6/4 \rceil$ = two sorted runs; one with SO pages and one with 28 pages.
Pass 3 merges the two runs produced in Pass 2 to produce the sorted file.

In each pass we read and write 108 pages; thus the total cost is 2* 108*4 = 864 l/Os. Applying our formula, we have *NI* 1108/51 22 and cost $2 * N * (\lceil log_{B-1} N1 \rceil + 1)$ = $2 * 108 * (\lceil log_4 22 \rceil + 1)$ = 864, as expected.

To emphasize the potential gains in using all available buffers, in Figure 13.7, we show the number of passes, computed using our formula., for several values of Nand *B.* To obtain the cost, the number of passes should be multiplied by *2N*. In practice, one would expect to have more than 257 buffers, but this table illustrates the importance of a high fan-in during merging.

N	B = 3	B = 5	B = 9	B = 17	B = 129	B = 257
100	7	4	3	2	1	1
1000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Figure 13.7 Number of Passes of External Merge Sort

Of course, the CPU cost of a multiway merge can be greater than that for a two-way merge, but in general the I/O costs tend to dominate. In doing a (B - I)-way merge, we have to repeatedly pick the 'lowest' record in the $B-1$ runs being merged and write it to the output buffer. This operation can be implemented simply by examining the first (remaining) element in each of the $B-1$ input buffers. In practice, for large values of B, more sophisticated techniques can be used, although we do not discuss them here. Further, as we will see shortly, there are other ways to utilize buffer pages to reduce I/O costs; these techniques involve allocating additional pages to each input (and output) run, thereby making the number of runs merged in each pass considerably smaller than the number of buffer pages B.

13.3.1 Minimizing the Number of Runs

In Pass 0 we read in B pages at a time and sort them internally to produce $\lceil N/B \rceil$ runs of B pages each (except for the last run, which may contain fewer pages). With a more aggressive implementation, called **replacement sort,** we can write out runs of approximately $2 \cdot B$ internally sorted pages on average. This improvement is achieved as follows. We begin by reading in pages of the file of tuples to be sorted, say R, until the buffer is full, reserving (say) one page for use as an input buffer and one page for use as an output buffer. We refer to the $B - 2$ pages of R tuples that are not in the input or output buffer as the *current set.* Suppose that the file is to be sorted in ascending order on some search key k. Tuples are appended to the output in ascending order by k value.

The idea is to repeatedly pick the tuple in the current set with the smallest k value that is still greater than the largest k value in the output buffer and append it to the output buffer. For the output buffer to remain sorted, the chosen tuple must satisfy the condition that its k value be greater than or

equal to the largest k value currently in the output buffer; of all tuples in the current set that satisfy this condition, we pick the one with the smallest k value and append it to the output buffer. Moving this tuple to the output buffer creates some space in the current set, which 've use to add the next input tuple to the current set. (\Ve assume for simplicity that all tuples are the same size.) This process is illustrated in Figure 13.8. The tuple in the current set that is going to be appended to the output next is highlighted, as is the most recently appended output tuple.

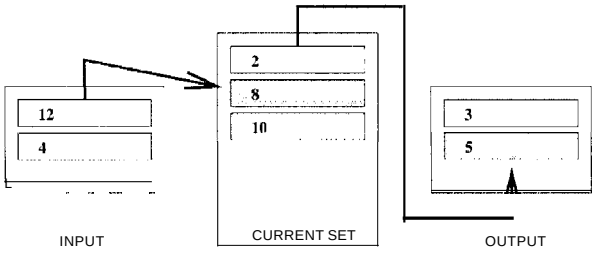


Figure 13.8 Generating Longer Runs

When all tuples in the input buffer have been consumed in this manner, the next page of the file is read in. Of course, the output buffer is written out when it is full, thereby extending the current run (which is gradually built up on disk).

The important question is this: When do we have to terminate the current run and start a new run? As long as some tuple t in the current set has a bigger k value than the most recently appended output tuple, we can append t to the output buffer and the current run can be extended.[2] In Figure 13.8, although a tuple (k = 2) in the current set has a smaller k value than the largest output tuple (k = 5), the current run can be extended because the current set also has a tuple (k = 8) that is larger than the largest output tuple.

When every tuple in the current set is smaller than the largest tuple in the output buffer, the output buffer is written out and becomes the last page in the current run. \Ve then start a new l'lm and continue the cycle of writing tuples from the input buffer to the current set to the output buffer. It is known that this algorithm produces runs that are about $2 \cdot B$ pages long, on average.

This refinement has not been implemented in commercial database systenls because managing the main memory available for sorting becOlnes difficult with

2If B is large, the CPU cost of finding such a tuple t can be significant unless appropriate in-memory data structures are used to organize the tuples in the buffer pool. \Ve will not discuss this issue further.

replacement sort, especially in the presence of variable length records. Recent work on this issue, however, shows promise and it could lead to the use of replacement sort in commercial systems.

13.4 MINIMIZING I/O COST VERSUS NUMBER OF I/OS

We have thus far used the number of page 1/Os as a cost metric. This metric is only an approximation of true I/O costs because it ignores the effect of *blocked* I/O--issuing a single request to read (or write) several consecutive pages can be much cheaper than reading (or writing) the same number of pages through independent I/O requests, as discussed in Chapter 8. This difference turns out to have some very important consequences for our external sorting algorithm.

Further, the time taken to perform I/O is only part of the time taken by the algorithm; we must consider CPU costs as well. Even if the time taken to do I/O accounts for most of the total time, the time taken for processing records is nontrivial and definitely worth reducing. In particular, we can use a technique called *double buffeTing* to keep the CPU busy while an I/O operation is in progress.

In this section, we consider how the external sorting algorithm can be refined using blocked I/O and double buffering. The motivation for these optimizations requires us to look beyond the number of I/Os as a cost metric. These optimizations can also be applied to other I/O intensive operations such as joins, which we study in Chapter 14.

13.4.1 Blocked I/O

If the number of page I/Os is taken to be the cost metric, the goal is clearly to minimize the number of passes in the sorting algorithm because each page in the file is read and written in each pass. It therefore makes sense to maximize the fan-in during merging by allocating just one buffer pool page per run (which is to be merged) and one buffer page for the output of the merge. Thus, we can merge $B-1$ runs, where B is the number of pages in the buffer pool. If we take into account the effect of blocked access, which reduces the average cost to read or write *a single page*, we are led to consider whether it might be better to read and write in units of more than one page.

Suppose we decide to read and write in units, which we call **buffer** blocks, of b pages. We must now set aside one buffer block per input run and one buffer block for the output of the merge, which means that we can merge at most $\lfloor \frac{B-b}{b} \rfloor$ runs in each pass. For example, if we have 10 buffer pages, we can either merge nine runs at a time with one-page input and output buffer

blocks, or we can merge four runs at a time with two-page input and output buffer blocks. If we choose larger buffer blocks, however, the number of passes increases, while we continue to read and write every page in the file in each pass! In the example, each merging pass reduces the number of runs by a factor of 4, rather than a factor of 9. Therefore, the number of page I/Os increases. This is the price we pay for decreasing the per-page I/O cost and is a trade-off we must take into account when designing an external sorting algorithm.

In practice, however, current main memory sizes are large enough that all but the largest files can be sorted in just two passes, even using blocked I/O. Suppose we have B buffer pages and choose to use a blocking factor of b pages. That is, we read and write b pages at a time, and all our input and output buffer blocks are b pages long. The first pass produces about $N2 = \lceil N/2B \rceil$ sorted runs, each of length $2B$ pages, if we use the optimization described in Section 13.3.1, and about $N1 = \lceil N/B \rceil$ sorted runs, each of length B pages, otherwise. For the purposes of this section, we assume that the optimization is used.

In subsequent passes we can merge $F = \lfloor B/b \rfloor - 1$ runs at a time. The number of passes is therefore $1 + \lceil \log_F N2 \rceil$, and in each pass we read and write all pages in the file. Figure 13.9 shows the number of passes needed to sort files of various sizes N, given B buffer pages, using a blocking factor b of 32 pages. It is quite reasonable to expect 5000 pages to be available for sorting purposes; with 4KB pages, 5000 pages is only 20MB. (With 50,000 buffer pages, we can do 1561-way merges; with 10,000 buffer pages, we can do 311-way merges; with 5000 buffer pages, we can do 155-way merges; and with 1000 buffer pages, we can do 30-way merges.)

N	B = 1000	B = 5000	B = 10,000	B = 50,000
100	1	1	1	1
1000	1	1	1	1
10,000	2	2	1	1
100,000	3	2	2	2
1,000,000	3	2	2	2
10,000,000	4	3	3	2
100,000,000	5	3	3	2
1,000,000,000	5	4	3	3

Figure 13.9 Number of Passes of External Merge Sort with Block Size $b = 32$

To compute the I/O cost, we need to calculate the number of 32-page blocks read or written and multiply this number by the cost of doing a 32-page block I/O. To find the number of block I/Os, we can find the total number of page

l/Os (number of passes rnultiplied by the number of pages in the file) and divide by the block size, 32. The cost of a 32-page block I/O is the seek time and rotational delay for the first page, plus transfer time for all 32 pages, as discussed in Chapter 8. The reader is invited to calculate the total I/O cost of sorting files of the sizes mentioned in Figure 13.9 with 5000 buffer pages for different block sizes (say, b = 1, 32, and 64) to get a feel for the benefits of using blocked I/O.

13.4.2 Double Buffering

Consider what happens in the external sorting algorithm when all the tuples in an input block have been consumed: An I/O request is issued for the next block of tuples in the corresponding input run, and the execution is forced to suspend until the I/O is complete. That is, for the duration of the time taken for reading in one block, the CPU remains idle (assuming that no other jobs are running). The overall time taken by an algorithm can be increased considerably because the CPU is repeatedly forced to wait for an I/O operation to complete. This effect becomes more and more important as CPU speeds increase relative to I/O speeds, which is a long-standing trend in relative speeds. It is therefore desirable to keep the CPU busy while an I/O request is being carried out; that is, to overlap CPU and I/O processing. Current hardware supports such overlapped computation, and it is therefore desirable to design algorithms to take advantage of this capability.

In the context of external sorting, we can achieve this overlap by allocating extra pages to each input buffer. Suppose a block size of b = 32 is chosen. The idea is to allocate an additional 32-page block to every input (and the output) buffer. Now, when all the tuples in a 32-page block have been consumed, the CPU can process the next 32 pages of the run by switching to the second, 'double,' block for this run. Meanwhile, an I/O request is issued to fill the empty block. Thus, assmning that the time to consume a block is greater than the time to read in a block, the CPU is never idle! On the other hand, the number of pages allocated to a buffer is doubled (for a given block size, which means the total I/O cost stays the same). This technique, called double buffering, can considerably reduce the total time taken to sort a file. The use of buffer pages is illustrated in Figure 13.10.

Note that although double buffering can considerably reduce the response tiule for a given query, it may not have a significant impact on throughput, because the CPU can be kept busy by working on other queries while waiting for one query's I/O operation to complete.

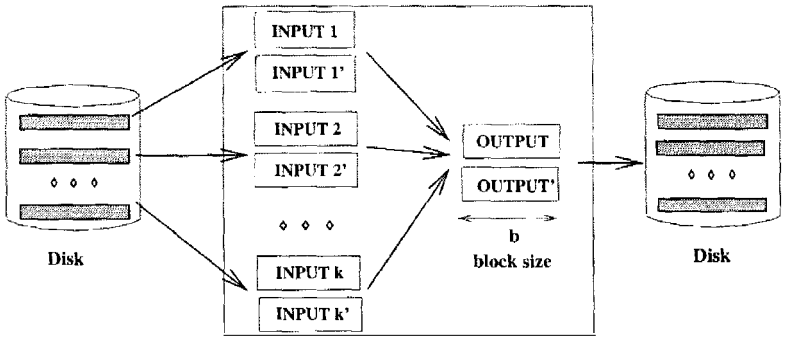


Figure 13.10 Double Buffering

13.5 USING B+ TREES FOR SORTING

Suppose that we have a B+ tree index on the (search) key to be used for sorting a file of records. Instead of using an external sorting algorithm, we could use the B+ tree index to retrieve the records in search key order by traversing the sequence set (i.e., the sequence of leaf pages). Whether this is a good strategy depends on the nature of the index.

13.5.1 Clustered Index

If the B+ tree index is clustered, then the traversal of the sequence set is very efficient. The search key order corresponds to the order in which the data records are stored, and for each page of data records we retrieve, we can read all the records on it in sequence. This correspondence between search key ordering and data record ordering is illustrated in Figure 13.11, with the assumption that data entries are *(key, rid)* pairs (i.e., Alternative (2) is used for data entries).

The cost of using the clustered B+ tree index to retrieve the data records in search key order is the cost to traverse the tree from root to the left-most leaf (which is usually less than four I/Os) plus the cost of retrieving the pages in the sequence set, plus the cost of retrieving the (say, *N)* pages containing the data records. Note that no data page is retrieved twice, thanks to the ordering of data entries being the same as the ordering of data records. The number of pages in the sequence set is likely to be much smaller than the number of data pages because data entries are likely to be smaller than typical data records. Thus, the strategy of using a dusterecl B+ tree inclex to retrieve the records in sorted order is a good one and should be used whenever such an index is ailable.

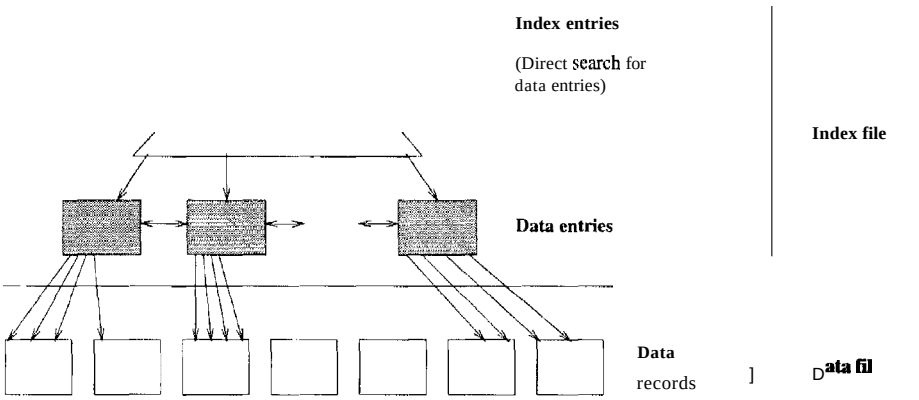


Figure 13.11 Clustered B+ Tree for Sorting

What if Alternative (1) is used for data entries? Then, the leaf pages would contain the actual data records, and retrieving the pages in the sequence set (a total of N pages) would be the only cost. (Note that the space utilization is about 67% in a B+ tree; the number of leaf pages is greater than the number of pages needed to hold the data records in a sorted file, where, in principle, 100% space utilization can be achieved.) In this case, the choice of the B+ tree for sorting is excellent!

13.5.2 Unclustered Index

What if the B+ tree index on the key to be used for sorting is unclustered? This is illustrated in Figure 13.12, with the assumption that data entries are $\langle key, rid \rangle$.

In this case each rid in a leaf page could point to a different data page. Should this happen, the cost (in disk l/Os) of retrieving all data records could equal the number of data records. That is, the worst-case cost is equal to the number of data records, because fetching each record could require a disk I/O. This cost is in addition to the cost of retrieving leaf pages of the B+ tree to get the data entries (which point to the data records).

If p is the average number of records per data page and there are N data pages, the number of data records is $p \cdot N$. If we take f to be the ratio of the size of a data entry to the size of a data record, we can approximate the number of leaf pages in the tree by $f \cdot N$. The total cost of retrieving records in sorted order

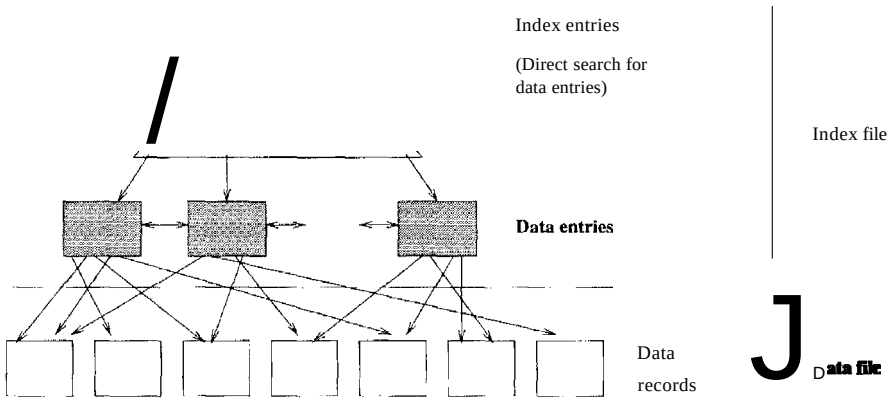


Figure 13.12 Unclustered B+ Tree for Sorting

using an unclustered B+ tree is therefore $(J + p) . N$. Since f is usually 0.1 or smaller and p is typically much larger than 10, $p . N$ is a good approximation.

In practice, the cost may be somewhat less because some rids in a leaf page lead to the same data page, and further, some pages are found in the buffer pool, thereby avoiding an I/O. Nonetheless, the usefulness of an unclustered B+ tree index for sorted retrieval highly depends on the extent to which the order of data entries corresponds and--this is just a matter of chance-to the physical ordering of data records.

We illustrate the cost of sorting a file of records using external sorting and unclustered B+ tree indexes in Figure 13.13. The costs shown for the unclustered index are worst-case numbers, based on the approximate formula $p . N$. For comparison, note that the cost for a clustered index is approximately equal to N, the number of pages of data records.

	Sorting	$p = 1$	$p = 10$	$p = 100$
100	200	100	1000	10,000
1000	2000	1000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

Figure 13.13 Cost of External Sorting ($13 = 1000$, $b = 32$) versus Unclustered Index

Keep in mind that p is likely to be doser to 100 and B is likely to be higher than 1,000 in practice. The ratio of the cost of sorting versus the cost of using an unclustered index is likely to be even lower than indicated by Figure 13.13 because the I/O for sorting is in 32-page buffer blocks, whereas the I/O for the undustered indexes is one page at a time. The value of p is determined by the page size and the size of a data record; for p to be 10, with 4KB pages, the average data record size must be about 400 bytes. In practice, p is likely to be greater than 10.

For even modest file sizes, therefore, sorting by using an unclustered index is clearly inferior to external sorting. Indeed, even if we want to retrieve only about 10--20% of the data records, for example, in response to a range query such as "Find all sailors whose rating is greater than 7," sorting the file may prove to be more efficient than using an unclustered index!

13.6 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What database operations utilize sorting? (Section 13.1)
- Describe how the *two-way merge sort* algorithm can sort a file of arbitrary length using only three main-memory pages at any time. Explain what a *run* is and how runs are created and merged. Discuss the cost of the algorithm in terms of the number of *passes* and the I/O cost per pass. (Section 13.2)
- How does the general *external merge sort* algorithm improve upon the two-way merge sort? Discuss the length of initial runs, and how memory is utilized in subsequent merging passes. Discuss the cost of the algorithm in terms of the number of passes and the I/O cost per pass. (Section 13.3)
- Discuss the use of *replacement sort* to increase the average length of initial runs and thereby reduce the number of runs to be merged. How does this affect the cost of external sorting? (Section 13.3.1)
- What is *blocked I/O?* Why is it cheaper to read a sequence of pages using blocked I/O than to read them through several independent requests? How does the use of blocking affect the external sorting algorithm, and how does it change the cost formula? (Section 13.4.1)
- What is *double buffering?* What is the motivation for using it? (Section 13.4.2)
- If we want to sort a file and there is a B+ tree with the same search key, we have the option of retrieving records in order through the index. Compare

the cost of this approach to retrieving the records in random order and then sorting them. Consider both clustered and unclustered B+ trees. What conclusions can you draw from your comparison? **(Section 13.5)**

EXERCISES

Exercise 13.1 Suppose you have a file with 10,000 pages and you have three buffer pages. Answer the following questions for each of these scenarios, assuming that our most general external sorting algorithm is used:

(a) A file with 10,000 pages and three available buffer pages.

(b) A file with 20,000 pages and five available buffer pages.

(c) A file with 2,000,000 pages and 17 available buffer pages.

1. How many runs will you produce in the first pass?
2. How many passes will it take to sort the file completely?
3. What is the total I/O cost of sorting the file?
4. How many buffer pages do you need to sort the file completely in just two passes?

Exercise 13.2 Answer Exercise 13.1 assuming that a two-way external sort is used.

Exercise 13.3 Suppose that you just finished inserting several records into a heap file and now want to sort those records. Assume that the DBMS uses external sort and makes efficient use of the available buffer space when it sorts a file. Here is some potentially useful information about the newly loaded file and the DBMS software available to operate on it:

> The number of records in the file is 4500. The sort key for the file is 4 bytes long. You can assume that rids are 8 bytes long and page ids are 4 bytes long. Each record is a total of 48 bytes long. The page size is 512 bytes. Each page has 12 bytes of control information on it. Four buffer pages are available.

1. How many sorted subfiles will there be after the initial pass of the sort, and how long will each subtile be?
2. How many passes (including the initial pass just considered) are required to sort this file?
3. What is the total I/O cost for sorting this file?
4. What is the largest file, in terms of the number of records, you can sort with just four buffer pages in two passes? How would your answer change if you had 257 buffer pages?
5. Suppose that you have a B+ tree index with the search key being the same as the desired sort key. Find the cost of using the index to retrieve the records in sorted order for each of the following cases:
   - The index uses Alternative (1) for data entries.
   - The index uses Alternative (2) and is unclustered. (You can compute the worst-case cost in this case.)

- How would the costs of using the index change if the file is the largest that you can sort in two passes of external sort with 257 buffer pages? Give your answer for both clustered and unclustered indexes.

Exercise 13.4 Consider a disk with an average seek time of lOms, average rotational delay of 5ms, and a transfer time of 1ms for a 41< page. Assume that the cost of reading/writing a page is the sum of these values (i.e., 16ms) unless a *sequence* of pages is read/written. In this case, the cost is the average seek time plus the average rotational delay (to find the first page in the sequence) plus 1ms per page (to transfer data). You are given 320 buffer pages and asked to sort a file with 10,000,000 pages.

1. Why is it a bad idea to use the 320 pages to support virtual memory, that is, to 'new' 10,000,000 4K bytes of memory, and to use an in-memory sorting algorithm such as Quicksort?
2. Assume that you begin by creating sorted runs of 320 pages each in the first pass. Evaluate the cost of the following approaches for the subsequent merging passes:
   (a) Do 31g-way merges.
   (b) Create 256 'input' buffers of 1 page each, create an 'output' buffer of 64 pages, and do 256-way merges.
   (c) Create 16 'input' buffers of 16 pages each, create an 'output' buffer of 64 pages, and do 16-way merges.
   (d) Create eight 'input' buffers of 32 pages each, create an 'output' buffer of 64 pages, and do eight-way merges.
   (e) Create four 'input' buffers of 64 pages each, create an 'output' buffer of 64 pages, and do four-way merges.

Exercise 13.5 Consider the refinement to the external sort algorithm that produces runs of length *2B* on average, where *B* is the number of buffer pages. This refinement was described in Section 11.2.1 under the assumption that all records are the same size. Explain why this assumption is required and extend the idea to cover the case of variable-length records.

PROJECT-BASED EXERCISES

Exercise 13.6 *(Note to instructors: Additional deta'ils must be provided if this exercise* is *assigned; see Appendix 30.)* Implement external sorting in Minibase.

BIBLIOGRAPHIC NOTES

Knuth's text [442] is the classic reference for sorting algorithms. Memory management for replacement sort is discussed in [471]. A number of papers discuss parallel external sorting algorithms, including [66, 71, 223, 494, 566, 647].

14

EVALUATING RELATIONAL OPERATORS

- What are the alternative algorithms for selection? Which alternatives are best under different conditions? How are complex selection conditions handled?
- How can we eliminate duplicates in projection? How do sorting and hashing approaches compare?
- What are the alternative join evaluation algorithms? Which alternatives are best under different conditions?
- How are the set operations (union, inter;section, set-difference, cross-product) implemented?
- How are aggregate operations and grouping handled?
- How does the size of the buffer pool and the buffer replacement policy affect algorithms for evaluating relational operators?
- **Key concepts:** selections, CNF; projections, sorting versus hashing; joins, block nested loops, index nested loops, sort-merge, hash; union, set-difference, duplicate elimination; aggregate operations, running information, partitioning into groups, using indexes; buffer management, concurrent execution, repeated access patterns

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*

In this chapter, we consider the implementation of individual relational operators in sufficient detail to understand how DBMSs are implemented. The discussion builds on the foundation laid in Chapter 12. We present implementation alternatives for the selection operator in Sections 14.1 and 14.2. It is instructive to see the variety of alternatives and the wide variation in per'formanee of these alternatives, for even such a simple operator. In Section 14.3, we consider the other unary operator in relational algebra, projection.

We then discuss the implementation of binary operators, beginning with joins in Section 14.4. 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 14.5. We discuss the implementation of grouping and aggregate operators, which are extensions of relational algebra, in Section 14.6. We conclude with a discussion of how buffer management affects operator evaluation costs in Section 14.7.

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 ill less depth can skip some of these alternatives without loss of continuity.

Preliminaries: Examples and Cost Calculations

We present a number of example queries using the same schema as in Chapter 12:

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 added a string field *rname* to Reserves. Intuitively, this field is the name of the person who 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 assume that each tuple of Reserves is 40 bytes lOllg, that a page can hold 100 Reserves tuples, alld that we have 1000 pages of such tuples. Similarly, we 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 Inind to understancl 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 l/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 uniformly ignore this cost in comparing alternatives.

14.1 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 14.1, which has the selection condition *rno:me='Joe'*.

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

Figure 14.1 Simple Selection Query

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 1000 l/Os, since Reserves contains 1000 pages. If only a few tuples have *rnarne= '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 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 $\sigma_{R.attr}$ **op** $_{value}(R)$ until Section 14.2, where we consider general selections. In terms of the general techniques listed in Section 12.2, the algorithms for selection use either iteration or indexing.

14.1.1 No Index, Unsorted Data

Given a selection of the form $\sigma_{R.attr}$ **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. Therefore,

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.

14.1.2 No Index, Sorted Data

Given a selection of the form *O'R.attr* **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 $\sigma_{R.attr}$ **op** *value(R).*

For example, suppose that the selection condition is *R.aUr!* > 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(l092M).* 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 14.1), the cost of the binary search is *[0921000* $\approx$ 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).

14.1.3 B+ Tree Index

If a clustereel B+ tree index is available on *R.attr,* the best strategy for selection conditions $\sigma_{R.attr}$ **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 later.

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 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 13.11 and 13.12. 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 *rnarne* < *'C%'* on the Reserves relation. Assuming that names are uniformly distributed with respect to the initial letter, for simplicity, we estimate that roughly 10% 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 *marne* 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 avoid retrieving the same page multiple times; nonetheless, the tuples to be retrieved are likely to be scattered across many more than 100 pages. Therefc)re, the use of an unclusterecl index for a range selection could be expensive; it might be cheaper to simply scan the entire relation (which is lOOn pages in our example).

14.1.4 Hash Index, Equality Selection

If a hash index is available on $R.attr$ and **op** is equality, the best way to implement the selection $_{CTR.attr}\,\mathbf{op}\,_{value}(R)$ 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 14.1. Suppose that there is an unclustered hash index on the *rname* attribute, that we have 10 buffer pages, and that 100 reservations were 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.

14.2 GENERAL SELECTION CONDITIONS

In our discussion of the selection operation thus far, we have considered selection conditions of the form $_{CTR.attr}\,\mathbf{op}\,_{value(R)}$. In general, a selection condition is a Boolean combination (Le., an expression using the logical connectives $1\backslash$ and V) of terms that have the form *attribute* **op** *constant* or *attributel* **op** *attrilmte2*. For example, if the WHERE clause in the query shown in Figure 14.1 contained the condition *R.rnarne='Joe'* AND *R.bid=r*, the equivalent algebra expression would be $_{CTR.rname='Joe'l\backslash R.bid=r(R)}$.

In Section 14.2.1, we provide a more rigorous definition of CNF, which we introduced in Section 12.2.2. We consider algorithms for applying selection conditions without disjunction in Section 14.2.2 and then discuss conditions with disjunction in Section 14.2.3.

14.2.1 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 *conjunets* that are connected through the use of the $\wedge$ operator. Each conjunct consists of one or more *terms* (of the form described previously) connected by V.[1] Conjuncts that contain V are said to be disjunctive or to contain disjunction.

As an example, suppose that we have a selection on Reserves with the condition *(day* < *8/9/02* $\wedge$ *rname* = *'Joe'*) $\vee$ *bid=5* $\vee$ *sid=3.* We can rewrite this in conjunctive normal form as *(day* < *8/9/02* $\vee$ *bid=5* $\vee$ *sid=3)* $\wedge$ *(rname* = *'Joe'V bid=5* $\vee$ *sid=3).*

We discussed when an index matches a CNF selection in Section 12.2.2 and introduced selectivity of access paths. The reader is urged to review that material now.

14.2.2 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 do 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 reduce the cardinality of the result of the selection.)
- 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, then retrieving those records whose rids are in the intersection. If additional conjuncts are present in the selection, we can apply these conjuncts to discard some of the candidate tuples from the result.

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

Intersecting rid Sets: Oracle 8 uses several techniques to do rid set intersection for selections with AND. One is to AND bit maps. 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 22.10.2). Sybase ASE does not do rid set intersection for AND selections; Sybase ASIQ does it using bitmap operations. Informix also does rid set intersection.

As an example, given the condition *day* < *8/9/02* ∧ *bid=5* ∧ *sid=,'J,* we can retrieve the rids of records that meet the condition *day* < *8/9/02* by using a B+ tree index on *day,* retrieve the rids of records that meet the condition *sid=,'J* 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* < *8/9/02* ∧ *bid=5* ∧ *sid=,'J.*

14.2.3 Selections with Disjunction

Now let us consider 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* < *8/9/02* V *rnarne='Joe').* We can retrieve tuples satisfying the condition *rname='Joe'* by using the index on *rnarne.* However, *day* < *8/9/02* requires a file scan. So we might as well do a file scan and check the condition *rname='Joe'* for each retrieved tuple. Therefore, the most selective access path in this example is a file scan.

On the other hand, if the selection condition is *(day* < *8/9/02* ∨ *rname='Joe')* ∧ *sid=,'J,* the index on *sid* matches the conjunct *sid=S.* We can use this index to find qualifying tuples and apply *day* < *8/9/02* ∨ *rname='Joe'* to just these tuples. The best access path in this example is the index on *sid* with the primary conjunct *sid=3.*

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, such as $sal < 5 \vee sal > 30$, 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 25.) (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.

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 $(day < 8/9/02 \vee rname='Joe')$ and we have B+ tree indexes on *day* and *rname*, we can retrieve all tuples such that $day < 8/9/02$ 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 $day < 8/9/02$ 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 $(day < 8/9/02 \vee rname='Joe')$.

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

14.3 THE PROJECTION OPERATION

Consider the query shown in Figure 14.2. The optimizer translates this query into the relational algebra expression $\pi_{sid,bid}Reserves$. In general the projection operator is of the form $\pi_{attr1,attr2,\ldots,attrm}(R)$. To implement projection, we have

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

Figure 14.2 Simple Projection Query

to do the following:

1. Remove unwanted attributes (i.e., those not specified in the projection).
2. Eliminate any duplicate tuples 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 12.2, 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 14.3.4.

14.3.1 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 $a(M)$. (The exact value of T depends on the number of fields retained and the sizes of these fields.) The second step costs *O(TlogT)* (which is also *0(MlogA1)*, 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 14.2. We can scall Reserves at a cost of 1000 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 we have 20 buffer pages. We can sort the temporary relation in two passes at a cost of $2 \cdot 2 \cdot 250 = 1000$ I/Os. The scan required in the third step costs an additional 250 I/Os. The total cost is 2500 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 presented in Chapter 13. 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 Rand write out *(T/!vI)* . 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 discussed in Section 13.3.1.)
- We can eliminate duplicates during the merging passes. In fact, this modification reduces the cost of the merging passes since fewer tuples are written out in each pass. (Most of the duplicates are eliminated in the very first merging pass.)

Let us consider our example again. In the first pass we scan Reserves, at a cost of 1000 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.

14.3.2 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 14.3.

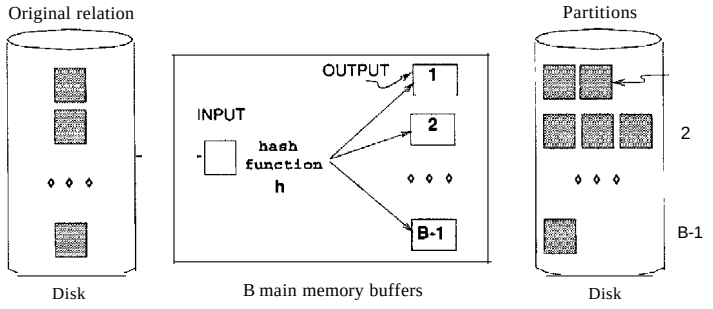


Figure 14.3 Partitioning Phase of Hash-Based Projection

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 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 $\frac{T}{B-1} \cdot f$; 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 $\frac{T}{B-1} \cdot f$ to avoid partition overflow. This observation implies that we require approximately $B > \sqrt{f \cdot T}$ 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 $M + T$ l/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 14.2), this cost is $1000 + 2 \cdot 250 = 1500$ l/Os.

14.3.3 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 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

Projection in Commercial Systems: Informix uses hashing. IBMDB2, Oracle 8, and Sybase ASE use sorting. Microsoft SQL Server and Sybase ASIQ implement both hash-based and sort-based algorithms.

same I/O cost. Sorting takes two passes. In the first pass, we read M pages of the original relation and write 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.

14.3.4 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 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, called an *index-only scan,* and was discussed in Sections 8.5.2 and 12.3.2. **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 15.4.1.

14.4 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, one of the most useful operations in relational algebra, is the primary means of combining information from two or more relations.

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 25). 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, 80rt-merge, hash join, and a technique called *hash teams.* Informix supports block nested loops, index nested loops, and hybrid hash join.

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.

We now 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 12.2.

The remaining join algorithms avoid enumerating the cross-product. They are instances of the indexing and partitioning techniques mentioned in Section 12.2. 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 $R_i = Sj$, using positional notation. (If we have more complex join conditions, the basic idea behind each algorithm remains essentially the same. \Ve discuss the details in Section 14.4.4.) We assmne M pages in R with p_R tuples per page and N

pages in S with PS tuples per page. We use R and S in our presentation of the algorithms, and the Reserves and Sailors relations for specific examples.

14.4.1 Nested Loops Join

The simplest join algorithm is a tuple-at-a-time nested loops evaluation. 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 $PR \cdot M$ times, and each scan costs N I/Os. Thus, the total cost is $M + PR \cdot M \cdot N$.

```
foreach tuple r ∈ R do
    foreach tuple s ∈ S do
        if ri==Sj then add (r, s) to result
```

Figure 14.4 Simple Nested Loops Join

Suppose we choose R to be Reserves and S to be Sailors. The value of M is then 1,000, PR is 100, and N is 500. The cost of simple nested loops join is $1000 + 100 \cdot 1000 \cdot 500$ page I/Os (plus the cost of writing out the result; we remind the reader again that we uniformly ignore this component of the cost). The cost is staggering: $1000 + (5 \cdot 10^7)$ I/Os. Note that each I/O costs about lams 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 (r, s) for all qualifying tuples $r \in R\text{-}page$ and $s \in S\text{-}page$. This way, the cost is M to scan R, as before. However, S is scanned only M times, and so the total cost is $M + M \cdot N$. Thus, the page-at-a-time refinement gives us an improvement of a factor of PR. In the example join of the Reserves and Sailors relations, the cost is reduced to $1000 + 1000 \cdot 500 = 501,000$ 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 $500 + 500 \cdot 1000 = 500,500$ 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 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 E 5, we check R and output a tuple $(1', s)$ for qualifying tuples s (i.e., $r_i = sj$). The second extra buffer page)s 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 have too little 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 5 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 $(1', s)$, where r E *R-block*, s E *S-page*, and $r_i = sj$, 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 = sj) 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 14.5. Buffer usage in this algorithm is illustrated in Figl.lre 14.6.

```
foreach block of B-2 pages of R do
    foreach page of 5 do {
        for all matching in-memory tuples r E R-block and s E S-page,
        add (1', s) to result
    }
```

Figure 14.5 Block Nested Loops Join

The cost of this strategy is M 1/0s for reading in R (which is scanned only once). 5 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 l/Os. The total cost is thus $M + N \cdot \lceil \frac{M}{B-2} \rceil$.

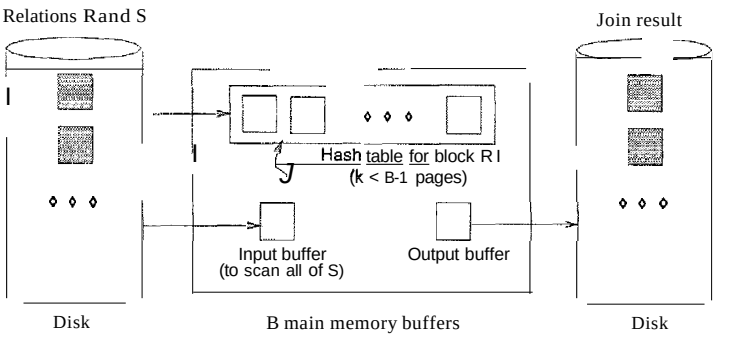


Figure 14.6 Buffer Usage in Block Nested Loops Join

Consider the join of the Reserves and Sailors relations. Let us choose Reserves to be the outer relation R and assume 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 1000 I/Os. For each 100-page block of Reserves, we have to scan Sailors. Therefore, we perform 10 scans of Sailors, each costing 500 I/Os. The total cost is $1000 + 10 \cdot 500 = 6000$ I/Os. If we had only enough buffers to hold 90 pages of Reserves, we would have to scan Sailors $\lceil 1000/90 \rceil = 12$ times, and the total cost would be $1000 + 12 \cdot 500 = 7000$ 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 $500 + 5 \cdot 1{,}000 = 5500$ 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 $500 + 6 \cdot 1000 = 6500$ 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 13 in the context of sorting) can also be used, but we do 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 we have a suitable index on *S;* Figure 14.7 describes the index nested loops join algorithm.

```
foreach tuple r E R do
    foreach tuple s E S where r_i == s_j
        add ⟨r, s) to result
```

Figure 14.7 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 Rand 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---4 l/Os. If the index is a hash index, the cost to find the appropriate bucket is 1-2 1/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 l/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 1000. There are 100 . 1000 tuples in Reserves. For each of these tuples, retrieving the index page containing the rid of the matching Sailors tuple costs 1.2 l/Os (on average); in addition, we have to retrieve the Sailors page containing the qualifying tuple. Therefore, we have

2This is a typical cost for hash-based indexes,

100,000 ·(1 + 1.2) l/Os to retrieve matching Sailors tuples. The total cost is 221,000 l/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 l/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 may have no reservations or 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 l/Os on average. The total cost thus far is 500 + 40,000 . 1.2 = 48,500 l/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 l/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 l/Os for retrieving qualifying tuples. Therefore, the total cost can vary from 48,500+40,000=88,500 to 48,500+100,000=148,500 l/Os. Assuming 10ms per I/O, this would take about 15 to 25 minutes.

So, 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.

14.4.2 Sort-Merge Join

The basic idea behind the sort-merge **join** algorithm is to *sort* both relations on the join attribute and 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 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 13 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 *Rand S)* looking for qualifying tuples (i.e., tuples *Tr* in *Rand Ts* in *S* such that $Tr_i = Tsj$). 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 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 $Tr_i = TSj'$

When we find tuples *Tr* and *Ts* such that $Tri = Tsj$, 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 *(r, s).* We then resume scanning Rand *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 14.8. 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 *Tri* is defined to evaluate to false.

We illustrate sort-merge join on the Sailors and Reserves instances shown in Figures 14.9 and 14.10, 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.

```
proc smjoin(R, B, 'Ri = Sj')

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

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

while Tr ≠ eof and Gs ≠ eof do {

    while Tri < GSj do
        Tr = next tuple in Rafter Tr;                   // continue scan of R

    while Tri > GSj do
        Gs = next tuple in S after Gs                   // continue scan of B

    Ts = Gs;                                    // Needed in case Tri ≠ GSj
    while Tri == GSj do {                       // process current R partition
        Ts = Gs;                                        // reset S partition scan
        while TSj == Tri do {                           // process current R tuple
            add (Tr, Ts) to result;                     // output joined tuples
            Ts = next tuple in S after Ts;} // advance S partition scan
        Tr = next tuple in Rafter Tr;                   // advance scan of R
        }                                       // done with current R partition

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

    }
```

Figure 14.8 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 14.9 An Instance of Sailors

sid	bid	day	rname
28	103	12/04/96	guppy
28	103	11/03/96	}'uppy
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 14.10 An Instance of Reserves

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=7'name.* Now, both relations contain more than one tuple in the partition with *sname=mame='lubber'.* The tuples with *rname='lubber'* in Reserves have to be scanned for each Sailors tuple with *sname='lubber'.*

Cost of Sort-Merge Join

The cost of sorting R is $O(MlogM)$ ancl 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 $2 \cdot 2 \cdot 1000 = 4000$ l/Os. Similarly, we can sort Sailors in two passes, at a cost of $2 \cdot 2 \cdot 500 = 2000$ l/Os. In addition, the seconcl phase of the sort-merge join algorithm requires an additional scan of both relations. Thus the total cost is $4000 + 2000 + 1000 + 500 = 7500$ l/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 7500 l/Os. However, the cost of the block nested loops join algorithm is more than 15,000 l/Os. On the other hand, if we have 300 buffer pages, the cost of the sort-merge join remains at 7500 I/Os, whereas the cost of the block nested loops join drops to 2500 l/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 partitiOli 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 many pages

of Reserves tuples are 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 the complete second relation for each *tuple* in the first relation, and the number of I/Os is $O(M \cdot N)$ 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 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 Rand 5. 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 Rand 5. We allocate one buffer page for each run of R *and* one for each run of 5. We then merge the runs of R (to generate the sorted version of R), merge the runs of 5, and merge the resulting Rand 5 streams as they are generated; we apply the join condition as we merge the Rand 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 13.3.1 we can produce sorted runs of size approximately $2 \cdot B$ for both Rand 5. 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 Rand S in the first pass and reading Rand 5 in the second pass. The total cost is thus $3 \cdot (M + N)$. In our example, the cost goes down from 7500 to 4500 I/Os.

Blocked Access and Double-Buffering

The blocked I/O and double-buffering optimizations, discussed in Chapter 13 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 do not discuss these refinements.

14.4.3 Hash Join

The hash join algorithm, like the sort-merge join algorithm, identifies partitions in Rand *S* in a partitioning phase and, in a subsequent probing phase, compares tuples in an *R* partition only with tuples in the corresponding *5* 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 14.3. The probing (sometimes called matching) phase is illustrated in Figure 14.11.

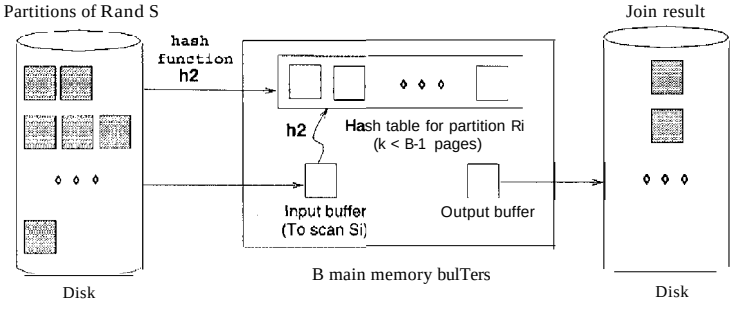


Figure 14.11 Probing Phase of Hash Join

The idea is to hash *both* relations on the join attribute, using the *same* hash function *h.* If we hash each relation (ideally 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 Rand *S* tuples again. Thus, once Rand *S* are partitioned, we can perform the join by reading in *R* and *5* just once, provided 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 ha'3h function *h2* that is different from *h* (since *h2* is intended to distribute tuples in a partition based on *h),* 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 14.12. (There are several variants on this idea; this version 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 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 + 1000) = 4500 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 ∈ R do
    read r and add it to buffer page h(ri);        // flushed as page fills

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

// Probing phase
for l = 1, ..., k do {

    // Build in-memory hash table for Rz, using h2
    foreach tuple r ∈ partition Rz do
        read r and insert into hash table using h2(ri) ;

    // Scan Sz and probe for matching Rz tuples
    foreach tuple s ∈ partition Sz do {
        read s and probe table using h2(sj);
        for matching R tuples r, output ⟨r, s⟩ };

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

Figure 14.12 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,

8) into k partitions, we need at least k output buffers and one input buffer. Therefore, 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 $\frac{f \cdot M}{B-1}$, 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 8 partition and an output buffer. Therefore, we require $B > \frac{f \cdot M}{B-1} + 2$. *We need approximately* $B > \sqrt{f} \cdot M$ 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 is somewhat larger than $\frac{M}{B-1}$, and the number of buffer pages required is a little more than $B > \sqrt{f} \cdot M$. 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 8 partition. That is, we first divide the Rand 8 partitions into subpartitions. Then, we join the subpartitions pairwise. All subpartitions of R 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 $B > \sqrt{f} \cdot M$. If more memory is available, a variant of hash join called **hybrid hash join** offers better performance. Suppose that $B > f \cdot (lYI/k)$, 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, 5) into k partitions, we need k output buHers and one input buffer; that is, $k + 1$ pages. This leaves us with $B - (k + 1)$ extra pages during the partitioning phase.

Suppose that $B - (k + 1) > f \cdot (M/k)$. 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 do not write this partition to disk. Similarly, while partitioning 8, rather than write out the tuples in the first partition of 5, 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 1000 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 if we assume that the partitions are of equal size. We then scan S and write out one partition; the cost is 1000 + 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 + 1500 + 750 = 3000. In contrast, the cost of hash join is 4500.

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 llleans that we read R only once, to build this hash table, and read S once, to probe the R hash table. The cost is 500 + 1000 = 1500.

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 saved in this case by using the hash join algorithm in comparison to a block nested loops join is illustrated in Figure 14.13. 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.

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

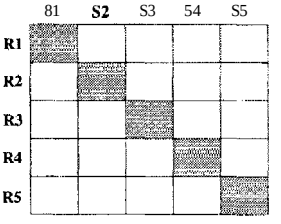


Figure 14.13 Hash Join Vs. Block Nested Loops for Large Relations

We note that this picture is rather simplistic. It does not capture the costs of scanning R in the block nested loops join and the partitioning phase in the 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 $3(M + N)$ l/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 14.4.2. 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.

14.4.4 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.s'id 1\ R.rname=S.sname:*

- For index nested loops join, we can **build** an index on Reserves on the combination of fields *(R.sid, R.rname)* 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.marne*. (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 ⟨*sid, rname*⟩ and Sailors on the combination of fields ⟨*sid, snarne)*. Similarly, for hash join, we partition on these combinations of fields.
- The other join algorithms we discussed are essentially unaffected.

If we have an inequality comparison, for example, a join of Reserves R and Sailors 5 with the join condition *R.rnarne* < *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 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 no one join algorithm 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.

14.5 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 cr08S-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 acldress 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 discussecl in Section 14.2 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 12.2.

14.5.1 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.

14.5.2 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.

14.6 AGGREGATE OPERATIONS

The SQL query shown in Figure 14.14 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 14.14 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 14.15. The cost of this operation is the cost of scanning all Sailors tuples.

Aggregate Operation	*Running Inforrniation*
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 14.15 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 14.14, 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 12.2.

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 *(gTOuping-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 cOlnplete, 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 *gTOuping-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.

14.6.1 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 15.4.1.

14.7 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 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; therefore, 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 discussed in Section 9.4.1. 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 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.

14.8 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Consider a simple selection query of the form $\sigma_{R.attr\ \mathbf{op}\ value}(R)$. What are the alternative access paths in each of these cases: (i) there is no index and the file is not sorted, (ii) there is no index but the file is sorted. (Section 14.1)

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

- If a B+ tree index matches the selection condition, how does clustering affect the cost? Discuss this in terms of the selectivity of the condition. **(Section 14.1)**
- Describe *conjunctive normal form* for general selections. Define the terms *conjunct* and *disjunct*. Under what conditions does a general selection condition match an index? **(Section 14.2)**
- Describe the various implementation options for general selections. **(Section 14.2)**
- Discuss the use of sorting versus hashing to eliminate duplicates during projection. **(Section 14.3)**
- When can an index be used to implement projections, without retrieving actual data records? When does the index additionally allow us to eliminate duplicates without sorting or hashing? **(Section 14.3)**
- Consider the join of relations Rand 5. Describe *simple nested loops join* and *block nested loops join.* What are the similarities and differences? How does the latter reduce I/O costs? Discuss how you would utilize buffers in block nested loops. **(Section 14.4.1)**
- Describe *index nested loops join.* How does it differ from block nested loops join? **(Section 14.4.1)**
- Describe sort-merge join of Rand 5. What join conditions are supported? What optimizations are possible beyond sorting both Rand 5 on the join attributes and then doing a merge of the two? In particular, discuss how steps in sorting can be combined with the merge pass. **(Section 14.4.2)**
- What is the idea behind *hash join?* What is the additional optimization in *hybrid hash join?* **(Section 14.4.3)**
- Discuss how the choice of join algorithm depends on the number of buffer pages available, the sizes of Rand 5, and the indexes available. Be specific in your discussion and refer to cost formulas for the I/O cost of each algorithm. **(Sections** 14.12 Section **14.13)**
- How are general join conditions handled? **(Section 14.4.4)**
- \Vhy are the set operations $R \cap 5$ and R x S special cases of joins? What is the similarity between the set operations $Ru5$ and R - 5? **(Section 14.5)**
- Discuss the use of sorting versus hashing in implementing $Ru5$ and R - S. Compare this with the ilnplementation of projection. **(Section 14.5)**
- Discuss the use of *running information* in implementing aggregate operations. Discuss the use of sorting versus hashing for dealing with grouping. **(Section 14.6)**

- Under what conditions can we use an index to implement aggregate operations without retrieving data records? Under what conditions do indexes allow us to avoid sorting or hashing? (Section 14.6)
- Using the cost formulas for the various relational operator evaluation algorithms, discuss which operators are most sensitive to the number of available buffer pool pages. How is this number influenced by the number of operators being evaluated concurrently? (Section 14.7)
- Explain how the choice of a good buffer pool replacement policy can influence overall performance. Identify the patterns of access in typical relational operator evaluation and how they influence the choice of replacement policy. (Section 14.7)

EXERCISES

Exercise 14.1 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 on 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 14.2 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 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 approaches (or combination thereof) 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.*
- Use a clustered B+ tree index on attributes *(R.a, R.b).*
- Use a linear hashed index on attributes *(R.a, R.b).*
- Use an unclustered B+ tree index on attribute *R.b.*

1. $O"u<50,OOOAb<50,ooo(R)$
2. $\sigma_{u=50,OOOAb<50,OOO}(R)$
3. $O"u>50,OOOAb=50,ooo(R)$
4. $\sigma_{u=50,OOOi\backslash a=50,OIO}(R)$
5. $\sigma_{a\neq 50,000\wedge b=50,000}(R)$
6. $\sigma_{a<50,OOOvb=50,OOO}(R)$

Exercise 14.3 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 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 are 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 14.4 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 2000 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 does the join of R and S produce, at most, and how many pages are 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 were told that *R.a* is a foreign key that refers to *S.b?*

Exercise 14.5 Consider the join of R and S described in Exercise 14.1.

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 2000 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 2000 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 14.6 Answer each of the questions—if some question is inapplicable, explain why--in Exercise 14.1 again but using the following information about Rand 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 14.7 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 14.8 *(Note to instructors: Additional details must be provided if this exen:ise* is *assigned; see Appendix 30.)* Implement the various join algorithms described in this chapter in Minibase. (As additional exercises, you Inay 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 [101]. The implementation techniques used in PRTV, which utilized relational algebra transformations and a form of multiple-query optimization, are discussed in [358]. The techniques used for aggregate operations in Ingres are described in [246]. [324] 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 [110], [222], [325], and [677]. Duplicate elimination is discussed in [99]. [277] discusses secondary storage access patterns arising in join implementations. Parallel algorithms for implementing relational operations are discussed in [99, 168, 220, 224, 233, 293,534].

15

A TYPICAL RELATIONAL QUERY OPTIMIZER

- How are SQL queries translated into relational algebra? As a consequence, what class of relation algebra queries does a query optimizer concentrate on?
- What information is stored in the system catalog of a DBMS and how is it used in query optimization?
- How does an optimizer estimate the cost of a query evaluation plan?
- How does an optimizer generate alternative plans for a query? What is the space of plans considered? What is the role of relational algebra equivalences in generating plans?
- How are nested SQL queries optimized?
- Key concepts: SQL to algebra, query block; system catalog, data dictionary, metadata, system statistics, relational representation of catalogs; cost estimation, size estimation, reduction factors; histograms, equiwidth, equidepth, compressed; algebra equivalences, pushing selections, join ordering; plan space, single-relation plans, multi-relation left-deep plans; enumerating plans, dynamic programming approach, alternative approaches

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

-John Lennon

In this chapter, we present a typical relational query optimizer in detail. We begin by discussing how SQL queries are converted into units called *blocks*

and how blocks are translated into (extended) relational algebra expressions (Section 15.1). The central task of an optimizer is to find a good plan for evaluating such expressions. Optimizing a relational algebra expression involves two basic steps:

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

We discuss how to use system statistics to estimate the properties of the result of a relational operation, in particular result sizes, in Section 15.2. After discussing how to estimate the cost of a given plan, we describe the space of plans considered by a typical relational query optimizer in Sections 15.3 and 15.4. We discuss how nested SQL queries are handled in Section 15.5. We briefly discuss some of the influential choices made in the System R query optimizer in Section 15.6. We conclude with a short discussion of other approaches to query optimization in Section 15.7.

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

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

As in Chapter 14, we assume that each tuple of Reserves is 40 bytes long, that a page can hold 100 Reserves tuples, and that we have 1000 pages of such tuples. Similarly, we 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.

15.1 TRANSLATING SQL QUERIES INTO ALGEBRA

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

15.1.1 Decomposition of a Query into Blocks

When a user submits an SQL query, the query is parsed into a collection of query blocks and then passed on to the query optimizer. A **query block**

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

Figure 15.1 Sailors Reserving Red Boats

(or simply block) is an SQL query with no nesting and exactly one SELECT clause and one FROM clause and at most one WHERE clause, GROUP BY clause, and HAVING clause. The WHERE clause is assumed to be in conjunctive normal form, as per the discussion in Section 14.2. We use the following query as a running example:

For each 8ailor' with the highe8t mting (oveT all sailors) and at least two reservations for Ted boats, find the sailoT id and the earliest date on which the sailor has a TeseTvat:ion for a red boat.

The SQL version of this query is shown in Figure 15.1. This query has two query blocks. The nested block is:

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

The nested block computes the highest sailor rating. The outer block is shown in Figure 15.2. Every SQL query can be decomposed into a collection of query blocks without nesting.

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

Figure 15.2 Outer Block of Red Boats Query

The optimizer examines the system catalogs to retrieve information about the types and lengths of fields, statistics about the referenced relations, and the access paths (indexes) available for them. The optimizer then considers each query block and chooses a query evaluation plan for that block. We focus 1nostly on optimizing a single query block and defer a discussion of nested queries to Section 15.5.

15.1.2 A Query Block as a Relational Algebra Expression

The first step in optimizing a query block is to express it as a relational algebra expression. For uniformity, let us assume that GROUP BY and HAVING are also operators in the extended algebra used for plans and that aggregate operations are allowed to appear in the argument list of the projection operator. The meaning of the operators should be clear from our discussion of SQL. The SQL query of Figure 15.2 can be expressed in the extended algebra as:

$$\pi_{S.sid,MIN(R.day)}($$
$$HAVINGcoUNT(*»2($$
$$GROUP\ BYs._{sid}($$
$$\sigma_{S.sid=R.sid\wedge R.bid=B.bidAB.coloT='Ted'AS.rating=value\_from\_nested\_block}($$
$$Sailors \times Reserves \times Boats))))$$

For brevity, we used *S*, *R*, and *B* (rather than Sailors, Reserves, and Boats) to prefix attributes. Intuitively, the selection is applied to the cross-product of the three relations. Then the qualifying tuples are grouped by *S.sid*, and the HAVING clause condition is used to discard some groups. For each remaining group, a result tuple containing the attributes (and count) mentioned in the projection list is generated. This algebra expression is a faithful summary of the semantics of an SQL query, which we discussed in Chapter 5.

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

The alternative plans examined by a typical relational query optimizer can be understood by recognizing that *a query is essentially treated as a* $\sigma\pi\times$ *algebm expression*, with the remaining operations (if any, in a given query) carried out on the result of the $\sigma\pi\times$ expression. The $\sigma\pi\times$ expression for the query in Figure 15.2 is:

$$\pi_{S.sid,R.day}($$
$$\sigma_{S.sid=R.sidAR.b'id=B.bid\wedge B.color='red'\wedge S.rating=value\_fTO/lLTlcste(Lblock}($$
$$Sailors \times Reserves \times Boats))$$

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

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

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

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

Some optimizations are possible if the FROM clause contains just one relation and the relation has some indexes that can be used to carry out the grouping operation. We discuss this situation further in Section 15.4.1.

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

15.2 ESTIMATING THE COST OF A PLAN

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

1. For each node in the tree, we must *estimate the cost* of performing the corresponding operation. Costs are affected significantly by whether pipelining is used or temporary relations are created to pass the output of an operator to its parent.
2. For each node in the tree, we must *estimate the size of the result* and whether it is sorted. This result is the input for the operation that corresponds to the parent of the current node, and the size and sort order in turn affect the estimation of size, cost, and sort order for the parent.

We discussed the cost of implementation techniques for relational operators in Chapter 14. As we saw there, estimating costs requires knowledge of various

parameters of the input relations, such as the number of pages and available indexes. Such statistics are maintained in the DBMS's system catalogs. In this section, we describe the statistics maintained by a typical DBMS and discuss how result sizes are estimated. As in Chapter 14, we use the number of page I/Os as the metric of cost and ignore issues such as blocked access, for the sake of simplicity.

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

15.2.1 Estimating Result Sizes

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

Consider a query block of the form:

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

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

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

- *column* = *value*: For a term of this form, the reduction factor can be approximated by $\frac{1}{Keys(I)}$ if there is an index I on *column* for the relation in question. This formula assumes uniform distribution of tuples among the

index key values; this uniform distribution assumption is frequently made in arriving at cost estimates in a typical relational query optimizer. If there is no index on *col'umn,* the System R optimizer arbitrarily assumes that the reduction factor is $\frac{1}{10}$. Of course, it is possible to maintain statistics such as the number of distinct values present for any attribute whether or not there is an index on that attribute. If such statistics are maintained, we can do better than the arbitrary choice of $\frac{1}{10}$.

- *column1 = column2:* In this case the reduction factor can be approximated by $\frac{1}{MAX(NKeys(I1),NKeys(I2))}$ if there are indexes I1 and 12 on *column1* and *column2,* respectively. This formula assumes that each key value in the smaller index, say, I1, has a matching value in the other index. Given a value for *columnl,* we assume that each of the $NKeys(I2)$ values for *column2* is equally likely. Therefore, the number of tuples that have the same value in *column2* as a given value in *columni* is $\frac{1}{NKeys(I2)}$. If only one of the two columns has an index *I,* we take the reduction factor to be $\frac{1}{NKeys(I)}$; if neither column has an index, we approximate it by the ubiquitous $\frac{1}{10}$. These formulas are used whether or not the two columns appear in the same relation.
- *column > value:* The reduction factor is approximated by $\frac{High(I) - value}{High(I) - Low(I)}$ if there is an index I on *column.* If the column is not of an arithmetic type or there is no index, a fraction less than half is arbitrarily chosen. Similar formulas for the reduction factor can be derived for other range selections.
- *column* IN *(list of values):* The reduction factor is taken to be the reduction factor for *column = value* multiplied by the number of items in the list. However, it is allowed to be at most half, reflecting the heuristic belief that each selection eliminates at least half the candidate tuples.

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

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

To summarize, regardless of the plan chosen, we can estimate the size of the final result by taking the product of the sizes of the relations in the FROM clause

Estimating Query Characteristics: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all usehistograms to estimate query characteristics such as result size and cost. As an example, Sybase ASE uses one-dimensional, equidepth histograms with some special attention paid to high frequency values, so that their count is estimated accurately. ASE also keeps the average count of duplicates for each prefix of an index to estimate correlations between histograms for composite keys (although it does not maintain such histograms). ASE also maintains estimates of the degree of clustering in tables and indexes. IBM DB2, Informix, and Oracle also use one-dimensional equidepth histograms; Oracle automatically switches to maintaining a count of duplicates for each value when there are few values in a column. Microsoft SQL Server uses one-dimensional equiarea histograms with some optimizations (adjacent buckets with similar distributions are sometimes combined to compress the histogram). In SQL Server, the creation and maintenance of histograms is done automatically with no need for user input.

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

and the reduction factors for the terms in the WHERE clause. We can similarly estimate the size of the result of each operator in a plan tree by using reduction factors, since the subtree rooted at that operator's node is itself a query block.

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

Improved Statistics: Histograms

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

Estimates can be improved considerably by maintaining more detailed statistics than just the low and high values in the index *I*. Intuitively, we want to approximate the distribution of key values *I* as accurately as possible. Consider the two distributions of values shown in Figure 15.3. The first is a nonuniform distribution *D* of values (say, for an attribute called *age).* The *frequency* of a value is the number of tuples with that *age* value; a distribution is represented by showing the frequency for each possible *age* value. In our example, the lowest *age* value is 0, the highest is 14, and all recorded *age* values are integers in the range 0 to 14. The second distribution approximates *D* by assuming that each *age* value in the range a to 14 appears equally often in the underlying collection of tuples. This approximation can be stored compactly because we need to record only the low and high values for the *age* range (0 and 14 respectively) and the total count of all frequencies (which is 45 in our example).

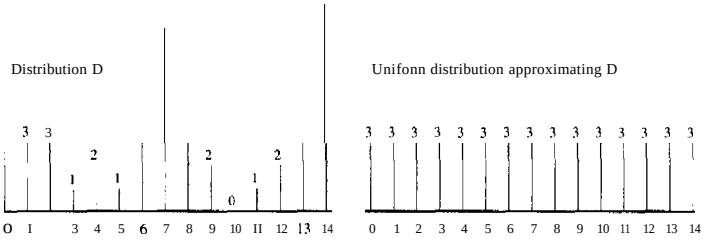


Figure 15.3 Uniform vs. Nonuniform Distributions

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

A histogram is a data structure maintained by a DBMS to approximate a data distribution. In Figure 15.4, we show how the data distribution from Figure 15.3 can be approximated by dividing the range of *age* values into subranges called buckets, and for each bucket, counting the number of tuples with *age* values within that bucket. Figure 15.4 shows two different kinds of histograms, called *equiwidth* and *equidepth,* respectively.

Consider the selection query $age > 13$ again and the first (equiwidth) histogram. We can estimate the size of the result to be 5 because the selected range includes a third of the range for Bucket 5. Since Bucket 5 represents a total of 15 tuples, the selected range corresponds to $\frac{1}{3} \cdot 15 = 5$ tuples. As this example shows, we a..ssume that the distribution *within* a histogram bucket is uniform. Therefore, when we simply maintain the high and low values for index

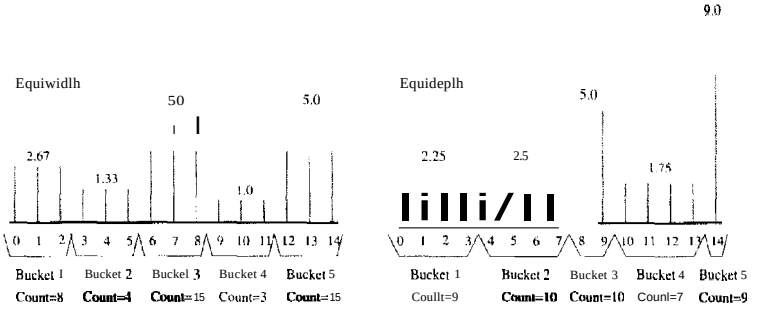


Figure 15.4 Histograms Approximating Distribution *D*

I, we effectively use a 'histogram' with a single bucket. Using histograms with a small number of buckets instead leads to much more accurate estimates, at the cost of a few hundred bytes per histogram. (Like all statistics in a DBMS, histograms are updated periodically rather than whenever the data is changed.)

One important question is how to divide the value range into buckets. In an equiwidth histogram, we divide the range into subranges of equal size (in terms of the *age* value range). We could also choose subranges such that the number of tuples within each subrange (i.e., bucket) is equal. Such a histogram, called an equidepth histogram, is also illustrated in Figure 15.4. Consider the selection *age* > 13 again. Using the equidepth histogram, we are led to Bucket 5, which contains only the *age* value 15, and thus we arrive at the exact answer, 9. While the relevant bucket (or buckets) generally contains more than one tuple, equidepth histograms provide better estimates than equiwidth histograms. Intuitively, buckets with very frequently occurring values contain fewer values, and thus the uniform distribution assumption is applied to a smaller range of values, leading to better approximations. Conversely, buckets with mostly infrequent values are approximated less accurately in an equidepth histogram, but for good estimation, the frequent values are important.

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

15.3 RELATIONAL ALGEBRA EQUIVALENCES

In this section, we present several equivalences among relational algebra expressions; and in Section 15.4, we discuss the space of alternative plans considered by a optimizer.

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

15.3.1 Selections

Two important equivalences involve the selection operation. The first one involves cascading of selections:

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

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

The second equivalence states that selections are commutative:

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

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

15.3.2 Projections

The rule for cascading projections says that successively elilninating columns from a relation is equivalent to sirnply eliminating all but the columns retained

by the final projection:

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

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

15.3.3 Cross-Products and Joins

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

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

$$Rx8 \qquad 8xR$$

$$RN8$$

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

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

$$R \times (8 \times T) \qquad (R \times 8) \times T$$

$$RN\,(8NT) \qquad (R \bowtie 8)\,NT$$

Thus we can either join Rand 8 first and then join T to the result, or join 8 and T first and then join R to the result. The intuition behind associativity of cross-products is that, regardless of the order in which the three relations are considered, the final result contains the same columns. Join associativity is based on the same intuition, with the additional observation that the selections specifying the join conditions can be cascaded. Thus the same rows appear in the final result, regardless of the order in which the relations are joined.

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

$$R \bowtie (8 \bowtie T) \equiv (T \bowtie R) \bowtie 8$$

From commutativity, we have:

$$RN\,(8NT) \qquad RN\,(TN\,8)$$

From associativity, we have:

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

Using commutativity again, we have:

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

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

15.3.4 Selects, Projects, and Joins

Some important equivalences involve two or more operators.

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

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

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

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

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

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

$$\sigma_c(R \text{ x } S) \qquad \sigma_c(R) \text{ x } S$$
$$\sigma_c(R \bowtie S) \qquad \sigma_c(R) \bowtie S$$

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

In general, a selection σ_c on *R* x *S* can be replaced by a cascade of selections σ_{c_1}, σ_{c_2}, and σ_{c_3} such that c1 involves attributes of both R and *S*, c2 involves only attributes of *R*, and c_3 involves only attributes of *S*:

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

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

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

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

$$\sigma_{C1}(\sigma_{C2}(R) \times \sigma_{C3}(S))$$

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

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

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

where a1 is the subset of attributes in *a* that appear in *R*, and *a2* is the subset of attributes in *a* that appear in *S*. We can also **commute** a projection with a join if the join condition involves only attributes retained by the projection:

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

where a1 is the subset of attributes in *a* that appear in *R*, and *a2* is the subset of attributes in *a* that appear in *S*. Further, every attribute mentioned in the join condition c must appear in *a*.

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

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

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

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

15.3.5 Other Equivalences

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

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

Figure 15.5 A Single-Relation Query

15.4 ENUMERATION OF ALTERNATIVE PLANS

We now come to an issue that is at the heart of an optimizer, namely, the space of alternative plans considered for a given query. Given a query, an optimizer essentially enumerates a certain set of plans and chooses the plan with the least estimated cost; the discussion in Section 12.1.1 indicated how the cost of a plan is estimated. The algebraic equivalences discussed in Section 15.3 form the basis for generating alternative plans, in conjunction with the choice of implementation technique for the relational operators (e.g., joins) present in the query. However, not all algebraically equivalent plans are considered, because doing so would make the cost of optimization prohibitively expensive for all but the simplest queries. This section describes the subset of plans considered by a typical optimizer.

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

15.4.1 Single-Relation Queries

If the query contains a single relation in the FROM clause, only selection, projection, grouping, and aggregate operations are involved; there are no joins. If we have just one selection or projection or aggregate operation applied to a relation, the alternative implementation techniques and cost estimates discussed in Chapter 14 cover all the plans that must be considered. We now consider how to optimize queries that involve a combination of several such operations, using the following query as an example:

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

The SQL version of this query is shown in Figure 15.5. Using the extended algebra notation introduced in Section 15.1.2, we can write this query as:

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

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

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

We are now ready to discuss the plans that an optimizer would consider. The main decision to be made is which access path to use in retrieving Sailors tuples. If we considered only the selections, we would simply choose the most selective access path, based on which available indexes *match* the conditions in the WHERE clause (as per the definition in Section 14.2.1). Given the additional operators in this query, we must also take into account the cost of subsequent sorting steps and consider whether these operations can be performed without sorting by exploiting some index. We first discuss the plans generated when there are no suitable indexes and then examine plans that utilize some index.

Plans without Indexes

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

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

The resulting tuples are then sorted according to the GROUP BY clause (in the example query, on *rating*), and one answer tuple is generated for each group that meets the condition in the HAVING clause. The computation of the aggregate functions in the SELECT and HAVING clauses is done for each group, using one of the techniques described in Section 14.6.

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

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

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

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

Plans Utilizing an Index

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

1. Single-Index Access Path: If several indexes match the selection conditions in the WHERE clause, each matching index offers an alternative access path. An optimizer can choose the access path that it estimates will result in retrieving the fewest pages, apply any projections and nonprimary selection terms (i.e., parts of the selection condition that do not match the index), and proceed to compute the grouping and aggregation operations (by sorting on the GROUP BY attributes).

2. Multiple-Index Access Path: If several indexes using Alternatives (2) or (3) for data entries match the selection condition, each such index can be used to retrieve a set of rids. We can *intersect* these sets of rids, then sort the result by page id (assuming that the rid representation includes the page id) and retrieve tuples that satisfy the primary selection terms of all the matching indexes. Any projections and nonprimary selection terms can then be applied, followed by grouping and aggregation operations.

3. Sorted Index Access Path: If the list of grouping attributes is a prefix of a trec index, the index can be used to retrieve tuples in the order required by the GROUP BY clause. All selection conditions can be applied on each

retrieved tuple, unwanted fields can be removed, and aggregate operations computed for each group. This strategy works well for clustered indexes.

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

We now illustrate each of these four cases, using the query shown in Figure 15.5 as a running example. We assume that the following indexes, all using Alternative (2) for data entries, are available: a B+ tree index on *rating*, a hash index on *age*, and a B+ tree index on *(rating, sname, age)*. For brevity, we do not present detailed cost calculations, but the reader should be able to calculate the cost of each plan. The steps in these plans are scans (a file scan, a scan retrieving tuples by using an index, or a scan of only index entries), sorting, and writing temporary relations; and we have already discussed how to estimate the costs of these operations.

As an example of the first case, we could choose to retrieve Sailors tuples such that *S.age=20* using the hash index on *age*. The cost of this step is the cost of retrieving the index entries plus the cost of retrieving the corresponding Sailors tuples, which depends on whether the index is clustered. We can then apply the condition *S.rating* > 5 to each retrieved tuple; project out fields not mentioned in the SELECT, GROUP BY, and HAVING clauses; and write the result to a temporary relation. In the example, only the *rating* and *sname* fields need to be retained. The temporary relation is then sorted on the *rating* field to identify the groups, and some groups are eliminated by applying the HAVING conclitioIl.

Utilizing Indexes: All of the main RDBMSs recognize the importance of index-only plans and look for such plans whenever possible. In IBM DD2, when creating an index a user can specify a set of 'include' columns that are to be kept in the index but are *not* part of the index key. This allows a richer set of index-only queries to be handled, because columns frequently accessed are included in the index even if they are ;notpart of the key. In Microsoft SQL Server, an interesting class of index-only plans is considered: Consider a query that selects attributes *sal* and *age* from a table, given an index on *sal* and another index on *age*. SQL Server uses the indexes by joining the entries on the rid of data records to identify *(sal, age)* pairs that appear in the table.

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

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

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

15.4.2 Multiple-Relation Queries

Query blocks that contain two or more relations in the FROM clause require joins (or cross-products). Finding a good plan for such queries is very important because these queries can be quite expensive. Regardless of the plan chosen, the size of the final result can be estimated by taking the product of the sizes

of the relations in the FROM clause and the reduction factors for the terms in the WHERE clause. But, depending on the order in which relations are joined, intermediate relations of widely varying sizes can be created, leading to plans with very different costs.

Enumeration of Left-Deep Plans

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

Consider a query block of the form:

SELECT *attribute list*
FROM *relation list*
WHERE $term_1$ 1\ *term2* 1\ ... 1\ $term_n$

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

Pass 1: We enumerate all single-relation plans (over some relation in the FROM clause). Intuitively, each single-relation plan is a partial left-deep plan for evaluating the query in which the given relation is the first (in the linear join order for the left-deep plan of which it is a part). When considering plans involving a relation *A*, we identify those selection terms in the WHERE clause that mention only attributes of *A*. These are the selections that can be performed when first accessing *A*, before any joins that involve *A*. We also identify those attributes of *A* not mentioned in the SELECT clause or in terms in the WHERE clause involving attributes of other relations. These attributes can be projected out when first accessing *A*, before any joins that involve *A*. We choose the best access method for *A* to carry out these selections and projections, as per the discussion in Section 15.4.1.

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

Pass 2: We generate all two-relation plans by considering each single-relation plan retained after Pass 1 as the outer relation and (successively) every other

relation as the inner relation. Suppose that A is the outer relation and B the inner relation for a particular two-relation plan. We examine the list of selections in the WHERE clause and identify:

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

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

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

An important point to note is that tuples generated by the outer plan are assumed to be *pipelined* into the join. That is, we avoid having the outer plan write its result to a file that is subsequently read by the join (to obtain outer tuples). For SOme join methods, the join operator might require materializing the outer tuples. For example, a hash join would partition the incoming tuples, and a sort-merge join would sort them if they are not already in the appropriate sort order. Nested loops joins, however, can use outer tuples as they are generated and avoid materializing them. Similarly, sort-merge joins can use outer tuples as they are generated if they are generated in the sorted order required for the join. We include the cost of materializing the outer relation, should this be necessary, in the cost of the join. The adjustments to the join costs discussed in Chapter 14 to reflect the use of pipelining or materialization of the outer are straightforward.

For each single-relation plan for A retained after Pass 1, for each join method that we consider, we must determine the best access method to use for B. The access method chosen for B retrieves, in general, a subset of the tuples in B, possibly with some fields eliminated, as discussed later. Consider relation B.

We have a collection of selections (some of which are the join conditions) and projections on a single relation, and the choice of the best access method is made as per the discussion in Section 15.4.1. The only additional consideration is that the join method might require tuples to be retrieved in some order. For example, in a sort-merge join, we want the inner tuples in sorted order on the join column(s). If a given access method does not retrieve inner tuples in this order, we must add the cost of an additional sorting step to the cost of the access method.

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

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

If a multiple-relation query contains a GROUP BY clause and aggregate functions such as MIN, MAX, and SUM in the SELECT clause, these are dealt with at the very end. If the query block includes a GROUP BY clause, a set of tuples is computed based on the rest of the query, as described above, and this set is sorted as per the GROUP BY clause. Of course, if there is a plan according to which the set of tuples is produced in the desired order, the cost of this plan is compared with the cost of the cheapest plan (assuming that the two are different) plus the sorting cost. Given the sorted set of tuples, partitions are identified and any aggregate functions in the SELECT clause are applied on a per-partition basis, as per the discussion in Chapter 14.

Examples of Multiple-Relation Query Optimization

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

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

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

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

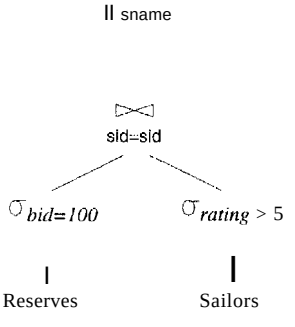


Figure 15.6 A Query Tree

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

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

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

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

We finally retain the cheapest plan overall.

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

```
SELECT    S.sid, COUNT(*) AS numres
FROM      Boats B, Reserves R, Sailors S
WHERE     R.sid = S.sid AND B.bid=R.bid AND Rcolor = 'red'
GROUP BY S.sid
```

This query finds the number of red boats reserved by each sailor. This query is shown in the form of a tree in Figure 15.7.

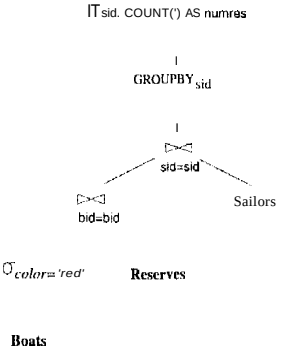


Figure 15.7 A Query Tree

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

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

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

index on *color* (outer) with Sailors (inner), Boats accessed via B+ tree index on *color* (outer) with Reserves (inner), and Boats accessed via hash index on *color* (outer) with Reserves (inner).

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

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

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

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

15.5 NESTED SUBQUERIES

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

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

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

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

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

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

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

```
SELECT S.snallle
```

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

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

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

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

A nested query often has an equivalent query without nesting, and a correlated query often has an equivalent query without correlation. We already saw cor-

Nested Queries: IBM DB2, Informix, Microsoft SQL Server, Orade 8, and Sybase ASE all use some version of correlated evaluation to handle nested queries, which are an important part of the TPC-D benchmark; IBM and Informix support a version in which the results of subqueries are stored in a 'memo' table and the same subquery is not executed multiple times. All these RDBMSs consider decqrrelation and "flattening" of nested queries as an option. Microsoft SQL Server, Oracle 8 and IBM DB2 also use rewriting techniques, e.g., Magic Sets (see Chapter 24) or variants, in conjunction with decorrelation.

related and uncorrelated versions of the example nested query. There is also an equivalent query without nesting:

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

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

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

15.6 THE SYSTEM R OPTIMIZER

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

1. The use of *statistics* about the database instance to estiInate the cost of a query evaluation plan.
2. A decision to consider only plans with binary joins in which the inner relation is a base relation (i.e., not a telnporary relation). This heuristic

reduces the (potentially very large) number of alternative plans that must be considered.

3. A decision to focus optimization on the class of SQL queries without nesting and treat nested queries in a relatively ad hoc way.
4. A decision not to perform duplicate elimination for projections (except as a final step in the query evaluation when required by a DISTINCT clause).
5. A model of cost that accounted for CPU costs as well as I/O costs.

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

15.7 OTHER APPROACHES TO QUERY OPTIMIZATION

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

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

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

15.8 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- \What is an *SQL query block*? Why is it important in the context of query optimization? (Section 15.1)
- Describe how a query block is translated into extended relational algebra. Describe and motivate the extensions to relational algebra. Why are $\sigma\pi$x expressions the focus of an optimizer? (Section 15.1)
- \What are the two parts to estimating the cost of a query plan? (Section 15.2)
- How is the result size estimated for a $\sigma\pi\times$ expression? Describe the use of *reduction factors,* and explain how they are calculated for different kinds of selections? (Section 15.2.1)
- What are *histograms?* How do they help in cost estimation? Explain the differences between the different kinds of histograms, with particular attention to the role of frequent data values. (Section 15.2.1)
- When are two relational algebra expressions considered *equivalent?* How is equivalence used in query optimization? What algebra equivalences that justify the common optimizations of pushing selections ahead of joins and re-ordering join expressions? (Section 15.3)
- Describe *left-deep* plans and explain why optimizers typically consider only such plans. (Section 15.4)
- What plans are considered for (sub)queries with a single relation? Of these, which plans are retained in the dynamic programming approach to enumerating left-deep plans? Discuss access methods and output order in your answer. In particular, explain *index-only plans* and why they are attractive. (Section 15.4)
- Explain how query plans are generated for queries with multiple relations. Discuss the space and time complexity of the dynamic programming approach, and how the plan generation process incorporates heuristics like pushing selections and join ordering. How are index-only plans for multiple-relation queries identified? How are pipelining opportunities identified? (Section 15.4)
- How are nested subqueries optimized and evaluated? Discuss correlated queries and the additional optimization challenges they present. \Vhy are plans produced for nested queries typically of poor quality? What is the lesson for application programmers? (Section 15.5)
- Discuss some of the influential design choices made in the System R optimizer. (Section 15.6)
- Briefly survey optimization techniques that go beyond the dynamic programming framework discussed in this chapter. (Section 15.7)

EXERCISES

Exercise **15.1** Briefly answer the following questions:

1. In the context of query optimization, what is an *SQL query block?*
2. Define the term *reduction factor.*
3. Describe a situation in which projection should precede selection in processing a project-select query, and describe a situation where the opposite processing order is better. (Assume that duplicate elimination for projection is done via sorting.)
4. If there are unclustered (secondary) B+ tree indexes on both *R.a* and *S.b,* the join $R \bowtie_{a=b} S$ could be processed by doing a sort-merge type of join-without doing any sorting-by using these indexes.
   (a) Would this be a good idea if Rand S each has only one tuple per page or would it be better to ignore the indexes and sort Rand S? Explain.
   (b) What if Rand S each have many tuples per page? Again, explain.
5. Explain the role of *interesting orders* in the System R optimizer.

Exercise **15.2** Consider a relation with this schema:

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

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

1. Consider each of the following selection conditions and, assuming that the reduction factor (RF) for each term that matches an index is 0.1, compute the cost of the most selective access path for retrieving all Employees tuples that satisfy the condition:
   (a) $sol > 100$
   (b) $age = 25$
   (c) $age > 20$
   (d) $eid = 1,000$
   (e) $sal > 200 \wedge age > 30$
   (f) $sal > 200 \wedge age = 20$
   (g) $sal > 200 \wedge title = 'CFO'$
   (h) $sal > 200 \wedge age > 30 \wedge title = 'CFO'$
2. Suppose that, for each of the preceding selection conditions, you want to retrieve the average salary of qualifying tuples. For each selection condition, describe the least expensive evaluation method and state its cost.
3. Suppose that, for each of the preceding selection conditions, you want to compute the average salary for each *age* group. For each selection condition, describe the least expensive evaluation method and state its cost.
4. Suppose that, for each of the preceding selection conditions, you want to compute the average age for each sa/level (Le.) group by *sal).* For each selection condition, describe the least expensive evaluation method and state its cost.

5. For each of the following selection conditions, describe the best evaluation method:
   (a) $sal > 200 \vee age = 20$
   (b) $sal > 200 \vee title = 'CFO'$
   (c) $title = 'CFO' \wedge ename = 'Joe'$

Exercise **15.3** For each of the following SQL queries, for each relation involved, list the attributes that must be examined to compute the answer. All queries refer to the following relations:

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

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

Exercise **15.4** You are given the following information:

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

1. Consider the following query:

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

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

   (a) Suppose that a clustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan? (In this and subsequent questions, be sure to describe the plan you have in mind.)
   (b) Suppose that an unclustered B+ tree index on *title* is (the only index) available. What is the cost of the best plan?
   (c) Suppose that a clustered B+ tree index on *enarne* is (the only index) available. What is the cost of the best plan?
   (d) Suppose that a clustered B+ tree index on *address* is (the only index) available. What is the cost of the best pian?
   (e) Suppose that a clustered B+ tree index on *(ename, title)* is (the only index) available. What is the cost of the best plan?

2. Suppose that the query is as follows:

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

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

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

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

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

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

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

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

3. Suppose that the query is as follows:

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

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

(b) Suppose that an unclustered B+ tree index on *t'ltle* is (the only index) available. What is the cost of the best plan?

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

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

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

4. Suppose that the query is as follows:

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

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

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

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

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

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

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

Exercise **15.5** Consider the query $\pi_{A,B,C,D}(R \bowtie_{A=C} S)$. Suppose that the projection routine is based on sorting and is smart enough to eliminate all but the desired attributes during the initial pass of the sort and also to toss out duplicate tuples on-the-fly while sorting, thus eliminating two potential extra passes. Finally, assume that you know the following:

R is 10 pages long, and R tuples are 300 bytes long.
S is 100 pages long, and S tuples are 500 bytes long.
C is a key for S, and A is a key for R.
The page size is 1024 bytes.
Each S tuple joins with exactly one R tuple.
The combined size of attributes A, B, G, and D is 450 bytes.
A and B are in R and have a combined size of 200 bytes; C and D are in S.

1. What is the cost of writing out the final result? (As usual, you should ignore this cost in answering subsequent questions.)
2. Suppose that three buffer pages are available, and the only join method that is implemented is simple (page-oriented) nested loops.
   (a) Compute the cost of doing the projection followed by the join.
   (b) Compute the cost of doing the join followed by the projection.
   (c) Compute the cost of doing the join first and then the projection on-the-fly.
   (d) Would your answers change if 11 buffer pages were available?

Exercise 15.6 Briefly answer the following questions:

1. Explain the role of relational algebra equivalences in the System R optimizer.
2. Consider a relational algebra expression of the form $\sigma_c(\pi_l(R \times S))$. Suppose that the equivalent expression with selections and projections pushed as much as possible, taking into accollnt only relational algebra equivalences, is in one of the following forms. In each case give an illustrative example of the selection conditions and the projection lists (c, l, el, 11, etc.).
   (a) *Equivalent maximally pushed form:* $\pi_{l1}(\sigma_{c1}(R) \times S)$,
   (b) *Equivalent maximally pushed form:* $\pi_{l1}(\sigma_{c1}(R) \times \sigma_{c2}(S))$.
   (c) *Equivalent maximally pushed form:* $\sigma_c(\pi_{l1}(\pi_{l2}(R) \times 8))$.
   (d) *Equivalent maximally pushed fONT!:* $\sigma_{c1}(\pi_{l1}(\sigma_{c2}(\pi_{l2}(R)) \times 8))$.
   (e) *Equivalent ma:rimally pushed form:* $\sigma_{c1}(\pi_{l1}(\pi_{l2}(\sigma_{c2}(R)) \times S))$.
   (f) *Equivalent maximally pushed form:* $\pi_l(\sigma_{c1}(\pi_{l1}(\pi_{l2}(\sigma_{c2}(R)) \times 8)))$.

Exercise 15.7 Consider the following relational schema and SQL query. The schema captures information about employees, departments, and company finances (organized on a per department basis).

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

Consider the following query:

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

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

Exercise 15.8 Consider the following relational schema and SQL query:

Suppliers(sid: integer, *snarne:* char(20), *city:* char(20»)
Supply(*sid:* integer, *pid:* integer)
Parts(pid: integer, *pnarne:* char(20), *price:* real)

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

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

Exercise 15.9 Consider the following scenario:

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

Assume that each Emp record is 20 bytes long, each Dept record is 40 bytes long, and each Proj record is 2000 bytes long on average. There are 20,000 tuples in Emp, 5000 tuples in Dept (note that *did* is not a key), and 1000 tuples in Proj. Each department, identified by *did,* has 10 projects on average. The file system supports 4000 byte pages, and 12 buffer pages are available. All following questions are based on this information. You can assume uniform distribution of values. State any additional assumptions. The cost metric to use is *the number of page I/Os.* Ignore the cost of writing out the final result.

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

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

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

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

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

4. Consider the following query:

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

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

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

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

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

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

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

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

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

5. Consider the following query:

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

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

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

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

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

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

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

(f) Suppose there is a clustered B+ tree index on *D.did,* a hash index on *D.b1ldget,* and a hash index on *P.projid.* Show the plan with the lowest estimated cost.

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

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

6. Consider the following query:

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

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

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

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

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

BIBLIOGRAPHIC NOTES

Query optimization is critical in a relational DBMS, and it has therefore been extensively studied. We concentrate in this chapter on the approach taken in System R, as described in [668], although our discussion incorporates subsequent refinements to the approach. [784] describes query optimization in Ingres. Good surveys can be found in [410J and [399J. [434] contains several articles on query processing and optimization.

From a theoretical standpoint, [155] shows that determining whether two *conjunctive queries* (queries involving only selections, projections, and cross-products) are equivalent is an NP-complete problem; if relations are *multisets*, rather than sets of tuples, it is not known whether the problem is decidable, although it is Π_2^p hard. The equivalence problem is shown to be decidable for queries involving selections, projections, cross-products, and unions in [643]; surprisingly, this problem is undecidable if relations are multisets [404]. Equivalence of conjunctive queries in the presence of integrity constraints is studied in [30], and equivalence of conjunctive queries with inequality selections is studied in [440].

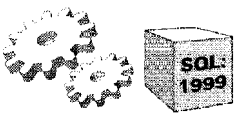
An important problem in query optimization is estimating the size of the result of a query expression. Approaches based on sampling are explored in [352, 353, 384, 481, 569]. The use of detailed statistics, in the form of histograms, to estimate size is studied in [405, 558, 598]. Unless care is exercised, errors in size estimation can quickly propagate and make cost estimates worthless for expressions with several operators, This problem is examined in [400]. [512] surveys several techniques for estimating result sizes and correlations between values in relations. There are a number of other papers in this area; for example, [26, 170, 594, 725], and our list is far from complete,

Semantic query optimization is based on transformations that preserve equivalence only when certain integrity constraints hold. The idea was introduced in [437] and developed further in [148,682, 688].

In recent years, there has been increasing interest in complex queries for decision support applications. Optimization of nested SQL queries is discussed in [298, 426, /130, 557, 760]. The use of the Magic Sets technique for optimizing SQL queries is studied in [553, 554, 555, 670, 673]. Rule-based query optimizers are studiecl in [287, 326, 490, 539, 596]. Finding a good join order for queries with a large number of joins is studied in [401, 402, 453, 726]. Optimization of multiple queries for simultaneous execution is considerecl in [585, 633, 669]. Determining query plans at run-time is discussed in [327, 403]. Re-optimization of running queries based on statistics gathered during query execution is considered by Kabra and DeWitt [413]. Probabilistic optimization of queries is proposed in [183, 229].

PART V

TRANSACTION MANAGEMENT



16

OVERVIEW OF TRANSACTION MANAGEMENT

- What four properties of transactions does a DBMS guarantee?
- Why does a DBMS interleave transactions?
- What is the correctness criterion for interleaved execution?
- What kinds of anomalies can interleaving transactions cause?
- How does a DBMS use locks to ensure correct interleavings?
- What is the impact of locking on performance?
- What SQL commands allow programmers to select transaction characteristics and reduce locking overhead?
- How does a DBMS guarantee transaction atomicity and recovery from system crashes?

-- **Key concepts:** ACID properties, atomicity, consistency, isolation, durability; schedules, serializability, recoverability, avoiding cascading aborts; anomalies, dirty reads, unrepeatable reads, lost updates; locking protocols, exclusive and shared locks, Strict Two-Phase Locking; locking performance, thrashing, hot spots; SQL transaction characteristics, savepoints, rollbacks, phantoms, access mode, isolation level; transaction manager, recovery manager, log, system crash, media failure; stealing frames, forcing pages; recovery phases, analysis, redo and undo.

I always say, keep a diary and someday it'11 keep you.

Mae West

In this chapter, we cover the concept of a *transaction*, which is the foundation for concurrent execution and recovery from system failure in a DBMS. A transaction is defined as *any one execution* of a user program in a DBMS and differs from an execution of a program outside the DBMS (e.g., a C program executing on Unix) in important ways. (Executing the same program several times generates several transactions.)

For performance reasons, a DBMS has to interleave the actions of several transactions. (We motivate interleaving of transactions in detail in Section 16.3.1.) However, to give users a simple way to understand the effect of running their programs, the interleaving is done carefully to ensure that the result of a concurrent execution of transactions is nonetheless equivalent (in its effect on the database) to some serial, or one-at-a-time, execution of the same set of transactions, How the DBMS handles concurrent executions is an important aspect of transaction management and the subject of *concurrency control.* A closely related issue is how the DBMS handles partial transactions, or transactions that are interrupted before they run to normal completion, The DBMS ensures that the changes made by such partial transactions are not seen by other transactions. How this is achieved is the subject of *crash r'ecovery.* In this chapter, we provide a broad introduction to concurrency control and crash recovery in a DBMS, The details are developed further in the next two chapters.

In Section 16.1, we discuss four fundamental properties of database transactions and how the DBMS ensures these properties. In Section 16.2, we present an abstract way of describing an interleaved execution of several transactions, called a *schedule.* In Section 16,3, we discuss various problems that can arise due to interleaved execution, \Ve introduce lock-based concurrency control, the most widely used approach, in Section 16.4. We discuss performance issues associated with lock-based concurrency control in Section 16.5. We consider locking and transaction properties in the context of SQL in Section 16.6, Finally, in Section 16.7, we present an overview of how a clatabase system recovers from crashes and what steps are taken during normal execution to support crash recovery.

16.1 THE ACID PROPERTIES

We introduced the concept of database transactions in Section 1.7, To recapitulate briefly, a transaction is an execution of a user program, seen by the DBMS as a series of read and write operations.

A DBMS must ensure four important properties of transactions to maintain data in the face of concurrent access and system failures:

1. Users should be able to regard the execution of each transaction as atomic: Either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).

2. Each transaction, run by itself with no concurrent execution of other transactions, lnust preserve the consistency of the database. The DBMS assumes that consistency holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.

3. Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as isolation: Transactions are isolated, or protected, from the effects of concurrently scheduling other transactions.

4. Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called durability.

The acronym ACID is sometimes used to refer to these four properties of transactions: atomicity, consistency, isolation and durability. We now consider how each of these properties is ensured in a DBMS.

16.1.1 Consistency and Isolation

Users are responsible for ensuring transaction consistency. That is, the user who submits a transaction must ensure that, when run to completion by itself against a 'consistent' database instance, the transaction will leave the databa.,se in a 'consistent' state. For example, the user may (naturally) have the consistency criterion that fund transfers between bank accounts should not change the total amount of money in the accounts. To transfer money from one account to another, a transaction must debit one account, temporarily leaving the database inconsistent in a global sense, even though the new account balance may satisfy any integrity constraints with respect to the range of acceptable account balances. The user's notion of a consistent database is preserved when the second account is credited with the transferred amount. If a faulty transfer program always credits the second account with one dollar less than the alllount debited frOlll the first account, the DBMS cannot be expected to detect inconsistencies due to such errors in the user program's logic.

The isolation property is ensured by guaranteeing that, even though actions of several transactions rnight be interleaved, the net effect is identical to executing all transactions one after the other in sorne serial order. (We discuss

how the DBMS implements this guarantee in Section 16.4.) For example, if two transactions *T1* and *T2* are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of) *T1* followed by executing *T2* or executing *T2* followed by executing T1. (The DBMS provides no guarantees about which of these orders is effectively chosen.) If each transaction maps a consistent database instance to another consistent database instance, executing several transactions one after the other (on a consistent initial database instance) results in a consistent final database instance.

Database consistency is the property that every transaction sees a consistent database instance. Database consistency follows from transaction atomicity, isolation, and transaction consistency. Next, we discuss how atomicity and durability are guaranteed in a DBMS.

16.1.2 Atomicity and Durability

Transactions can be incomplete for three kinds of reasons. First, a transaction can be **aborted,** or terminated unsuccessfully, by the DBMS because some anomaly arises during execution. If a transaction is aborted by the DBMS for some internal reason, it is automatically restarted and executed anew. Second, the system may crash (e.g., because the power supply is interrupted) while one or more transactions are in progress. Third, a transaction may encounter an unexpected situation (for example, read an unexpected data value or be unable to access some disk) and decide to abort (i.e., terminate itself).

Of course, since users think of transactions as being atomic, a transaction that is interrupted in the middle may leave the database in an inconsistent state. Therefore, a DBMS must find a way to remove the effects of partial transactions from the database. That is, it must ensure transaction atomicity: Either all of a transaction's actions are carried out or none are. A DBMS ensures transaction atomicity by *undoing* the actions of incomplete transactions. This means that users can ignore incomplete transactions in thinking about how the database is modified by transactions over time. To be able to do this, the DBMS maintains a record, called the *log*, of all writes to the database. The log is also used to ensure durability: If the system crashes before the changes made by a completed transaction are written to disk, the log is used to remember and restore these changes when the system restarts.

The DBMS component that ensures atomicity and durability, called the *recovery manager*, is discussed further in Section 16.7.

16.2 TRANSACTIONS AND SCHEDULES

A transaction is seen by the DBMS as a series, or *list*, of **actions.** The actions that can be executed by a transaction include **reads** and **writes** of *database objects.* To keep our notation simple, we assume that an object 0 is always read into a program variable that is also named O. *'Ne* can therefore denote the action of a transaction *T* reading an object 0 as *RT(O);* similarly, we can denote writing as $W_T(O)$. When the transaction *T* is clear from the context, we omit the subscript.

In addition to reading and writing, each transaction *must* specify as its final action either **commit** (i.e., complete successfully) or **abort** (i.e., terminate and undo all the actions carried out thus far). *AbortT* denotes the action of *T* aborting, and *CommitT* denotes *T* committing.

We make two important assumptions:

1. Transactions interact with each other *only* via database read and write operations; for example, they are not allowed to exchange messages.
2. A database is a *fiJ;ed* collection of *independent* objects. When objects are added to or deleted from a database or there are relationships between database objects that we want to exploit for performance, some additional issues arise.

If the first assumption is violated, the DBMS has no way to detect or prevent inconsistencies cause by such external interactions between transactions, and it is upto the writer of the application to ensure that the program is well-behaved. We relax the second assumption in Section 16.6.2.

A **schedule** is a list of actions (reading, writing, aborting, or committing) from a set of transactions, and the order in which two actions of a transaction *T* appear in a schedule must be the same as the order in which they appear in *T*. Intuitively, a schedule represents an actual or potential execution sequence. For example, the schedule in Figure 16.1 shows an execution order for actions of two transactions *T1* and *T2*. We move forward in time as we go down from one row to the next. We emphasize that a schedule describes the actions of transactions *as seen by the DBMS.* In addition to these actions, a transaction rnay carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on; however, we assume that these actions do not affect other transactions; that is, the effect of a transaction on another transaction can be understood solely in terms of the cornmon database objects that they read and write.

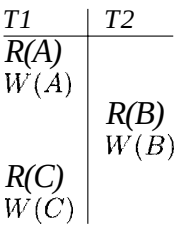


Figure 16.1 A Schedule Involving Two Transactions

Note that the schedule in Figure 16.1 does not contain an abort or commit action for either transaction. A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a complete schedule. A complete schedule must contain all the actions of every transaction that appears in it. If the actions of different transactions are not interleaved-that is, transactions are executed from start to finish, one by one-we call the schedule a serial schedule.

16.3 CONCURRENT EXECUTION OF TRANSACTIONS

Now that we have introduced the concept of a schedule, we have a convenient way to describe interleaved executions of transactions. The DBMS interleaves the actions of different transactions to improve performance, but not all interleavings should be allowed. In this section, we consider what interleavings, or schedules, a DBMS should allow.

16.3.1 Motivation for Concurrent Execution

The schedule shown in Figure 16.1 represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult but necessary for performance reasons. First, while one transa.etion is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the amount of time disks and processors are idle and increases system throughput (the average number of transactions completed in a given time). Second, interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short transaction could get stuck behind a long transaction, leading to unpredictable delays in response time, or average time taken to complete a transaction.

16.3.2 SerializabHity

A **serializable schedule** over a set S of cormnitted transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S. That is, the database instance that results from executing the given schedule is identical to the database instance that results frOlll executing the transactions in *some* serial order.[1]

As an example, the schedule shown in Figure 16.2 is serializable. Even though the actions of *T1* and *T2* are interleaved, the result of this schedule is equivalent to running *T1* (in its entirety) and then running $T2$. Intuitively, *T1*'s read and write of *B* is not influenced by *T2's* actions on *A*, and the net effect is the same if these actiolls are 'swapped' to obtain the serial schedule *Tl; T2*.

Tl	*T2*
R(A)	
$W(A)$	
	R(A)
	W(A)
R(B)	
$W(B)$	
	R(B)
	W(B)
	Commit
Commit	

Figure 16.2 A Serializable Schedule

Executing transactions serially in different orders may produce different results, but all are presumed to be acceptable: the DBMS makes no guarantees ahout which of them will be the outcome of an interleaved execution. To see this, note that the two example transactions from Figure 16.2 can be interleaved as shown in Figure 16.3. This schedule, also serializable, is equivalent to the serial schedule *T2;* Tl. If *T1* and *T2* are submitted concurrently to a DBMS, either of these schedules (among others) could be chosen.

The preceding definition of a serializable schedule does not cover the case of schedules containing aborted transactions. We extend the definition of serializable schedules to cover aborted transactions in Section 16.3.4.

---

[1] If a transaction prints a value to the screen, this 'effect' is not directly captured in the database. For simplicity, we assume that such values are also written into the database.

T1	T2
	R(A)
	W(A)
R(A)	
	R(B)
	W(B)
W(A)	
R(B)	
W(B)	
	Commit
Commit	

Figure 16.3 Another Serializable Schedule

Finally, we note that a DBMS might sometimes execute transactions in a way that is not equivalent to any serial execution; that is, using a schedule that is not serializable. This can happen for two reasons. First, the DBMS might use a concurrency control method that ensures the executed schedule, though not itself serializable, is equivalent to some serializable schedule (e.g., see Section 17.6.2). Second, SQL gives application programmers the ability to instruct the DBMS to choose non-serializable schedules (see Section 16.6).

16.3.3 Anomalies Due to Interleaved Execution

We now illustrate three main ways in which a schedule involving two consistency preserving, committed transactions could run against a consistent database and leave it in an inconsistent state. Two actions on the same data object conflict if at least one of them is a write. The three anomalous situations can be described in terms of when the actions of two transactions *T1* and *T2* conflict with each other: In a write-read (WR) conflict, *T2* reads a data object previously written by *T1;* we define read-write (RW) and write-write (WW) conflicts similarly.

Reading Uncommitted Data (WR Conflicts)

The first source of anomalies is that a transaction *T2* could read a database object *A* that has been modified by another transaction *T1,* which has not yet committed. Such a read is called a dirty read. A simple example illustrates how such a schedule could lead to an inconsistent database state. Consider two transactions *T1* and *T2*. each of which, run alone, preserves database consistency: *T1* transfers $100 from *A* to *B*, and *T2* increments both *A* and *B* by 6% (e.g., annual interest is deposited into these two accounts). Suppose

that the actions are interleaved so that (1) the account transfer program T1 deducts $100 from account *A*, then (2) the interest deposit program *T2* reads the current values of accounts *A* and *B* and adds 6% interest to each, and then (3) the account transfer program credits $100 to account *B*. The corresponding schedule, which is the view the DBMS has of this series of events, is illustrated in Figure 16.4. The result of this schedule is different from any result that we would get by running one of the two transactions first and then the other. The problem can be traced to the fact that the value of *A* written by *T1* is read by *T2* before *T1* has completed all its changes.

T1	*T2*
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Figure 16.4 Reading Uncommitted Data

The general problem illustrated here is that T1 may write some value into *A* that makes the database inconsistent. As long as *T1* overwrites this value with a 'correct' value of *A* before committing, no harm is done if *T1* and *T2* run in some serial order, because *T2* would then not see the (temporary) inconsistency. On the other hand, interleaved execution can expose this inconsistency and lead to an inconsistent final database state.

Note that although a transaction must leave a database in a consistent state *after* it completes, it is not required to keep the database consistent while it is still in progress. Such a requirement would be too restrictive: To transfer money from one account to another, a transaction *must* debit one account, temporarily leaving the database inconsistent, and then credit the second account, restoring consistency.

Unrepeatable Reads (RW Conflicts)

The second way in which anomalous behavior could result is that a transaction $T2$ could change the value of an object A that has been read by a transaction $T1$, while $T1$ is still in progress.

If $T1$ tries to read the value of A again, it will get a different result, even though it has not modified A in the meantime. This situation could not arise in a serial execution of two transactions; it is called an unrepeatable read.

To see why this can cause problems, consider the following example. Suppose that A is the number of available copies for a book. A transaction that places an order first reads A, checks that it is greater tha,n 0, and then decrements it. Transaction $T1$ reads A and sees the value 1. Transaction $T2$ also reads A and sees the value 1, decrements A to 0 and commits. Transaction $T1$ then tries to decrement A and gets an error (if there is an integrity constraint that prevents A from becoming negative).

This situation can never arise in a serial execution of $T1$ and $T2$; the second transaction would read A and see 0 and therefore not proceed with the order (and so would not attempt to decrement A).

Overwriting Uncommitted Data (WW Conflicts)

The third source of anomalous behavior is that a transaction $T2$ could overwrite the value of an object A, which has already been modified by a transaction $T1$, while $T1$ is still in progress. Even if $T2$ does not read the value of A written by $T1$, a potential problem exists as the following example illustrates.

Suppose that Harry and Larry are two employees, and their salaries must be kept equal. Transaction $T1$ sets their salaries to $2000 and transaction $T2$ sets their salaries to $1000. If we execute these in the serial order $T1$ followed by $T2$, both receive the salary $1000: the serial order $T2$ followed by $T1$ gives each the salary $2000. Either of these is acceptable from a consistency standpoint (although Harry and Larry may prefer a higher salary!). Note that neither transaction reads a salary value before writing it----such a write is called a blind write, tor obvious reasons.

Now, consider the following interleaving of the actions of $T1$ and $T2$: $T2$ sets Harry's salary to $1000, $T1$ sets Larry's salary to $2000, $T2$ sets La.rry's salary to $1000 and commits, and finally $T1$ sets Harry's salary to $2000 and connnits. The result is not identical to the result of either of the two possible serial

executions, and the interleaved schedule is therefore not serializable. It violates the desired consistency criterion that the two salaries must be equal.

The problem is that we have a **lost update.** The first transaction to commit, *T2,* overwrote Larry's salary as set by T1. **In** the serial order T2 followed by *T1,* Larry's salary should reflect T1's update rather than $T2$'s, but $T1$'s update is 'lost'.

16.3.4 Schedules Involving Aborted Transactions

We now extend our definition of serializability to include aborted trallsactions.[2] Intuitively, all actions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with. Using this intuition, we extend the definition of a serializable schedule as follows: A **serializable schedule** over a set *S* of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of *committed* transactions in S.

This definition of serializability relies on the actions of aborted transactions being undone completely, which may be impossible in some situations. For example, suppose that (1) an account transfer program T1 deducts $100 from account *A,* then (2) an interest deposit program T2 reads the current values of accounts *A* and *B* and adds 6% interest to each, then commits, and then (3) *T1* is aborted. The corresponding schedule is shown in Figure 16.5.

T1	*T2*
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

Figure 16.5 An Unrecoverable Schedule

[2] We must also consider incomplete transactions for a rigorous discussion of system failures, because transactions that are active when the system fails are neither aborted nor committed. However, system recovery usually begins by aborting all active transactions. and for our informal discussion, considering schedules involving committed and aborted transactions is sufficient.

Now, T2 has read a value for *A* that should never have been there. (Recall that aborted transactions' effects are not supposed to be visible to other transactions.) If *T2* had not yet committed, we could deal with the situation by *cascading* the abort of TI and also aborting *T2;* this process recursively aborts any transaction that read data written by *T2,* and so on. But *T2* has already committed, and so we cannot undo its actions. We say that such a schedule is *unrecoverable.* In a recoverable schedule, transactions commit only after (and if!) all transactions whose changes they read commit. If transactions read only the changes of committed transactions, not only is the schedule recoverable, but also aborting a transaction can be accomplished without cascading the abort to other transactions. Such a schedule is said to avoid cascading aborts.

There is another potential problem in undoing the actions of a transaction. Suppose that a transaction *T2* overwrites the value of an object *A* that has been modified by a transaction *T1,* while *T1* is still in progress, and *T1* subsequently aborts. All of *T1's* changes to database objects are undone by restoring the value of any object that it modified to the value of the object before *T1's* changes. (We look at the details of how a transaction abort is handled in Chapter 18.) When *T1* is aborted and its changes are undone in this manner, *T2's* changes are lost as well, even if *T2* decides to commit. So, for example, if *A* originally had the value 5, then was changed by *T1* to 6, and by *T2* to 7, if *T1* now aborts, the value of *A* becomes 5 again. Even if *T2* commits, its change to *A* is inadvertently lost. A concurrency control technique called Strict 2PL, introduced in Section 16.4, can prevent this problem (as discussed in Section 17.1).

16.4 LOCK-BASED CONCURRENCY CONTROL

A DBMS must be able to ensure that only serializable, recoverable schedules are allowed and that no actions of committed transactions are lost while undoing aborted transactions. A DBMS typically uses a *locking protocol* to achieve this. A lock is a small bookkeeping object associated with a database object. A locking protocol is a set of rules to be followed by each transaction (and enforced by the DBlVIS) to ensure that, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in sOlne serial order. Different locking protocols use different types of locks, such as shared locks or exclusive locks, as we see next, when we discuss the Strict 2PL protocol.

16.4.1 Strict Two-Phase Locking (Strict 2PL)

The most widely used locking protocol, called *Strict Two-Phase Locking*, or *Strict 2PL*, has two rules. The first rule is

> 1. If a transaction T wants to *read* (respectively, *modify*) an object, it first requests a shared (respectively, exclusive) lock on the object.

Of course, a transaction that has an exclusive lock can also read the object; an additional shared lock is not required. A transaction that requests a lock is suspended until the DBMS is able to grant it the requested lock. The DBMS keeps track of the locks it has granted and ensures that if a transaction holds an exclusive lock on an object, no other transaction holds a shared or exclusive lock on the same object. The second rule in Strict 2PL is

> 2. All locks held by a transaction are released when the transaction is completed.

Requests to acquire and release locks can be automatically inserted into transactions by the DBMS; users need not worry about these details. eWe discuss how application programmers can select properties of transactions and control locking overhead in Section 16.6.3.)

In effect, the locking protocol allows only 'safe' interleavings of transactions. If two transactions access completely independent parts of the database, they concurrently obtain the locks they need and proceed merrily on their ways. On the other band, if two transactions access the same object, and one wants to modify it, their actions are effectively ordered serially·all actions of one of these transactions (the one that gets the lock on the common object first) are completed before (this lock is released and) the other transaction can proceed.

We denote the action of a transaction T requesting a shared (respectively, exclusive) lock on object 0 as $5T(0)$ (respectively, $XT(O))$ and omit the subscript denoting the transaction when it is clear from the context. As an example, consider the schedule shown in Figure 16.4. This interleaving could result in a state that cannot result from any serial execution of the three transactions. For instance, $T1$ could change A from 10 to 20, then $T2$ (which reads the value 20 for $A)$ could change B from 100 to 200, and then $T1$ would read the value 200 for B. If run serially, either T1 or $T2$ would execute first, and read the values 10 for A and 100 for B: Clearly, the interleaved execution is not equivalent to either serial execution.

If the Strict 2PL protocol is used, such interleaving is disallowed. Let us see why. Assuming that the transactions proceed at the same relative speed as

before, *T1* would obtain an exclusive lock on A first and then read and write A (Figure 16.6). Then, $T2$ would request a lock on A. However, this request

T1	*T2*
X(A)	
R(A)	
W(A)	

Figure 16.6 Schedule Illustrating Strict 2PL

cannot be granted until *T1* releases its exclusive lock on A, and the DBMS therefore suspends *T2*. *T1* now proceeds to obtain an exclusive lock on B, reads and writes B, then finally commits, at which time its locks are released. *T2's* lock request is now granted, and it proceeds. In this example the locking protocol results in a serial execution of the two transactions, shown in Figure 16.7.

T1	*T2*
X(A)	
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit

Figure 16.7 Schedule Illustrating Strict 2PL with Serial Execution

In general, however, the actions of different transactions could be interleaved. As an example, consider the interleaving of two transactions shown in Figure 16.8, which is permitted by the Strict 2PL protocol.

It can be shown that the Strict 2PL algorithm allows only serializable schedules. None of the anomalies discussed in Section 16.3.3 can arise if the DBMS implements Strict 2PL.

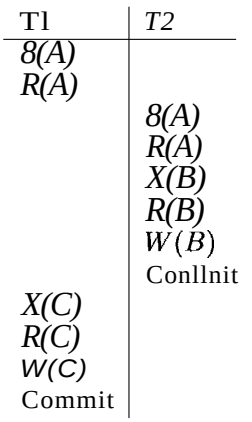


Figure 16.8 Schedule Following Strict 2PL with Interleaved Actions

16.4.2 Deadlocks

Consider the following example. Transaction *T1* sets an exclusive lock on object *A*, *T2* sets an exclusive lock on *B*, *T1* requests an exclusive lock on *B* and is queued, and *T2* requests an exclusive lock on *A* and is queued. Now, *T1* is waiting for *T2* to release its lock and *T2* is waiting for *T1* to release its lock. Such a cycle of transactions waiting for locks to be released is called a **deadlock.** Clearly, these two transactions will make no further progress. Worse, they hold locks that may be required by other transactions. The DBMS must either prevent or detect (and resolve) such deadlock situations; the common approach is to detect and resolve deadlocks.

A simple way to identify deadlocks is to use a timeout mechanism. If a transaction has been waiting too long for a lock, we can assume (pessimistically) that it is in a deadlock cycle and abort it. We discuss deadlocks in more detail in Section 17.2.

16.5 PERFORMANCE OF LOCKING

Lock-based schemes are designed to resolve conflicts between transactions and use two basic mechanisms: *blocking* and *aborting.* Both mechanisrns involve a performance penalty: Blocked transactions may hold locks that force other transactions to wait, and aborting and restarting a transaction obviously wastes the work done thus far by that transaction. A deadlock represents an extreme instance of blocking in which a set of transactions is forever blocked unless one of the deadlocked transactions is aborted by the DBMS.

In practice, fewer than 1% of transactions are involved in a deadlock, and there are relatively few aborts. Therefore, the overhead of locking comes primarily from delays due to blocking.[3] Consider how blocking delays affect throughput. The first few transactions are unlikely to conflict, and throughput rises in proportion to the number of active transactions. As more and more transactions execute concurrently on the same number of database objects, the likelihood of their blocking each other goes up. Thus, delays due to blocking increase with the number of active transactions, and throughput increases more slowly than the number of active transactions. In fact, there comes a point when adding another active transaction actually reduces throughput; the new transaction is blocked and effectively competes with (and blocks) existing transactions. We say that the system thrashes at this point, which is illustrated in Figure 16.9.

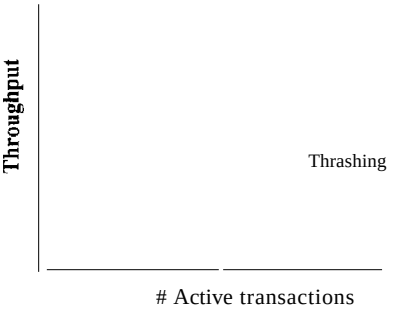


Figure 16.9 Lock Thrashing

If a database system begins to thrash, the database administrator should reduce the number of transactions allowed to run concurrently. Empirically, thrashing is seen to occur when 30% of active transactions are blocked, and a DBA should monitor the fraction of blocked transactions to see if the system is at risk of thrashing.

Throughput can be increased in three ways (other than buying a faster system):

- By locking the smallest sized objects possible (reducing the likelihood that two transactions need the same lock).
- By reducing the time that transaction hold locks (so that other transactions are blocked for a shorter time).

[3] Many common deadlocks can be avoided using a technique called *lock downgrade8,* implemented in most cOlnmercial systems (Section 17.3).

- By reducing **hot** spots. A hot spot is a databa.ge object that is frequently accessed and modified, and causes a lot of blocking delays. Hot spots can significantly affect performance.

The granularity of locking is largely determined by the database system's implementation of locking, and application programmers and the DBA have little control over it. We discuss how to improve performance by minimizing the duration locks are held and using techniques to deal with hot spots in Section 20.10.

16.6 TRANSACTION SUPPORT IN SQL

We have thus far studied transactions and transaction management using an abstract model of a transaction as a sequence of read, write, and abort/commit actions. We now consider what support SQL provides for users to specify transaction-level behavior.

16.6.1 Creating and Terminating Transactions

A transaction is automatically started when a user executes a statement that accesses either the database or the catalogs, such as a SELECT query, an UPDATE command, or a CREATE TABLE statement.[4]

Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either a COMMIT command or a ROLLBACK (the SQL keyword for abort) command.

In SQL:1999, two new features are provided to support applications that involve long-running transactions, or that must run several transactions one after the other. To understand these extensions, recall that all the actions of a given transaction are executed in order, regardless of how the actions of different transactions are interleaved. We can think of each transaction as a sequence of steps.

The first feature, called a savepoint, allows us to identify a point in a transaction and selectively roll back operations carried out after this point. This is especially useful if the transaction carries out what-if kinds of operations, and wishes to undo or keep the changes based on the results. This can be accomplished by defining savepoints.

4Some SQL statements ..... e.g., the CONNECT statement, which connects an application program to a database server do not require the creation of a transaction.

SQL:1999 Nested Transactions: The concept of a transaction as an atomic sequence of actions has been extended in SQL:1999 thrQugh the introduction of the *savepoint* feature. This allows parts of a transaction to be selectively rolled back. The introduction of savepoints represents the first SQL support for the concept of nested transactions, which have been extensively studied in the research community. The idea is that a transaction can have several nested subtransactions, each of which can be selectively rolled back. Savepoints snpport a simple form of one-level nesting.

In a long-running transaction, we may want to define a series of savepoints. The savepoint command allows us to give each savepoint a name:

SAVEPDINT *(savepoint name)*

A subsequent rollback command can specify the savepoint to roll back to

ROLLBACK TO SAVEPDINT *(savepoint name)*

If we define three savepoints A, B, and C in that order, and then rollback to A, all operations since A are undone, including the creation of savepoints B and C. Indeed, the savepoint A is itself undone when we roll hack to it, and we must re-establish it (through another savepoint conunand) if we wish to be able to roll back to it again. From a locking standpoint, locks obtained after savepoint A can be released when we roll back to A.

It is instructive to compare the use of savepoints with the alternative of executing a series of transactions (i.e., treat all operations in between two consecutive savepoints as a new transaction). The savepoint mechanism offers two advantages. First, we can roll back over several savepoints. In the alternative approach, we can roll back only the most recent transaction, which is equivalent to rolling back to the most recent savepoint. Second, the overhead of initiating several transactions is avoided.

Even with the use of savepoints, certain applications might require us to run several transactions one after the other. To minimize the overhead in such situations, SQL:1999 introduces another feature, called chained transactions, \Ve can cornmit or roll back a transaction and immediately initiate another transaction. This is done by using the optional keywords AND CHAIN in the COMMIT and ROLLBACK statements.

16.6.2 What Should We Lock?

Until now, we have discussed transactions and concurrency control in tenus of an abstract model in which a database contains a fixed collection of objects, and each transaction is a series of read and write operations on individual objects. An important question to consider in the context of SQL is what the DBMS should treat as an *object* when setting locks for a given SQL statement (that is part of a transaction).

Consider the following query:

```
SELECT  S.rating, MIN  (S.age)
FROM    Sailors S
WHERE   S.rating = 8
```

Suppose that this query runs as part of transaction *T1* and an SQL statement that modifies the age of a given sailor, say Joe, with *rating=8* runs as part of transaction *T2*. What 'objects' should the DBMS lock when executing these transactions? Intuitively, we must detect a conflict between these transactions.

The DBMS could set a shared lock on the entire Sailors table for *T1* and set an exclusive lock on Sailors for *T2,* which would ensure that the two transactions are executed in a serializable manner. However, this approach yields low concurrency, and we can do better by locking smaller objects, reflecting what each transaction actually accesses. Thus, the DBMS could set a shared lock on every row with *rating=8* for transaction *T1* and set an exclusive lock on just the row for the modified tuple for transaction *T2*. Now, other read-only transactions that do not involve *rating=8* rows can proceed without waiting for *T1* or *T2*.

As this example illustrates, the DBMS can lock objects at different **granularities**: We can lock entire tables or set row-level locks. The latter approach is taken in current systems because it offers much better performance. **In** practice, while row-level locking is generally better, the choice of locking granularity is complicated. For example, a transaction that examines several rows and modifies those that satisfy some condition might be best served by setting shared locks on the entire table and setting exclusive locks on those rows it wants to lllodify. We diseuss this issue further in Section 17.5.3.

A second point to note is that SQL statements conceptually access a collection of rows described by a *selection predicate.* In the preceding example, transaction *T1* accesses all rows with *rating=8.* We suggested that this could be dealt with by setting shared locks on all rows in Sailors that had *rating=8.* Unfortunately, this is a little too silnplistic. To sec why, consider an SQL statelnent that inserts

a new sailor with *rating=8* and runs as transaction *T3*. (Observe that this example violates our assumption of a fixed number of objects in the database, but we must obviously deal with such situations in practice.)

Suppose that the DBJ\iIS sets shared locks on every existing Sailors row with *rating=8* for Tl. This does not prevent transaction *T3* from creating a brand new row with *rating=8* and setting an exclusive lock on this row. If this new row has a smaller *age* value than existing rows, *TI* returns an answer that depends on when it executed relative to *T2*. However, our locking scheme imposes no relative order on these two transactions.

This phenomenon is called the **phantom** problem: A transaction retrieves a collection of objects (in SQL terms, a collection of tuples) twice and sees different results, even though it does not modify any of these tuples itself. To prevent phantoms, the DBMS must conceptually lock *all possible* rows with *rating=8* on behalf of Tl. One way to do this is to lock the entire table, at the cost of low concurrency. It is possible to take advantage of indexes to do better, as we will see in Section 17.5.1, but in general preventing phantoms can have a significant impact on concurrency.

It may well be that the application invoking *T1* can accept the potential inaccuracy due to phantoms. If so, the approach of setting shared locks on existing tuples for *TI* is adequate, and offers better performance. SQL allows a programmer to make this choice---and other similar choices'--explicitly, as we see next.

16.6.3 Transaction Characteristics in SQL

In order to give programmers control over the locking overhead incurred by their transactions, SQL allows them to specify three characteristics of a transaction: access mode, diagnostics size, and isolation level. The **diagnostics** size determines the number of error conditions that can be recorded; we will not discuss this feature further.

If the access **mode** is READ ONLY, the transaction is not allowed to modify the database. Thus, INSERT, DELETE, UPDATE, and CREATE comlnands cannot be executed. If we have to execute one of these commands, the access mode should be set to READ WRITE. For transactions with READ ONLY access mode. only shared locks need to be obtained, thereby increasing concurrency.

The **isolation level** controls the extent to which a given transaction is exposed to the actions of other transactions executing concurrently. By choosing one of four possible isolation level settings, a user can obtain greater concur-

rencyat the cost of increasing the transaction's exposure to other transactions' uncommitted changes.

Isolation level choices are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. The effect of these levels is summarized in Figure 16.10. In this context, *dirty read* and *unrepeatable read* are defined as usuaL

Level	Dirty Read	Unrepeatable Read	
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No

Figure 16.10 Transaction Isolation Levels in SQL-92

The highest degree of isolation from the effects of other transactions is achieved by setting the isolation level for a transaction T to SERIALIZABLE. This isolation level ensures that T reads only the changes made by committed transactions, that no value read or written by T is changed by any other transaction until T is complete, and that if T reads a set of values based on some search condition, this set is not changed by other transactions until T is complete (i.e., T avoids the phantom phenomenon).

In terms of a lock-based implementation, a SERIALIZABLE transaction obtains locks before reading or writing objects, including locks on sets of objects that it requires to be unchanged (see Section 17.5.1) and holds them until the end, according to Strict 2PL.

REPEATABLE READ ensures that T reads only the changes made by committed transactions and no value read or written by T is changed by any other transaction until T is complete. However, T could experience the phantom phenomenon; for example, while T examines all Sailors records with $rating=1$, another transaction might add a new such Sailors record, which is missed by T.

A REPEATABLE READ transaction sets the same locks as a SERIALIZABLE transaction, except that it does not do index locking; that is, it locks only individual objects, not sets of objects. We discuss index locking in detail in Section 17.5.1.

READ COMMITTED ensures that T reads only the changes made by committed transactions, and that no value written by T is changed by any other transaction until T is complete. However, a value read by T may well be modified by

another transaction while T is still in progress, and T is exposed to the phantom problem.

A READ COMMITTED transaction obtains exclusive locks before writing objects and holds these locks until the end. It also obtains shared locks before reading objects, but these locks are released immediately; their only effect is to guarantee that the transaction that last modified the object is complete. (This guarantee relies on the fact that *every* SQL transaction obtains exclusive locks before writing objects and holds exclusive locks until the end.)

A READ UNCOMMITTED transaction T can read changes made to an object by an ongoing transaction; obviously, the object can be changed further while T is in progress, and T is also vulnerable to the phantom problem.

A READ UNCOMMITTED transaction does not obtain shared locks before reading objects. This mode represents the greatest exposure to uncommitted changes of other transactions; so much so that SQL prohibits such a transaction from making any changes itself-a READ UNCOMMITTED transaction is required to have an access mode of READ ONLY. Since such a transaction obtains no locks for reading objects and it is not allowed to write objects (and therefore never requests exclusive locks), it never makes any lock requests.

The SERIALIZABLE isolation level is generally the safest and is recommended for most transactions. Some transactions, however, can run with a lower isolation level, and the smaller number of locks requested can contribute to improved system performance. For example, a statistical query that finds the average sailor age can be run at the READ COMMITTED level or even the READ UNCOMMITTED level, because a few incorrect or missing values do not significantly affect the result if the number of sailors is large.

The isolation level and access mode can be set using the SET TRANSACTION command. For example, the following command declares the current transaction to be SERIALIZABLE and READ ONLY:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY
```

When a transaction is started, the default is SERIALIZABLE and READ WRITE.

16.7 INTRODUCTION TO CRASH RECOVERY

The recovery manager of a DBMS is responsible for ensuring transaction *atomicity* and *durability*. It ensures atomicity by undoing the actions of transactions that do not commit, and durability by making sure that all actions of

committed transactions survive **systenl crashes,** (e.g., a core dump caused by a bus error) and **media failures** (e.g., a disk is corrupted).

\\Then a DBMS is restarted after crashes. the recovery manager is given control and must bring the database to a consistent state. The recovery manager is also responsible for undoing the actions of an aborted transaction. To see what it takes to implement a recovery manager, it is necessary to understand what happens during normal execution.

The **transaction manager** of a DBMS controls the execution of transactions. Before reading and writing objects during normal execution, locks must be acquired (and released at some later time) according to a chosen locking protocol.[5] For simplicity of exposition, we make the following assumption:

Atomic Writes: Writing a page to disk is an atomic action.

This implies that the system does not crash while a write is in progress and is unrealistic. **In** practice, disk writes do not have this property, and steps must be taken during restart after a crash (Section 18.6) to verify that the most recent write to a given page was completed successfully, and to deal with the consequences if not.

16.7.1 Stealing Frames and Forcing Pages

\Vith respect to writing objects, two additional questions arise:

1. Can the changes made to an object 0 in the buffer pool by a transaction *T* be written to disk before *T* commits? Such writes are executed when another transaction wants to bring in a page and the buffer manager chooses to replace the frame containing 0; of course, this page must have been unpinned by *T*. **If** such writes are allowed, we say that a **steal** approach is used. (Informally, the second transaction 'steals' a frame from *T.)*
2. When a transaction cOl1units, must we ensure that all the changes it has made to objects in the buffer pool are immediately forced to disk? **If** so. we say that a **force** approach is used.

From the standpoint of implementing a recovery manager, it is simplest to use a buffer manager with a no-steaL force approach. **If** a no-steal approach is used, we do not have to undo the changes of an aborted transaction (because these changes have not been written to disk), and if a force approach is used, we do

[5] A concurrency control technique that does not involve locking could be used instead, but we assume that locking is llsed.

not have to redo the changes of a committed transaction if there is a subsequent crash (because all these changes are guaranteed to have been written to disk at commit time).

However, these policies have important drawbacks. The no-steal approach assumes that all pages modified by ongoing transactions can be accommodated in the buffer pool, and in the presence of large transactions (typically run in batch mode, e.g., payroll processing), this assumption is unrealistic. The force approach results in excessive page I/O costs. If a highly used page is updated in succession by 20 transactions, it would be written to disk 20 times. With a no-force approach, on the other hand, the in-memory copy of the page would be successively modified and written to disk just once, reflecting the effects of all 20 updates, when the page is eventually replaced in the buffer pool (in accordance with the buffer manager's page replacement policy).

For these reasons, most systems use a steal, no-force approach. Thus, if a frame is dirty and chosen for replacement, the page it contains is written to disk even if the modifying transaction is still active *(steal);* in addition, pages in the buffer pool that are modified by a transaction are not forced to disk when the transaction commits *(no-force).*

16.7.2 Recovery-Related Steps during Normal Execution

The recovery manager of a DBMS maintains some information during normal execution of transactions to enable it to perform its task in the event of a failure. In particular, a log of all modifications to the database is saved on stable storage, which is guaranteed[6] to survive crashes and media failures. Stable storage is implemented by maintaining multiple copies of information (perhaps in different locations) on nonvolatile storage devices such as disks or tapes.

As discussed earlier in Section 16.7, it is important to ensure that the log entries describing a change to the database are written to stable storage *before* the change is made; otherwise, the system might crash just after the change, leaving us without a record of the change. (Recall that this is the Write-Ahead Log, or WAL, property.)

The log enables the recovery manager to undo the actions of aborted and incomplete transactions and redo the actions of committed transactions. For example, a transaction that committed before the crash may have made updates

(jNothing in life is really guaranteed except death and taxes. However, we can reduce the chance of log failure to be vanishingly small by taking steps such as duplexing the log and storing the copies in different secure locations.

Tuning the Recovery Subsystem: DBMS performance can be greatly affected by the overhead imposed by the recoverysubsystem. A DBA can take several steps to tune this subsystem, such as correctlysizing the log and how it is managed on disk, controlling the rate at which buffer pages are forced to disk, choosing a good frequency for checkpointing, and so forth.

to a copy (of a database object) in the buffer pool, and this change may not have been written to disk before the crash, because of a no-force approach. Such changes must be identified using the log and written to disk. Further, changes of transactions that did not commit prior to the crash might have been written to disk because of a steal approach. Such changes must be identified using the log and then undone.

The amount of work involved during recovery is proportional to the changes made by committed transactions that have not been written to disk at the time of the crash. To reduce the time to recover from a crash, the DBMS periodically forces buffer pages to disk during normal execution using a background process (while making sure that any log entries that describe changes these pages are written to disk first, i.e., following the WAL protocol). A process called *checkpointing,* which saves information about active transactions and dirty buffer pool pages, also helps reduce the time taken to recover from a crash. Checkpoints are discussed in Section 18.5.

16.7.3 Overview of ARIES

ARIES is a recovery algorithm that is designed to work with a steal, no-force approach. When the recovery manager is invoked after a crash, restart proceeds in three phases. In the Analysis phase, it identifies dirty pages in the buffer pool (i.e., changes that have not been written to disk) and active transactions at the time of the crash. In the Redo phase, it repeats all actions, starting from an appropriate point in the log, and restores the database state to what it was at the time of the crash. Finally, in the Undo phase, it undoes the actions of transactions that did not commit, so that the database reflects only the actions of committed transactions. The ARIES algorithm is discussed further in Chapter 18.,

16.7.4 Atomicity: Implementing Rollback

It is important to recognize that the recovery subsystem is also responsible for executing the ROLLBACK command, which aborts a single transaction. Indeed,

the logic (and code) involved in undoing a single transaction is identical to that used during the Undo phase in recovering from a system crash. All log records for a given transaction are organized in a linked list and can be efficiently accessed in reverse order to facilitate transaction rollback.

16.8 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are the ACID properties? Define *atomicity, consistency, isolation,* and *durability* and illustrate them through examples, **(Section 16.1)**
- Define the terms *transaction, schedule, complete schedule,* and *seTial schedule.* **(Section 16.2)**
- Why does a DBMS interleave concurrent transactions? **(Section 16.3)**
- When do two actions on the same data object *conflict?* Define the anomalies that can be caused by conflicting actions *(dirty reads, unrepeatable reads, lost updates).* **(Section 16.3)**
- What is a *serializable schedule?* What is a *Tecoverable schedule?* What is a schedule that *avoids cascading abor'ts?* What is a *strict schedule?* **(Section 16.3)**
- What is a *locking protocol?* Describe the *Strict Two-Phase Locking (StTict 2PL)* protocol. What can you say about the schedules allowed by this protocol? **(Section 16.4)**
- What overheads are associated with lock-based concurrency control? Discuss *blocking* and *aborting* overheads specifically and explain which is more important in practice. **(Section 16.5)**
- What is thrashing? What should a DBA do if the system thrashes? **(Section 16.5)**
- How can throughput be increased? **(Section 16.5)**
- How are transactions created and terminated in SQL? What are savepoints? What are chained transactions? Explain why savepoints and chained transactions are useful. **(Section 16.6)**
- What are the considerations in determining the locking granularity when executing SQL statements? What is the phantom problem? What irnpact does it have on performance? **(Section 16.6.2)**

- What transaction characteristics can a programmer control in SQL? Discuss the different *access modes* and *isolat'ion levels* in particular. What issues should be considered in selecting an access mode and an isolation level for a transaction? (Section 16.6.3)
- Describe how different isolation levels are implemented in terms of the locks that are set. 'What can you say about the corresponding locking overheads? (Section 16.6.3)
- What functionality does the *recovery manager* of a DBMS provide? What does the *transaction manager* do? (Section 16.7)
- Describe the *steal* and *force* policies in the context of a buffer manager. What policies are used in practice and how does this affect recovery? (Section 16.7.1)
- What recovery-related steps are taken during normal execution? What can a DBA control to reduce the time to recover from a crash? (Section 16.7.2)
- How is the log used in transaction rollback and crash recovery? (Sections 16.7.2, 16.7.3, and 16.7.4)

EXERCISES

Exercise 16.1 Give brief answers to the following questions:

1. What is a transaction? In what ways is it different from an ordinary program (in a language such as C)?
2. Define these terms: *atomicity, consistency, isolation, durability, schedule, blind write, dirty read, unrepeatable read, serializable schedule, recoverable schedule, avoidsvcascading-aborts schedule.*
3. Describe Strict 2PL.
4. What is the phantom problem? Can it occur in a database where the set of database objects is fixed and only the values of objects can be changed?

Exercise 16.2 Consider the following actions taken by transaction 1'1 on database objects *X* and *Y:*

R(X), W(X),R(Y), W(Y)

1. Give an example of another transaction 1'2 that, if run concurrently to transaction *T* without some form of concurrency control, could interfere with 1'1.
2. Explain how the use of Strict 2PL would prevent interference between the two transactions.
3. Strict 2PL is lIsed in many database systems. Give two reasons for its popularity.

Exercise 16.3 Consider a database with objects X and Y and assume that there are two transactions $T1$ and $T2$. Transaction $T1$ reads objects X and Y and then writes object X. Transaction $T2$ reads objects X and Y and then writes objects X and Y.

1. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a write-read conflict.
2. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a read-write conflict.
3. Give an example schedule with actions of transactions $T1$ and $T2$ on objects X and Y that results in a write-write conflict.
4. For each of the three schedules, show that Strict 2PL disallows the schedule.

Exercise 16.4 We call a transaction that only reads database object a read-only transaction, otherwise the transaction is called a read-write transaction. Give brief answers to the following questions:

1. What is lock thrashing and when does it occur?
2. What happens to the database system throughput if the number of read-write transactions is increased?
3. What happens to the datbase system throughput if the number of read-only transactions is increased?
4. Describe three ways of tuning your system to increase transaction throughput.

Exercise 16.5 Suppose that a DBMS recognizes *increment*, which increments an integer-valued object by 1, and *decrement* as actions, in addition to reads and writes. A transaction that increments an object need not know the value of the object; increment and decrement are versions of blind writes. In addition to shared and exclusive locks, two special locks are supported: An object must be locked in I mode before incrementing it and locked in D mode before decrementing it. An I lock is compatible with another I or D lock on the same object, but not with S and X locks.

1. Illustrate how the use of I and D locks can increase concurrency. (Show a schedule allowed by Strict 2PL that only uses S and X locks. Explain how the use of I and D locks can allow more actions to be interleaved, while continuing to follow Strict 2PL.)
2. Informally explain how Strict 2PL guarantees serializability even in the presence of I and D locks. (Identify which pairs of actions conflict, in the sense that their relative order can affect the result, and show that the use of S, X, I, and D locks according to Strict 2PL orders all conflicting pairs of actions to be the same as the order in some serial schedule.)

Exercise 16.6 Answer the following questions: SQL supports four isolation-levels and two access-modes, for a total of eight combinations of isolation-level and access-mode. Each combination impiicitly defines a class of transactions; the following questions refer to these eight classes:

1. Consider the four SQL isolation levels. Describe which of the plHmomena can occur at each of these isolation levels: *dirty read, unrepeatable read, phantom problem.*
2. For each of the four isolation levels, give examples of transactions that could be run safely at that level.
3. Why does the access mode of a transaction matter?

Exercise 16.7 Consider the university enrollment database schema:

Student(snurn: integer, *snarne:* string, *majoT:* string, *level:* string, *age:* integer)
Class(*name:* string, *meets_at:* time, *room:* string, *fid:* integer)
Enrolled(snum: integer, *cname:* string)
Faculty(*fid:* integer, *fname:* string, *deptid:* integer)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

For each of the following transactions, state the SQL isolation level you would use and explain why you chose it.

1. Enroll a student identified by her *snum* into the class named 'Introduction to Database Systems'.
2. Change enrollment for a student identified by her *snum* from one class to another class,
3. Assign a new faculty member identified by his *fid* to the class with the least number of students.
4. For each class, show the number of students enrolled in the class.

Exercise 16.8 Consider the following schema:

Suppliers(sid: integer, *sname:* string, *addTess:* string)
Parts(pid: integer, *pname:* string, *coloT:* string)
Catalog(sid: integer, *pid:* integer, *cost:* real)

The Catalog relation lists the prices charged for parts by Suppliers.

For each of the following transactions, state the SQL isolation level that you would use and explain why you chose it.

1. A transaction that adds a new part to a supplier's catalog.
2. A transaction that increases the price that a supplier charges for a part.
3. A transaction that determines the total number of items for a given supplier.
4. A transaction that shows, for each part, the supplier that supplies the part at the lowest price.

Exercise 16.9 Consider a database with the following schema:

Suppliers(*sid:* integer, *sname:* string, *addTess:* string)
Parts(pid: integer, *pname:* string, *coloT:* string)
Catalog(*sid:* integer, *pid:* integer, *cost:* real)

The Catalog relation lists the prices charged for parts by Suppliers.

Consider three transactions *1'1,1'2,* and *1'3; 1'1* always has SQL isolation level SERIALIZABLE. We first run *1'1* concurrently with *1'2* and then we run *1'1* concurrently with *1'2* but we change the isolation level of *1'2* as specified below. Give a database instance and SQL statements for *1'1* and *1'2* such that result of running *1'2* with the first SQL isolation level is different from running 1'2 with the second SQL isolation level. Also specify the common schedule of *1'1* and *1'2* and explain why the results are different.

1. SERIALIZABLE versus REPEATABLE READ.
2. REPEATABLE READ versus READ COMMITTED.
3. READ COMMITTED versus READ UNCOMMITTED.

BIBLIOGRAPHIC NOTES

The transaction concept and some of its limitations are discussed in [332J. A formal transaction model that generalizes several earlier transaction models is proposed in [182].

Two-phase locking was introduced in [252], a fundamental paper that also discusses the concepts of transactions, phantoms, and predicate locks. Formal treatments of serializability appear in [92, 581].

Excellent in-depth presentations of transaction processing can be found in [90] and [770]. [338] is a classic, encyclopedic treatment of the subject.

17

CONCURRENCY CONTROL

- How does Strict 2PL ensure serializability and recoverability?
- How are locks implemented in a DBMS?
- What are lock conversions and why are they important?
- How does a DBMS resolve deadlocks?
- How do current systerns deal with the phantom problerrl?
- Why are specialized locking techniques used on tree indexes?
- How does multiple-granularity locking work?
- What is Optimistic concurrency control?
- What is Timestamp-Based concurrency control?
- What is Multiversion concurrency control?
- Key concepts: Two-phase locking (2PL), serializability, recoverability, precedence graph, strict schedule, view equivalence, view serializable, lock nlanager, lock table, transaction table, latch, convoy, lock upgrade, deadlock, waits-for graph, conservative 2PL, index locking, predicate locking, multiple-granularity locking, lock escalation, SQL isolation level, phantom problerrl, optirnistic concurrency control, Thornas Write Rule, recoverability

Pooh was sitting in his house one day, counting his pots of honey, when there carne a knock on the door.
"Fourteen," said Pooh. "Come in. Fourteen. Or was it fifteen? Bother. T'hat's rnuddled rnc."

"Hallo, Pooh," said Rabbit. "Halla, R,abbit. Fourteen, wasn't it?"
"What was?" "My pots of honey what I was counting."
"Fourteen, that's right."
"Are you sure?"
"No," said Rabbit. "Does it matter?"

—A.A. Milne, *The House at Pooh Comer*

In this chapter, we look at concurrency control in more detail. We begin by looking at locking protocols and how they guarantee various irnportant properties of schedules in Section 17.1. Section 17.2 is an introduction to how locking protocols are implemented in a DBMS. Section 17.3 discusses the issue of lock conversions, and Section 17.4 covers deadlock handling. Section 17.5 discusses three specialized locking protocols---for locking sets of objects identified by some predicate, for locking nodes in tree-structured indexes, and for locking collections of related objects. Section 17.6 examines some alternatives to the locking approach.

17.1 2PL, SERIALIZABILITY, AND RECOVERABILITY

In this section, we consider how locking protocols guarantee some important properties of schedules; namely, serializability and recoverability. Two schedules are said to be conflict equivalent if they involve the (sarne set of) actions of the same transactions and they order every pair of conflicting actions of two committed transactions in the sanle way.

As we saw in Section 16.3.3, two actions conflict if they operate on the same data object and at least one of them is a write. The outcome of a schedule depends only on the order of conflicting operations; we can interchange any pair of nonconflicting operations without altering the effect of the schedule on the database. If two schedules are conflict equivalent, it is easy to see that they have the same effect on a database. Indeed, because they order all pairs of conflicting operations in the same way, we can obtain one of thern frorn the other by repeatedly swapping pairs of nonconflicting actions, that is, by swapping pairs of actions whose relative order does not alter the outcome.

A schedule is conflict serializable if it is conflict equivalent to some serial schedule. Every conflict serializable schedule is serializable, if we assurne that the set of items in the database does not grow or shrink; that is, values can be nlodified but items are not added or deleted. We lllake this assurnption for now and consider its consequences in Section 17.5.1. However, sonle serializable schedules are not conflict serializable, as illustrated in Figure 17.1. This schedule is equivalent to executing the transactions serially in the order $T1$, $T2$,

T1	T2	T3
R(A)		
	W(A)	
	COllirnit	
W(A)		
COllirnit		
		W(A)
		Commit

Figure 1.7.1 Serializable Schedule That Is Not Conflict Serializable

T3, but it is not conflict equivalent to this serial schedule because the writes of *T1* and *T2* are ordered differently.

It is useful to capture all potential conflicts between the transactions in a schedule in a precedence graph, also called a serializability graph. The precedence graph for a schedule *S* contains:

- A node for each comnlitted transaction in *S.*
- An arc franl *Ti* to *Tj* if an action of *Ti* precedes and conflicts with one of *Tj*'s actions.

The precedence graphs for the schedules shown in Figures 16.7, 16.8, and 17.1 are shown in Figure 17.2 (parts a, b, and c, respectively).

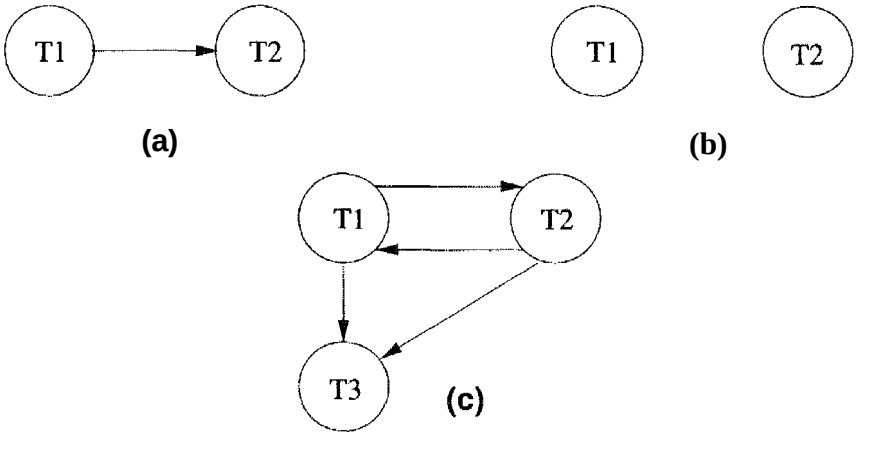


Figure 17.2 Examples of Precedence Graphs

'I'he Strict 2PL protocol (introduced in Section 16.4) allows only conflict serializable schedules, as is seen frcHu the following two results:

1. A schedule S' is conflict serializable if and only if its precedence graph is acyclic. (An equivalent serial schedule in this case is given by any topological sort over the precedence graph.)

2. Strict 2PL ensures t.hat the precedence graph for any schedule that it allows is acyclic.

A widely studied variant of Strict 2PL, called Two-Phase Locking (2PL), relaxes the second rule of Strict 2PL to allow transactions to release locks before the end, that is, before the comnlit or abort action. For 2PL, the second rule is replaced by the following rule:

> (2PL) (2) A transaction cannot request additional locks once it releases *any* lock.

Thus, every transaction has a 'growing' phase in which it acquires locks, followed by a 'shrinking' phase in which it releases locks.

It can be shown that even nonstrict 2PL ensures acyclicity of the precedence graph and therefore allows only conflict serializable schedules. Intuitively, an equivalent serial order of transactions is given by the order in which transactions enter their shrinking phase: If *T2* reads or writes an object written by *Tl*, *Tl* IllUSt have released its lock on the object before *T*'2 requested a lock on this object. Thus, *Tl* precedes *T2*. (A sirnilar argulnent shows that *Tl* precedes *T2* if 7'2 writes an object previously read by *Tl*. A forIllal proof of the claim would have to show that there is no cycle of transactions that 'precede' each other by this argurnent.)

A schedule is said to be strict if a value written by a transaction *T* is not read or overwritten by other transactions until *T* either aborts or eOlnrnits. Strict schedules are recoverable, do not require cascading aborts, and actions of aborted transactions can be undone by restoring the original values of lnodified objects. (See the last exaInple in Section 16.3.4.) Strict 2PL irnproves on 2PL by guaranteeing that every allowed schedule is strict in addition to being conflict serializable. The reason is that when a transaction *T* writes an object under Strict 2PL, it holds the (exclusive) lock until it conunits or aborts. Thus, no other transaction can see or rnodify this object until *T* is cornplete.

The reader is invited to revisit the exarnples in Section 16.3.3 to see how the corresponding schedules are disallowed by Strict 2PL and 2PL. Sirnilarly, it would be instructive to \vork out how the schedules for the exarnples in Section 16.3.4 are disallowed by Strict 2PL but not by 2PL.

17.1.1 View Serializability

Conflict serializability is sufficient but not necessary for serializability. A 1n01'e general sufficient condition is view serializability. Two schedules 81 and 82 over the saIne set of transactions—any transaction that appears in either 81 or 82 rnust also appear in the other----------are view equivalent under these conditions:

1. If *fTi* reads the initial value of object *A* in 81, it Blust also read the initial value of *A* in 82.
2. If *Ti* reads a value of *A* written by *Tj* in 81, it IllUst also read the value of *A* written by *Tj* in 82.
3. For each data object *A*, the transaction (if any) that perforlns the final write on *A* in 81 must also perform the final write on *A* in 82.

A schedule is view serializable if it is view equivalent to SaIne serial schedule. Every conflict serializable schedule is view serializable, although the converse is not true. For example, the schedule shown in Figure 17.1 is view serializable, although it is not conflict serializable. Incidentally, note that this exalnple contains blind writes. This is not a coincidence; it can be shown that any view serializable schedule that is not conflict serializable contains a blind write.

As we saw in Section 17.1, efficient locking protocols allow us to ensure that only conflict serializable schedules are allowed. Enforcing or testing vie\v serializability turns out to be lIluch lnore expensive, and the concept therefore has little practical use, although it increases our understanding of serializability.

17.2 INTRODUCTION TO LOCK MANAGEMENT

The part of the DBMS that keeps track of the locks issued to transactions is called the lock manager. The lock lnanager rnaintains a lock table, which is a hash table with the data object identifier as the key. The DBMS also Inaintains a descriptive entry for each transaction in a transaction table, and alIlong other things, the entry contains a pointer to a list of locks held by the transaction. This list is checked before requesting a lock, to ensure that a transaction does not request the saIne lock twice.

A lock table entry for an object—which can be a page, a record, and so on, depending on the DBMS---contains the following inforrnation: the nurnber of transactions currently holding a lock on the object (this can be rnore than one if the object is locked in shared rnode), the nature of the lock (shared or exclusive), and a pointer to a queue of lock requests.

17.2.1 Implementing Lock and Unlock Requests

According to the Strict 2PL protocol, before a transaction *T* reads or writes a database object 0, it must obtain a shared or exclusive lock on 0 and Inust hold on to the lock until it commits or aborts. When a transaction needs a lock on an object, it issues a lock request to the lock manager:

1. If a shared lock is requested, the queue of requests is ernpty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object (indicating that the object is locked in shared mode, and incrernenting the number of transactions holding a lock by one).
2. If an exclusive lock is requested and no transaction currently holds a lock on the object (which also implies the queue of requests is empty), the lock rnanager grants the lock and updates the lock table entry.
3. Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object. The transaction requesting the lock is suspended.

When a transaction aborts or comrnits, it releases all its locks. When a lock on an object is released, the lock manager updates the lock table entry for the object and exarnines the lock request at the head of the queue for this object. If this request can now be granted, the transaction that made the request is woken up and given the lock. Indeed, if several requests for a shared lock on the object are at the front of the queue, all of these requests can now be granted together.

Note that if *TI* has a shared lock on 0 and 1'2 requests an exclusive lock, *T2's* request is queued. Now, if *T3* requests a shared lock, its request enters the queue behind that of *T2*, even though the requested lock is cornpatible with the lock held by TI. This rule ensures that *T2* does not *starve*, that is, wait indefinitely while a stream of other transactions acquire shared locks and thereby prevent *T2* frorn getting the exclusive lock for which it is waiting.

Atomicity of Locking and Unlocking

The irnplernentation of *lock* and *unlock* cornrnands rnust ensure that these are atomic operations. To ensure atornicity of these operations when several instances of the lock rnanager code can exccute concurrently, access to the lock table has to be guarded by an operating systern synchronization rnechanisrn such as a sernaphore.

To understand why, suppose that a transaction requests an exclusive lock. The lock manager checks and finds that no other transaction holds a lock on the object and therefore decides to grant the request. But, in the 11leantirne, another transaction rnight have requested and *received* a conflicting lock. To prevent this, the entire sequence of actions .in a lock request call (checking to see if the request can be granted, updating the lock table, etc.) must be irnplernented as an atornic operation.

Other Issues: Latches, Convoys

In addition to locks, which are held over a long duration, a DBMS also supports short-duration latches. Setting a latch before reading or writing a page ensures that the physical read or write operation is atomic; otherwise, two read/write operations rnight conflict if the objects being locked do not correspond to disk pages (the units of I/O). Latches are unset immediately after the physical read or write operation is cOlnpleted.

We concentrated thus far on how the DBMS schedules transactions based on their requests for locks. This interleaving interacts with the operating system's scheduling of processes' access to the CPU and can lead to a situation called a convoy, where most of the CPU cycles are spent on process switching. The problem is that a transaction *T* holding a heavily used lock may be suspended by the operating system. UntH *T* is resurned, every other transaction that needs this lock is queued. Such queues, called convoys, can quickly become very long; a convoy, once forrned, tends to be stable. Convoys are one of the drawbacks of building a DBMS on top of a general-purpose operating system with preeruptive scheduling.

17.3 LOCK CONVERSIONS

A transaction rnay need to acquire an exclusive lock on an object for which it already holds a shared lock. For exarnple, a SQL update statenlent could result in shared locks being set on each row in a table. If a row satisfies the condition (in the WHERE clause) for being updated, an exclusive lock must be obtained for that row.

Such a lock upgrade request lnust be handled specially by granting the exclusive lock illunediately if no other transaction holds a shared lock on the object and inserting the request at the front of the queue other\vise. The rationale for favoring the transaction thus is that it already 1101ds a shared lock on the object and queuing it behind. another transaction that wants an exclusive lock on the same object causes both a deadlock. UnfortunatelY,while favoring lock upgrades helps, it does not prevent deadlocks caused by two conflicting upgrade

requests. For exalnplc, if two transactions that hold a shared lock on an object both request an upgrade to an exclusive lock, this leads to a deadlock.

A better approach is to avoid the need for lock upgrades altogether by obtaining exclusive locks initially, and downgrading to a shared lock once it is clear that this is sufficient. In our exalnple of an SQL update statelnent, rows in a table are locked in exclusive rnode first. If a row does *not* satisfy the condition for being updated, the lock on the row is dnwngraded to a shared lock. Does the downgrade approach violate the 2PL requirernent? On the surface, it does, because downgrading reduces the locking privileges held by a transaction, and the transaction Illay go on to acquire other locks. However, this is a special case, because the transaction did nothing but read the object that it downgraded, even though it conservatively obtained an exclusive lock. We can safely expand our definition of 2PL from Section 17.1 to allow lock downgrades in the growing phase, provided that the transaction has not lnodified the object.

The downgrade approach reduces concurrency by obtaining write locks in some cases where they are not required. On the whole, however, it irnproves throughput by reducing deadlocks. This approach is therefore widely used in current commercial systems. Concurrency can be increased by introducing a new kind of lock, called an update lock, that is cornpatible with shared locks but not other update and exclusive locks. By setting an update lock initially, rather than exclusive locks, we prevent conflicts with other read operations. Once we are sure we need not update the object, we can downgrade to a shared lock. If we need to update the object, we rnust first upgrade to an exclusive lock. This upgrade does not lead to a deadlock because no other transaction can have an upgrade or exclusive lock on the object.

17.4 DEALING WITH DEADLOCKS

Deadlocks tend to be rare and typically involve very few transactions. In practice, therefore, database systerns periodically check for deadlocks. When a transaction Ti is suspended because a lock that it requests cannot be granted, it rnust wait until all transactions Tj that currently hold conflicting locks release thcrn. The lock rnanager rnaintains a structure called a waits-for graph to detect deadlock cycles. The nodes correspond to active transactions, and there is an arc frolnTi to Tj if (and only if)Ti is \vaiting for Tj to release a lock. The lock rnanagcr adds edges to this graph when it queues lock requests and rernoves edges when it gra,nts lock requests.

Consider the schedule shown in F'igure 17.:3, The last step, shown below the line, creates a cycle in the \vaits-for graph. Figure 17.4 shows the ·waits-for graph before and after this step.

TI	*T2*	*T3*	*T4*
S(A)			
$R(A)$			
	X(B)		
	$W(B)$		
8(B)			
		8(C)	
		R(C)	
	X(C)		
			X(B)
		X(A)	

Figure 17.3 Schedule Illustrating Deadlock

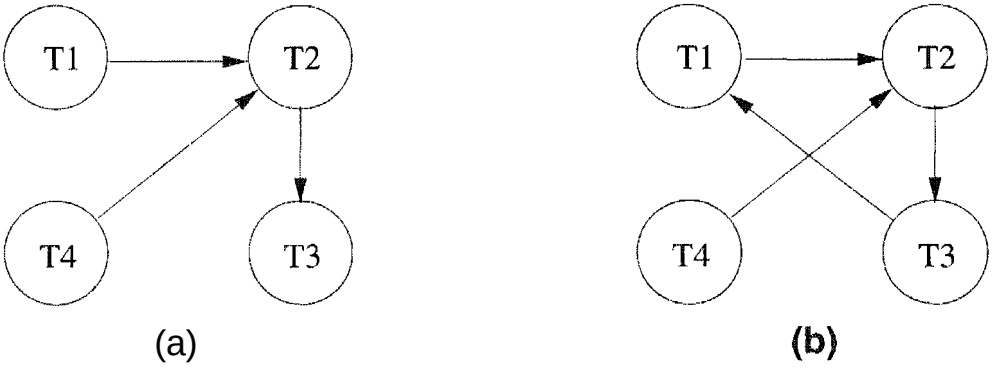


Figure 17.4 Waits-for Graph Before and After Deadlock

Observe that the \vaits-for graph describes all active transactions, some of which eventually abort. If there is an edge froIn *Ti* to *Tj* in the waits-for graph, and both *Ti* and *Tj* eventually commit, there is an edge in the opposite direction (froIn *I'j* to *Ti)* in the precedence graph (which involves only cOlluuitted transactions).

The waits-for graph is periodically checked for cycles, which indicate deadlock. A deadlock is resolved by aborting a transaction that is on a cycle and releasing its locks; this action allows SOlne of the waiting transactions to proceed. The choice of which transaction to abort can be made using several criteria: the one with the fewest locks, the one that has done the least work, the one that is farthest from completion, and so all. Further, a transaction might have been repeatedly restarted; if so, it should eventually be favored during deadlock detection and allowed to complete.

A simple alternative to maintaining a waits-for graph is to identify deadlocks through a timeout mechanism: If a transaction has been waiting too long for a lock, we assume (pessiruistically) that it is in a deadlock cycle and abort it.

17.4.1 Deadlock Prevention

Elnpirical results indicate that deadlocks are relatively infrequent, and detection-based schemes work well in practice. However, if there is a high level of contention for locks and therefore an increased likelihood of deadlocks, prevention-based schelnes could perform better. We can prevent deadlocks by giving each transaction a priority and ensuring that lower-priority transactions are not allowed to wait for higher-priority transactions (or vice versa). One way to assign priorities is to give each transaction a timestamp when it starts up. The lower the timestamp, the higher is the transaction's priority; that is, the oldest transaction has the highest priority.

If a transaction *Ti* requests a lock and transaction *Tj* holds a conflicting lock, the lock lnanager can use one of the following two policies:

- Wait-die: If *Ti* has higher priority, it is allowed to wait; otherwise, it is aborted.
- Wound-wait: If *Ti* has higher priority, abort *Tj*; otherwise, *Ti* waits.

In the \vait-die scherne, lower-priority transactions can never wait for higher-priority transactions. In the wound-wait scherne, higher-priority transactions never wait for lower-priority transactions. In either ease, no deadlock cyc.le develops.

A subtle point is that we nlust also ensure that no transaction is perennially aborted because it never has a sufficiently high priority. (Note that, in both schernes, the higher-priority transaction is never aborted.) When a transaction is aborted and restarted, it should be given the same timestamp it had originally. Reissuing timestarnps in this way ensures that each transaction will eventually becorne the oldest transaction, and therefore the one with the highest priority, and will get all the locks it requires.

The wait-die scheme is nonpreemptive; only a transaction requesting a lock can be aborted. As a transaction grows older (and its priority increases), it tends to wait for more and rnore younger transactions. A younger transaction that conflicts with an older transaction may be repeatedly aborted (a disadvantage with respect to wound-wait), but on the other hand, a transaction that has all the locks it needs is never aborted for deadlock reasons (an advantage with respect to wound-wait, which is preemptive).

A variant of 2PL, called Conservative 2PL, can also prevent deadlocks. Under Conservative 2PL, a transaction obtains all the locks it will ever need when it begins, or blocks waiting for these locks to become available. This scheme ensures that there will be no deadlocks, and, perhaps lllore important, that a transaction that already holds some locks will not block waiting for other locks. If lock contention is heavy, Conservative 2PL can reduce the time that locks are held on average, because transactions that hold locks are never blocked. The trade-off is that a transaction acquires locks earlier, and if lock contention is low, locks are held longer under Conservative 2PL. From a practical perspective, it is hard to know exactly what locks are needed ahead of time, and this approach leads to setting more locks than necessary. It also has higher overhead for setting locks because a transaction has to release all locks and try to obtain thern all over if it fails to obtain even one lock that it needs. This approach is therefore not used in practice.

17.5 SPECIALIZED LOCKING TECHNIQUES

Thus far we have treated a database as a *fixed* collection of *independent* data objects in our presentation of locking protocols. We now relax each of these restrictions and discuss the consequences.

If the collection of database objects is not fixed, but can grow and shrink through the insertion and deletion of objects, we must deal with a subtle cOlnplication known as the *phantom problem,* which was illustrated in Section 16.6.2. We discuss this problern in Section 17.5.1.

Although treating a database as an independent collection of objects is adequate for a discussion of serializability and recoverability, luuch better perfc)l'rnance can sOlnetilnes be obtained using protocols that recognize and exploit the relationships between objects. We discuss two such cases, nalnely, locking in tree-structured indexes (Section 17.5.2) and locking a collection of objects with containnent relationships between the1n (Section 17.5.3).

17.5.1 Dynamic Databases and the Phantom Problem

Consider the following exarnple: rrransaction *TI* scans the Sailors relation to find the oldest sailor for each of the *rating* levels 1 and 2. First, *TI* identifies and locks all pages (assurning that page-level locks are set) containing sailors with rating 1 and then finds the age of the oldest sailor, which is, say, 71. Next, transaction *T2* inserts a new sailor with rating 1 and age 96. Observe that this new Sailors record can be inserted onto a page that does not contain other sailors with rating 1; thus, an exclusive lock on this page does not conflict with any of the locks held by *TI*. *T2* also locks the page containing the oldest sailor with rating 2 and deletes this sailor (whose age is, say, 80). *T2* then comrnits and releases its locks. Finally, transaction *T1* identifies and locks pages containing (all remaining) sailors with rating 2 and finds the age of the oldest such sailor, which is, say, 63.

The result of the interleaved execution is that ages 71 and 63 are printed in response to the query. If *T1* had run first, then *T2*, we would have gotten the ages 71 and 80; if *T2* had run first, then *T*1, we would have gotten the ages 96 and 63. Thus, the result of the interleaved execution is not identical to any serial exection of *TI* and 1'2, even though both transactions follow Strict 2PL and cOlnmit. The problem is that *T1* assurnes that the pages it has locked include *all* pages containing Sailors records with rating 1, and this assurnption is violated when *rT2* inserts a new such sailor on a different page.

'rhe flaw is not in the Strict 2PL protocol. R,ather, it is in *T1*'s irnplicit assurnption that it has locked the set of all Sailors records with *rating* value 1. *T1*'s sernantics requires it to identify all such records, but locking pages that contain such records *at a given tirne* does not prevent new "phantom' records frorn being added on other pages. *T*1 has therefore *not* locked the set of desired Sailors records.

Strict 2PL guarantees conflict serializability; indeed, there are no cycles in the precedence graph for this exarnple because conflicts are defined with respect to objects (in this example, pages) read/written by the traJlsactions. However, because the set of objects that *should* have been locked by *TI* was altered by the actions ofT2, the olltcorne of the schedule differed frolll the outcorne of any

serial execution. This exalllple brings out an irnportant point about conflict serializability: If new itenls are added to the database, conflict serializability does not guarantee serializability.

A closer look at how a transaction identifies pages containing Sailors records with *rating* 1 suggests how the problenl can be handled:

- If there is no index and all pages in the file rnust be scanned, *T1* IUUSt someho\v ensure that no new pages are added to the file, in addition to locking all existing pages.
- If there is an index on the *rating* field, *T1* can obtain a lock on the index page—again, assurning that physical locking is done at the page level-that contains a data entry with *rating*= 1. If there are no such data entries, that is, no records with this *rating* value, the page that *would* contain a data entry for *rating=l* is locked to prevent such a record from being inserted. Any transaction that tries to insert a record with *rating=-l* into the Sailors relation 11lUSt insert a data entry pointing to the new record into this index page and is blocked until *T1* releases its locks. This technique is called index locking.

Both techniques effectively give *T1* a lock on the set of Sailors records with *rating=l:* Each existing record with *rating=l* is protected frolll changes by other transactions, and additionally, new records with *rating=l* cannot be inserted.

An independent issue is how transaction 7'1 can efficiently identify and lock the index page containing *rating*=1. We discuss this issue for the case of tree-structured indexes in Section 17.5.2.

\Ve note that index locking is a special case of a luore general concept called predicate locking. In our exalnple, the lock on the index page irnplieitly locked all Sailors records that satisfy the logical predicate *rating*= 1. 1V101'e generally, we can support irnplicit locking of all records that rnatch an arbitra,ry predicate. General predicate locking is expensive to irnplenlent and therefore not cOllullonly used.

17.5.2 Concurrency Control in B+ Trees

A straightforward approach to concurrency control for B+ trees and ISAM indexes is to ignore the index structure, treat each page as a data object, and use senne version of 2PL. This siInplistic locking strategy would lead to very high lock contention in the higher levels of the tree, because every tree search begins at the root and proceeds along sorne path to a leaf node. Fortunately, Innch Inore efficient locking protocols that exploit the hierarchical structure of a tree

index are known to reduce the locking overhead while ensuring seriaIizability and recoverability.We discuss sorne of these approaches briefly, concentrating on the search and insert operations.

Two observations provide the necessary insight:

1. The higher levels of the tree only direct searches. All the 'real' data is in the leaf levels (in the fornat of one of the three alternatives for data entries).

2. For inserts, a node must be locked (in exclusive rnode, of course) only if a split can propagate up to it frorn the modified leaf.

Searches should obtain shared locks on nodes, starting at the root and proceeding along a path to the desired leaf. The first observation suggests that a lock on a node can be released as soon as a lock on a child node is obtained, because searches never go back up the tree.

A conservative locking strategy for inserts would be to obtain exclusive locks on all nodes as we go down from the root to the leaf node to be modified, because splits can propagate all the way from a leaf to the root. However, once we lock the child of a node, the lock on the node is required only in the event that a split propagates back to it. In particular, if the child of this node (on the path to the modified leaf) is not full when it is locked, any split that propagates up to the child can be resolved at the child, and does not propagate further to the current node. Therefore, when we lock a child node, we can release the lock on the parent if the child is not full. The locks held thus by an insert force any other transaction following the sarne path to wait at the earliest point (i.e., the node nearest the root) that rnight be affected by the insert. The technique of locking a child node and (if possible) releasing the lock on the parent is called lock-coupling, or crabbing (think of how a crab walks, and cornpare it to how we proceed down a tree, alternately releasing a lock on a parent and setting a lock on a child).

We illustrate B+ tree locking using the tree in Figure 17.5. To search for data entry 38*, a transaction Ti rnust obtain an *S* lock on node *A*, read the contents and deterrnine that it needs to examine node *B*, obtain an *S* lock on node *B* and release the lock on *A*, then obtain an *S* lock on node C' and release the lock on *B*, then obtain an *S* lock on nodeD and release the lock on ().

Ti always rnaintains a lock all one node in the path, to force new transactions that want to read or nlodify nodes on the sarne path to wait until the current transaction is done. If transaction Tj wants to delete 38*, for exarnple, it rnust also traverse the path fro111 the root to node *D* and is forced to wait until Ti

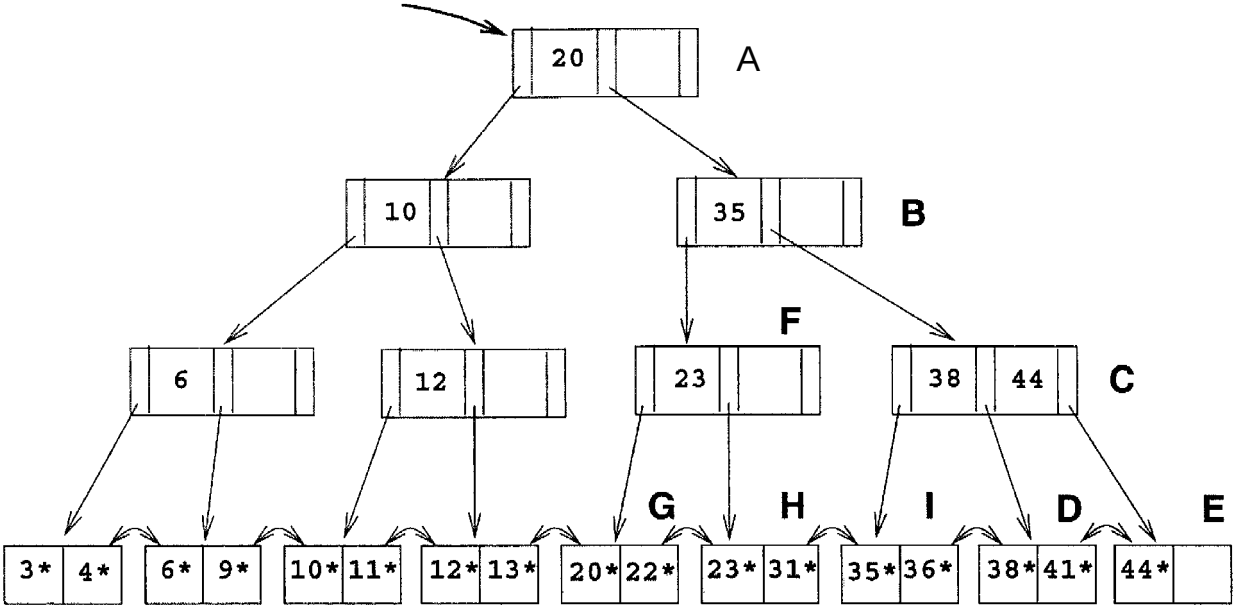


Figure 17.5 B+ Tree Locking Example

is done. Of course, if SOl11e transaction *Tk* holds a lock on, say, node C before *Ti* reaches this node, *Ti* is similarly forced to wait for *Tk* to complete.

To insert data entry 45*, a transaction 111USt obtain an *S* lock on node *A*, obtain an *S* lock on node *B* and release the lock on *A*, then obtain an *S* lock on node C (observe that the lock on *B* is *not* released, because C is full), then obtain an X lock on node *E* and release the locks on C and then *B*. Because node *E* has space for the new entry, the insert is accomplished by modifying this node.

In contrast, consider the insertion of data entry 25*. Proceeding as for the insert of 45*, we obtain an X lock on node *H*. Unfortunately, this node is full and must be split. Splitting *H* requires that we also rnodify the parent, node *F*, but the transaction has only an *S* lock on *F*. Thus, it must request an upgrade of this lock to an X lock. If no other tra,nsaction holds an *S* lock on *F*, the upgrade is granted, and since *F* has space, the split does not propagate further and the insertion of 25* can proceed (by splitting *If* and locking G to modify the sibling pointer in *I* to point to the newly created node). However, if another transaction holds an *S* lock on node *F*, the first transaction is suspended until this transaction releases its Slack.

Observe that if another transaction holds an *S* lock on *F* and also wants to access node *H*, we have a deadlock because the first transaction has an X lock on *If*. The preceding exarnple also illustrates an interesting point about sibling pointers: When we split leaf node *H*, the new node *must* be added to the *left* of *Ii*, since otherwise the node whose sibling pointer is to be changed would be node 1, which has a different parent. To rnodify a sibling pointer on *I*, we

'would have to lock its parent, node C (and possibly ancestors of C, in order to lock C).

Except for the locks on intermediate nodes that we indicated could be released early, senne variant of 2PL HUlst be used to govern when locks can be released, to ensure serializability and recoverability.

T'his approach irllproves considerably on the naive use of 2PL, but several exclusive locks are still set unnecessarily and, although they are quickly released, affect perfon.nance substantially. One way to iInprove perforlllance is for inserts to obtain shared locks instead of exclusive locks, except for the leaf, which is locked in exclusive 11lode. In the vast rnajority of cases, a split is not required and this approach works very well. If the leaf is full, however, we Blust upgrade from shared locks to exclusive locks for all nodes to which the split propagates. Note that such lock upgrade requests can also lead to deadlocks.

The tree locking ideas that we describe illustrate the potential for efficient locking protocols in this very important special case, but they are not the current state of the art. The interested reader should pursue the leads in the bibliography.

17.5.3 Multiple-Granularity Locking

Another specialized locking strategy, called **multiple-granularity locking,** allows us to efficiently set locks on objects that contain other objects.

For instance, a database contains several files, a file is a collection of pages, and a page is a collection of records. A transaction that expects to access rnost of the pages in a file should probably set a lock on the entire file, rather than locking individual pages (or records) when it needs thern. Doing so reduces the locking overhead considerably. On the other hand, other tra,nsactions that require access to parts of the file---even parts not needed by this transaction---are blocked. If a transaction accesses relatively few pages of the file, it is better to lock only those pages. Sirnilarly, if a transaction accesses several records on a page, it should lock the entire page, and if it accesses just a few records, it should lock just those records.

The question to be addressed is how a lock rnanager can efficiently ensure that a page, for exaruple, is not locked by a transaction while another transaction holds a conflicting lock on the file containing the page (a.nd therefore, irnplicitly, on the page).

The idea is to exploit the hierarchical nature of the 'contains' relationship. A database contains a set of files, each file contains a set of pages, and each page contains a set of records. This containunent hierarchy can be thought of as a tree of objects, \vhere each node contains all its children. (The approach can easily be extended to cover hierarchies that are not trees, but we do not discuss this extension.) A lock on a node locks that node and, irnplicitly, all its descendants. (Note that this interpretation of a lock is very different fron1 B+ tree locking, where locking a node does *not* lock any descendants ilnplicitly.)

In addition to shared (8) and exclusive *(XO)* locks, rnultiple-granularity locking protocols also use two new kinds of locks, called intention shared (18) and intention exclusive *(IX)* locks. 18 locks conflict only with X locks. IX locks conflict with 8 and X locks. To lock a node in *S* (respectively, X) luode, a transaction must first lock all its ancestors in 18 (respectively, 1*X)* rllode. Thus, if a transaction locks a node in 8 rIlode, no other transaction can have locked any ancestor in X rnode; siInilarly, if a transaction locks a node in X mode, no other transaction can have locked any ancestor in 8 or X mode. This ensures that no other transaction holds a lock on an ancestor that conflicts with the requested 8 or X lock on the node.

A common situation is that a transaction needs to read an entire file and modify a few of the records in it; that is, it needs an 8 lock on the file and an 1X lock so that it can subsequently lock sorne of the contained objects in X mode. It is useful to define a new kind of lock, called an 81*X* lock, that is logically equivalent to holding an 8 lock and an *IX* lock. A transaction can obtain a single 81*X* lock (which conflicts with any lock that conflicts with either *S* or *IX)* instead of an 8 lock and an *IX* lock.

A subtle point is that locks rnust be released in leaf-to-root order for this protocol to work correctly. To see this, consider what happens when a transaction *Ti* locks all nodes on a path frolH the root (corresponding to the entire database) to the node corresponding to sorne page *p* in 18 rnode, locks *p* in *S* rHode, and then releases the lock on the root node. Another transaction *T*j could now obtain an *X* lock on the root. This lock iInplicitly gives *Tj* an *X* lock on page *p,* which conflicts with the 8 lock currently held by *Ti*.

Multiple-granularity locking lllust be used with 2PL to ensure serializability. The 2PL protocol dictates when locks can be released. At that tirne, locks obtained using rIlultiple-granularity locking can be released and IllUSt be released in leaf-to-root order.

Finally, there is the question of how to decide what granularity of locking is appropriate for a given transaction. One approach is to begin by obtaining fine granularity locks (e.g., at the record level) and, after the transaction requests