15.2.4 Exercises for Section 15.2

Exercise 15.2.1: For each of the operations below, write an iterator that uses the algorithm described in this section: (a) distinct (δ) (b) grouping (γ_L) (c) set union (d) set intersection (e) set difference (f) bag intersection (g) bag difference (h) product (i) natural join. (j) projection

Exercise 15.2.2: For each of the operators in Exercise 15.2.1, tell whether the operator is *blocking*, by which we mean that the first output cannot be produced until all the input has been read. Put another way, a blocking operator is one whose only possible iterators have all the important work done by `Open`.

! **Exercise 15.2.3:** Give one-pass algorithms for each of the following join-like operators:

a) $R \overset{\circ}{\bowtie}_L S$, assuming R fits in memory (see Section 5.2.7 for definitions involving outerjoins).

b) $R \overset{\circ}{\bowtie}_L S$, assuming S fits in memory.

c) $R \overset{\circ}{\bowtie}_R S$, assuming R fits in memory.

d) $R \overset{\circ}{\bowtie}_R S$, assuming S fits in memory.

e) $R \overset{\circ}{\bowtie} S$, assuming R fits in memory.

f) $R \ltimes S$, assuming R fits in memory (see Exercise 2.4.8 for a definition of the semijoin).

g) $R \ltimes S$, assuming S fits in memory.

h) $R \overline{\ltimes} S$, assuming R fits in memory (see Exercise 2.4.9 for a definition of the antisemijoin).

i) $R \overline{\ltimes} S$, assuming S fits in memory.

Exercise 15.2.4: Figure 15.9 summarizes the memory and disk-I/O requirements of the algorithms of this section and the next. However, it assumes all arguments are clustered. How would the entries change if one or both arguments were not clustered?

15.3 Nested-Loop Joins

Before proceeding to the more complex algorithms in the next sections, we shall turn our attention to a family of algorithms for the join operator called "nested-loop" joins. These algorithms are, in a sense, "one-and-a-half" passes, since in each variation one of the two arguments has its tuples read only once, while the other argument will be read repeatedly. Nested-loop joins can be used for relations of any size; it is not necessary that one relation fit in main memory.

15.3.1 Tuple-

The simplest variat
tuples of the relati
nested-loop join, we

```
FOR each tu
    FOR eac
        IF
```

If we are careles
this algorithm coul
are many situation
cost. One case is w
of R to find the tu
read the entire rela
second improvemen
are divided among
the number of disk
this block-based ve

15.3.2 An Ite

One advantage of a
work, and thus, as
intermediate relatio
easy to build from t
and so on, as in Se
nested-loop join is
R nor S is empty.

15.3.3 Block-

We can improve o
compute $R \bowtie S$ by:

1. Organizing ac

2. Using as muc
relation S, the

Point (1) makes su
loop, we use as few
each tuple of R tha
tuples of S as will f

cost is proportional to the product of the sizes of the two relations, divided by the amount of available main memory. We can do much better than a nested-loop join when both relations are large. But for reasonably small examples such as Example 15.4, the cost of the nested-loop join is not much greater than the cost of a one-pass join, which is 1500 disk I/O's for this example. In fact, if $B(S) \leq M - 1$, the nested-loop join becomes identical to the one-pass join algorithm of Section 15.2.3.

Although nested-loop join is generally not the most efficient join algorithm possible, we should note that in some early relational DBMS's, it was the only method available. Even today, it is needed as a subroutine in more efficient join algorithms in certain situations, such as when large numbers of tuples from each relation share a common value for the join attribute(s). For an example where nested-loop join is essential, see Section 15.4.6.

15.3.5 Summary of Algorithms so Far

The main-memory and disk I/O requirements for the algorithms we have discussed in Sections 15.2 and 15.3 are shown in Fig. 15.9. The memory requirements for γ and δ are actually more complex than shown, and $M = B$ is only a loose approximation. For γ, M depends on the number of groups, and for δ, M depends on the number of distinct tuples.

Operators	Approximate M required	Disk I/O	Section
σ, π	1	B	15.2.1
γ, δ	B	B	15.2.2
$\cup$, $\cap$, $-$, $\times$, $\bowtie$	$\min(B(R), B(S))$	$B(R) + B(S)$	15.2.3
$\bowtie$	any $M \geq 2$	$B(R)B(S)/M$	15.3.3

Figure 15.9: Main memory and disk I/O requirements for one-pass and nested-loop algorithms

15.3.6 Exercises for Section 15.3

Exercise 15.3.1: Suppose $B(R) = B(S) = 10{,}000$, and $M = 1000$. Calculate the disk I/O cost of a nested-loop join.

Exercise 15.3.2: For the relations of Exercise 15.3.1, what value of M would we need to compute $R \bowtie S$ using the nested-loop algorithm with no more than (a) 200,000 ! (b) 25,000 ! (c) 15,000 disk I/O's?

Exercise 15.3.3: Give the three iterator methods for the block-based version of nested-loop join.

! **Exercise 15.3.4:**
or S is empty. Re
relations are empty

! **Exercise 15.3.5:**
join would require

a) How can you

b) If only one o
loop join? C
the smaller i

15.4 Two-

We shall now begi
algebra operation
rithms of Section
where data from t
some way, written
operation. We ca
the data is read
two-pass algorith

a) Two passes

b) Generalizin
sions in Sec

We begin with an
general approach:
M, sort them, an
only one block of

15.4.1 Two-

It is possible to
called *Two-Phas*
memory buffers t

- *Phase 1*: R
them, usin
sublist to s

- *Phase 2*: M
at most M
one input

! **Exercise 15.3.4:** The iterator of Fig. 15.7 will not work properly if either R or S is empty. Rewrite the methods so they will work, even if one or both relations are empty.

! **Exercise 15.3.5:** If R and S are both unclustered, it seems that nested-loop join would require about $T(R)T(S)/M$ disk I/O's.

a) How can you do significantly better than this cost?

b) If only one of R and S is unclustered, how would you perform a nested-loop join? Consider both the cases that the larger is unclustered and that the smaller is unclustered.

15.4 Two-Pass Algorithms Based on Sorting

We shall now begin the study of multipass algorithms for performing relational-algebra operations on relations that are larger than what the one-pass algorithms of Section 15.2 can handle. We concentrate on *two-pass algorithms*, where data from the operand relations is read into main memory, processed in some way, written out to disk again, and then reread from disk to complete the operation. We can naturally extend this idea to any number of passes, where the data is read many times into main memory. However, we concentrate on two-pass algorithms because:

a) Two passes are usually enough, even for very large relations.

b) Generalizing to more than two passes is not hard; we discuss these extensions in Section 15.4.1 and more generally in Section 15.8.

We begin with an implementation of the sorting operator τ that illustrates the general approach: divide a relation R for which $B(R) > M$ into chucks of size M, sort them, and then process the sorted sublists in some fashion that requires only one block of each sorted sublist in main memory at any one time.

15.4.1 Two-Phase, Multiway Merge-Sort

It is possible to sort very large relations in two passes using an algorithm called *Two-Phase, Multiway Merge-Sort* (TPMMS), Suppose we have M main-memory buffers to use for the sort. TPMMS sorts a relation R as follows:

- *Phase 1*: Repeatedly fill the M buffers with new tuples from R and sort them, using any main-memory sorting algorithm. Write out each *sorted sublist* to secondary storage.

- *Phase 2*: Merge the sorted sublists. For this phase to work, there can be at most $M - 1$ sorted sublists, which limits the size of R. We allocate one input block to each sorted sublist and one block to the output. The

2. Bring the first block of each sublist into a buffer; we assume there are no more than M sublists in all.

3. Repeatedly find the least Y-value y among the first available tuples of all the sublists. Identify all the tuples of both relations that have Y-value y, perhaps using some of the M available buffers to hold them, if there are fewer than M sublists. Output the join of all tuples from R with all tuples from S that share this common Y-value. If the buffer for one of the sublists is exhausted, then replenish it from disk.

Example 15.7: Let us again consider the problem of Example 15.4: joining relations R and S of sizes 1000 and 500 blocks, respectively, using 101 buffers. We divide R into 10 sublists and S into 5 sublists, each of length 100, and sort them.[3] We then use 15 buffers to hold the current blocks of each of the sublists. If we face a situation in which many tuples have a fixed Y-value, we can use the remaining 86 buffers to store these tuples.

We perform three disk I/O's per block of data. Two of those are to create the sorted sublists. Then, every block of every sorted sublist is read into main memory one more time in the multiway merging process. Thus, the total number of disk I/O's is 4500. □

This sort-join algorithm is more efficient than the algorithm of Section 15.4.6 when it can be used. As we observed in Example 15.7, the number of disk I/O's is $3\big(B(R) + B(S)\big)$. We can perform the algorithm on data that is almost as large as that of the previous algorithm. The sizes of the sorted sublists are M blocks, and there can be at most M of them among the two lists. Thus, $B(R) + B(S) \leq M^2$ is sufficient.

15.4.9 Summary of Sort-Based Algorithms

In Fig. 15.11 is a table of the analysis of the algorithms we have discussed in Section 15.4. As discussed in Sections 15.4.6 and 15.4.8, the join algorithms have limitiations on how many tuples can share a common value of the join attribute(s). If this limit is violated, we may have to use a nest-loop join instead.

15.4.10 Exercises for Section 15.4

Exercise 15.4.1: For each of the following operations, write an iterator that uses the algorithm described in this section: (a) grouping (γ_L) (b) set intersection (c) bag difference (d) natural join. (e) distinct (δ)

[3] Technically, we could have arranged for the sublists to have length 101 blocks each, with the last sublist of R having 91 blocks and the last sublist of S having 96 blocks, but the costs would turn out exactly the same.

Operators

τ, γ, δ

$\cup, \cap, -$

$\bowtie$

$\bowtie$

Figure 15.11: I
rithms

Exercise 15.4
algorithm for r
(c) a binary op

Exercise 15.4
join-like operat

Exercise 15.4
disk I/O requi
Section 15.4.8.

! **Exercise 15.4**
in this section
sublists. How

! **Exercise 15.4**
and S, with 1
need additiona
value that nei
total number

a) There a
half the

b) There a

c) There a

! **Exercise 15.**
Section 15.4.8

Operators	Approximate M required	Disk I/O	Section
τ, γ, δ	$\sqrt{B}$	$3B$	15.4.1, 15.4.2, 15.4.3
$\cup$, $\cap$, $-$	$\sqrt{B(R)+B(S)}$	$3\big(B(R)+B(S)\big)$	15.4.4, 15.4.5
$\bowtie$	$\sqrt{\max(B(R),B(S))}$	$5\big(B(R)+B(S)\big)$	15.4.6
$\bowtie$	$\sqrt{B(R)+B(S)}$	$3\big(B(R)+B(S)\big)$	15.4.8

Figure 15.11: Main memory and disk I/O requirements for sort-based algorithms

Exercise 15.4.2: How much memory do we need to use a two-pass, sort-based algorithm for relations of 20,000 blocks each, if the operation is: (a) δ (b) γ (c) a binary operation such as join or union.

Exercise 15.4.3: Describe a two-pass, sort-based algorithm for each of the join-like operators of Exercise 15.2.3.

Exercise 15.4.4: If $B(R) = B(S) = 10{,}000$ and $M = 500$, what are the disk I/O requirements of: (a) simple sort-join (b) the more efficient sort-join of Section 15.4.8. (c) set union

! **Exercise 15.4.5:** Suppose that the second pass of an algorithm described in this section does not need all M buffers, because there are fewer than M sublists. How might we save disk I/O's by using the extra buffers?

! **Exercise 15.4.6:** In Example 15.6 we discussed the join of two relations R and S, with 1000 and 500 blocks, respectively, and $M = 101$. However, we need additional additional disk I/O's if there are so many tuples with a given value that neither relation's tuples could fit in main memory. Calculate the total number of disk I/O's needed if:

a) There are only two Y-values, each appearing in half the tuples of R and half the tuples of S (recall Y is the join attribute or attributes).

b) There are five Y-values, each equally likely in each relation.

c) There are 10 Y-values, each equally likely in each relation.

! **Exercise 15.4.7:** Repeat Exercise 15.4.6 for the more efficient sort-join of Section 15.4.8.

!! **Exercise 15.4.8:** Sometimes, it is possible to save some disk I/O's if we leave the last sublist in memory. It may even make sense to use sublists of fewer than M blocks to take advantage of this effect. How many disk I/O's can be saved this way?

! **Exercise 15.4.9:** Suppose records could be larger than blocks, i.e., we could have spanned records. How would the memory requirements of two-pass, sort-based algorithms change?

15.5 Two-Pass Algorithms Based on Hashing

There is a family of hash-based algorithms that attack the same problems as in Section 15.4. The essential idea behind all these algorithms is as follows. If the data is too big to store in main-memory buffers, hash all the tuples of the argument or arguments using an appropriate hash key. For all the common operations, there is a way to select the hash key so all the tuples that need to be considered together when we perform the operation fall into the same bucket.

We then perform the operation by working on one bucket at a time (or on a pair of buckets with the same hash value, in the case of a binary operation). In effect, we have reduced the size of the operand(s) by a factor equal to the number of buckets, which is roughly M. Notice that the sort-based algorithms of Section 15.4 also gain a factor of M by preprocessing, although the sorting and hashing approaches achieve their similar gains by rather different means.

15.5.1 Partitioning Relations by Hashing

To begin, let us review the way we would take a relation R and, using M buffers, partition R into $M - 1$ buckets of roughly equal size. We shall assume that h is the hash function, and that h takes complete tuples of R as its argument (i.e., all attributes of R are part of the hash key). We associate one buffer with each bucket. The last buffer holds blocks of R, one at a time. Each tuple t in the block is hashed to bucket $h(t)$ and copied to the appropriate buffer. If that buffer is full, we write it out to disk, and initialize another block for the same bucket. At the end, we write out the last block of each bucket if it is not empty. The algorithm is given in more detail in Fig. 15.12.

15.5.2 A Hash-Based Algorithm for Duplicate Elimination

We shall now consider the details of hash-based algorithms for the various operations of relational algebra that might need two-pass algorithms. First, consider duplicate elimination, that is, the operation $\delta(R)$. We hash R to $M - 1$ buckets, as in Fig. 15.12. Note that two copies of the same tuple t will hash to the same bucket. Thus, we can examine one bucket at a time, perform δ on that bucket in isolation, and take as the answer the union of $\delta(R_i)$, where

2. Sort-based algorithms sometimes allow us to produce a result in sorted order and take advantage of that sort later. The result might be used in another sort-based algorithm for a subsequent operator, or it could be the answer to a query that is required to be produced in sorted order.

3. Hash-based algorithms depend on the buckets being of equal size. Since there is generally at least a small variation in size, it is not possible to use buckets that, on average, occupy M blocks; we must limit them to a slightly smaller figure. This effect is especially prominent if the number of different hash keys is small, e.g., performing a group-by on a relation with few groups or a join with very few values for the join attributes.

4. In sort-based algorithms, the sorted sublists may be written to consecutive blocks of the disk if we organize the disk properly. Thus, one of the three disk I/O's per block may require little rotational latency or seek time and therefore may be much faster than the I/O's needed for hash-based algorithms.

5. Moreover, if M is much larger than the number of sorted sublists, then we may read in several consecutive blocks at a time from a sorted sublist, again saving some latency and seek time.

6. On the other hand, if we can choose the number of buckets to be less than M in a hash-based algorithm, then we can write out several blocks of a bucket at once. We thus obtain the same benefit on the write step for hashing that the sort-based algorithms have for the second read, as we observed in (5). Similarly, we may be able to organize the disk so that a bucket eventually winds up on consecutive blocks of tracks. If so, buckets can be read with little latency or seek time, just as sorted sublists were observed in (4) to be writable efficiently.

15.5.8 Exercises for Section 15.5

Exercise 15.5.1: If $B(S) = B(R) = 10{,}000$ and $M = 500$, what is the number of disk I/O's required for a hybrid hash join?

Exercise 15.5.2: Write iterators that implement the two-pass, hash-based algorithms for (a) $\cap_B$ (b) $-_S$ (c) $\bowtie$ (d) δ (e) γ.

Exercise 15.5.3: The hybrid-hash-join idea, storing one bucket in main memory, can also be applied to other operations. Show how to save the cost of storing and reading one bucket from each relation when implementing a two-pass, hash-based algorithm for: (a) $\cap_B$ (b) $-_S$. (c) δ (d) γ

! **Exercise 15.5.4:** Suppose we are performing a two-pass, hash-based grouping operation on a relation R of the appropriate size; i.e., $B(R) \leq M^2$. However, there are so few groups, that some groups are larger than M; i.e., they will not fit in main memory at once. What modifications, if any, need to be made to the algorithm given here?

! **Exercise 15.5.5:** Suppose that we are using a disk where the time to move the head to a block is 100 milliseconds, and it takes 1/2 millisecond to read one block. Therefore, it takes $k/2$ milliseconds to read k consecutive blocks, once the head is positioned. Suppose we want to compute a two-pass hash-join $R \bowtie S$, where $B(R) = 1000$, $B(S) = 500$, and $M = 101$. To speed up the join, we want to use as few buckets as possible (assuming tuples distribute evenly among buckets), and read and write as many blocks as we can to consecutive positions on disk. Counting 100.5 milliseconds for a random disk I/O and $100 + k/2$ milliseconds for reading or writing k consecutive blocks from or to disk:

a) How much time does the disk I/O take?

b) How much time does the disk I/O take if we use a hybrid hash-join as described in Example 15.9?

c) How much time does a sort-based join take under the same conditions, assuming we write sorted sublists to consecutive blocks of disk?

15.6 Index-Based Algorithms

The existence of an index on one or more attributes of a relation makes available some algorithms that would not be feasible without the index. Index-based algorithms are especially useful for the selection operator, but algorithms for join and other binary operators also use indexes to very good advantage. In this section, we shall introduce these algorithms. We also continue with the discussion of the index-scan operator for accessing a stored table with an index that we began in Section 15.1.1. To appreciate many of the issues, we first need to digress and consider "clustering" indexes.

15.6.1 Clustering and Nonclustering Indexes

Recall from Section 15.1.3 that a relation is "clustered" if its tuples are packed into roughly as few blocks as can possibly hold those tuples. All the analyses we have done so far assume that relations are clustered.

We may also speak of *clustering indexes*, which are indexes on an attribute or attributes such that all the tuples with a fixed value for the search key of this index appear on roughly as few blocks as can hold them. Note that a relation that isn't clustered cannot have a clustering index,[5] but even a clustered relation

[5]Technically, if the index is on a key for the relation, so only one tuple with a given value in the index key exists, then the index is always "clustering," even if the relation is not clustered. However, if there is only one tuple per index-key value, then there is no advantage from clustering, and the performance measure for such an index is the same as if it were considered nonclustering.

total number of disk I/O's at 3500, which is less than that for other methods considered so far.

Now, assume that both R and S have indexes on Y. Then there is no need to sort either relation. We use just 1500 disk I/O's to read the blocks of R and S through their indexes. In fact, if we determine from the indexes alone that a large fraction of R or S cannot match tuples of the other relation, then the total cost could be considerably less than 1500 disk I/O's. However, in any event we should add the small number of disk I/O's needed to read the indexes themselves. □

15.6.5 Exercises for Section 15.6

Exercise 15.6.1: Suppose $B(R) = 10{,}000$ and $T(R) = 500{,}000$. Let there be an index on $R.a$, and let $V(R,a) = k$ for some number k. Give the cost of $\sigma_{a=0}(R)$, as a function of k, under the following circumstances. You may neglect disk I/O's needed to access the index itself.

a) The index is not clustering.

b) The index is clustering.

c) R is clustered, and the index is not used.

Exercise 15.6.2: Repeat Exercise 15.6.1 if the operation is the range query $\sigma_{C \leq a \texttt{ AND } a \leq D}(R)$. You may assume that C and D are constants such that $k/10$ of the values are in the range.

Exercise 15.6.3: Suppose there is an index on attribute $R.a$. Describe how this index could be used to improve the execution of the following operations. Under what circumstances would the index-based algorithm be more efficient than sort- or hash-based algorithms?

a) $\delta(R)$.

b) $R \cup_S S$ (assume that R and S have no duplicates, although they may have tuples in common).

c) $R \cap_S S$ (again, with R and S sets).

! **Exercise 15.6.4:** If R is clustered, but the index on $R.a$ is *not* clustering, then depending on k we may prefer to implement a query by performing a table-scan of R or using the index. For what values of k would we prefer to use the index if the relation and query are as in (a) Exercise 15.6.1 (b) Exercise 15.6.2.

Exercise 15.6.5: Consider the SQL query:

```
SELECT birthdate FROM StarsIn, MovieStar
WHERE movieTitle = 'King Kong' AND starName = name;
```

This query uses the "movie" relations:

```
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
```

If we translate it to relational algebra, the heart is an equijoin between

$$\sigma_{movieTitle=\texttt{'King Kong'}}(\texttt{StarsIn})$$

and `MovieStar`, which can be implemented much as a natural join $R \bowtie S$. Since there were only three movies named "King Kong," $T(R)$ is very small. Suppose that S, the relation `MovieStar`, has an index on `name`. Compare the cost of an index-join for this $R \bowtie S$ with the cost of a sort- or hash-based join.

! **Exercise 15.6.6:** In Example 15.14 we discussed the disk-I/O cost of a join $R \bowtie S$ in which one or both of R and S had sorting indexes on the join attribute(s). However, the methods described in that example can fail if there are too many tuples with the same value in the join attribute(s). What are the limits (in number of blocks occupied by tuples with the same value) under which the methods described will not need to do additional disk I/O's?

15.7 Buffer Management

We have assumed that operators on relations have available some number M of main-memory buffers that they can use to store needed data. In practice, these buffers are rarely allocated in advance to the operator, and the value of M may vary depending on system conditions. The central task of making main-memory buffers available to processes, such as queries, that act on the database is given to the *buffer manager*. It is the responsibility of the buffer manager to allow processes to get the memory they need, while minimizing the delay and unsatisfiable requests. The role of the buffer manager is illustrated in Fig. 15.16.

15.7.1 Buffer Management Architecture

There are two broad architectures for a buffer manager:

1. The buffer manager controls main memory directly, as in many relational DBMS's, or

2. The buffer manager allocates buffers in virtual memory, allowing the operating system to decide which buffers are actually in main memory at any time and which are in the "swap space" on disk that the operating system manages. Many "main-memory" DBMS's and "object-oriented" DBMS's operate this way.

available to hold blocks of R. As we read each block of R, in order, the blocks that remain in buffers at the end of this iteration of the outer loop will be the last k blocks of R. We next reload the $M - 1$ buffers for S with new blocks of S and start reading the blocks of R again, in the next iteration of the outer loop. However, if we start from the beginning of R again, then the k buffers for R will need to be replaced, and we do not save disk I/O's just because $k > 1$.

A better implementation of nested-loop join, when an LRU buffer-replacement strategy is used, visits the blocks of R in an order that alternates: first-to-last and then last-to-first (called *rocking*). In that way, if there are k buffers available to R, we save k disk I/O's on each iteration of the outer loop except the first. That is, the second and subsequent iterations require only $B(R) - k$ disk I/O's for R. Notice that even if $k = 1$ (i.e., no *extra* buffers are available to R), we save one disk I/O per iteration. □

Other algorithms also are impacted by the fact that M can vary and by the buffer-replacement strategy used by the buffer manager. Here are some useful observations.

- If we use a sort-based algorithm for some operator, then it is possible to adapt to changes in M. If M shrinks, we can change the size of a sublist, since the sort-based algorithms we discussed do not depend on the sublists being the same size. The major limitation is that as M shrinks, we could be forced to create so many sublists that we cannot then allocate a buffer for each sublist in the merging process.

- If the algorithm is hash-based, we can reduce the number of buckets if M shrinks, as long as the buckets do not then become so large that they do not fit in allotted main memory. However, unlike sort-based algorithms, we cannot respond to changes in M while the algorithm runs. Rather, once the number of buckets is chosen, it remains fixed throughout the first pass, and if buffers become unavailable, the blocks belonging to some of the buckets will have to be swapped out.

15.7.4 Exercises for Section 15.7

Exercise 15.7.1: Suppose that we wish to execute a join $R \bowtie S$, and the available memory will vary between M and $M/2$. In terms of M, $B(R)$, and $B(S)$, give the conditions under which we can guarantee that the following algorithms can be executed:

a) A two-pass, hash-based join.

b) A two-pass, sort-based join.

c) A one-pass join.

! **Exercise 15.7.2:** How would the number of disk I/O's taken by a nested-loop join improve if extra buffers became available and the buffer-replacement policy were:

a) The clock algorithm.

b) First-in-first-out.

!! **Exercise 15.7.3:** In Example 15.15, we suggested that it was possible to take advantage of extra buffers becoming available during the join by keeping more than one block of R buffered and visiting the blocks of R in reverse order on even-numbered iterations of the outer loop. However, we could also maintain only one buffer for R and increase the number of buffers used for S. Which strategy yields the fewest disk I/O's?

15.8 Algorithms Using More Than Two Passes

While two passes are enough for operations on all but the largest relations, we should observe that the principal techniques discussed in Sections 15.4 and 15.5 generalize to algorithms that, by using as many passes as necessary, can process relations of arbitrary size. In this section we shall consider the generalization of both sort- and hash-based approaches.

15.8.1 Multipass Sort-Based Algorithms

In Section 15.4.1 we alluded to how 2PMMS could be extended to a three-pass algorithm. In fact, there is a simple recursive approach to sorting that will allow us to sort a relation, however large, completely, or if we prefer, to create n sorted sublists for any desired n.

Suppose we have M main-memory buffers available to sort a relation R, which we shall assume is stored clustered. Then do the following:

BASIS: If R fits in M blocks (i.e., $B(R) \leq M$), then read R into main memory, sort it using any main-memory sorting algorithm, and write the sorted relation to disk.

INDUCTION: If R does not fit into main memory, partition the blocks holding R into M groups, which we shall call $R_1, R_2, \ldots, R_M$. Recursively sort R_i for each $i = 1, 2, \ldots, M$. Then, merge the M sorted sublists, as in Section 15.4.1.

If we are not merely sorting R, but performing a unary operation such as γ or δ on R, then we modify the above so that at the final merge we perform the operation on the tuples at the front of the sorted sublists. That is,

- For a δ, output one copy of each distinct tuple, and skip over copies of the tuple.

passes; that is, the buckets are of size $u(M, k-1)$. Since R is divided into $M-1$ buckets, we must have $u(M,k) = (M-1)u(M, k-1)$.

If we expand the recurrence above, we find that $u(M,k) = M(M-1)^{k-1}$, or approximately, assuming M is large, $u(M,k) = M^k$. Equivalently, we can perform one of the unary relational operations on relation R in k passes with M buffers, provided $M \geq \big(B(R)\big)^{1/k}$.

We may perform a similar analysis for binary operations. As in Section 15.8.2, let us consider the join. Let $j(M,k)$ be an upper bound on the size of the smaller of the two relations R and S involved in $R(X,Y) \bowtie S(Y,Z)$. Here, as before, M is the number of available buffers and k is the number of passes we can use.

BASIS: $j(M,1) = M-1$; that is, if we use the one-pass algorithm to join, then either R or S must fit in $M-1$ blocks, as we discussed in Section 15.2.3.

INDUCTION: $j(M,k) = (M-1)j(M,k-1)$; that is, on the first of k passes, we can divide each relation into $M-1$ buckets, and we may expect each bucket to be $1/(M-1)$ of its entire relation, but we must then be able to join each pair of corresponding buckets in $M-1$ passes.

By expanding the recurrence for $j(M,k)$, we conclude that $j(M,k) = (M-1)^k$. Again assuming M is large, we can say approximately $j(M,k) = M^k$. That is, we can join $R(X,Y) \bowtie S(Y,Z)$ using k passes and M buffers provided $\min\big(B(R), B(S)\big) \leq M^k$.

15.8.5 Exercises for Section 15.8

Exercise 15.8.1: Suppose $B(R) = 10{,}000$, $B(S) = 40{,}000$, and $M = 101$. Describe the behavior of the following algorithms to compute $R \bowtie S$:

a) A three-pass, hash-based algorithm.

b) A three-pass, sort-based algorithm.

! **Exercise 15.8.2:** There are several "tricks" we have discussed for improving the performance of two-pass algorithms. For the following, tell whether the trick could be used in a multipass algorithm, and if so, how?

a) Improving a sort-based algorithm by storing blocks consecutively on disk (Section 15.5.7).

b) Improving a hash-based algorithm by storing blocks consecutively on disk (Section 15.5.7).

c) The hybrid-hash-join trick of Section 15.5.6.