Chapter 15

Query Execution

The broad topic of query processing will be covered in this chapter and Chapter 16. The *query processor* is the group of components of a DBMS that turns user queries and data-modification commands into a sequence of database operations and executes those operations. Since SQL lets us express queries at a very high level, the query processor must supply much detail regarding how the query is to be executed. Moreover, a naive execution strategy for a query may take far more time than necessary.

Figure 15.1 suggests the division of topics between Chapters 15 and 16. In this chapter, we concentrate on query execution, that is, the algorithms that manipulate the data of the database. We focus on the operations of the extended relational algebra, described in Section 5.2. Because SQL uses a bag model, we also assume that relations are bags, and thus use the bag versions of the operators from Section 5.1.

We shall cover the principal methods for execution of the operations of relational algebra. These methods differ in their basic strategy; scanning, hashing, sorting, and indexing are the major approaches. The methods also differ on their assumption as to the amount of available main memory. Some algorithms assume that enough main memory is available to hold at least one of the relations involved in an operation. Others assume that the arguments of the operation are too big to fit in memory, and these algorithms have significantly different costs and structures.

Preview of Query Compilation

To set the context for query execution, we offer a very brief outline of the content of the next chapter. Query compilation is divided into the three major steps shown in Fig. 15.2.

a) *Parsing*. A *parse tree* for the query is constructed.

b) *Query Rewrite*. The parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then

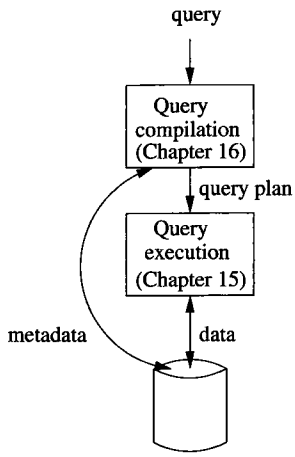

Figure 15.1: The major parts of the query processor

transformed into an equivalent plan that is expected to require less time to execute.

c) *Physical Plan Generation.* The abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.

Parts (b) and (c) are often called the *query optimizer*, and these are the hard parts of query compilation. To select the best query plan we need to decide:

1. Which of the algebraically equivalent forms of a query leads to the most efficient algorithm for answering the query?

2. For each operation of the selected form, what algorithm should we use to implement that operation?

3. How should the operations pass data from one to the other, e.g., in a pipelined fashion, in main-memory buffers, or via the disk?

Each of these choices depends on the metadata about the database. Typical metadata that is available to the query optimizer includes: the size of each relation; statistics such as the approximate number and frequency of different values for an attribute; the existence of certain indexes; and the layout of data on disk.

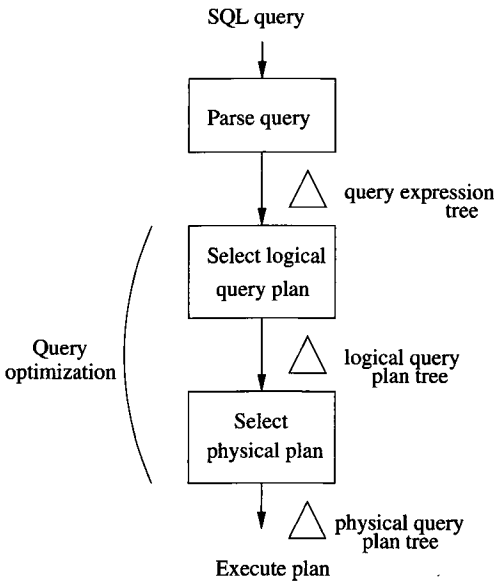

Figure 15.2: Outline of query compilation

15.1 Introduction to Physical-Query-Plan Operators

Physical query plans are built from operators, each of which implements one step of the plan. Often, the physical operators are particular implementations for one of the operations of relational algebra. However, we also need physical operators for other tasks that do not involve an operation of relational algebra. For example, we often need to "scan" a table, that is, bring into main memory each tuple of some relation. The relation is typically an operand of some other operation.

In this section, we shall introduce the basic building blocks of physical query plans. Later sections cover the more complex algorithms that implement operators of relational algebra efficiently; these algorithms also form an essential part of physical query plans. We also introduce here the "iterator" concept, which is an important method by which the operators comprising a physical query plan can pass requests for tuples and answers among themselves.

15.1.1 Scanning Tables

Perhaps the most basic thing we can do in a physical query plan is to read the entire contents of a relation R. A variation of this operator involves a simple predicate, where we read only those tuples of the relation R that satisfy the

predicate. There are two basic approaches to locating the tuples of a relation R.

1. In many cases, the relation R is stored in an area of secondary memory, with its tuples arranged in blocks. The blocks containing the tuples of R are known to the system, and it is possible to get the blocks one by one. This operation is called *table-scan*.

2. If there is an index on any attribute of R, we may be able to use this index to get all the tuples of R. For example, a sparse index on R, as discussed in Section 14.1.3, can be used to lead us to all the blocks holding R, even if we don't know otherwise which blocks these are. This operation is called *index-scan*.

We shall take up index-scan again in Section 15.6.2, when we talk about implementing selection. However, the important observation for now is that we can use the index not only to get *all* the tuples of the relation it indexes, but to get only those tuples that have a particular value (or sometimes a particular range of values) in the attribute or attributes that form the search key for the index.

15.1.2 Sorting While Scanning Tables

There are a number of reasons why we might want to sort a relation as we read its tuples. For one, the query could include an `ORDER BY` clause, requiring that a relation be sorted. For another, some approaches to implementing relational-algebra operations require one or both arguments to be sorted relations. These algorithms appear in Section 15.4 and elsewhere.

The physical-query-plan operator *sort-scan* takes a relation R and a specification of the attributes on which the sort is to be made, and produces R in that sorted order. There are several ways that sort-scan can be implemented. If relation R must be sorted by attribute a, and there is a B-tree index on a, then a scan of the index allows us to produce R in the desired order. If R is small enough to fit in main memory, then we can retrieve its tuples using a table scan or index scan, and then use a main-memory sorting algorithm. If R is too large to fit in main memory, then we can use a multiway merge-sort, as will be discussed Section 15.4.1.

15.1.3 The Computation Model for Physical Operators

A query generally consists of several operations of relational algebra, and the corresponding physical query plan is composed of several physical operators. Since choosing physical-plan operators wisely is an essential of a good query processor, we must be able to estimate the "cost" of each operator we use. We shall use the number of disk I/O's as our measure of cost for an operation. This measure is consistent with our view (see Section 13.3.1) that it takes longer to

get data from disk than to do anything useful with it once the data is in main memory.

When comparing algorithms for the same operations, we shall make an assumption that may be surprising at first:

- We assume that the arguments of any operator are found on disk, but the result of the operator is left in main memory.

If the operator produces the final answer to a query, and that result is indeed written to disk, then the cost of doing so depends only on the size of the answer, and not on how the answer was computed. We can simply add the final write-back cost to the total cost of the query. However, in many applications, the answer is not stored on disk at all, but printed or passed to some formatting program. Then, the disk I/O cost of the output either is zero or depends upon what some unknown application program does with the data. In either case, the cost of writing the answer does not influence our choice of algorithm for executing the operator.

Similarly, the result of an operator that forms part of a query (rather than the whole query) often is not written to disk. In Section 15.1.6 we shall discuss "iterators," where the result of one operator O_1 is constructed in main memory, perhaps a small piece at a time, and passed as an argument to another operator O_2. In this situation, we never have to write the result of O_1 to disk, and moreover, we save the cost of reading from disk an argument of O_2.

15.1.4 Parameters for Measuring Costs

Now, let us introduce the parameters (sometimes called statistics) that we use to express the cost of an operator. Estimates of cost are essential if the optimizer is to determine which of the many query plans is likely to execute fastest. Section 16.5 will show how to exploit these cost estimates.

We need a parameter to represent the portion of main memory that the operator uses, and we require other parameters to measure the size of its argument(s). Assume that main memory is divided into buffers, whose size is the same as the size of disk blocks. Then M will denote the number of main-memory buffers available to an execution of a particular operator.

Sometimes, we can think of M as the entire main memory, or most of the main memory. However, we shall also see situations where several operations share the main memory, so M could be much smaller than the total main memory. In fact, as we shall discuss in Section 15.7, the number of buffers available to an operation may not be a predictable constant, but may be decided during execution, based on what other processes are executing at the same time. If so, M is really an estimate of the number of buffers available to the operation.

Next, let us consider the parameters that measure the cost of accessing argument relations. These parameters, measuring size and distribution of data in a relation, are often computed periodically to help the query optimizer choose physical operators.

We shall make the simplifying assumption that data is accessed one block at a time from disk. In practice, one of the techniques discussed in Section 13.3 might be able to speed up the algorithm if we are able to read many blocks of the relation at once, and they can be read from consecutive blocks on a track. There are three parameter families, B, T, and V:

- When describing the size of a relation R, we most often are concerned with the number of blocks that are needed to hold all the tuples of R. This number of blocks will be denoted $B(R)$, or just B if we know that relation R is meant. Usually, we assume that R is *clustered*; that is, it is stored in B blocks or in approximately B blocks.

- Sometimes, we also need to know the number of tuples in R, and we denote this quantity by $T(R)$, or just T if R is understood. If we need the number of tuples of R that can fit in one block, we can use the ratio T/B.

- Finally, we shall sometimes want to refer to the number of distinct values that appear in a column of a relation. If R is a relation, and one of its attributes is a, then $V(R, a)$ is the number of distinct values of the column for a in R. More generally, if $[a_1, a_2, \ldots, a_n]$ is a list of attributes, then $V(R, [a_1, a_2, \ldots, a_n])$ is the number of distinct n-tuples in the columns of R for attributes $a_1, a_2, \ldots, a_n$. Put formally, it is the number of tuples in $\delta(\pi_{a_1, a_2, \ldots, a_n}(R))$.

15.1.5 I/O Cost for Scan Operators

As a simple application of the parameters that were introduced, we can represent the number of disk I/O's needed for each of the table-scan operators discussed so far. If relation R is clustered, then the number of disk I/O's for the table-scan operator is approximately B. Likewise, if R fits in main-memory, then we can implement sort-scan by reading R into memory and performing an in-memory sort, again requiring only B disk I/O's.

However, if R is not clustered, then the number of required disk I/O's is generally much higher. If R is distributed among tuples of other relations, then a table-scan for R may require reading as many blocks as there are tuples of R; that is, the I/O cost is T. Similarly, if we want to sort R, but R fits in memory, then T disk I/O's are what we need to get all of R into memory.

Finally, let us consider the cost of an index-scan. Generally, an index on a relation R occupies many fewer than $B(R)$ blocks. Therefore, a scan of the entire R, which takes at least B disk I/O's, will require significantly more I/O's than does examining the entire index. Thus, even though index-scan requires examining both the relation and its index,

- We continue to use B or T, respectively, to estimate the cost of accessing a clustered or unclustered relation in its entirety, using an index.

However, if we only want part of R, we often are able to avoid looking at the entire index and the entire R. We shall defer analysis of these uses of indexes to Section 15.6.2.

15.1.6 Iterators for Implementation of Physical Operators

Many physical operators can be implemented as an *iterator*, which is a group of three methods that allows a consumer of the result of the physical operator to get the result one tuple at a time. The three methods forming the iterator for an operation are:

1. `Open()`. This method starts the process of getting tuples, but does not get a tuple. It initializes any data structures needed to perform the operation and calls `Open()` for any arguments of the operation.

2. `GetNext()`. This method returns the next tuple in the result and adjusts data structures as necessary to allow subsequent tuples to be obtained. In getting the next tuple of its result, it typically calls `GetNext()` one or more times on its argument(s). If there are no more tuples to return, `GetNext()` returns a special value `NotFound`, which we assume cannot be mistaken for a tuple.

3. `Close()`. This method ends the iteration after all tuples, or all tuples that the consumer wanted, have been obtained. Typically, it calls `Close()` on any arguments of the operator.

When describing iterators and their methods, we shall assume that there is a "class" for each type of iterator (i.e., for each type of physical operator implemented as an iterator), and the class defines `Open()`, `GetNext()`, and `Close()` methods on instances of the class.

Example 15.1: Perhaps the simplest iterator is the one that implements the table-scan operator. The iterator is implemented by a class `TableScan`, and a table-scan operator in a query plan is an instance of this class parameterized by the relation R we wish to scan. Let us assume that R is a relation clustered in some list of blocks, which we can access in a convenient way; that is, the notion of "get the next block of R" is implemented by the storage system and need not be described in detail. Further, we assume that within a block there is a directory of records (tuples), so it is easy to get the next tuple of a block or tell that the last tuple has been reached.

Figure 15.3 sketches the three methods for this iterator. We imagine a block pointer b and a tuple pointer t that points to a tuple within block b. We assume that both pointers can point "beyond" the last block or last tuple of a block, respectively, and that it is possible to identify when these conditions occur. Notice that `Close()` in this example does nothing. In practice, a `Close()` method for an iterator might clean up the internal structure of the DBMS in various ways. It might inform the buffer manager that certain buffers are no

```
Open() {
    b := the first block of R;
    t := the first tuple of block b;
}

GetNext() {
    IF (t is past the last tuple on block b) {
        increment b to the next block;
        IF (there is no next block)
            RETURN NotFound;
        ELSE /* b is a new block */
            t := first tuple on block b;
    }  /* now we are ready to return t and increment */
    oldt := t;
    increment t to the next tuple of b;
    RETURN oldt;
}

Close() {
}
```

Figure 15.3: Iterator methods for the table-scan operator over relation R

longer needed, or inform the concurrency manager that the read of a relation has completed. □

Example 15.2: Now, let us consider an example where the iterator does most of the work in its `Open()` method. The operator is sort-scan, where we read the tuples of a relation R but return them in sorted order. We cannot return even the first tuple until we have examined each tuple of R. For simplicity, assume that R is small enough to fit in main memory.

`Open()` must read the entire R into main memory. It might also sort the tuples of R, in which case `GetNext()` needs only to return each tuple in turn, in the sorted order. Alternatively, `Open()` could leave R unsorted, and `GetNext()` could select the first of the remaining tuples, in effect performing one pass of a selection sort. □

Example 15.3: Finally, let us consider a simple example of how iterators can be combined by calling other iterators. The operation is the bag union $R \cup S$, in which we produce first all the tuples of R and then all the tuples of S, without regard for the existence of duplicates. Let $\mathcal{R}$ and $\mathcal{S}$ denote the iterators that produce relations R and S, and thus are the "children" of the union operator in a query plan for $R \cup S$. Iterators $\mathcal{R}$ and $\mathcal{S}$ could be table scans applied to stored relations R and S, or they could be iterators that call a network

Why Iterators?

We shall see in Section 16.7 how iterators support efficient execution when they are composed within query plans. They contrast with a *materialization* strategy, where the result of each operator is produced in its entirety — and either stored on disk or allowed to take up space in main memory. When iterators are used, many operations are active at once. Tuples pass between operators as needed, thus reducing the need for storage. Of course, as we shall see, not all physical operators support the iteration approach, or "pipelining," in a useful way. In some cases, almost all the work would need to be done by the `Open()` method, which is tantamount to materialization.

of other iterators to compute R and S. Regardless, all that is important is that we have available methods `R.Open()`, `R.GetNext()`, and `R.Close()`, and analogous methods for iterator S.

The iterator methods for the union are sketched in Fig. 15.4. One subtle point is that the methods use a shared variable `CurRel` that is either R or S, depending on which relation is being read from currently. □

15.2 One-Pass Algorithms

We shall now begin our study of a very important topic in query optimization: how should we execute each of the individual steps — for example, a join or selection — of a logical query plan? The choice of algorithm for each operator is an essential part of the process of transforming a logical query plan into a physical query plan. While many algorithms for operators have been proposed, they largely fall into three classes:

1. Sorting-based methods (Section 15.4).
2. Hash-based methods (Sections 15.5 and 20.1).
3. Index-based methods (Section 15.6).

In addition, we can divide algorithms for operators into three "degrees" of difficulty and cost:

a) Some methods involve reading the data only once from disk. These are the *one-pass* algorithms, and they are the topic of this section. Usually, they require at least one of the arguments to fit in main memory, although there are exceptions, especially for selection and projection as discussed in Section 15.2.1.

```
Open() {
    R.Open();
    CurRel := R;
}

GetNext() {
    IF (CurRel = R) {
        t := R.GetNext();
        IF (t <> NotFound) /* R is not exhausted */
            RETURN t;
        ELSE /* R is exhausted */ {
            S.Open();
            CurRel := S;
        }
    }
    /* here, we must read from S */
    RETURN S.GetNext();
    /* notice that if S is exhausted, S.GetNext()
       will return NotFound, which is the correct
       action for our GetNext as well */
}

Close() {
    R.Close();
    S.Close();
}
```

Figure 15.4: Building a union iterator from iterators $\mathcal{R}$ and $\mathcal{S}$

b) Some methods work for data that is too large to fit in available main memory but not for the largest imaginable data sets. These *two-pass* algorithms are characterized by reading data a first time from disk, processing it in some way, writing all, or almost all, of it to disk, and then reading it a second time for further processing during the second pass. We meet these algorithms in Sections 15.4 and 15.5.

c) Some methods work without a limit on the size of the data. These methods use three or more passes to do their jobs, and are natural, recursive generalizations of the two-pass algorithms. We shall study multipass methods in Section 15.8.

In this section, we shall concentrate on the one-pass methods. Here and subsequently, we shall classify operators into three broad groups:

1. *Tuple-at-a-time, unary operations.* These operations — selection and projection — do not require an entire relation, or even a large part of it, in memory at once. Thus, we can read a block at a time, use one main-memory buffer, and produce our output.

2. *Full-relation, unary operations.* These one-argument operations require seeing all or most of the tuples in memory at once, so one-pass algorithms are limited to relations that are approximately of size M (the number of main-memory buffers available) or less. The operations of this class are γ (the grouping operator) and δ (the duplicate-elimination operator).

3. *Full-relation, binary operations.* All other operations are in this class: set and bag versions of union, intersection, difference, joins, and products. Except for bag union, each of these operations requires at least one argument to be limited to size M, if we are to use a one-pass algorithm.

15.2.1 One-Pass Algorithms for Tuple-at-a-Time Operations

The tuple-at-a-time operations $\sigma(R)$ and $\pi(R)$ have obvious algorithms, regardless of whether the relation fits in main memory. We read the blocks of R one at a time into an input buffer, perform the operation on each tuple, and move the selected tuples or the projected tuples to the output buffer, as suggested by Fig. 15.5. Since the output buffer may be an input buffer of some other operator, or may be sending data to a user or application, we do not count the output buffer as needed space. Thus, we require only that $M \geq 1$ for the input buffer, regardless of B.

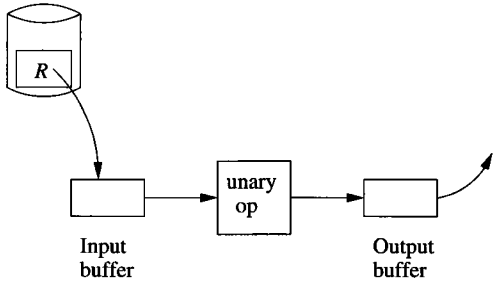

Figure 15.5: A selection or projection being performed on a relation R

The disk I/O requirement for this process depends only on how the argument relation R is provided. If R is initially on disk, then the cost is whatever it takes to perform a table-scan or index-scan of R. The cost was discussed in Section 15.1.5; typically, the cost is B if R is clustered and T if it is not clustered. However, remember the important exception where the operation being performed is a selection, and the condition compares a constant to an

Extra Buffers Can Speed Up Operations

Although tuple-at-a-time operations can get by with only one input buffer and one output buffer, as suggested by Fig. 15.5, we can often speed up processing if we allocate more input buffers. The idea appeared first in Section 13.3.2. If R is stored on consecutive blocks within cylinders, then we can read an entire cylinder into buffers, while paying for the seek time and rotational latency for only one block per cylinder. Similarly, if the output of the operation can be stored on full cylinders, we waste almost no time writing.

attribute that has an index. In that case, we can use the index to retrieve only a subset of the blocks holding R, thus improving performance, often markedly.

15.2.2 One-Pass Algorithms for Unary, Full-Relation Operations

Now, let us consider the unary operations that apply to relations as a whole, rather than to one tuple at a time: duplicate elimination (δ) and grouping (γ).

Duplicate Elimination

To eliminate duplicates, we can read each block of R one at a time, but for each tuple we need to make a decision as to whether:

1. It is the first time we have seen this tuple, in which case we copy it to the output, or
2. We have seen the tuple before, in which case we must not output this tuple.

To support this decision, we need to keep in memory one copy of every tuple we have seen, as suggested in Fig. 15.6. One memory buffer holds one block of R's tuples, and the remaining $M - 1$ buffers can be used to hold a single copy of every tuple seen so far.

When storing the already-seen tuples, we must be careful about the main-memory data structure we use. Naively, we might just list the tuples we have seen. When a new tuple from R is considered, we compare it with all tuples seen so far, and if it is not equal to any of these tuples we both copy it to the output and add it to the in-memory list of tuples we have seen.

However, if there are n tuples in main memory, each new tuple takes processor time proportional to n, so the complete operation takes processor time proportional to n^2. Since n could be very large, this amount of time calls into serious question our assumption that only the disk I/O time is significant. Thus,

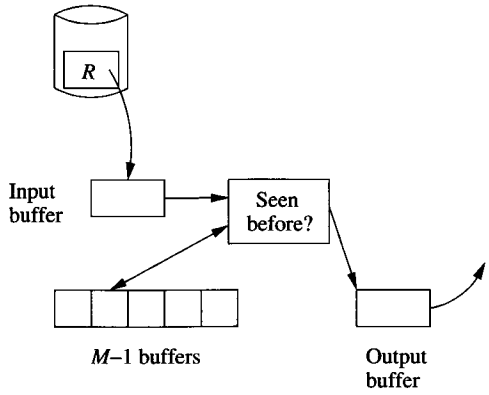

Figure 15.6: Managing memory for a one-pass duplicate-elimination

we need a main-memory structure that allows each us to add a new tuple and to tell whether a given tuple is already there, in time that grows slowly with n.

For example, we could use a hash table with a large number of buckets, or some form of balanced binary search tree.[1] Each of these structures has some space overhead in addition to the space needed to store the tuples; for instance, a main-memory hash table needs a bucket array and space for pointers to link the tuples in a bucket. However, the overhead tends to be small compared with the space needed to store the tuples, and we shall in this chpater neglect this overhead.

On this assumption, we may store in the $M - 1$ available buffers of main memory as many tuples as will fit in $M - 1$ blocks of R. If we want one copy of each distinct tuple of R to fit in main memory, then $B\big(\delta(R)\big)$ must be no larger than $M - 1$. Since we expect M to be much larger than 1, a simpler approximation to this rule, and the one we shall generally use, is:

- $B\big(\delta(R)\big) \leq M$

Note that we cannot in general compute the size of $\delta(R)$ without computing $\delta(R)$ itself. Should we underestimate that size, so $B\big(\delta(R)\big)$ is actually larger than M, we shall pay a significant penalty due to thrashing, as the blocks holding the distinct tuples of R must be brought into and out of main memory frequently.

[1]See Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983 for discussions of suitable main-memory structures. In particular, hashing takes on average $O(n)$ time to process n items, and balanced trees take $O(n \log n)$ time; either is sufficiently close to linear for our purposes.

Grouping

A grouping operation γ_L gives us zero or more grouping attributes and presumably one or more aggregated attributes. If we create in main memory one entry for each group — that is, for each value of the grouping attributes — then we can scan the tuples of R, one block at a time. The *entry* for a group consists of values for the grouping attributes and an accumulated value or values for each aggregation, as follows:

- For a `MIN(a)` or `MAX(a)` aggregate, record the minimum or maximum value, respectively, of attribute a seen for any tuple in the group so far. Change this minimum or maximum, if appropriate, each time a tuple of the group is seen.
- For any `COUNT` aggregation, add one for each tuple of the group that is seen.
- For `SUM(a)`, add the value of attribute a to the accumulated sum for its group, provided a is not `NULL`.
- `AVG(a)` is the hard case. We must maintain two accumulations: the count of the number of tuples in the group and the sum of the a-values of these tuples. Each is computed as we would for a `COUNT` and `SUM` aggregation, respectively. After all tuples of R are seen, we take the quotient of the sum and count to obtain the average.

When all tuples of R have been read into the input buffer and contributed to the aggregation(s) for their group, we can produce the output by writing the tuple for each group. Note that until the last tuple is seen, we cannot begin to create output for a γ operation. Thus, this algorithm does not fit the iterator framework very well; the entire grouping has to be done by the `Open` method before the first tuple can be retrieved by `GetNext`.

In order that the in-memory processing of each tuple be efficient, we need to use a main-memory data structure that lets us find the entry for each group, given values for the grouping attributes. As discussed above for the δ operation, common main-memory data structures such as hash tables or balanced trees will serve well. We should remember, however, that the search key for this structure is the grouping attributes only.

The number of disk I/O's needed for this one-pass algorithm is B, as must be the case for any one-pass algorithm for a unary operator. The number of required memory buffers M is not related to B in any simple way, although typically M will be less than B. The problem is that the entries for the groups could be longer or shorter than tuples of R, and the number of groups could be anything equal to or less than the number of tuples of R. However, in most cases, group entries will be no longer than R's tuples, and there will be many fewer groups than tuples.

Operations on Nonclustered Data

All our calculations regarding the number of disk I/O's required for an operation are predicated on the assumption that the operand relations are clustered. In the (typically rare) event that an operand R is not clustered, then it may take us $T(R)$ disk I/O's, rather than $B(R)$ disk I/O's to read all the tuples of R. Note, however, that any relation that is the result of an operator may always be assumed clustered, since we have no reason to store a temporary relation in a nonclustered fashion.

15.2.3 One-Pass Algorithms for Binary Operations

Let us now take up the binary operations: union, intersection, difference, product, and join. Since in some cases we must distinguish the set- and bag-versions of these operators, we shall subscript them with B or S for "bag" and "set," respectively; e.g., $\cup_B$ for bag union or $-_S$ for set difference. To simplify the discussion of joins, we shall consider only the natural join. An equijoin can be implemented the same way, after attributes are renamed appropriately, and theta-joins can be thought of as a product or equijoin followed by a selection for those conditions that cannot be expressed in an equijoin.

Bag union can be computed by a very simple one-pass algorithm. To compute $R \cup_B S$, we copy each tuple of R to the output and then copy every tuple of S, as we did in Example 15.3. The number of disk I/O's is $B(R) + B(S)$, as it must be for a one-pass algorithm on operands R and S, while $M = 1$ suffices regardless of how large R and S are.

Other binary operations require reading the smaller of the operands R and S into main memory and building a suitable data structure so tuples can be both inserted quickly and found quickly, as discussed in Section 15.2.2. As before, a hash table or balanced tree suffices. Thus, the approximate requirement for a binary operation on relations R and S to be performed in one pass is:

- $\min\big(B(R), B(S)\big) \leq M$

More preceisely, one buffer is used to read the blocks of the larger relation, while approximately M buffers are needed to house the entire smaller relation and its main-memory data structure.

We shall now give the details of the various operations. In each case, we assume R is the larger of the relations, and we house S in main memory.

Set Union

We read S into $M - 1$ buffers of main memory and build a search structure whose search key is the entire tuple. All these tuples are also copied to the output. We then read each block of R into the Mth buffer, one at a time. For

each tuple t of R, we see if t is in S, and if not, we copy t to the output. If t is also in S, we skip t.

Set Intersection

Read S into $M - 1$ buffers and build a search structure with full tuples as the search key. Read each block of R, and for each tuple t of R, see if t is also in S. If so, copy t to the output, and if not, ignore t.

Set Difference

Since difference is not commutative, we must distinguish between $R -_S S$ and $S -_S R$, continuing to assume that R is the larger relation. In each case, read S into $M - 1$ buffers and build a search structure with full tuples as the search key.

To compute $R -_S S$, we read each block of R and examine each tuple t on that block. If t is in S, then ignore t; if it is not in S then copy t to the output.

To compute $S -_S R$, we again read the blocks of R and examine each tuple t in turn. If t is in S, then we delete t from the copy of S in main memory, while if t is not in S we do nothing. After considering each tuple of R, we copy to the output those tuples of S that remain.

Bag Intersection

We read S into $M - 1$ buffers, but we associate with each distinct tuple a *count*, which initially measures the number of times this tuple occurs in S. Multiple copies of a tuple t are not stored individually. Rather we store one copy of t and associate with it a count equal to the number of times t occurs.

This structure could take slightly more space than $B(S)$ blocks if there were few duplicates, although frequently the result is that S is compacted. Thus, we shall continue to assume that $B(S) \leq M$ is sufficient for a one-pass algorithm to work, although the condition is only an approximation.

Next, we read each block of R, and for each tuple t of R we see whether t occurs in S. If not we ignore t; it cannot appear in the intersection. However, if t appears in S, and the count associated with t is still positive, then we output t and decrement the count by 1. If t appears in S, but its count has reached 0, then we do not output t; we have already produced as many copies of t in the output as there were copies in S.

Bag Difference

To compute $S -_B R$, we read the tuples of S into main memory, and count the number of occurrences of each distinct tuple, as we did for bag intersection. When we read R, for each tuple t we see whether t occurs in S, and if so, we decrement its associated count. At the end, we copy to the output each tuple

in main memory whose count is positive, and the number of times we copy it equals that count.

To compute $R -_B S$, we also read the tuples of S into main memory and count the number of occurrences of distinct tuples. We may think of a tuple t with a count of c as c reasons not to copy t to the output as we read tuples of R. That is, when we read a tuple t of R, we see if t occurs in S. If not, then we copy t to the output. If t does occur in S, then we look at the current count c associated with t. If $c = 0$, then copy t to the output. If $c > 0$, do not copy t to the output, but decrement c by 1.

Product

Read S into $M - 1$ buffers of main memory; no special data structure is needed. Then read each block of R, and for each tuple t of R concatenate t with each tuple of S in main memory. Output each concatenated tuple as it is formed.

This algorithm may take a considerable amount of processor time per tuple of R, because each such tuple must be matched with $M - 1$ blocks full of tuples. However, the output size is also large, and the time per output tuple is small.

Natural Join

In this and other join algorithms, let us take the convention that $R(X, Y)$ is being joined with $S(Y, Z)$, where Y represents all the attributes that R and S have in common, X is all attributes of R that are not in the schema of S, and Z is all attributes of S that are not in the schema of R. We continue to assume that S is the smaller relation. To compute the natural join, do the following:

1. Read all the tuples of S and form them into a main-memory search structure with the attributes of Y as the search key. Use $M - 1$ blocks of memory for this purpose.

2. Read each block of R into the one remaining main-memory buffer. For each tuple t of R, find the tuples of S that agree with t on all attributes of Y, using the search structure. For each matching tuple of S, form a tuple by joining it with t, and move the resulting tuple to the output.

Like all the one-pass, binary algorithms, this one takes $B(R) + B(S)$ disk I/O's to read the operands. It works as long as $B(S) \leq M - 1$, or approximately, $B(S) \leq M$.

We shall not discuss joins other than the natural join. Remember that an equijoin is executed in essentially the same way as a natural join, but we must account for the fact that "equal" attributes from the two relations may have different names. A theta-join that is not an equijoin can be replaced by an equijoin or product followed by a selection.

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) projection (b) distinct (δ) (c) grouping (γ_L) (d) set union (e) set intersection (f) set difference (g) bag intersection (h) bag difference (i) product (j) natural join.

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: 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?

! **Exercise 15.2.4:** Give one-pass algorithms for each of the following join-like operators:

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

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

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

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

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

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

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

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

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

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-Based Nested-Loop Join

The simplest variation of nested-loop join has loops that range over individual tuples of the relations involved. In this algorithm, which we call *tuple-based nested-loop join*, we compute the join $R(X,Y) \bowtie S(Y,Z)$ as follows:

```
FOR each tuple s in S DO
    FOR each tuple r in R DO
        IF r and s join to make a tuple t THEN
            output t;
```

If we are careless about how we buffer the blocks of relations R and S, then this algorithm could require as many as $T(R)T(S)$ disk I/O's. However, there are many situations where this algorithm can be modified to have much lower cost. One case is when we can use an index on the join attribute or attributes of R to find the tuples of R that match a given tuple of S, without having to read the entire relation R. We discuss index-based joins in Section 15.6.3. A second improvement looks much more carefully at the way tuples of R and S are divided among blocks, and uses as much of the memory as it can to reduce the number of disk I/O's as we go through the inner loop. We shall consider this block-based version of nested-loop join in Section 15.3.3.

15.3.2 An Iterator for Tuple-Based Nested-Loop Join

One advantage of a nested-loop join is that it fits well into an iterator framework, and thus, as we shall see in Section 16.7.3, allows us to avoid storing intermediate relations on disk in some situations. The iterator for $R \bowtie S$ is easy to build from the iterators for R and S, which support methods `R.Open()`, and so on, as in Section 15.1.6. The code for the three iterator methods for nested-loop join is in Fig. 15.7. It makes the assumption that neither relation R nor S is empty.

15.3.3 Block-Based Nested-Loop Join Algorithm

We can improve on the tuple-based nested-loop join of Section 15.3.1 if we compute $R \bowtie S$ by:

1. Organizing access to both argument relations by blocks, and
2. Using as much main memory as we can to store tuples belonging to the relation S, the relation of the outer loop.

Point (1) makes sure that when we run through the tuples of R in the inner loop, we use as few disk I/O's as possible to read R. Point (2) enables us to join each tuple of R that we read with not just one tuple of S, but with as many tuples of S as will fit in memory.

```
Open() {
    R.Open();
    S.Open();
    s := S.GetNext();
}

GetNext() {
    REPEAT {
        r := R.GetNext();
        IF (r = NotFound) { /* R is exhausted for
                the current s */
            R.Close();
            s := S.GetNext();
            IF (s = NotFound) RETURN NotFound;
                /* both R and S are exhausted */
            R.Open();
            r := R.GetNext();
        }
    }
    UNTIL (r and s join);
    RETURN the join of r and s;
}

Close() {
    R.Close();
    S.Close();
}
```

Figure 15.7: Iterator methods for tuple-based nested-loop join of R and S

As in Section 15.2.3, let us assume $B(S) \leq B(R)$, but now let us also assume that $B(S) > M$; i.e., neither relation fits entirely in main memory. We repeatedly read $M-1$ blocks of S into main-memory buffers. A search structure, with search key equal to the common attributes of R and S, is created for the tuples of S that are in main memory. Then we go through all the blocks of R, reading each one in turn into the last block of memory. Once there, we compare all the tuples of R's block with all the tuples in all the blocks of S that are currently in main memory. For those that join, we output the joined tuple. The nested-loop structure of this algorithm can be seen when we describe the algorithm more formally, in Fig. 15.8. The algorithm of Fig. 15.8 is sometimes called "nested-block join." We shall continue to call it simply *nested-loop join*, since it is the variant of the nested-loop idea most commonly implemented in practice.

```
FOR each chunk of M-1 blocks of S DO BEGIN
    read these blocks into main-memory buffers;
    organize their tuples into a search structure whose
        search key is the common attributes of R and S;
    FOR each block b of R DO BEGIN
        read b into main memory;
        FOR each tuple t of b DO BEGIN
            find the tuples of S in main memory that
                join with t;
            output the join of t with each of these tuples;
        END;
    END;
END;
```

Figure 15.8: The nested-loop join algorithm

The program of Fig. 15.8 appears to have three nested loops. However, there really are only two loops if we look at the code at the right level of abstraction. The first, or outer loop, runs through the tuples of S. The other two loops run through the tuples of R. However, we expressed the process as two loops to emphasize that the order in which we visit the tuples of R is not arbitrary. Rather, we need to look at these tuples a block at a time (the role of the second loop), and within one block, we look at all the tuples of that block before moving on to the next block (the role of the third loop).

Example 15.4: Let $B(R) = 1000$, $B(S) = 500$, and $M = 101$. We shall use 100 blocks of memory to buffer S in 100-block chunks, so the outer loop of Fig. 15.8 iterates five times. At each iteration, we do 100 disk I/O's to read the chunk of S, and we must read R entirely in the second loop, using 1000 disk I/O's. Thus, the total number of disk I/O's is 5500.

Notice that if we reversed the roles of R and S, the algorithm would use slightly more disk I/O's. We would iterate 10 times through the outer loop and do 600 disk I/O's at each iteration, for a total of 6000. In general, there is a slight advantage to using the smaller relation in the outer loop. □

15.3.4 Analysis of Nested-Loop Join

The analysis of Example 15.4 can be repeated for any $B(R)$, $B(S)$, and M. Assuming S is the smaller relation, the number of chunks, or iterations of the outer loop is $B(S)/(M-1)$. At each iteration, we read $M-1$ blocks of S and $B(R)$ blocks of R. The number of disk I/O's is thus $B(S)\big(M-1+B(R)\big)/(M-1)$, or $B(S)+\big(B(S)B(R)\big)/(M-1)$.

Assuming all of M, $B(S)$, and $B(R)$ are large, but M is the smallest of these, an approximation to the above formula is $B(S)B(R)/M$. That is, the

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
$\cup$, $\cap$, $-$, $\times$, $\bowtie$	$\min\big(B(R), B(S)\big)$	$B(R) + B(S)$	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: Give the three iterator methods for the block-based version of nested-loop join.

Exercise 15.3.2: Suppose $B(R) = B(S) = 10{,}000$, and $M = 1000$. Calculate the disk I/O cost of a nested-loop join.

Exercise 15.3.3: For the relations of Exercise 15.3.2, what value of M would we need to compute $R \bowtie S$ using the nested-loop algorithm with no more than (a) 100,000 ! (b) 25,000 ! (c) 15,000 disk I/O's?

! **Exercise 15.3.4:** 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.

! **Exercise 15.3.5:** 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.

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

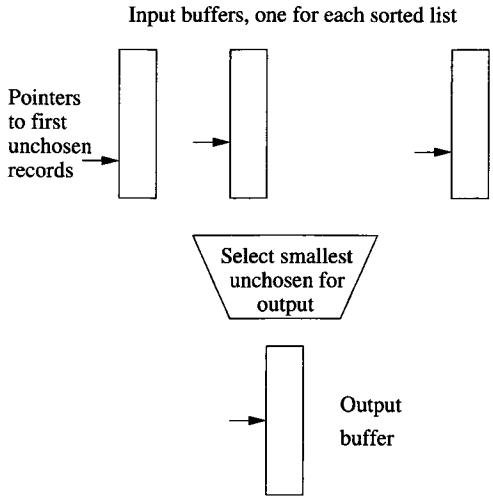

Figure 15.10: Main-memory organization for multiway merging

use of buffers is suggested by Fig. 15.10. A pointer to each input block indicates the first element in the sorted order that has not yet been moved to the output. We merge the sorted sublists into one sorted list with all the records as follows.

1. Find the smallest key among the first remaining elements of all the lists. Since this comparison is done in main memory, a linear search is sufficient, taking a number of machine instructions proportional to the number of sublists. However, if we wish, there is a method based on "priority queues"[2] that takes time proportional to the logarithm of the number of sublists to find the smallest element.
2. Move the smallest element to the first available position of the output block.
3. If the output block is full, write it to disk and reinitialize the same buffer in main memory to hold the next output block.
4. If the block from which the smallest element was just taken is now exhausted of records, read the next block from the same sorted sublist into the same buffer that was used for the block just exhausted. If no blocks remain, then leave its buffer empty and do not consider elements from that list in any further competition for smallest remaining elements.

In order for TPMMS to work, there must be no more than $M - 1$ sublists. Suppose R fits on B blocks. Since each sublist consists of M blocks, the number

[2]See Aho, A. V. and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, 1992.

of sublists is B/M. We thus require $B/M \leq M - 1$, or $B \leq M(M-1)$ (or about $B \leq M^2$).

The algorithm requires us to read B blocks in the first pass, and another B disk I/O's to write the sorted sublists. The sorted sublists are each read again in the second pass, resulting in a total of $3B$ disk I/O's. If, as is customary, we do not count the cost of writing the result to disk (since the result may be pipelined and never written to disk), then $3B$ is all that the sorting operator τ requires. However, if we need to store the result on disk, then the requirement is $4B$.

Example 15.5: Suppose blocks are 64K bytes, and we have one gigabyte of main memory. Then we can afford M of 16K. Thus, a relation fitting in B blocks can be sorted as long as B is no more than $(16\text{K})^2 = 2^{28}$. Since blocks are of size $64\text{K} = 2^{14}$, a relation can be sorted as long as its size is no greater than 2^{42} bytes, or 4 terabytes. □

Example 15.5 shows that even on a modest machine, 2PMMS is sufficient to sort all but an incredibly large relation in two passes. However, if you have an even bigger relation, then the same idea can be applied recursively. Divide the relation into chunks of size $M(M-1)$, use 2PMMS to sort each one, and then treat the resulting sorted lists as sublists for a third pass. The idea extends similarly to any number of passes.

15.4.2 Duplicate Elimination Using Sorting

To perform the $\delta(R)$ operation in two passes, we sort the tuples of R in sublists as in 2PMMS. In the second pass, we use the available main memory to hold one block from each sorted sublist and one output block, as we did for 2PMMS. However, instead of sorting on the second pass, we reapeatedly select the first (in sorted order) unconsidered tuple t among all the sorted sublists. We write one copy of t to the output and eliminate from the input blocks all occurrences of t. Thus, the output will consist of exactly one copy of any tuple in R; they will in fact be produced in sorted order. When an output block is full or an input block empty, we manage the buffers exactly as in 2PMMS.

The number of disk I/O's performed by this algorithm, as always ignoring the handling of the output, is the same as for sorting: $3B(R)$. This figure can be compared with $B(R)$ for the single-pass algorithm of Section 15.2.2. On the other hand, we can handle much larger files using the two-pass algorithm than with the one-pass algorithm. As for 2PMMS, approximately $B \leq M^2$ is required for the two-pass algorithm to be feasible, compared with $B \leq M$ for the one-pass algorithm. Put another way, to eliminate duplicates with the two-pass algorithm requires only $\sqrt{B(R)}$ blocks of main memory, rather than the $B(R)$ blocks required for a one-pass algorithm.

15.4.3 Grouping and Aggregation Using Sorting

The two-pass algorithm for $\gamma_L(R)$ is quite similar to the algorithm for $\delta(R)$ or 2PMMS. We summarize it as follows:

1. Read the tuples of R into memory, M blocks at a time. Sort the tuples in each set of M blocks, using the grouping attributes of L as the sort key. Write each sorted sublist to disk.

2. Use one main-memory buffer for each sublist, and initially load the first block of each sublist into its buffer.

3. Repeatedly find the least value of the sort key (grouping attributes) present among the first available tuples in the buffers. This value, v, becomes the next group, for which we:

   (a) Prepare to compute all the aggregates on list L for this group. As in Section 15.2.2, use a count and sum in place of an average.

   (b) Examine each of the tuples with sort key v, and accumulate the needed aggregates.

   (c) If a buffer becomes empty, replace it with the next block from the same sublist.

   When there are no more tuples with sort key v available, output a tuple consisting of the grouping attributes of L and the associated values of the aggregations we have computed for the group.

As for the δ algorithm, this two-pass algorithm for γ takes $3B(R)$ disk I/O's, and will work as long as $B(R) \leq M^2$.

15.4.4 A Sort-Based Union Algorithm

When bag-union is wanted, the one-pass algorithm of Section 15.2.3, where we simply copy both relations, works regardless of the size of the arguments, so there is no need to consider a two-pass algorithm for $\cup_B$. However, the one-pass algorithm for $\cup_S$ only works when at least one relation is smaller than the available main memory, so we must consider a two-pass algorithm for set union. The methodology we present works for the set and bag versions of intersection and difference as well, as we shall see in Section 15.4.5. To compute $R \cup_S S$, we modify 2PMMS as follows:

1. In the first phase, create sorted sublists from both R and S.

2. Use one main-memory buffer for each sublist of R and S. Initialize each with the first block from the corresponding sublist.

3. Repeatedly find the first remaining tuple t among all the buffers. Copy t to the output, and remove from the buffers all copies of t (if R and S are sets there should be at most two copies). Manage empty input buffers and a full output buffer as for 2PMMS.

We observe that each tuple of R and S is read twice into main memory, once when the sublists are being created, and the second time as part of one of the sublists. The tuple is also written to disk once, as part of a newly formed sublist. Thus, the cost in disk I/O's is $3\big(B(R) + B(S)\big)$.

The algorithm works as long as the total number of sublists among the two relations does not exceed $M - 1$, because we need one buffer for each sublist and one for the output Thus, approximately, the sum of the sizes of the two relations must not exceed M^2; that is, $B(R) + B(S) \leq M^2$.

15.4.5 Sort-Based Intersection and Difference

Whether the set version or the bag version is wanted, the algorithms are essentially the same as that of Section 15.4.4, except that the way we handle the copies of a tuple t at the fronts of the sorted sublists differs. For each algorithm, we repeatedly consider the tuple t that is least in the sorted order among all tuples remaining in the input buffers. We produce output as follows, and then remove all copies of t from the input buffers.

- For set intersection, output t if it appears in both R and S.
- For bag intersection, output t the minimum of the number of times it appears in R and in S. Note that t is not output if either of these counts is 0; that is, if t is missing from one or both of the relations.
- For set difference, $R -_S S$, output t if and only if it appears in R but not in S.
- For bag difference, $R -_B S$, output t the number of times it appears in R minus the number of times it appears in S. Of course, if t appears in S at least as many times as it appears in R, then do not output t at all.

One subtlely must be remembered for the bag operations. When counting occurrences of t, it is possible that all remaining tuples in an input buffer are t. If so, there may be more t's on the next block for that sublist. Thus, when a buffer has only t's remaining, we must load the next block for that sublist, continuing the count of t's. This process may continue for several blocks and may need to be done for several sublists.

The analysis of this family of algorithms is the same as for the set-union algorithm described in Section 15.4.4:

- $3\big(B(R) + B(S)\big)$ disk I/O's.
- Approximately $B(R) + B(S) \leq M^2$ for the algorithm to work.

15.4.6 A Simple Sort-Based Join Algorithm

There are several ways that sorting can be used to join large relations. Before examining the join algorithms, let us observe one problem that can occur when we compute a join but was not an issue for the binary operations considered so far. When taking a join, the number of tuples from the two relations that share a common value of the join attribute(s), and therefore need to be in main memory simultaneously, can exceed what fits in memory. The extreme example is when there is only one value of the join attribute(s), and every tuple of one relation joins with every tuple of the other relation. In this situation, there is really no choice but to take a nested-loop join of the two sets of tuples with a common value in the join-attribute(s).

To avoid facing this situation, we can try to reduce main-memory use for other aspects of the algorithm, and thus make available a large number of buffers to hold the tuples with a given join-attribute value. In this section we shall discuss the algorithm that makes the greatest possible number of buffers available for joining tuples with a common value. In Section 15.4.8 we consider another sort-based algorithm that uses fewer disk I/O's, but can present problems when there are large numbers of tuples with a common join-attribute value.

Given relations $R(X,Y)$ and $S(Y,Z)$ to join, and given M blocks of main memory for buffers, we do the following:

1. Sort R, using 2PMMS, with Y as the sort key.
2. Sort S similarly.
3. Merge the sorted R and S. We use only two buffers: one for the current block of R and the other for the current block of S. The following steps are done repeatedly:
   (a) Find the least value y of the join attributes Y that is currently at the front of the blocks for R and S.
   (b) If y does not appear at the front of the other relation, then remove the tuple(s) with sort key y.
   (c) Otherwise, identify all the tuples from both relations having sort key y. If necessary, read blocks from the sorted R and/or S, until we are sure there are no more y's in either relation. As many as M buffers are available for this purpose.
   (d) Output all the tuples that can be formed by joining tuples from R and S that have a common Y-value y.
   (e) If either relation has no more unconsidered tuples in main memory, reload the buffer for that relation.

Example 15.6: Let us consider the relations R and S from Example 15.4. Recall these relations occupy 1000 and 500 blocks, respectively, and there are $M = 101$ main-memory buffers. When we use 2PMMS on a relation and store

the result on disk, we do four disk I/O's per block, two in each of the two phases. Thus, we use $4\big(B(R)+B(S)\big)$ disk I/O's to sort R and S, or 6000 disk I/O's.

When we merge the sorted R and S to find the joined tuples, we read each block of R and S a fifth time, using another 1500 disk I/O's. In this merge we generally need only two of the 101 blocks of memory. However, if necessary, we could use all 101 blocks to hold the tuples of R and S that share a common Y-value y. Thus, it is sufficient that for no y do the tuples of R and S that have Y-value y together occupy more than 101 blocks.

Notice that the total number of disk I/O's performed by this algorithm is 7500, compared with 5500 for nested-loop join in Example 15.4. However, nested-loop join is inherently a quadratic algorithm, taking time proportional to $B(R)B(S)$, while sort-join has linear I/O cost, taking time proportional to $B(R)+B(S)$. It is only the constant factors and the small size of the example (each relation is only 5 or 10 times larger than a relation that fits entirely in the allotted buffers) that make nested-loop join preferable. □

15.4.7 Analysis of Simple Sort-Join

As we noted in Example 15.6, the algorithm of Section 15.4.6 performs five disk I/O's for every block of the argument relations. We also need to consider how big M needs to be in order for the simple sort-join to work. The primary constraint is that we need to be able to perform the two-phase, multiway merge sorts on R and S. As we observed in Section 15.4.1, we need $B(R) \leq M^2$ and $B(S) \leq M^2$ to perform these sorts. In addition, we require that all the tuples with a common Y-value must fit in M buffers. In summary:

- The simple sort-join uses $5\big(B(R)+B(S)\big)$ disk I/O's.
- It requires $B(R) \leq M^2$ and $B(S) \leq M^2$ to work.
- It also requires that the tuples with a common value for the join attributes fit in M blocks.

15.4.8 A More Efficient Sort-Based Join

If we do not have to worry about very large numbers of tuples with a common value for the join attribute(s), then we can save two disk I/O's per block by combining the second phase of the sorts with the join itself. We call this algorithm *sort-join*; other names by which it is known include "merge-join" and "sort-merge-join." To compute $R(X,Y) \bowtie S(Y,Z)$ using M main-memory buffers:

1. Create sorted sublists of size M, using Y as the sort key, for both R and S.

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 $3(B(R) + B(S))$. 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) distinct (δ) (b) grouping (γ_L) (c) set intersection (d) bag difference (e) natural join.

[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	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
$\bowtie$	$\sqrt{\max\big(B(R),B(S)\big)}$	$5\big(B(R)+B(S)\big)$	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: If $B(R) = B(S) = 10{,}000$ and $M = 1000$, what are the disk I/O requirements of: (a) set union (b) simple sort-join (c) the more efficient sort-join of Section 15.4.8.

! **Exercise 15.4.3:** 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 15.4.4:** 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 15.4.5:** Repeat Exercise 15.4.4 for the more efficient sort-join of Section 15.4.8.

Exercise 15.4.6: How much memory do we need to use a two-pass, sort-based algorithm for relations of 10,000 blocks each, if the operation is: (a) δ (b) γ (c) a binary operation such as join or union.

Exercise 15.4.7: Describe a two-pass, sort-based algorithm for each of the join-like operators of Exercise 15.2.4.

! **Exercise 15.4.8:** 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?

!! **Exercise 15.4.9:** 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?

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

```
initialize M-1 buckets using M-1 empty buffers;
FOR each block b of relation R DO BEGIN
    read block b into the Mth buffer;
    FOR each tuple t in b DO BEGIN
        IF the buffer for bucket h(t) has no room for t THEN
            BEGIN
                copy the buffer to disk;
                initialize a new empty block in that buffer;
            END;
        copy t to the buffer for bucket h(t);
    END;
END;
FOR each bucket DO
    IF the buffer for this bucket is not empty THEN
        write the buffer to disk;
```

Figure 15.12: Partitioning a relation R into $M-1$ buckets

R_i is the portion of R that hashes to the ith bucket. The one-pass algorithm of Section 15.2.2 can be used to eliminate duplicates from each R_i in turn and write out the resulting unique tuples.

This method will work as long as the individual R_i's are sufficiently small to fit in main memory and thus allow a one-pass algorithm. Since we may assume the hash function h partitions R into equal-sized buckets, each R_i will be approximately $B(R)/(M-1)$ blocks in size. If that number of blocks is no larger than M, i.e., $B(R) \leq M(M-1)$, then the two-pass, hash-based algorithm will work. In fact, as we discussed in Section 15.2.2, it is only necessary that the number of distinct tuples in one bucket fit in M buffers. Thus, a conservative estimate (assuming M and $M-1$ are essentially the same) is $B(R) \leq M^2$, exactly as for the sort-based, two-pass algorithm for δ.

The number of disk I/O's is also similar to that of the sort-based algorithm. We read each block of R once as we hash its tuples, and we write each block of each bucket to disk. We then read each block of each bucket again in the one-pass algorithm that focuses on that bucket. Thus, the total number of disk I/O's is $3B(R)$.

15.5.3 Hash-Based Grouping and Aggregation

To perform the $\gamma_L(R)$ operation, we again start by hashing all the tuples of R to $M-1$ buckets. However, in order to make sure that all tuples of the same group wind up in the same bucket, we must choose a hash function that depends only on the grouping attributes of the list L.

Having partitioned R into buckets, we can then use the one-pass algorithm for γ from Section 15.2.2 to process each bucket in turn. As we discussed

for δ in Section 15.5.2, we can process each bucket in main memory provided $B(R) \leq M^2$.

However, on the second pass, we need only one record per group as we process each bucket. Thus, even if the size of a bucket is larger than M, we can handle the bucket in one pass provided the records for all the groups in the bucket take no more than M buffers. As a consequence, if groups are large, then we may actually be able to handle much larger relations R than is indicated by the $B(R) \leq M^2$ rule. On the other hand, if M exceeds the number of groups, then we cannot fill all buckets. Thus, the actual limitation on the size of R as a function of M is complex, but $B(R) \leq M^2$ is a conservative estimate. Finally, we observe that the number of disk I/O's for γ, as for δ, is $3B(R)$.

15.5.4 Hash-Based Union, Intersection, and Difference

When the operation is binary, we must make sure that we use the same hash function to hash tuples of both arguments. For example, to compute $R \cup_S S$, we hash both R and S to $M-1$ buckets each, say $R_1, R_2, \ldots, R_{M-1}$ and $S_1, S_2, \ldots, S_{M-1}$. We then take the set-union of R_i with S_i for all i, and output the result. Notice that if a tuple t appears in both R and S, then for some i we shall find t in both R_i and S_i. Thus, when we take the union of these two buckets, we shall output only one copy of t, and there is no possibility of introducing duplicates into the result. For $\cup_B$, the simple bag-union algorithm of Section 15.2.3 is preferable to any other approach for that operation.

To take the intersection or difference of R and S, we create the $2(M-1)$ buckets exactly as for set-union and apply the appropriate one-pass algorithm to each pair of corresponding buckets. Notice that all these one-pass algorithms require $B(R) + B(S)$ disk I/O's. To this quantity we must add the two disk I/O's per block that are necessary to hash the tuples of the two relations and store the buckets on disk, for a total of $3\big(B(R) + B(S)\big)$ disk I/O's.

In order for the algorithms to work, we must be able to take the one-pass union, intersection, or difference of R_i and S_i, whose sizes will be approximately $B(R)/(M-1)$ and $B(S)/(M-1)$, respectively. Recall that the one-pass algorithms for these operations require that the smaller operand occupies at most $M-1$ blocks. Thus, the two-pass, hash-based algorithms require that $\min\big(B(R), B(S)\big) \leq M^2$, approximately.

15.5.5 The Hash-Join Algorithm

To compute $R(X,Y) \bowtie S(Y,Z)$ using a two-pass, hash-based algorithm, we act almost as for the other binary operations discussed in Section 15.5.4. The only difference is that we must use as the hash key just the join attributes, Y. Then we can be sure that if tuples of R and S join, they will wind up in corresponding buckets R_i and S_i for some i. A one-pass join of all pairs of

corresponding buckets completes this algorithm, which we call *hash-join*.[4]

Example 15.8: Let us renew our discussion of the two relations R and S from Example 15.4, whose sizes were 1000 and 500 blocks, respectively, and for which 101 main-memory buffers are made available. We may hash each relation to 100 buckets, so the average size of a bucket is 10 blocks for R and 5 blocks for S. Since the smaller number, 5, is much less than the number of available buffers, we expect to have no trouble performing a one-pass join on each pair of buckets.

The number of disk I/O's is 1500 to read each of R and S while hashing into buckets, another 1500 to write all the buckets to disk, and a third 1500 to read each pair of buckets into main memory again while taking the one-pass join of corresponding buckets. Thus, the number of disk I/O's required is 4500, just as for the efficient sort-join of Section 15.4.8. □

We may generalize Example 15.8 to conclude that:

- Hash join requires $3(B(R) + B(S))$ disk I/O's to perform its task.
- The two-pass hash-join algorithm will work as long as approximately $\min(B(R), B(S)) \leq M^2$.

The argument for the latter point is the same as for the other binary operations: one of each pair of buckets must fit in $M - 1$ buffers.

15.5.6 Saving Some Disk I/O's

If there is more memory available on the first pass than we need to hold one block per bucket, then we have some opportunities to save disk I/O's. One option is to use several blocks for each bucket, and write them out as a group, in consecutive blocks of disk. Strictly speaking, this technique doesn't save disk I/O's, but it makes the I/O's go faster, since we save seek time and rotational latency when we write.

However, there are several tricks that have been used to avoid writing some of the buckets to disk and then reading them again. The most effective of them, called *hybrid hash-join*, works as follows. In general, suppose we decide that to join $R \bowtie S$, with S the smaller relation, we need to create k buckets, where k is much less than M, the available memory. When we hash S, we can choose to keep m of the k buckets entirely in main memory, while keeping only one block for each of the other $k - m$ buckets. We can manage to do so provided the expected size of the buckets in memory, plus one block for each of the other buckets, does not exceed M; that is:

$$mB(S)/k + k - m \leq M \tag{15.1}$$

[4]Sometimes, the term "hash-join" is reserved for the variant of the one-pass join algorithm of Section 15.2.3 in which a hash table is used as the main-memory search structure. Then, the two-pass hash-join algorithm described here is called "partition hash-join."

In explanation, the expected size of a bucket is $B(S)/k$, and there are m buckets in memory.

Now, when we read the tuples of the other relation, R, to hash that relation into buckets, we keep in memory:

1. The m buckets of S that were never written to disk, and
2. One block for each of the $k-m$ buckets of R whose corresponding buckets of S were written to disk.

If a tuple t of R hashes to one of the first m buckets, then we immediately join it with all the tuples of the corresponding S-bucket, as if this were a one-pass, hash-join. It is necessary to organize each of the in-memory buckets of S into an efficient search structure to facilitate this join, just as for the one-pass hash-join. If t hashes to one of the buckets whose corresponding S-bucket is on disk, then t is sent to the main-memory block for that bucket, and eventually migrates to disk, as for a two-pass, hash-based join.

On the second pass, we join the corresponding buckets of R and S as usual. However, there is no need to join the pairs of buckets for which the S-bucket was left in memory; these buckets have already been joined and their result output.

The savings in disk I/O's is equal to two for every block of the buckets of S that remain in memory, and their corresponding R-buckets. Since m/k of the buckets are in memory, the savings is $2(m/k)\big(B(R)+B(S)\big)$. We must thus ask how to maximize m/k, subject to the constraint of Equation (15.1). The surprising answer is: pick $m=1$, and then make k as small as possible.

The intuitive justification is that all but $k-m$ of the main-memory buffers can be used to hold tuples of S in main memory, and the more of these tuples, the fewer the disk I/O's. Thus, we want to minimize k, the total number of buckets. We do so by making each bucket about as big as can fit in main memory; that is, buckets are of size M, and therefore $k=B(S)/M$. If that is the case, then there is only room for one bucket in the extra main memory; i.e., $m=1$.

In fact, we really need to make the buckets slightly smaller than $B(S)/M$, or else we shall not quite have room for one full bucket and one block for the other $k-1$ buckets in memory at the same time. Assuming, for simplicity, that k is about $B(S)/M$ and $m=1$, the savings in disk I/O's is

$$2M\big(B(R)+B(S)\big)/B(S)$$

and the total cost is $\big(3-2M/B(S)\big)\big(B(R)+B(S)\big)$.

Example 15.9: Consider the problem of Example 15.4, where we had to join relations R and S, of 1000 and 500 blocks, respectively, using $M=101$. If we use a hybrid hash-join, then we want k, the number of buckets, to be about $500/101$. Suppose we pick $k=5$. Then the average bucket will have 100 blocks

of S's tuples. If we try to fit one of these buckets and four extra blocks for the other four buckets, we need 104 blocks of main memory, and we cannot take the chance that the in-memory bucket will overflow memory.

Thus, we are advised to choose $k = 6$. Now, when hashing S on the first pass, we have five buffers for five of the buckets, and we have up to 96 buffers for the in-memory bucket, whose expected size is 500/6 or 83. The number of disk I/O's we use for S on the first pass is thus 500 to read all of S, and $500 - 83 = 417$ to write five buckets to disk. When we process R on the first pass, we need to read all of R (1000 disk I/O's) and write 5 of its 6 buckets (833 disk I/O's).

On the second pass, we read all the buckets written to disk, or $417 + 833 = 1250$ additional disk I/O's. The total number of disk I/O's is thus 1500 to read R and S, 1250 to write 5/6 of these relations, and another 1250 to read those tuples again, or 4000 disk I/O's. This figure compares with the 4500 disk I/O's needed for the straightforward hash-join or sort-join. □

15.5.7 Summary of Hash-Based Algorithms

Figure 15.13 gives the memory requirements and disk I/O's needed by each of the algorithms discussed in this section. As with other types of algorithms, we should observe that the estimates for γ and δ may be conservative, since they really depend on the number of duplicates and groups, respectively, rather than on the number of tuples in the argument relation.

Operators	Approximate M required	Disk I/O	Section
γ, δ	$\sqrt{B}$	$3B$	15.5.2, 15.5.3
$\cup$, $\cap$, $-$	$\sqrt{B(S)}$	$3\big(B(R) + B(S)\big)$	15.5.4
$\bowtie$	$\sqrt{B(S)}$	$3\big(B(R) + B(S)\big)$	15.5.5
$\bowtie$	$\sqrt{B(S)}$	$\big(3 - 2M/B(S)\big)\big(B(R) + B(S)\big)$	15.5.6

Figure 15.13: Main memory and disk I/O requirements for hash-based algorithms; for binary operations, assume $B(S) \leq B(R)$

Notice that the requirements for sort-based and the corresponding hash-based algorithms are almost the same. The significant differences between the two approaches are:

1. Hash-based algorithms for binary operations have a size requirement that depends only on the smaller of two arguments rather than on the sum of the argument sizes, that sort-based algorithms require.

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: 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) δ (b) γ (c) $\cap_B$ (d) $-_S$.

Exercise 15.5.2: If $B(S) = B(R) = 10{,}000$ and $M = 1000$, what is the number of disk I/O's required for a hybrid hash join?

Exercise 15.5.3: Write iterators that implement the two-pass, hash-based algorithms for (a) δ (b) γ (c) $\cap_B$ (d) $-_S$ (e) $\bowtie$.

! **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.

can have nonclustering indexes.

Example 15.10: A relation $R(a, b)$ that is sorted on attribute a and stored in that order, packed into blocks, is surely clustered. An index on a is a clustering index, since for a given a-value a_1, all the tuples with that value for a are consecutive. They thus appear packed into blocks, except possibly for the first and last blocks that contain a-value a_1, as suggested in Fig. 15.14. However, an index on b is unlikely to be clustering, since the tuples with a fixed b-value will be spread all over the file unless the values of a and b are very closely correlated. □

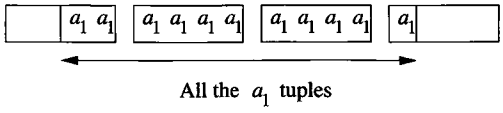

Figure 15.14: A clustering index has all tuples with a fixed value packed into (close to) the minimum possible number of blocks

15.6.2 Index-Based Selection

In Section 15.1.1 we discussed implementing a selection $\sigma_C(R)$ by reading all the tuples of relation R, seeing which meet the condition C, and outputting those that do. If there are no indexes on R, then that is the best we can do; the number of disk I/O's used by the operation is $B(R)$, or even $T(R)$, the number of tuples of R, should R not be a clustered relation.[6] However, suppose that the condition C is of the form $a = v$, where a is an attribute for which an index exists, and v is a value. Then one can search the index with value v and get pointers to exactly those tuples of R that have a-value v. These tuples constitute the result of $\sigma_{a=v}(R)$, so all we have to do is retrieve them.

If the index on $R.a$ is a clustering index, then the number of disk I/O's to retrieve the set $\sigma_{a=v}(R)$ will average $B(R)/V(R, a)$. The actual number may be somewhat higher for several reasons:

1. Often, the index is not kept entirely in main memory, and some disk I/O's are needed to support the index lookup.

2. Even though all the tuples with $a = v$ might fit in b blocks, they could be spread over $b + 1$ blocks because they don't start at the beginning of a block.

[6] Recall from Section 15.1.3 the notation we developed: $T(R)$ for the number of tuples in R, $B(R)$ for the number of blocks in which R fits, and $V(R, L)$ for the number of distinct tuples in $\pi_L(R)$.

3. Even though the tuples of R may be clustered, they may not be packed as tightly as possible into blocks. For example, there could be extra space for tuples to be inserted into R later, or R could be in a clustered file, as discussed in Section 14.1.6.

Moreover, we of course must round up if the ratio $B(R)/V(R,a)$ is not an integer. Most significant is that should a be a key for R, then $V(R,a) = T(R)$, which is presumably much bigger than $B(R)$, yet we surely require one disk I/O to retrieve the tuple with key value v, plus whatever disk I/O's are needed to access the index.

Now, let us consider what happens when the index on $R.a$ is nonclustering. To a first approximation, each tuple we retrieve will be on a different block, and we must access $T(R)/V(R,a)$ tuples. Thus, $T(R)/V(R,a)$ is an estimate of the number of disk I/O's we need. The number could be higher because we may also need to read some index blocks from disk; it could be lower because fortuitously some retrieved tuples appear on the same block, and that block remains buffered in memory.

Example 15.11: Suppose $B(R) = 1000$, and $T(R) = 20{,}000$. That is, R has 20,000 tuples, packed at most 20 to a block. Let a be one of the attributes of R, suppose there is an index on a, and consider the operation $\sigma_{a=0}(R)$. Here are some possible situations and the worst-case number of disk I/O's required. We shall ignore the cost of accessing the index blocks in all cases.

1. If R is clustered, but we do not use the index, then the cost is 1000 disk I/O's. That is, we must retrieve every block of R.

2. If R is not clustered and we do not use the index, then the cost is 20,000 disk I/O's.

3. If $V(R,a) = 100$ and the index is clustering, then the index-based algorithm uses $1000/100 = 10$ disk I/O's, plus whatever is needed to access the index.

4. If $V(R,a) = 10$ and the index is nonclustering, then the index-based algorithm uses $20{,}000/10 = 2000$ disk I/O's. Notice that this cost is higher than scanning the entire relation R, if R is clustered but the index is not.

5. If $V(R,a) = 20{,}000$, i.e., a is a key, then the index-based algorithm takes 1 disk I/O plus whatever is needed to access the index, regardless of whether the index is clustering or not.

□

Index-scan as an access method can help in several other kinds of selection operations.

a) An index such as a B-tree lets us access the search-key values in a given range efficiently. If such an index on attribute a of relation R exists, then we can use the index to retrieve just the tuples of R in the desired range for selections such as $\sigma_{a \geq 10}(R)$, or even $\sigma_{a \geq 10 \text{ AND } a \leq 20}(R)$.

b) A selection with a complex condition C can sometimes be implemented by an index-scan followed by another selection on only those tuples retrieved by the index-scan. If C is of the form $a = v$ AND C', where C' is any condition, then we can split the selection into a cascade of two selections, the first checking only for $a = v$, and the second checking condition C'. The first is a candidate for use of the index-scan operator. This splitting of a selection operation is one of many improvements that a query optimizer may make to a logical query plan; it is discussed particularly in Section 16.7.1.

15.6.3 Joining by Using an Index

All the binary operations we have considered, and the unary full-relation operations of γ and δ as well, can use certain indexes profitably. We shall leave most of these algorithms as exercises, while we focus on the matter of joins. In particular, let us examine the natural join $R(X,Y) \bowtie S(Y,Z)$; recall that X, Y, and Z can stand for sets of attributes, although it is sufficient to think of them as single attributes.

For our first index-based join algorithm, suppose that S has an index on the attribute(s) Y. Then one way to compute the join is to examine each block of R, and within each block consider each tuple t. Let t_Y be the component or components of t corresponding to the attribute(s) Y. Use the index to find all those tuples of S that have t_Y in their Y-component(s). These are exactly the tuples of S that join with tuple t of R, so we output the join of each of these tuples with t.

The number of disk I/O's depends on several factors. First, assuming R is clustered, we shall have to read $B(R)$ blocks to get all the tuples of R. If R is not clustered, then up to $T(R)$ disk I/O's may be required.

For each tuple t of R we must read an average of $T(S)/V(S,Y)$ tuples of S. If S has a nonclustered index on Y, then the number of disk I/O's required to read S is $T(R)T(S)/V(S,Y)$, but if the index is clustered, then only $T(R)B(S)/V(S,Y)$ disk I/O's suffice.[7] In either case, we may have to add a few disk I/O's per Y-value, to account for the reading of the index itself.

Regardless of whether or not R is clustered, the cost of accessing tuples of S dominates. Ignoring the cost of reading R, we shall take $T(R)T(S)/V(S,Y)$ or $T(R)\big(\max(1, B(S)/V(S,Y))\big)$ as the cost of this join method, for the cases of nonclustered and clustered indexes on S, respectively.

[7] But remember that $B(S)/V(S,Y)$ must be replaced by 1 if it is less, as discussed in Section 15.6.2.

Example 15.12: Let us consider our running example, relations $R(X,Y)$ and $S(Y,Z)$ covering 1000 and 500 blocks, respectively. Assume ten tuples of either relation fit on one block, so $T(R) = 10{,}000$ and $T(S) = 5000$. Also, assume $V(S,Y) = 100$; i.e., there are 100 different values of Y among the tuples of S.

Suppose that R is clustered, and there is a clustering index on Y for S. Then the approximate number of disk I/O's, excluding what is needed to access the index itself, is 1000 to read the blocks of R plus $10{,}000 \times 500 \ / \ 100 = 50{,}000$ disk I/O's. This number is considerably above the cost of other methods for the same data discussed previously. If either R or the index on S is not clustered, then the cost is even higher. □

While Example 15.12 makes it look as if an index-join is a very bad idea, there are other situations where the join $R \bowtie S$ by this method makes much more sense. Most common is the case where R is very small compared with S, and $V(S,Y)$ is large. We discuss in Exercise 15.6.5 a typical query in which selection before a join makes R tiny. In that case, most of S will never be examined by this algorithm, since most Y-values don't appear in R at all. However, both sort- and hash-based join methods will examine every tuple of S at least once.

15.6.4 Joins Using a Sorted Index

When the index is a B-tree, or any other structure from which we easily can extract the tuples of a relation in sorted order, we have a number of other opportunities to use the index. Perhaps the simplest is when we want to compute $R(X,Y) \bowtie S(Y,Z)$, and we have such an index on Y for either R or S. We can then perform an ordinary sort-join, but we do not have to perform the intermediate step of sorting one of the relations on Y.

As an extreme case, if we have sorting indexes on Y for both R and S, then we need to perform only the final step of the simple sort-based join of Section 15.4.6. This method is sometimes called *zig-zag join*, because we jump back and forth between the indexes finding Y-values that they share in common. Notice that tuples from R with a Y-value that does not appear in S need never be retrieved, and similarly, tuples of S whose Y-value does not appear in R need not be retrieved.

Example 15.13: Suppose that we have relations $R(X,Y)$ and $S(Y,Z)$ with indexes on Y for both relations. In a tiny example, let the search keys (Y-values) for the tuples of R be in order $1,3,4,4,4,5,6$, and let the search key values for S be $2,2,4,4,6,7$. We start with the first keys of R and S, which are 1 and 2, respectively. Since $1 < 2$, we skip the first key of R and look at the second key, 3. Now, the current key of S is less than the current key of R, so we skip the two 2's of S to reach 4.

At this point, the key 3 of R is less than the key of S, so we skip the key of R. Now, both current keys are 4. We follow the pointers associated with all the keys 4 from both relations, retrieve the corresponding tuples, and join

them. Notice that until we met the common key 4, no tuples of the relation were retrieved.

Having dispensed with the 4's, we go to key 5 of R and key 6 of S. Since $5 < 6$, we skip to the next key of R. Now the keys are both 6, so we retrieve the corresponding tuples and join them. Since R is now exhausted, we know there are no more pairs of tuples from the two relations that join. □

If the indexes are B-trees, then we can scan the leaves of the two B-trees in order from the left, using the pointers from leaf to leaf that are built into the structure, as suggested in Fig. 15.15. If R and S are clustered, then retrieval of all the tuples with a given key will result in a number of disk I/O's proportional to the fractions of these two relations read. Note that in extreme cases, where there are so many tuples from R and S that neither fits in the available main memory, we shall have to use a fixup like that discussed in Section 15.4.6. However, in typical cases, the step of joining all tuples with a common Y-value can be carried out with only as many disk I/O's as it takes to read them.

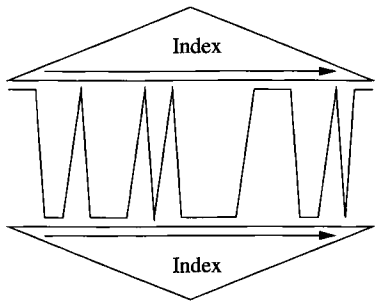

Figure 15.15: A zig-zag join using two indexes

Example 15.14: Let us continue with Example 15.12, to see how joins using a combination of sorting and indexing would typically perform on this data. First, assume that there is an index on Y for S that allows us to retrieve the tuples of S sorted by Y. We shall, in this example, also assume both relations and the index are clustered. For the moment, we assume there is no index on R.

Assuming 101 available blocks of main memory, we may use them to create 10 sorted sublists for the 1000-block relation R. The number of disk I/O's is 2000 to read and write all of R. We next use 11 blocks of memory — 10 for the sublists of R and one for a block of S's tuples, retrieved via the index. We neglect disk I/O's and memory buffers needed to manipulate the index, but if the index is a B-tree, these numbers will be small anyway. In this second pass, we read all the tuples of R and S, using a total of 1500 disk I/O's, plus the small amount needed for reading the index blocks once each. We thus estimate the

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

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

c) $\delta(R)$.

Exercise 15.6.2: 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 clustering.

b) The index is not clustering.

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

Exercise 15.6.3: Repeat Exercise 15.6.2 if the operation is the range query $\sigma_{C \leq a \text{ AND } a \leq D}(R)$. You may assume that C and D are constants such that $k/10$ of the values are in the range.

! **Exercise 15.6.4:** 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 15.6.2 (b) Exercise 15.6.3.

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.

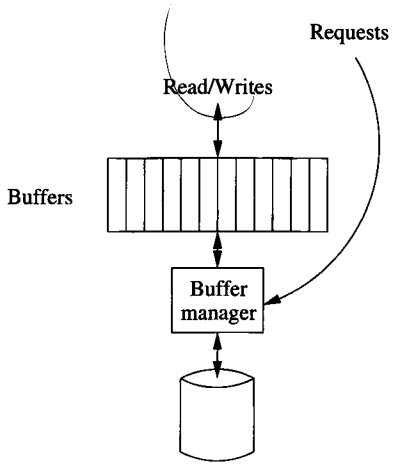

Figure 15.16: The buffer manager responds to requests for main-memory access to disk blocks

Whichever approach a DBMS uses, the same problem arises: the buffer manager should limit the number of buffers in use so they fit in the available main memory. When the buffer manager controls main memory directly, and requests exceed available space, it has to select a buffer to empty, by returning its contents to disk. If the buffered block has not been changed, then it may simply be erased from main memory, but if the block has changed it must be written back to its place on the disk. When the buffer manager allocates space in virtual memory, it has the option to allocate more buffers than can fit in main memory. However, if all these buffers are really in use, then there will be "thrashing," a common operating-system problem, where many blocks are moved in and out of the disk's swap space. In this situation, the system spends most of its time swapping blocks, while very little useful work gets done.

Normally, the number of buffers is a parameter set when the DBMS is initialized. We would expect that this number is set so that the buffers occupy the available main memory, regardless of whether the buffers are allocated in main or virtual memory. In what follows, we shall not concern ourselves with which mode of buffering is used, and simply assume that there is a fixed-size *buffer pool*, a set of buffers available to queries and other database actions.

15.7.2 Buffer Management Strategies

The critical choice that the buffer manager must make is what block to throw out of the buffer pool when a buffer is needed for a newly requested block. The *buffer-replacement strategies* in common use may be familiar to you from other applications of scheduling policies, such as in operating systems. These include:

Memory Management for Query Processing

We are assuming that the buffer manager allocates to an operator M main-memory buffers, where the value for M depends on system conditions (including other operators and queries underway), and may vary dynamically. Once an operator has M buffers, it may use some of them for bringing in disk pages, others for index pages, and still others for sort runs or hash tables. In some DBMS's, memory is not allocated from a single pool, but rather there are separate pools of memory — with separate buffer managers — for different purposes. For example, an operator might be allocated D buffers from a pool to hold pages brought in from disk and H buffers to build a hash table. This approach offers more opportunities for system configuration and "tuning," but may not make the best global use of memory.

Least-Recently Used (LRU)

The LRU rule is to throw out the block that has not been read or written for the longest time. This method requires that the buffer manager maintain a table indicating the last time the block in each buffer was accessed. It also requires that each database access make an entry in this table, so there is significant effort in maintaining this information. However, LRU is an effective strategy; intuitively, buffers that have not been used for a long time are less likely to be accessed sooner than those that have been accessed recently.

First-In-First-Out (FIFO)

When a buffer is needed, under the FIFO policy the buffer that has been occupied the longest by the same block is emptied and used for the new block. In this approach, the buffer manager needs to know only the time at which the block currently occupying a buffer was loaded into that buffer. An entry into a table can thus be made when the block is read from disk, and there is no need to modify the table when the block is accessed. FIFO requires less maintenance than LRU, but it can make more mistakes. A block that is used repeatedly, say the root block of a B-tree index, will eventually become the oldest block in a buffer. It will be written back to disk, only to be reread shortly thereafter into another buffer.

The "Clock" Algorithm ("Second Chance")

This algorithm is a commonly implemented, efficient approximation to LRU. Think of the buffers as arranged in a circle, as suggested by Fig. 15.17. A "hand" points to one of the buffers, and will rotate clockwise if it needs to find a buffer in which to place a disk block. Each buffer has an associated "flag,"

which is either 0 or 1. Buffers with a 0 flag are vulnerable to having their contents sent back to disk; buffers with a 1 are not. When a block is read into a buffer, its flag is set to 1. Likewise, when the contents of a buffer is accessed, its flag is set to 1.

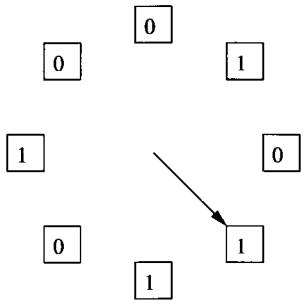

Figure 15.17: The clock algorithm visits buffers in a round-robin fashion and replaces $01\cdots1$ with $10\cdots0$

When the buffer manager needs a buffer for a new block, it looks for the first 0 it can find, rotating clockwise. If it passes 1's, it sets them to 0. Thus, a block is only thrown out of its buffer if it remains unaccessed for the time it takes the hand to make a complete rotation to set its flag to 0 and then make another complete rotation to find the buffer with its 0 unchanged. For instance, in Fig. 15.17, the hand will set to 0 the 1 in the buffer to its left, and then move clockwise to find the buffer with 0, whose block it will replace and whose flag it will set to 1.

System Control

The query processor or other components of a DBMS can give advice to the buffer manager in order to avoid some of the mistakes that would occur with a strict policy such as LRU, FIFO, or Clock. Recall from Section 13.6.5 that there are sometimes technical reasons why a block in main memory can *not* be moved to disk without first modifying certain other blocks that point to it. These blocks are called "pinned," and any buffer manager has to modify its buffer-replacement strategy to avoid expelling pinned blocks. This fact gives us the opportunity to force other blocks to remain in main memory by declaring them "pinned," even if there is no technical reason why they could not be written to disk. For example, a cure for the problem with FIFO mentioned above regarding the root of a B-tree is to "pin" the root, forcing it to remain in memory at all times. Similarly, for an algorithm like a one-pass hash-join, the query processor may "pin" the blocks of the smaller relation in order to assure that it will remain in main memory during the entire time.

More Tricks Using the Clock Algorithm

The "clock" algorithm for choosing buffers to free is not limited to the scheme described in Section 15.7.2, where flags had values 0 and 1. For instance, one can start an important page with a number higher than 1 as its flag, and decrement the flag by 1 each time the "hand" passes that page. In fact, one can incorporate the concept of pinning blocks by giving the pinned block an infinite value for its flag, and then having the system release the pin at the appropriate time by setting the flag to 0.

15.7.3 The Relationship Between Physical Operator Selection and Buffer Management

The query optimizer will eventually select a set of physical operators that will be used to execute a given query. This selection of operators may assume that a certain number of buffers M is available for execution of each of these operators. However, as we have seen, the buffer manager may not be willing or able to guarantee the availability of these M buffers when the query is executed. There are thus two related questions to ask about the physical operators:

1. Can the algorithm adapt to changes in the value of M, the number of main-memory buffers available?
2. When the expected M buffers are not available, and some blocks that are expected to be in memory have actually been moved to disk by the buffer manager, how does the buffer-replacement strategy used by the buffer manager impact the number of additional I/O's that must be performed?

Example 15.15: As an example of the issues, let us consider the block-based nested-loop join of Fig. 15.8. The basic algorithm does not really depend on the value of M, although its performance depends on M. Thus, it is sufficient to find out what M is just before execution begins.

It is even possible that M will change at different iterations of the outer loop. That is, each time we load main memory with a portion of the relation S (the relation of the outer loop), we can use all but one of the buffers available at that time; the remaining buffer is reserved for a block of R, the relation of the inner loop. Thus, the number of times we go around the outer loop depends on the average number of buffers available at each iteration. However, as long as M buffers are available *on average*, then the cost analysis of Section 15.3.4 will hold. In the extreme, we might have the good fortune to find that at the first iteration, enough buffers are available to hold all of S, in which case nested-loop join gracefully becomes the one-pass join of Section 15.2.3.

As another example of how nested-loop join interacts with buffering, suppose that we use an LRU buffer-replacement strategy, and there are k buffers

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 one-pass join.

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

c) A two-pass, sort-based 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) First-in-first-out.

b) The clock algorithm.

!! **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.

- For a γ, sort on the grouping attributes only, and combine the tuples with a given value of these grouping attributes in the appropriate manner, as discussed in Section 15.4.3.

When we want to perform a binary operation, such as intersection or join, we use essentially the same idea, except that the two relations are first divided into a total of M sublists. Then, each sublist is sorted by the recursive algorithm above. Finally, we read each of the M sublists, each into one buffer, and we perform the operation in the manner described by the appropriate subsection of Section 15.4.

We can divide the M buffers between relations R and S as we wish. However, to minimize the total number of passes, we would normally divide the buffers in proportion to the number of blocks taken by the relations. That is, R gets $M \times B(R)/\big(B(R) + B(S)\big)$ of the buffers, and S gets the rest.

15.8.2 Performance of Multipass, Sort-Based Algorithms

Now, let us explore the relationship between the number of disk I/O's required, the size of the relation(s) operated upon, and the size of main memory. Let $s(M,k)$ be the maximum size of a relation that we can sort using M buffers and k passes. Then we can compute $s(M,k)$ as follows:

BASIS: If $k = 1$, i.e., one pass is allowed, then we must have $B(R) \leq M$. Put another way, $s(M,1) = M$.

INDUCTION: Suppose $k > 1$. Then we partition R into M pieces, each of which must be sortable in $k-1$ passes. If $B(R) = s(M,k)$, then $s(M,k)/M$, which is the size of each of the M pieces of R, cannot exceed $s(M,k-1)$. That is: $s(M,k) = Ms(M,k-1)$.

If we expand the above recursion, we find

$$s(M,k) = Ms(M,k-1) = M^2 s(M,k-2) = \cdots = M^{k-1}s(M,1)$$

Since $s(M,1) = M$, we conclude that $s(M,k) = M^k$. That is, using k passes, we can sort a relation R if $B(R) \leq M^k$. Put another way, if we want to sort R in k passes, then the minimum number of buffers we can use is $M = \big(B(R)\big)^{1/k}$.

Each pass of a sorting algorithm reads all the data from disk and writes it out again. Thus, a k-pass sorting algorithm requires $2kB(R)$ disk I/O's.

Now, let us consider the cost of a multipass join $R(X,Y) \bowtie S(Y,Z)$, as representative of a binary operation on relations. Let $j(M,k)$ be the largest number of blocks such that in k passes, using M buffers, we can join relations of $j(M,k)$ or fewer total blocks. That is, the join can be accomplished provided $B(R) + B(S) \leq j(M,k)$.

On the final pass, we merge M sorted sublists from the two relations. Each of the sublists is sorted using $k-1$ passes, so they can be no longer than $s(M,k-1) = M^{k-1}$ each, or a total of $Ms(M,k-1) = M^k$. That is,

$B(R) + B(S) \leq M^k$. Reversing the role of the parameters, we can also state that to compute the join in k passes requires $\big(B(R) + B(S)\big)^{1/k}$ buffers.

To calculate the number of disk I/O's needed in the multipass algorithms, we should remember that, unlike for sorting, we do not count the cost of writing the final result to disk for joins or other relational operations. Thus, we use $2(k-1)\big(B(R)+B(S)\big)$ disk I/O's to sort the sublists, and another $B(R)+B(S)$ disk I/O's to read the sorted sublists in the final pass. The result is a total of $(2k-1)\big(B(R)+B(S)\big)$ disk I/O's.

15.8.3 Multipass Hash-Based Algorithms

There is a corresponding recursive approach to using hashing for operations on large relations. We hash the relation or relations into $M-1$ buckets, where M is the number of available memory buffers. We then apply the operation to each bucket individually, in the case of a unary operation. If the operation is binary, such as a join, we apply the operation to each pair of corresponding buckets, as if they were the entire relations. We can describe this approach recursively as:

BASIS: For a unary operation, if the relation fits in M buffers, read it into memory and perform the operation. For a binary operation, if either relation fits in $M-1$ buffers, perform the operation by reading this relation into main memory and then read the second relation, one block at a time, into the Mth buffer.

INDUCTION: If no relation fits in main memory, then hash each relation into $M-1$ buckets, as discussed in Section 15.5.1. Recursively perform the operation on each bucket or corresponding pair of buckets, and accumulate the output from each bucket or pair.

15.8.4 Performance of Multipass Hash-Based Algorithms

In what follows, we shall make the assumption that when we hash a relation, the tuples divide as evenly as possible among the buckets. In practice, this assumption will be met approximately if we choose a truly random hash function, but there will always be some unevenness in the distribution of tuples among buckets.

First, consider a unary operation, like γ or δ on a relation R using M buffers. Let $u(M,k)$ be the number of blocks in the largest relation that a k-pass hashing algorithm can handle. We can define u recursively by:

BASIS: $u(M,1) = M$, since the relation R must fit in M buffers; i.e., $B(R) \leq M$.

INDUCTION: We assume that the first step divides the relation R into $M-1$ buckets of equal size. Thus, we can compute $u(M,k)$ as follows. The buckets for the next pass must be sufficiently small that they can be handled in $k-1$

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 $B(R) = 20{,}000$, $B(S) = 50{,}000$, and $M = 101$. Describe the behavior of the following algorithms to compute $R \bowtie S$:

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

b) A three-pass, hash-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) The hybrid-hash-join trick of Section 15.5.6.

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

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

15.9 Summary of Chapter 15

- *Query Processing*: Queries are compiled, which involves extensive optimization, and then executed. The study of query execution involves knowing methods for executing operations of relational algebra with some extensions to match the capabilities of SQL.

- *Query Plans*: Queries are compiled first into logical query plans, which are often like expressions of relational algebra, and then converted to a physical query plan by selecting an implementation for each operator, ordering joins and making other decisions, as will be discussed in Chapter 16.

- *Table Scanning*: To access the tuples of a relation, there are several possible physical operators. The table-scan operator simply reads each block holding tuples of the relation. Index-scan uses an index to find tuples, and sort-scan produces the tuples in sorted order.

- *Cost Measures for Physical Operators*: Commonly, the number of disk I/O's taken to execute an operation is the dominant component of the time. In our model, we count only disk I/O time, and we charge for the time and space needed to read arguments, but not to write the result.

- *Iterators*: Several operations involved in the execution of a query can be meshed conveniently if we think of their execution as performed by an iterator. This mechanism consists of three methods, to open the construction of a relation, to produce the next tuple of the relation, and to close the construction.

- *One-Pass Algorithms*: As long as one of the arguments of a relational-algebra operator can fit in main memory, we can execute the operator by reading the smaller relation to memory, and reading the other argument one block at a time.

- *Nested-Loop Join*: This simple join algorithm works even when neither argument fits in main memory. It reads as much as it can of the smaller relation into memory, and compares that with the entire other argument; this process is repeated until all of the smaller relation has had its turn in memory.

- *Two-Pass Algorithms*: Except for nested-loop join, most algorithms for arguments that are too large to fit into memory are either sort-based, hash-based, or index-based.

- *Sort-Based Algorithms*: These partition their argument(s) into main-memory-sized, sorted sublists. The sorted sublists are then merged appropriately to produce the desired result. For instance, if we merge the tuples of all sublists in sorted order, then we have the important two-phase-multiway-merge sort.

- *Hash-Based Algorithms*: These use a hash function to partition the argument(s) into buckets. The operation is then applied to the buckets individually (for a unary operation) or in pairs (for a binary operation).
- *Hashing Versus Sorting*: Hash-based algorithms are often superior to sort-based algorithms, since they require only one of their arguments to be "small." Sort-based algorithms, on the other hand, work well when there is another reason to keep some of the data sorted.
- *Index-Based Algorithms*: The use of an index is an excellent way to speed up a selection whose condition equates the indexed attribute to a constant. Index-based joins are also excellent when one of the relations is small, and the other has an index on the join attribute(s).
- *The Buffer Manager*: The availability of blocks of memory is controlled by the buffer manager. When a new buffer is needed in memory, the buffer manager uses one of the familiar replacement policies, such as least-recently-used, to decide which buffer is returned to disk.
- *Coping With Variable Numbers of Buffers*: Often, the number of main-memory buffers available to an operation cannot be predicted in advance. If so, the algorithm used to implement an operation needs to degrade gracefully as the number of available buffers shrinks.
- *Multipass Algorithms*: The two-pass algorithms based on sorting or hashing have natural recursive analogs that take three or more passes and will work for larger amounts of data.

15.10 References for Chapter 15

Two surveys of query optimization are [6] and [2]. [8] is a survey of distributed query optimization.

An early study of join methods is in [5]. Buffer-pool management was analyzed, surveyed, and improved by [3].

The use of sort-based techniques was pioneered by [1]. The advantage of hash-based algorithms for join was expressed by [7] and [4]; the latter is the origin of the hybrid hash-join.

1. M. W. Blasgen and K. P. Eswaran, "Storage access in relational databases," *IBM Systems J.* **16**:4 (1977), pp. 363–378.
2. S. Chaudhuri, "An overview of query optimization in relational systems," *Proc. Seventeenth Annual ACM Symposium on Principles of Database Systems*, pp. 34–43, June, 1998.
3. H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," *Intl. Conf. on Very Large Databases*, pp. 127–141, 1985.

4. D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. Wood, "Implementation techniques for main-memory database systems," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1984), pp. 1–8.

5. L. R. Gotlieb, "Computing joins of relations," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1975), pp. 55–63.

6. G. Graefe, "Query evaluation techniques for large databases," *Computing Surveys* **25**:2 (June, 1993), pp. 73–170.

7. M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Application of hash to data base machine and its architecture," *New Generation Computing* **1**:1 (1983), pp. 66–74.

8. D. Kossman, "The state of the art in distributed query processing," *Computing Surveys* **32**:4 (Dec., 2000), pp. 422–469.