```
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.

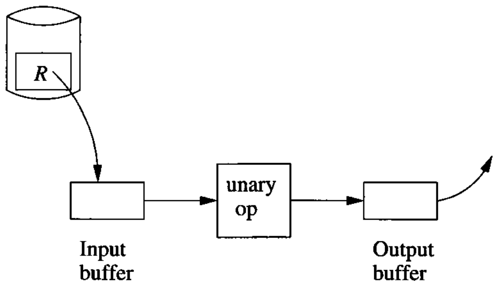

Figure 15.5: A selection or projection being performed on a relation R

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,

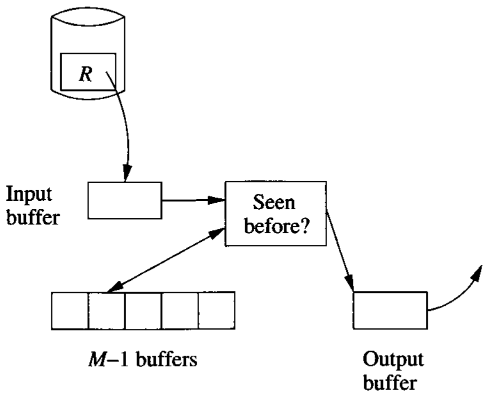

Figure 15.6: Managing memory for a one-pass duplicate-elimination

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
$\cup$, $\cap$, $-$, $\times$, $\bowtie$	$\min(B(R), B(S))$	$B(R) + B(S)$	15.2.3
$\bowtie$	any $M \geq 2$	$B(R)B(S)/M$	15.3.3

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

15.3.6 Exercises for Section 15.3

Exercise 15.3.1: 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

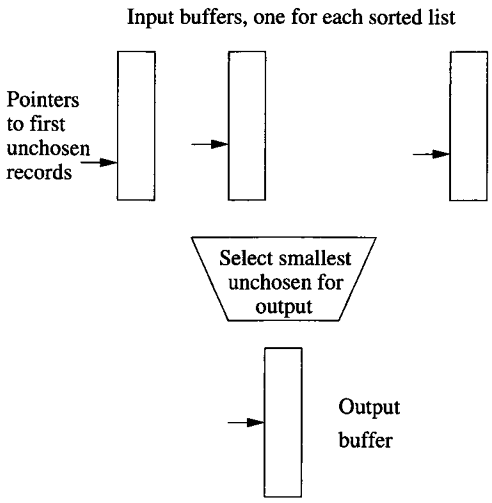
Input buffers, one for each sorted list
Pointers to first unchosen records
Select smallest unchosen for output
Output buffer

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 $4(B(R)+B(S))$ 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 $5(B(R) + B(S))$ 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 $3\big(B(R) + B(S)\big)$. We can perform the algorithm on data that is almost as large as that of the previous algorithm. The sizes of the sorted sublists are M blocks, and there can be at most M of them among the two lists. Thus, $B(R) + B(S) \leq M^2$ is sufficient.

15.4.9 Summary of Sort-Based Algorithms

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

15.4.10 Exercises for Section 15.4

Exercise 15.4.1: For each of the following operations, write an iterator that uses the algorithm described in this section: (a) 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
$\cup$, $\cap$, $-$	$\sqrt{B(R)+B(S)}$	$3(B(R)+B(S))$	15.4.4, 15.4.5
$\bowtie$	$\sqrt{\max(B(R),B(S))}$	$5(B(R)+B(S))$	15.4.6
$\bowtie$	$\sqrt{B(R)+B(S)}$	$3(B(R)+B(S))$	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 $3\big(B(R) + B(S)\big)$ disk I/O's to perform its task.
- The two-pass hash-join algorithm will work as long as approximately $\min\big(B(R), B(S)\big) \leq M^2$.

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
$\cup, \cap, -$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	15.5.4
$\bowtie$	$\sqrt{B(S)}$	$3(B(R) + B(S))$	15.5.5
$\bowtie$	$\sqrt{B(S)}$	$(3 - 2M/B(S))(B(R) + B(S))$	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. □

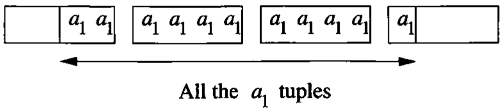

Figure 15.14: A clustering index has all tuples with a fixed value packed into (close to) the minimum possible number of blocks

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 $\sigma_{a \geq 10 \texttt{ AND } a \leq 20}(R)$.

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.

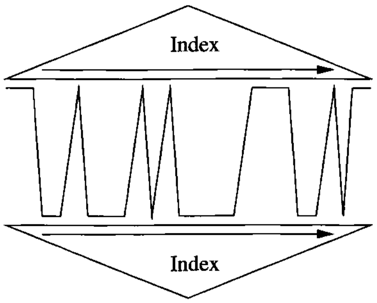

Figure 15.15: A zig-zag join using two indexes

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 $\sigma_{C \leq a \texttt{ AND } a \leq D}(R)$. You may assume that C and D are constants such that $k/10$ of the values are in the range.

! **Exercise 15.6.4:** 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.

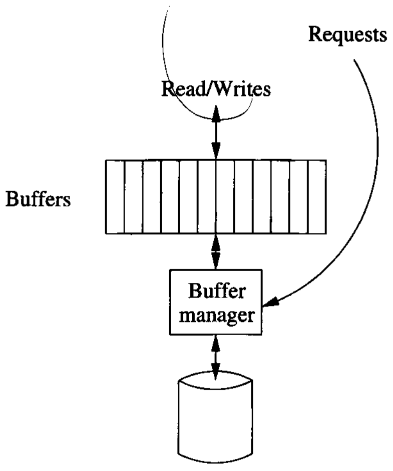

Figure 15.16: The buffer manager responds to requests for main-memory access to disk blocks

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.

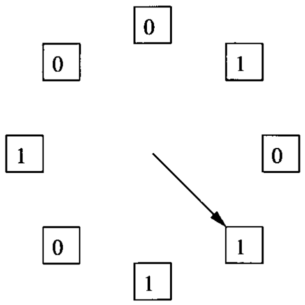

Figure 15.17: The clock algorithm visits buffers in a round-robin fashion and replaces $01 \cdots 1$ with $10 \cdots 0$

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 $\bigl(B(R) + B(S)\bigr)^{1/k}$ 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 $2(k-1)\bigl(B(R)+B(S)\bigr)$ 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 $(2k-1)\bigl(B(R)+B(S)\bigr)$ 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.

Chapter 16

The Query Compiler

We shall now take up the architecture of the query compiler and its optimizer. As we noted in Fig. 15.2, there are three broad steps that the query processor must take:

1. The query, written in a language like SQL, is *parsed*, that is, turned into a parse tree representing the structure of the query in a useful way.

2. The parse tree is transformed into an expression tree of relational algebra (or a similar notation), which we term a *logical query plan*.

3. The logical query plan must be turned into a *physical query plan*, which indicates not only the operations performed, but the order in which they are performed, the algorithm used to perform each step, and the ways in which stored data is obtained and data is passed from one operation to another.

The first step, parsing, is the subject of Section 16.1. The result of this step is a parse tree for the query. The other two steps involve a number of choices. In picking a logical query plan, we have opportunities to apply many different algebraic operations, with the goal of producing the best logical query plan. Section 16.2 discusses the algebraic laws for relational algebra in the abstract. Then, Section 16.3 discusses the conversion of parse trees to initial logical query plans and shows how the algebraic laws from Section 16.2 can be used in strategies to improve the initial logical plan.

When producing a physical query plan from a logical plan, we must evaluate the predicted cost of each possible option. Cost estimation is a science of its own, which we discuss in Section 16.4. We show how to use cost estimates to evaluate plans in Section 16.5, and the special problems that come up when we order the joins of several relations are the subject of Section 16.6. Finally, Section 16.7 covers additional issues and strategies for selecting the physical query plan: algorithm choice, and pipelining versus materialization.

16.1 Parsing and Preprocessing

The first stages of query compilation are illustrated in Fig. 16.1. The four boxes in that figure correspond to the first two stages of Fig. 15.2.

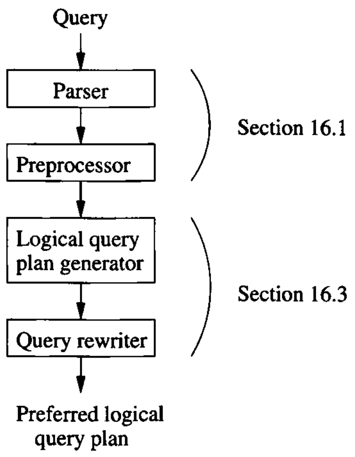

Figure 16.1: From a query to a logical query plan

In this section, we discuss parsing of SQL and give rudiments of a grammar that can be used for that language. We also discuss how to handle a query that involves a virtual view and other steps of preprocessing.

16.1.1 Syntax Analysis and Parse Trees

The job of the parser is to take text written in a language such as SQL and convert it to a *parse tree*, which is a tree whose nodes correspond to either:

1. *Atoms*, which are lexical elements such as keywords (e.g., SELECT), names of attributes or relations, constants, parentheses, operators such as + or <, and other schema elements, or

2. *Syntactic categories*, which are names for families of query subparts that all play a similar role in a query. We shall represent syntactic categories by triangular brackets around a descriptive name. For example, <Query> will be used to represent some queries in the common select-from-where form, and <Condition> will represent any expression that is a condition; i.e., it can follow WHERE in SQL.

If a node is an atom, then it has no children. However, if the node is a syntactic category, then its children are described by one of the *rules* of the grammar for the language. We shall present these ideas by example. The details of how one designs grammars for a language, and how one "parses," i.e., turns a program or query into the correct parse tree, is properly the subject of a course on compiling.[1]

16.1.2 A Grammar for a Simple Subset of SQL

We shall illustrate the parsing process by giving some rules that describe a small subset of SQL queries.

Queries

The syntactic category <Query> is intended to represent (some of the) queries of SQL. We give it only one rule:

```
<Query> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>
```

Symbol `::=` means "can be expressed as." The syntactic categories <SelList> and <FromList> represent lists that can follow `SELECT` and `FROM`, respectively. We shall describe limited forms of such lists shortly. The syntactic category <Condition> represents SQL conditions (expressions that are either true or false); we shall give some simplified rules for this category later.

Note this rule does not provide for the various optional clauses such as `GROUP BY`, `HAVING`, or `ORDER BY`, nor for options such as `DISTINCT` after `SELECT`, nor for query expressions using `UNION`, `JOIN`, or other binary operators.

Select-Lists

```
    <SelList> ::= <Attribute> , <SelList>
    <SelList> ::= <Attribute>
```

These two rules say that a select-list can be any comma-separated list of attributes: either a single attribute or an attribute, a comma, and any list of one or more attributes. Note that in a full SQL grammar we would also need provision for expressions and aggregation functions in the select-list and for aliasing of attributes and expressions.

From-Lists

```
    <FromList> ::= <Relation> , <FromList>
    <FromList> ::= <Relation>
```

Here, a from-list is defined to be any comma-separated list of relations. For simplification, we omit the possibility that elements of a from-list can be expressionsa, such as joins or subqueries. Likewise, a full SQL grammar would have to allow tuple variables for relations.

[1]Those unfamiliar with the subject may wish to examine A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 2007, although the examples of Section 16.1.2 should be sufficient to place parsing in the context of the query processor.

Conditions

The rules we shall use are:

```
<Condition> ::= <Condition> AND <Condition>
<Condition> ::= <Attribute> IN ( <Query> )
<Condition> ::= <Attribute> = <Attribute>
<Condition> ::= <Attribute> LIKE <Pattern>
```

Although we have listed more rules for conditions than for other categories, these rules only scratch the surface of the forms of conditions. We have omitted rules introducing operators OR, NOT, and EXISTS, comparisons other than equality and LIKE, constant operands, and a number of other structures that are needed in a full SQL grammar.

Base Syntactic Categories

Syntactic categories <Attribute>, <Relation>, and <Pattern> are special, in that they are not defined by grammatical rules, but by rules about the atoms for which they can stand. For example, in a parse tree, the one child of <Attribute> can be any string of characters that identifies an attribute of the current database schema. Similarly, <Relation> can be replaced by any string of characters that makes sense as a relation in the current schema, and <Pattern> can be replaced by any quoted string that is a legal SQL pattern.

Example 16.1: Recall two relations from the running movies example:

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

Our study of parsing and query rewriting will center around two versions of the query "find the titles of movies that have at least one star born in 1960." We identify stars born in 1960 by asking if their birthdate (a SQL string) ends in '1960', using the LIKE operator.

One way to ask this query is to construct the set of names of those stars born in 1960 as a subquery, and ask about each StarsIn tuple whether the starName in that tuple is a member of the set returned by this subquery. The SQL for this variation of the query is shown in Fig. 16.2.

The parse tree for the query of Fig. 16.2, according to the grammar we have sketched, is shown in Fig. 16.3. At the root is the syntactic category <Query>, as must be the case for any parse tree of a query. Working down the tree, we see that this query is a select-from-where form; the select-list consists of only the attribute movieTitle, and the from-list is only the one relation StarsIn.

The condition in the outer WHERE-clause is more complex. It has the form of attribute-IN-parenthesized-query. The subquery has its own singleton select- and from-lists and a simple condition involving a LIKE operator. □

```
SELECT movieTitle
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);
```

Figure 16.2: Find the movies with stars born in 1960

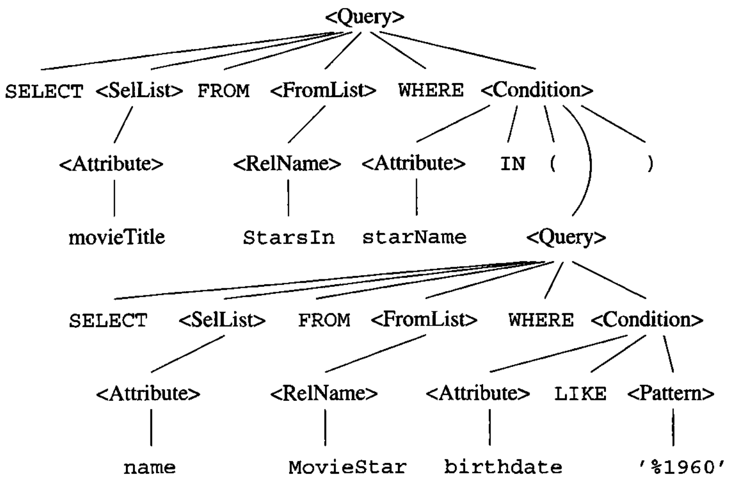

Figure 16.3: The parse tree for Fig. 16.2

Example 16.2: Now, let us consider another version of the query of Fig. 16.2, this time without using a subquery. We may instead equijoin the relations StarsIn and MovieStar, using the condition starName = name, to require that the star mentioned in both relations be the same. Note that starName is an attribute of relation StarsIn, while name is an attribute of MovieStar. This form of the query of Fig. 16.2 is shown in Fig. 16.4.[2]

The parse tree for Fig. 16.4 is seen in Fig. 16.5. Many of the rules used in this parse tree are the same as in Fig. 16.3. However, notice a from-list with more than one relation and two conditions connected by AND. □

[2]There is a small difference between the two queries in that Fig. 16.4 can produce duplicates if a movie has more than one star born in 1960. Strictly speaking, we should add DISTINCT to Fig. 16.4, but our example grammar was simplified to the extent of omitting that option.

```
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name AND
    birthdate LIKE '%1960';
```

Figure 16.4: Another way to ask for the movies with stars born in 1960

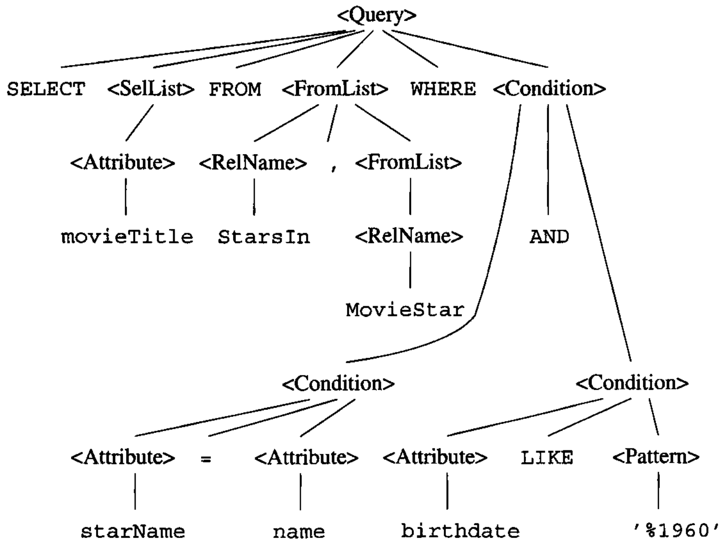

Figure 16.5: The parse tree for Fig. 16.4

16.1.3 The Preprocessor

The *preprocessor* has several important functions. If a relation used in the query is actually a virtual view, then each use of this relation in the from-list must be replaced by a parse tree that describes the view. This parse tree is obtained from the definition of the view, which is essentially a query. We discuss the preprocessing of view references in Section 16.1.4.

The preprocessor is also responsible for *semantic checking.* Even if the query is valid syntactically, it actually may violate one or more semantic rules on the use of names. For instance, the preprocessor must:

1. *Check relation uses.* Every relation mentioned in a FROM-clause must be a relation or view in the current schema.

2. *Check and resolve attribute uses.* Every attribute that is mentioned in the SELECT- or WHERE-clause must be an attribute of some relation in the current scope. For instance, attribute movieTitle in the first select-list of Fig. 16.3 is in the scope of only relation StarsIn. Fortunately,

movieTitle is an attribute of StarsIn, so the preprocessor validates this use of movieTitle. The typical query processor would at this point *resolve* each attribute by attaching to it the relation to which it refers, if that relation was not attached explicitly in the query (e.g., StarsIn.movieTitle). It would also check ambiguity, signaling an error if the attribute is in the scope of two or more relations with that attribute.

3. *Check types.* All attributes must be of a type appropriate to their uses. For instance, birthdate in Fig. 16.3 is used in a LIKE comparison, which requires that birthdate be a string or a type that can be coerced to a string. Since birthdate is a date, and dates in SQL normally can be treated as strings, this use of an attribute is validated. Likewise, operators are checked to see that they apply to values of appropriate and compatible types.

16.1.4 Preprocessing Queries Involving Views

When an operand in a query is a virtual view, the preprocessor needs to replace the operand by a piece of parse tree that represents how the view is constructed from base tables. The idea is illustrated in Fig. 16.6. A query Q is represented by its expression tree in relational algebra, and that tree may have some leaves that are views. We have suggested two such leaves, the views V and W. To interpret Q in terms of base tables, we find the definition of the views V and W. These definitions are also queries, so they can be expressed in relational algebra or as parse trees.

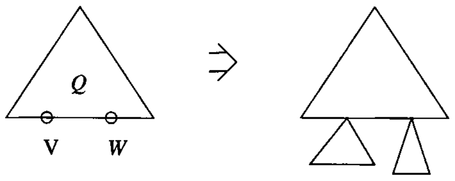

Figure 16.6: Substituting view definitions for view references

To form the query over base tables, we substitute, for each leaf in the tree for Q that is a view, the root of a copy of the tree that defines that view. Thus, in Fig. 16.6 we have shown the leaves labeled V and W replaced by the definitions of these views. The resulting tree is a query over base tables that is equivalent to the original query about views.

Example 16.3: Let us consider the view definition and query of Example 8.3. Recall the definition of view ParamountMovies is:

```
CREATE VIEW ParamountMovies AS
   SELECT title, year
```

```
    FROM Movies
    WHERE studioName = 'Paramount';
```

The tree in Fig. 16.7 is a relational-algebra expression for the query; we use relational algebra here because it is more succinct than the parse trees we have been using.

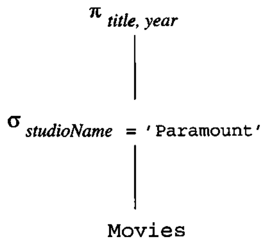

Figure 16.7: Expression tree for view `ParamountMovies`

The query of Example 8.3 is

```
SELECT title
FROM ParamountMovies
WHERE year = 1979;
```

asking for the Paramount movies made in 1979. This query has the expression tree shown in Fig. 16.8. Note that the one leaf of this tree represents the view `ParamountMovies`.

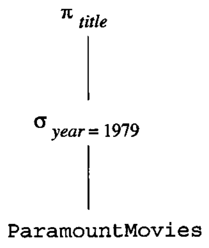

Figure 16.8: Expression tree for the query

We substitute the tree of Fig. 16.7 for the leaf `ParamountMovies` in Fig. 16.8. The resulting tree is shown in Fig. 16.9.

This tree, while the formal result of the view preprocessing, is not a very good way to express the query. In Section 16.2 we shall discuss ways to improve expression trees such as Fig. 16.9. In particular, we can push selections and projections down the tree, and combine them in many cases. Figure 16.10 is an improved representation that we can obtain by standard query-processing techniques. □

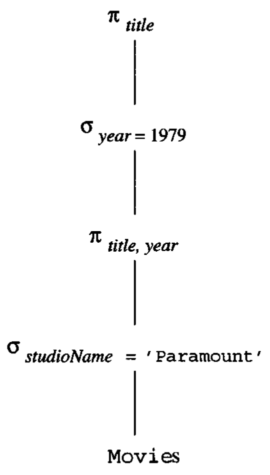

Figure 16.9: Expressing the query in terms of base tables

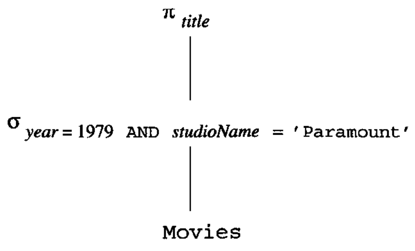

Figure 16.10: Simplifying the query over base tables

16.1.5 Exercises for Section 16.1

Exercise 16.1.1: Add to or modify the rules for <Query> to include simple versions of the following features of SQL select-from-where expressions:

a) The ability to produce a set with the `DISTINCT` keyword.

b) A `GROUP BY` clause and a `HAVING` clause.

c) Sorted output with the `ORDER BY` clause.

d) A query with no where-clause.

Exercise 16.1.2: Add to the rules for <Condition> to allow the following features of SQL conditionals:

a) Logical operators OR and NOT.

b) Comparisons other than =.

c) Parenthesized conditions.

d) EXISTS expressions.

Exercise 16.1.3: Using the simple SQL grammar exhibited in this section, give parse trees for the following queries about relations $R(a,b)$ and $S(b,c)$:

a)
```
SELECT a, c FROM R, S WHERE R.b = S.b;
```

b)
```
SELECT a FROM R WHERE b IN
    (SELECT a FROM R, S WHERE R.b = S.b);
```

16.2 Algebraic Laws for Improving Query Plans

We resume our discussion of the query compiler in Section 16.3, where we shall transform the parse tree into an expression of the extended relational algebra. Also in Section 16.3, we shall see how to apply heuristics that we hope will improve the algebraic expression of the query, using some of the many algebraic laws that hold for relational algebra. As a preliminary, this section catalogs algebraic laws that turn one expression tree into an equivalent expression tree that may have a more efficient physical query plan. The result of applying these algebraic transformations is the logical query plan that is the output of the query-rewrite phase.

16.2.1 Commutative and Associative Laws

A *commutative law* about an operator says that it does not matter in which order you present the arguments of the operator; the result will be the same. For instance, $+$ and $\times$ are commutative operators of arithmetic. More precisely, $x+y=y+x$ and $x\times y=y\times x$ for any numbers x and y. On the other hand, $-$ is not a commutative arithmetic operator: $x-y\neq y-x$.

An *associative law* about an operator says that we may group two uses of the operator either from the left or the right. For instance, $+$ and $\times$ are associative arithmetic operators, meaning that $(x+y)+z=x+(y+z)$ and $(x\times y)\times z = x\times(y\times z)$. On the other hand, $-$ is not associative: $(x-y)-z\neq x-(y-z)$. When an operator is both associative and commutative, then any number of operands connected by this operator can be grouped and ordered as we wish without changing the result. For example, $\big((w+x)+y\big)+z=(y+x)+(z+w)$.

Several of the operators of relational algebra are both associative and commutative. Particularly:

- $R\times S=S\times R$; $(R\times S)\times T=R\times(S\times T)$.
- $R\bowtie S=S\bowtie R$; $(R\bowtie S)\bowtie T=R\bowtie(S\bowtie T)$.
- $R\cup S=S\cup R$; $(R\cup S)\cup T=R\cup(S\cup T)$.

- $R \cap S = S \cap R$; $(R \cap S) \cap T = R \cap (S \cap T)$.

Note that these laws hold for both sets and bags. We shall not prove each of these laws, although we give one example of a proof, below.

Example 16.4: Let us verify the commutative law for $\bowtie$: $R \bowtie S = S \bowtie R$. First, suppose a tuple t is in the result of $R \bowtie S$, the expression on the left. Then there must be a tuple r in R and a tuple s in S that agree with t on every attribute that each shares with t. Thus, when we evaluate the expression on the right, $S \bowtie R$, the tuples s and r will again combine to form t.

We might imagine that the order of components of t will be different on the left and right, but formally, tuples in relational algebra have no fixed order of attributes. Rather, we are free to reorder components, as long as we carry the proper attributes along in the column headers, as was discussed in Section 2.2.5.

We are not done yet with the proof. Since our relational algebra is an algebra of bags, not sets, we must also verify that if t appears n times on the left, then it appears n times on the right, and vice-versa. Suppose t appears n times on the left. Then it must be that the tuple r from R that agrees with t appears some number of times n_R, and the tuple s from S that agrees with t appears some n_S times, where $n_R n_S = n$. Then when we evaluate the expression $S \bowtie R$ on the right, we find that s appears n_S times, and r appears n_R times, so we get $n_S n_R$ copies of t, or n copies.

We are still not done. We have finished the half of the proof that says everything on the left appears on the right, but we must show that everything on the right appears on the left. Because of the obvious symmetry, the argument is essentially the same, and we shall not go through the details here. □

We did not include the theta-join among the associative-commutative operators. True, this operator is commutative:

- $R \bowtie_C S = S \bowtie_C R$.

Moreover, if the conditions involved make sense where they are positioned, then the theta-join is associative. However, there are examples, such as the following, where we cannot apply the associative law because the conditions do not apply to attributes of the relations being joined.

Example 16.5: Suppose we have three relations $R(a,b)$, $S(b,c)$, and $T(c,d)$. The expression

$$(R \bowtie_{R.b>S.b} S) \bowtie_{a<d} T$$

is transformed by a hypothetical associative law into:

$$R \bowtie_{R.b>S.b} (S \bowtie_{a<d} T)$$

However, we cannot join S and T using the condition $a < d$, because a is an attribute of neither S nor T. Thus, the associative law for theta-join cannot be applied arbitrarily. □

Laws for Bags and Sets Can Differ

Be careful about applying familiar laws about sets to relations that are bags. For instance, you may have learned set-theoretic laws such as $A \cap_S (B \cup_S C) = (A \cap_S B) \cup_S (A \cap_S C)$, which is formally the "distributive law of intersection over union." This law holds for sets, but not for bags.

As an example, suppose bags A, B, and C were each $\{x\}$. Then $A \cap_B (B \cup_B C) = \{x\} \cap_B \{x, x\} = \{x\}$. But $(A \cap_B B) \cup_B (A \cap_B C) = \{x\} \cup_B \{x\} = \{x, x\}$, which differs from the left-hand-side, $\{x\}$.

16.2.2 Laws Involving Selection

Since selections tend to reduce the size of relations markedly, one of the most important rules of efficient query processing is to move the selections down the tree as far as they will go without changing what the expression does. Indeed early query optimizers used variants of this transformation as their primary strategy for selecting good logical query plans. As we shall see shortly, the transformation of "push selections down the tree" is not quite general enough, but the idea of "pushing selections" is still a major tool for the query optimizer.

To start, when the condition of a selection is complex (i.e., it involves conditions connected by AND or OR), it helps to break the condition into its constituent parts. The motivation is that one part, involving fewer attributes than the whole condition, may be moved to a convenient place where the entire condition cannot be evaluated. Thus, our first two laws for σ are the *splitting laws*:

- $\sigma_{C_1 \text{ AND } C_2}(R) = \sigma_{C_1}\big(\sigma_{C_2}(R)\big)$.
- $\sigma_{C_1 \text{ OR } C_2}(R) = \big(\sigma_{C_1}(R)\big) \cup_S \big(\sigma_{C_2}(R)\big)$.

However, the second law, for OR, works only if the relation R is a set. Notice that if R were a bag, the set-union would have the effect of eliminating duplicates incorrectly.

Notice that the order of C_1 and C_2 is flexible. For example, we could just as well have written the first law above with C_2 applied after C_1, as $\sigma_{C_2}\big(\sigma_{C_1}(R)\big)$. In fact, more generally, we can swap the order of any sequence of σ operators:

- $\sigma_{C_1}\big(\sigma_{C_2}(R)\big) = \sigma_{C_2}\big(\sigma_{C_1}(R)\big)$.

Example 16.6: Let $R(a, b, c)$ be a relation. Then $\sigma_{(a=1 \text{ OR } a=3) \text{ AND } b<c}(R)$ can be split as $\sigma_{a=1 \text{ OR } a=3}\big(\sigma_{b<c}(R)\big)$. We can then split this expression at the OR into $\sigma_{a=1}\big(\sigma_{b<c}(R)\big) \cup \sigma_{a=3}\big(\sigma_{b<c}(R)\big)$. In this case, because it is impossible for a tuple to satisfy both $a = 1$ and $a = 3$, this transformation holds regardless

of whether or not R is a set, as long as $\cup_B$ is used for the union. However, in general the splitting of an OR requires that the argument be a set and that $\cup_S$ be used.

Alternatively, we could have started to split by making $\sigma_{b<c}$ the outer operation, as $\sigma_{b<c}\big(\sigma_{a=1 \text{ OR } a=3}(R)\big)$. When we then split the OR, we would get $\sigma_{b<c}\big(\sigma_{a=1}(R) \cup \sigma_{a=3}(R)\big)$, an expression that is equivalent to, but somewhat different from the first expression we derived. □

The next family of laws involving σ allow us to push selections through the binary operators: product, union, intersection, difference, and join. There are three types of laws, depending on whether it is optional or required to push the selection to each of the arguments:

1. For a union, the selection *must* be pushed to both arguments.
2. For a difference, the selection must be pushed to the first argument and optionally may be pushed to the second.
3. For the other operators it is only required that the selection be pushed to one argument. For joins and products, it may not make sense to push the selection to both arguments, since an argument may or may not have the attributes that the selection requires. When it is possible to push to both, it may or may not improve the plan to do so; see Exercise 16.2.1.

Thus, the law for union is:

- $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$.

Here, it is mandatory to move the selection down both branches of the tree.

For difference, one version of the law is:

- $\sigma_C(R - S) = \sigma_C(R) - S$.

However, it is also permissible to push the selection to both arguments, as:

- $\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S)$.

The next laws allow the selection to be pushed to one or both arguments. If the selection is σ_C, then we can only push this selection to a relation that has all the attributes mentioned in C, if there is one. We shall show the laws below assuming that the relation R has all the attributes mentioned in C.

- $\sigma_C(R \times S) = \sigma_C(R) \times S$.
- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$.
- $\sigma_C(R \bowtie_D S) = \sigma_C(R) \bowtie_D S$.
- $\sigma_C(R \cap S) = \sigma_C(R) \cap S$.

If C has only attributes of S, then we can instead write:

- $\sigma_C(R \times S) = R \times \sigma_C(S)$.

and similarly for the other three operators $\bowtie$, $\bowtie_D$, and $\cap$. Should relations R and S both happen to have all attributes of C, then we can use laws such as:

- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie \sigma_C(S)$.

Note that it is impossible for this variant to apply if the operator is $\times$ or $\bowtie_D$, since in those cases R and S have no shared attributes. On the other hand, for $\cap$ this form of law always applies, since the schemas of R and S must then be the same.

Example 16.7: Consider relations $R(a, b)$ and $S(b, c)$ and the expression

$$\sigma_{(a=1 \texttt{ OR } a=3) \texttt{ AND } b<c}(R \bowtie S)$$

The condition $b < c$ applies only to to S, and the condition $a = 1$ OR $a = 3$ applies only to R. We thus begin by splitting the AND of the two conditions as we did in the first alternative of Example 16.6:

$$\sigma_{a=1 \texttt{ OR } a=3}\big(\sigma_{b<c}(R \bowtie S)\big)$$

Next, we can push the selection $\sigma_{b<c}$ to S, giving us the expression:

$$\sigma_{a=1 \texttt{ OR } a=3}\big(R \bowtie \sigma_{b<c}(S)\big)$$

Finally, push the first condition to R, yielding: $\sigma_{a=1 \texttt{ OR } a=3}(R) \bowtie \sigma_{b<c}(S)$. □

16.2.3 Pushing Selections

As was illustrated in Example 16.3, pushing a selection down an expression tree — that is, replacing the left side of one of the rules in Section 16.2.2 by its right side — is one of the most powerful tools of the query optimizer. However, when queries involve virtual views, it is sometimes necessary first to move a selection as far *up* the tree as it can go, and then push the selections down all possible branches. An example will illustrate the proper selection-pushing approach.

Example 16.8: Suppose we have the relations

```
StarsIn(title, year, starName)
Movies(title, year, length, genre, studioName, producerC#)
```

Note that we have altered the first two attributes of `StarsIn` from the usual `movieTitle` and `movieYear` to make this example simpler to follow. Define view `MoviesOf1996` by:

Some Trivial Laws

We are not going to state every true law for the relational algebra. The reader should be alert, in particular, for laws about extreme cases: a relation that is empty, a selection or theta-join whose condition is always true or always false, or a projection onto the list of all attributes, for example. A few of the many possible special-case laws:

- Any selection on an empty relation is empty.
- If C is an always-true condition (e.g., $x > 10$ OR $x \leq 10$ on a relation that forbids $x =$ NULL), then $\sigma_C(R) = R$.
- If R is empty, then $R \cup S = S$.

```
CREATE VIEW MoviesOf1996 AS
    SELECT *
    FROM Movies
    WHERE year = 1996;
```

We can ask the query "which stars worked for which studios in 1996?" by the SQL query:

```
SELECT starName, studioName
FROM MoviesOf1996 NATURAL JOIN StarsIn;
```

The view `MoviesOf1996` is defined by the relational-algebra expression

$$\sigma_{year=1996}(\texttt{Movies})$$

Thus, the query, which is the natural join of this expression with `StarsIn`, followed by a projection onto attributes `starName` and `studioName`, has the expression shown in Fig. 16.11.

Here, the selection is already as far down the tree as it will go, so there is no way to "push selections down the tree." However, the rule $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$ can be applied "backwards," to bring the selection $\sigma_{year=1996}$ above the join in Fig. 16.11. Then, since *year* is an attribute of both `Movies` and `StarsIn`, we may push the selection down to *both* children of the join node. The resulting logical query plan is shown in Fig. 16.12. It is likely to be an improvement, since we reduce the size of the relation `StarsIn` before we join it with the movies of 1996. □

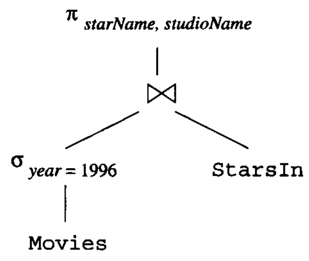

Figure 16.11: Logical query plan constructed from definition of a query and view

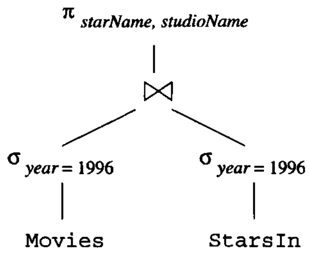

Figure 16.12: Improving the query plan by moving selections up and down the tree

16.2.4 Laws Involving Projection

Projections, like selections, can be "pushed down" through many other operators. Pushing projections differs from pushing selections in that when we push projections, it is quite usual for the projection also to remain where it is. Put another way, "pushing" projections really involves introducing a new projection somewhere below an existing projection.

Pushing projections is useful, but generally less so than pushing selections. The reason is that while selections often reduce the size of a relation by a large factor, projection keeps the number of tuples the same and only reduces the length of tuples. In fact, the extended projection operator of Section 5.2.5 can actually increase the length of tuples.

To describe the transformations of extended projection, we need to introduce some terminology. Consider a term $E \to x$ on the list for a projection, where E is an attribute or an expression involving attributes and constants. We say all attributes mentioned in E are *input* attributes of the projection, and x is an *output* attribute. If a term is a single attribute, then it is both an input and output attribute. If a projection list consists only of attributes, with no renaming or expressions other than a single attribute, then we say the projection is *simple.*

Example 16.9: Projection $\pi_{a,b,c}(R)$ is simple; a, b, and c are both its input attributes and its output attributes. On the other hand, $\pi_{a+b\rightarrow x,\ c}(R)$ is not simple. It has input attributes a, b, and c, and its output attributes are x and c. □

The principle behind laws for projection is that:

- We may introduce a projection anywhere in an expression tree, as long as it eliminates only attributes that are neither used by an operator above nor are in the result of the entire expression.

In the most basic form of these laws, the introduced projections are always simple, although the pre-existing projections, such as L below, need not be.

- $\pi_L(R \bowtie S) = \pi_L\big(\pi_M(R) \bowtie \pi_N(S)\big)$, where M and N are the join attributes and the input attributes if L that are found among the attributes of R and S, respectively.

- $\pi_L(R \bowtie_C S) = \pi_L\big(\pi_M(R) \bowtie_C \pi_N(S)\big)$, where M and N are the join attributes (i.e., those mentioned in condition C) and the input attributes of L that are found among the attributes of R and S respectively.

- $\pi_L(R \times S) = \pi_L\big(\pi_M(R) \times \pi_N(S)\big)$, where M and N are the lists of all attributes of R and S, respectively, that are input attributes of L.

Example 16.10: Let $R(a,b,c)$ and $S(c,d,e)$ be two relations. Consider the expression $\pi_{a+e\rightarrow x,\ b\rightarrow y}(R \bowtie S)$. The input attributes of the projection are a, b, and e, and c is the only join attribute. We may apply the law for pushing projections below joins to get the equivalent expression:

$$\pi_{a+e\rightarrow x,\ b\rightarrow y}\big(\pi_{a,b,c}(R) \bowtie \pi_{c,e}(S)\big)$$

Notice that the projection $\pi_{a,b,c}(R)$ is trivial; it projects onto all the attributes of R. We may thus eliminate this projection and get a third equivalent expression: $\pi_{a+e\rightarrow x,\ b\rightarrow y}\big(R \bowtie \pi_{c,e}(S)\big)$. That is, the only change from the original is that we remove the attribute d from S before the join. □

We can perform a projection entirely before a bag union. That is:

- $\pi_L(R \cup_B S) = \pi_L(R) \cup_B \pi_L(S)$.

On the other hand, projections cannot be pushed below set unions or either the set or bag versions of intersection or difference at all.

Example 16.11: Let $R(a,b)$ consist of the one tuple $\{(1,2)\}$ and $S(a,b)$ consist of the one tuple $\{(1,3)\}$. Then $\pi_a(R \cap S) = \pi_a(\emptyset) = \emptyset$. However, $\pi_a(R) \cap \pi_a(S) = \{(1)\} \cap \{(1)\} = \{(1)\}$. □

If the projection involves some computations, and the input attributes of a term on the projection list belong entirely to one of the arguments of a join or product below the projection, then we have the option, although not the obligation, to perform the computation directly on that argument. An example should help illustrate the point.

Example 16.12: Again let $R(a,b,c)$ and $S(c,d,e)$ be relations, and consider the join and projection $\pi_{a+b\rightarrow x,\ d+e\rightarrow y}(R \bowtie S)$. We can move the sum $a+b$ and its renaming to x directly onto the relation R, and move the sum $d+e$ to S similarly. The resulting equivalent expression is

$$\pi_{x,y}\big(\pi_{a+b\rightarrow x,\ c}(R) \bowtie \pi_{d+e\rightarrow y,\ c}(S)\big)$$

One special case to handle is if x or y were c. Then, we could not rename a sum to c, because a relation cannot have two attributes named c. Thus, we would have to invent a temporary name and do another renaming in the projection above the join. For example, $\pi_{a+b\rightarrow c,\ d+e\rightarrow y}(R \bowtie S)$ could become $\pi_{z\rightarrow c,\ y}\big(\pi_{a+b\rightarrow z,\ c}(R) \bowtie \pi_{d+e\rightarrow y,\ c}(S)\big)$. □

It is also possible to push a projection below a selection.

- $\pi_L\big(\sigma_C(R)\big) = \pi_L\Big(\sigma_C\big(\pi_M(R)\big)\Big)$, where M is the list of all attributes that are either input attributes of L or mentioned in condition C.

As in Example 16.12, we have the option of performing computations on the list L in the list M instead, provided the condition C does not need the input attributes of L that are involved in a computation.

16.2.5 Laws About Joins and Products

We saw in Section 16.2.1 many of the important laws involving joins and products: their commutative and associative laws. However, there are a few additional laws that follow directly from the definition of the join, as was mentioned in Section 2.4.12.

- $R \bowtie_C S = \sigma_C(R \times S)$.
- $R \bowtie S = \pi_L\big(\sigma_C(R \times S)\big)$, where C is the condition that equates each pair of attributes from R and S with the same name, and L is a list that includes one attribute from each equated pair and all the other attributes of R and S.

In practice, we usually want to apply these rules from right to left. That is, we identify a product followed by a selection as a join of some kind. The reason for doing so is that the algorithms for computing joins are generally much faster than algorithms that compute a product followed by a selection on the (very large) result of the product.

16.2.6 Laws Involving Duplicate Elimination

The operator δ, which eliminates duplicates from a bag, can be pushed through many, but not all operators. In general, moving a δ down the tree reduces the size of intermediate relations and may therefore be beneficial. Moreover, we can sometimes move the δ to a position where it can be eliminated altogether, because it is applied to a relation that is known not to possess duplicates:

- $\delta(R) = R$ if R has no duplicates. Important cases of such a relation R include

  a) A stored relation with a declared primary key, and

  b) The result of a γ operation, since grouping creates a relation with no duplicates.

  c) The result of a set union, intersection, or difference.

Several laws that "push" δ through other operators are:

- $\delta(R \times S) = \delta(R) \times \delta(S)$.
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$.
- $\delta(R \bowtie_C S) = \delta(R) \bowtie_C \delta(S)$.
- $\delta\big(\sigma_C(R)\big) = \sigma_C\big(\delta(R)\big)$.

We can also move the δ to either or both of the arguments of an intersection:

- $\delta(R \cap_B S) = \delta(R) \cap_B S = R \cap_B \delta(S) = \delta(R) \cap_B \delta(S)$.

On the other hand, δ generally cannot be pushed through the operators $\cup_B$, $-_B$, or π.

Example 16.13: Let R have two copies of the tuple t and S have one copy of t. Then $\delta(R \cup_B S)$ has one copy of t, while $\delta(R) \cup_B \delta(S)$ has two copies of t. Also, $\delta(R -_B S)$ has one copy of t, while $\delta(R) -_B \delta(S)$ has no copy of t.

Now, consider relation $T(a, b)$ with one copy each of the tuples $(1, 2)$ and $(1, 3)$, and no other tuples. Then $\delta\big(\pi_a(T)\big)$ has one copy of the tuple (1), while $\pi_a\big(\delta(T)\big)$ has two copies of (1). □

16.2.7 Laws Involving Grouping and Aggregation

When we consider the operator γ, we find that the applicability of many transformations depends on the details of the aggregate operators used. Thus, we cannot state laws in the generality that we used for the other operators. One exception is the law, mentioned in Section 16.2.6, that a γ absorbs a δ. Precisely:

- $\delta(\gamma_L(R)) = \gamma_L(R)$.

Another general rule is that we may project useless attributes from the argument should we wish, prior to applying the γ operation. This law can be written:

- $\gamma_L(R) = \gamma_L(\pi_M(R))$ if M is a list containing at least all those attributes of R that are mentioned in L.

The reason that other transformations depend on the aggregation(s) involved in a γ is that some aggregations — `MIN` and `MAX` in particular — are not affected by the presence or absence of duplicates. The other aggregations — `SUM`, `COUNT`, and `AVG` — generally produce different values if duplicates are eliminated prior to application of the aggregation.

Thus, let us call an operator γ_L *duplicate-impervious* if the only aggregations in L are `MIN` and/or `MAX`. Then:

- $\gamma_L(R) = \gamma_L(\delta(R))$ provided γ_L is duplicate-impervious.

Example 16.14: Suppose we have the relations

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

and we want to know for each year the birthdate of the youngest star to appear in a movie that year. We can express this query as

```
SELECT movieYear, MAX(birthdate)
FROM MovieStar, StarsIn
WHERE name = starName
GROUP BY movieYear;
```

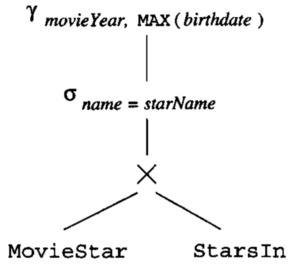

Figure 16.13: Initial logical query plan for the query of Example 16.14

An initial logical query plan constructed directly from the query is shown in Fig. 16.13. The `FROM` list is expressed by a product, and the `WHERE` clause by a selection above it. The grouping and aggregation are expressed by the γ operator above those. Some transformations that we could apply to Fig. 16.13 if we wished are:

1. Combine the selection and product into an equijoin.

2. Generate a δ below the γ, since the γ is duplicate-impervious.

3. Generate a π between the γ and the introduced δ to project onto `movieYear` and `birthdate`, the only attributes relevant to the γ.

The resulting plan is shown in Fig. 16.14.

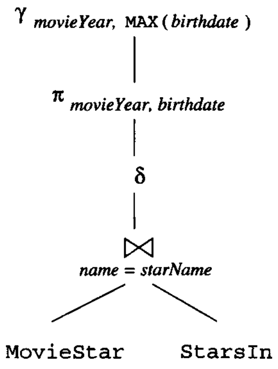

Figure 16.14: Another query plan for the query of Example 16.14

We can now push the δ below the $\bowtie$ and introduce π's below that if we wish. This new query plan is shown in Fig. 16.15. If `name` is a key for `MovieStar`, the δ can be eliminated along the branch leading to that relation. □

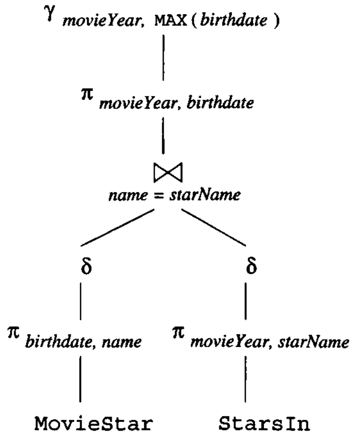

Figure 16.15: A third query plan for Example 16.14

16.2.8 Exercises for Section 16.2

Exercise 16.2.1: When it is possible to push a selection to both arguments of a binary operator, we need to decide whether or not to do so. How would the existence of indexes on one of the arguments affect our choice? Consider, for instance, an expression $\sigma_C(R \cap S)$, where there is an index on S.

Exercise 16.2.2: Give examples to show that:

a) Projection cannot be pushed below set union.

b) Projection cannot be pushed below set or bag difference.

c) Duplicate elimination (δ) cannot be pushed below projection.

d) Duplicate elimination cannot be pushed below bag union or difference.

! **Exercise 16.2.3:** Prove that we can always push a projection below both branches of a bag union.

! **Exercise 16.2.4:** Some laws that hold for sets hold for bags; others do not. For each of the laws below that are true for sets, tell whether or not it is true for bags. Either give a proof the law for bags is true, or give a counterexample.

a) $R \cup R = R$ (the idempotent law for union).

b) $R \cap R = R$ (the idempotent law for intersection).

c) $R - R = \emptyset$.

d) $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$ (distribution of union over intersection).

! **Exercise 16.2.5:** We can define $\subseteq$ for bags by: $R \subseteq S$ if and only if for every element x, the number of times x appears in R is less than or equal to the number of times it appears in S. Tell whether the following statements (which are all true for sets) are true for bags; give either a proof or a counterexample:

a) If $R \subseteq S$, then $R \cup S = S$.

b) If $R \subseteq S$, then $R \cap S = R$.

c) If $R \subseteq S$ and $S \subseteq R$, then $R = S$.

Exercise 16.2.6: Starting with an expression $\pi_L\big(R(a,b,c) \bowtie S(b,c,d,e)\big)$, push the projection down as far as it can go if L is:

a) $b + c \rightarrow x,\ c + d \rightarrow y$.

b) $a,\ b,\ a + d \rightarrow z$.

! Exercise 16.2.7: We mentioned in Example 16.14 that none of the plans we showed is necessarily the best plan. Can you think of a better plan?

! Exercise 16.2.8: The following are possible equalities involving operations on a relation $R(a, b)$. Tell whether or not they are true; give either a proof or a counterexample.

a) $\gamma_{MIN(a)\rightarrow y,\ x}\big(\gamma_{a,\ SUM(b)\rightarrow x}(R)\big) = \gamma_{y,SUM(b)\rightarrow x}\big(\gamma_{MIN(a)\rightarrow y,\ b}(R)\big)$.

b) $\gamma_{MIN(a)\rightarrow y,\ x}\big(\gamma_{a,\ MAX(b)\rightarrow x}(R)\big) = \gamma_{y,MAX(b)\rightarrow x}\big(\gamma_{MIN(a)\rightarrow y,\ b}(R)\big)$.

!! Exercise 16.2.9: The join-like operators of Exercise 15.2.4 obey some of the familiar laws, and others do not. Tell whether each of the following is or is not true. Give either a proof that the law holds or a counterexample.

a) $\sigma_C(R \ltimes S) = \sigma_C(R) \ltimes S$.

b) $\sigma_C(R \overset{\circ}{\bowtie} S) = \sigma_C(R) \overset{\circ}{\bowtie} S$.

c) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = \sigma_C(R) \overset{\circ}{\bowtie}_L S$, where C involves only attributes of R.

d) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = R \overset{\circ}{\bowtie}_L \sigma_C(S)$, where C involves only attributes of S.

e) $\pi_L(R \overline{\ltimes} S) = \pi_L(R) \overline{\ltimes} S$.

f) $(R \overset{\circ}{\bowtie} S) \overset{\circ}{\bowtie} T = R \overset{\circ}{\bowtie} (S \overset{\circ}{\bowtie} T)$.

g) $R \overset{\circ}{\bowtie} S = S \overset{\circ}{\bowtie} R$.

h) $R \overset{\circ}{\bowtie}_L S = S \overset{\circ}{\bowtie}_L R$.

i) $R \ltimes S = S \ltimes R$.

!! Exercise 16.2.10: While it is not precisely an algebraic law, because it involves an indeterminate number of operands, it is generally true that

$$\mathtt{SUM}(a_1, a_2, \ldots, a_n) = a_1 + a_2 + \cdots + a_n$$

SQL has both a `SUM` operator and addition for integers and reals. Considering the possibility that one or more of the a_i's could be `NULL`, rather than an integer or real, does this "law" hold in SQL?

16.3 From Parse Trees to Logical Query Plans

We now resume our discussion of the query compiler. Having constructed a parse tree for a query in Section 16.1, we next need to turn the parse tree into the preferred logical query plan. There are two steps, as was suggested in Fig. 16.1.

The first step is to replace the nodes and structures of the parse tree, in appropriate groups, by an operator or operators of relational algebra. We shall suggest some of these rules and leave some others for exercises. The second step is to take the relational-algebra expression produced by the first step and to turn it into an expression that we expect can be converted to the most efficient physical query plan.

16.3.1 Conversion to Relational Algebra

We shall now describe informally some rules for transforming SQL parse trees to algebraic logical query plans. The first rule, perhaps the most important, allows us to convert all "simple" select-from-where constructs to relational algebra directly. Its informal statement:

- If we have a <Query> with a <Condition> that has no subqueries, then we may replace the entire construct — the select-list, from-list, and condition — by a relational-algebra expression consisting, from bottom to top, of:

  1. The product of all the relations mentioned in the <FromList>, which is the argument of:
  2. A selection σ_C, where C is the <Condition> expression in the construct being replaced, which in turn is the argument of:
  3. A projection π_L, where L is the list of attributes in the <SelList>.

Example 16.15: Let us consider the parse tree of Fig. 16.5. The select-from-where transformation applies to the entire tree of Fig. 16.5. We take the product of the two relations `StarsIn` and `MovieStar` of the from-list, select for the condition in the subtree rooted at <Condition>, and project onto the select-list, `movieTitle`. The resulting relational-algebra expression is Fig. 16.16.

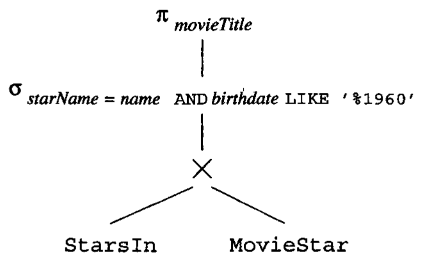

Figure 16.16: Translation of a parse tree to an algebraic expression tree

The same transformation does not apply to the outer query of Fig. 16.3. The reason is that the condition involves a subquery, a matter we defer to Section 16.3.2. However, we can apply the transformation to the subquery in

Limitations on Selection Conditions

One might wonder why we do not allow C, in a selection operator σ_C, to involve a subquery. It is conventional in relational algebra for the *arguments* of an operator — the elements that do not appear in subscripts — to be expressions that yield relations. On the other hand, *parameters* — the elements that appear in subscripts — have a type other than relations. For instance, parameter C in σ_C is a boolean-valued condition, and parameter L in π_L is a list of attributes or formulas.

If we follow this convention, then whatever calculation is implied by a parameter can be applied to each tuple of the relation argument(s). That limitation on the use of parameters simplifies query optimization. Suppose, in contrast, that we allowed an operator like $\sigma_C(R)$, where C involves a subquery. Then the application of C to each tuple of R involves computing the subquery. Do we compute it anew for every tuple of R? That would be unnecessarily expensive, unless the subquery were *correlated*, i.e., its value depends on something defined outside the query, as the subquery of Fig. 16.3 depends on the value of `starName`. Even correlated subqueries can be evaluated without recomputation for each tuple, in most cases, provided we organize the computation correctly.

Fig. 16.3. The expression of relational algebra that we get from the subquery is $\pi_{name}\big(\sigma_{birthdate\ \texttt{LIKE '\%1960'}}(\texttt{MovieStar})\big)$. □

16.3.2 Removing Subqueries From Conditions

For parse trees with a <Condition> that has a subquery, we shall introduce an intermediate form of operator, between the syntactic categories of the parse tree and the relational-algebra operators that apply to relations. This operator is often called *two-argument selection.* We shall represent a two-argument selection in a transformed parse tree by a node labeled σ, with no parameter. Below this node is a left child that represents the relation R upon which the selection is being performed, and a right child that is an expression for the condition applied to each tuple of R. Both arguments may be represented as parse trees, as expression trees, or as a mixture of the two.

Example 16.16: In Fig. 16.17 is a rewriting of the parse tree of Fig. 16.3 that uses a two-argument selection. Several transformations have been made to construct Fig. 16.17 from Fig. 16.3:

1. The subquery in Fig. 16.3 has been replaced by an expression of relational algebra, as discussed at the end of Example 16.15.

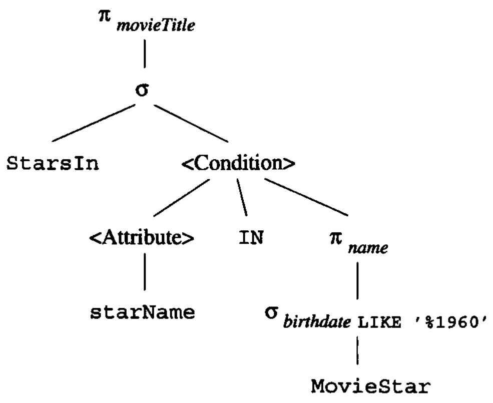

Figure 16.17: An expression using a two-argument σ, midway between a parse tree and relational algebra

2. The outer query has also been replaced, using the rule for select-from-where expressions from Section 16.3.1. However, we have expressed the necessary selection as a two-argument selection, rather than by the conventional σ operator of relational algebra. As a result, the upper node of the parse tree labeled <Condition> has not been replaced, but remains as an argument of the selection, with its parentheses and <Query> replaced by relational algebra, per point (1).

This tree needs further transformation, which we discuss next. □

We need rules that allow us to replace a two-argument selection by a one-argument selection and other operators of relational algebra. Each form of condition may require its own rule. In common situations, it is possible to remove the two-argument selection and reach an expression that is pure relational algebra. However, in extreme cases, the two-argument selection can be left in place and considered part of the logical query plan.

We shall give, as an example, the rule that lets us deal with the condition in Fig. 16.17 involving the `IN` operator. Note that the subquery in this condition is uncorrelated; that is, the subquery's relation can be computed once and for all, independent of the tuple being tested. The rule for eliminating such a condition is stated informally as follows:

- Suppose we have a two-argument selection in which the first argument represents some relation R and the second argument is a <Condition> of the form t `IN` S, where expression S is an uncorrelated subquery, and t is a tuple composed of (some) attributes of R. We transform the tree as follows:

  a) Replace the <Condition> by the tree that is the expression for S. If S may have duplicates, then it is necessary to include a δ operation at the root of the expression for S, so the expression being formed does not produce more copies of tuples than the original query does.

b) Replace the two-argument selection by a one-argument selection σ_C, where C is the condition that equates each component of the tuple t to the corresponding attribute of the relation S.

c) Give σ_C an argument that is the product of R and S.

Figure 16.18 illustrates this transformation.

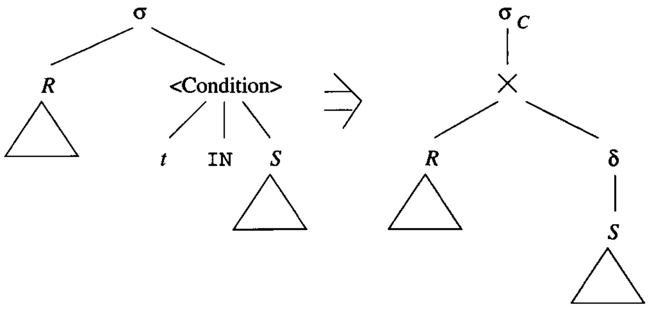

Figure 16.18: This rule handles a two-argument selection with a condition involving IN

Example 16.17: Consider the tree of Fig. 16.17, to which we shall apply the rule for IN conditions described above. In this figure, relation R is StarsIn, and relation S is the result of the relational-algebra expression consisting of the subtree rooted at π_{name}. The tuple t has one component, the attribute starName.

The two-argument selection is replaced by $\sigma_{starName=name}$; its condition C equates the one component of tuple t to the attribute of the result of query S. The child of the σ node is a $\times$ node, and the arguments of the $\times$ node are the node labeled StarsIn and the root of the expression for S. Notice that, because name is the key for MovieStar, there is no need to introduce a duplicate-eliminating δ in the expression for S. The new expression is shown in Fig. 16.19. It is completely in relational algebra, and is equivalent to the expression of Fig. 16.16, although its structure is quite different. □

The strategy for translating subqueries to relational algebra is more complex when the subquery is correlated. Since correlated subqueries involve unknown values defined outside themselves, they cannot be translated in isolation. Rather, we need to translate the subquery so that it produces a relation in which certain extra attributes appear — the attributes that must later be compared with the externally defined attributes. The conditions that relate attributes from the subquery to attributes outside are then applied to this relation, and

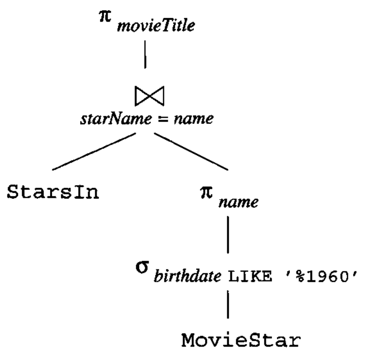

Figure 16.19: Applying the rule for IN conditions

the extra attributes that are no longer necessary can then be projected out. During this process, we must avoid introducing duplicate tuples, if the query does not eliminate duplicates at the end. The following example illustrates this technique.

```
SELECT DISTINCT m1.movieTitle, m1.movieYear
FROM StarsIn m1
WHERE m1.movieYear - 40 <= (
    SELECT AVG(birthdate)
    FROM StarsIn m2, MovieStar s
    WHERE m2.starName = s.name AND
        m1.movieTitle = m2.movieTitle AND
        m1.movieYear = m2.movieYear
);
```

Figure 16.20: Finding movies with high average star age

Example 16.18: Figure 16.20 is a SQL rendition of the query: "find the movies where the average age of the stars was at most 40 when the movie was made." To simplify, we treat `birthdate` as a birth year, so we can take its average and get a value that can be compared with the `movieYear` attribute of `StarsIn`. We have also written the query so that each of the three references to relations has its own tuple variable, in order to help remind us where the various attributes come from.

Fig. 16.21 shows the result of parsing the query and performing a partial translation to relational algebra. During this initial translation, we split the WHERE-clause of the subquery in two, and used part of it to convert the product of relations to an equijoin. We have retained the aliases m1, m2, and s in the nodes of this tree, in order to make clearer the origin of each attribute.

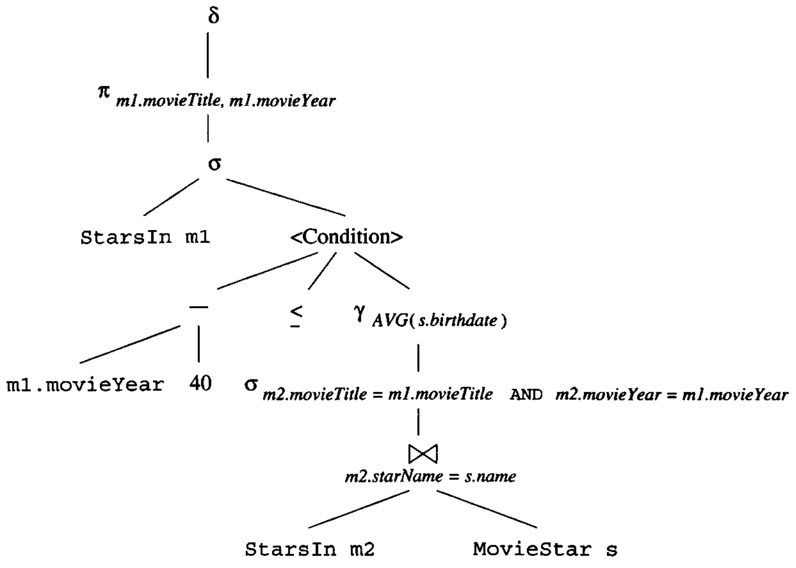

Figure 16.21: Partially transformed parse tree for Fig. 16.20

Alternatively, we could have used projections to rename attributes and thus avoid conflicting attribute names, but the result would be harder to follow.

In order to remove the <Condition> node and eliminate the two-argument σ, we need to create an expression that describes the relation in the right branch of the <Condition>. However, because the subquery is correlated, there is no way to obtain the attributes `m1.movieTitle` or `m1.movieYear` from the relations mentioned in the subquery, which are `StarsIn` (with alias `m2`) and `MovieStar`. Thus, we need to defer the selection

$$\sigma_{m2.movieTitle=m1.movieTitle \text{ AND } m2.movieYear=m1.movieYear}$$

until after the relation from the subquery is combined with the copy of `StarsIn` from the outer query (the copy aliased `m1`). To transform the logical query plan in this way, we need to modify the γ to group by the attributes `m2.movieTitle` and `m2.movieYear`, so these attributes will be available when needed by the selection. The net effect is that we compute for the subquery a relation consisting of movies, each represented by its title and year, and the average star birth year for that movie.

The modified group-by operator appears in Fig. 16.22; in addition to the two grouping attributes, we need to rename the average abd (average birthdate) so we can refer to it later. Figure 16.22 also shows the complete translation to relational algebra. Above the γ, the `StarsIn` from the outer query is joined with the result of the subquery. The selection from the subquery is then applied to the product of `StarsIn` and the result of the subquery; we show this selection as

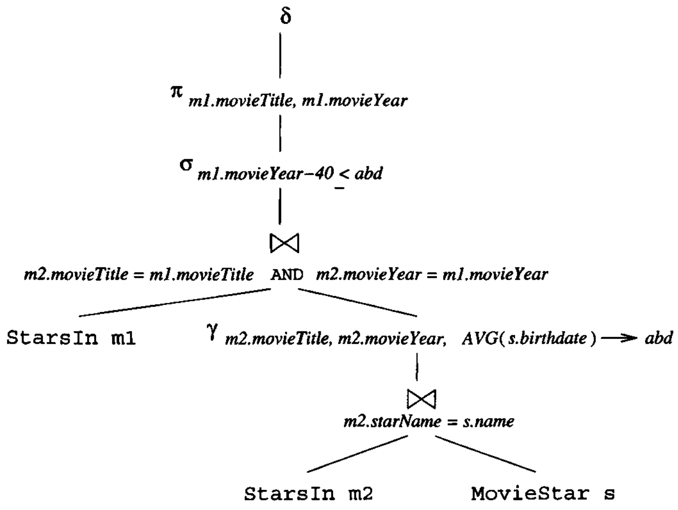
δ
π m1.movieTitle, m1.movieYear
σ m1.movieYear−40 ≤ abd
⋈ m2.movieTitle = m1.movieTitle AND m2.movieYear = m1.movieYear
StarsIn m1
γ m2.movieTitle, m2.movieYear, AVG(s.birthdate) → abd
⋈ m2.starName = s.name
StarsIn m2
MovieStar s

Figure 16.22: Translation of Fig. 16.21 to a logical query plan

a theta-join, which it would become after normal application of algebraic laws. Above the theta-join is another selection, this one corresponding to the selection of the outer query, in which we compare the movie's year to the average birth year of its stars. The algebraic expression finishes at the top like the expression of Fig. 16.21, with the projection onto the desired attributes and the elimination of duplicates.

As we shall see in Section 16.3.3, there is much more that a query optimizer can do to improve the query plan. This particular example satisfies three conditions that let us improve the plan considerably. The conditions are:

1. Duplicates are eliminated at the end,
2. Star names from `StarsIn m1` are projected out, and
3. The join between `StarsIn m1` and the rest of the expression equates the title and year attributes from `StarsIn m1` and `StarsIn m2`.

Because these conditions hold, we can replace all uses of `m1.movieTitle` and `m1.movieYear` by `m2.movieTitle` and `m2.movieYear`, respectively. Thus, the upper join in Fig. 16.22 is unnecessary, as is the argument `StarsIn m1`. This logical query plan is shown in Fig. 16.23. □

16.3.3 Improving the Logical Query Plan

When we convert our query to relational algebra we obtain one possible logical query plan. The next step is to rewrite the plan using the algebraic laws outlined

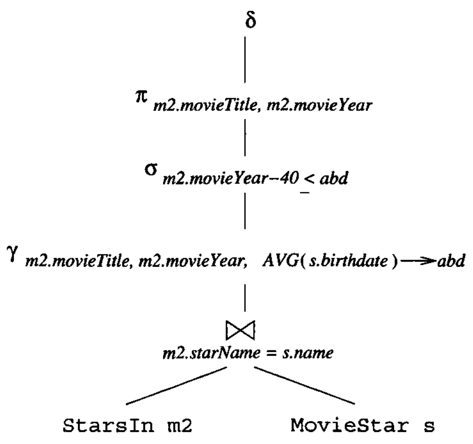
δ
π m2.movieTitle, m2.movieYear
σ m2.movieYear−40 ≤ abd
γ m2.movieTitle, m2.movieYear, AVG(s.birthdate)→abd
⋈ m2.starName = s.name
StarsIn m2
MovieStar s

Figure 16.23: Simplification of Fig. 16.22

in Section 16.2. Alternatively, we could generate more than one logical plan, representing different orders or combinations of operators. But in this book we shall assume that the query rewriter chooses a single logical query plan that it believes is "best," meaning that it is likely to result ultimately in the cheapest physical plan.

We do, however, leave open the matter of what is known as "join ordering," so a logical query plan that involves joining relations can be thought of as a family of plans, corresponding to the different ways a join could be ordered and grouped. We discuss choosing a join order in Section 16.6. Similarly, a query plan involving three or more relations that are arguments to the other associative and commutative operators, such as union, should be assumed to allow reordering and regrouping as we convert the logical plan to a physical plan. We begin discussing the issues regarding ordering and physical plan selection in Section 16.4.

There are a number of algebraic laws from Section 16.2 that tend to improve logical query plans. The following are most commonly used in optimizers:

- Selections can be pushed down the expression tree as far as they can go. If a selection condition is the AND of several conditions, then we can split the condition and push each piece down the tree separately. This strategy is probably the most effective improvement technique, but we should recall the discussion in Section 16.2.3, where we saw that in some circumstances it was necessary to push the selection up the tree first.

- Similarly, projections can be pushed down the tree, or new projections can be added. As with selections, the pushing of projections should be done with care, as discussed in Section 16.2.4.

- Duplicate eliminations can sometimes be removed, or moved to a more convenient position in the tree, as discussed in Section 16.2.6.

- Certain selections can be combined with a product below to turn the pair of operations into an equijoin, which is generally much more efficient to evaluate than are the two operations separately. We discussed these laws in Section 16.2.5.

Example 16.19: Let us consider the query of Fig. 16.16. First, we may split the two parts of the selection into $\sigma_{starName=name}$ and $\sigma_{birthdate}$ `LIKE '%1960'`. The latter can be pushed down the tree, since the only attribute involved, `birthdate`, is from the relation `MovieStar`. The first condition involves attributes from both sides of the product, but they are equated, so the product and selection is really an equijoin. The effect of these transformations is shown in Fig. 16.24. □

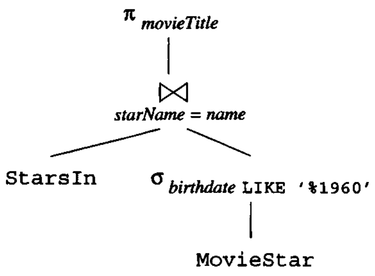
$\pi_{movieTitle}$
$\bowtie_{starName = name}$
StarsIn
$\sigma_{birthdate}$ LIKE '%1960'
MovieStar

Figure 16.24: The effect of query rewriting

16.3.4 Grouping Associative/Commutative Operators

An operator that is associative and commutative operators may be thought of as having any number of operands. Thinking of an operator such as join as having any number of operands lets us reorder those operands so that when the multiway join is executed as a sequence of binary joins, they take less time than if we had executed the joins in the order implied by the parse tree. We discuss ordering multiway joins in Section 16.6.

Thus, we shall perform a last step before producing the final logical query plan: for each portion of the subtree that consists of nodes with the same associative and commutative operator, we group the nodes with these operators into a single node with many children. Recall that the usual associative/commutative operators are natural join, union, and intersection. Natural joins and theta-joins can also be combined with each other under certain circumstances:

1. We must replace the natural joins with theta-joins that equate the attributes of the same name.

2. We must add a projection to eliminate duplicate copies of attributes involved in a natural join that has become a theta-join.

3. The theta-join conditions must be associative. Recall there are cases, as discussed in Section 16.2.1, where theta-joins are not associative.

In addition, products can be considered as a special case of natural join and combined with joins if they are adjacent in the tree. Figure 16.25 illustrates this transformation in a situation where the logical query plan has a cluster of two union operators and a cluster of three natural join operators. Note that the letters R through W stand for any expressions, not necessarily for stored relations.

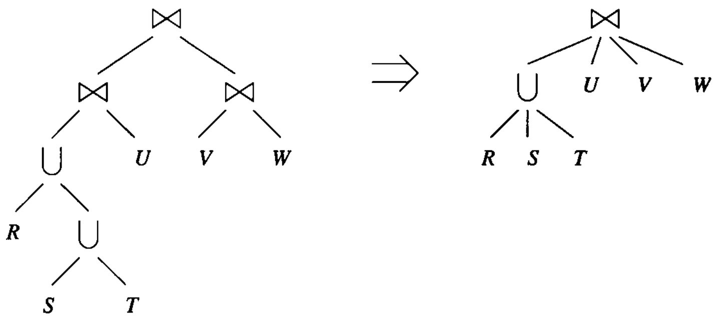

Figure 16.25: Final step in producing the logical query plan: group the associative and commutative operators

16.3.5 Exercises for Section 16.3

Exercise 16.3.1: Replace the natural joins in the following expressions by equivalent theta-joins and projections. Tell whether the resulting theta-joins form a commutative and associative group.

a) $\big(R(a,b) \bowtie S(b,c)\big) \bowtie_{S.c>T.c} T(c,d)$.

b) $\big(R(a,b) \bowtie S(b,c)\big) \bowtie \big(T(c,d) \bowtie U(d,e)\big)$.

c) $\big(R(a,b) \bowtie S(b,c)\big) \bowtie \big(T(c,d) \bowtie U(a,d)\big)$.

Exercise 16.3.2: Convert to relational algebra your parse trees from Exercise 16.1.3(a) and (b). For (b), show both the form with a two-argument selection and its eventual conversion to a one-argument (conventional σ_C) selection.

! **Exercise 16.3.3:** Give a rule for converting each of the following forms of <Condition> to relational algebra. All conditions may be assumed to be applied (by a two-argument selection) to a relation R. You may assume that the subquery is not correlated with R. Be careful that you do not introduce or eliminate duplicates in opposition to the formal definition of SQL.

a) A condition of the form `EXISTS`(<Query>).

b) A condition of the form a = `ANY` <Query>, where a is an attribute of R.

c) A condition of the form a = `ALL` <Query>, where a is an attribute of R.

!! **Exercise 16.3.4:** Repeat Exercise 16.3.3, but allow the subquery to be corollated with R. For simplicity, you may assume that the subquery has the simple form of select-from-where expression described in this section, with *no* further subqueries.

!! **Exercise 16.3.5:** From how many different expression trees could the grouped tree on the right of Fig. 16.25 have come? Remember that the order of children after grouping is not necessarily reflective of the ordering in the original expression tree.

16.4 Estimating the Cost of Operations

Having parsed a query and transformed it into a logical query plan, we must next turn the logical plan into a physical plan. We normally do so by considering many different physical plans that are derived from the logical plan, and evaluating or estimating the cost of each. After this evaluation, often called *cost-based enumeration*, we pick the physical query plan with the least estimated cost; that plan is the one passed to the query-execution engine. When enumerating possible physical plans derivable from a given logical plan, we select for each physical plan:

1. An order and grouping for associative-and-commutative operations like joins, unions, and intersections.

2. An algorithm for each operator in the logical plan, for instance, deciding whether a nested-loop join or a hash-join should be used.

3. Additional operators — scanning, sorting, and so on — that are needed for the physical plan but that were not present explicitly in the logical plan.

4. The way in which arguments are passed from one operator to the next, for instance, by storing the intermediate result on disk or by using iterators and passing an argument one tuple or one main-memory buffer at a time.

To make each of these choices, we need to understand what the costs of the various physical plans are. We cannot know these costs exactly without executing the plan. But almost always, the cost of executing a query plan is

Review of Notation

Recall from Section 15.1.3 the following size parameters:

- $B(R)$ is the number of blocks needed to hold relation R.
- $T(R)$ is the number of tuples of relation R.
- $V(R, a)$ is the *value count* for attribute a of relation R, that is, the number of distinct values relation R has in attribute a. Also, $V(R, [a_1, a_2, \ldots, a_n])$ is the number of distinct values R has when all of attributes $a_1, a_2, \ldots, a_n$ are considered together, that is, the number of tuples in $\delta(\pi_{a_1,a_2,\ldots,a_n}(R))$.

significantly greater than all the work done by the query compiler in selecting a plan. Thus, we do not want to execute more than one plan for one query, and we are forced to estimate the cost of any plan without executing it.

Therefore, our first problem is how to estimate costs of plans accurately. Such estimates are based on parameters of the data (see the box on "Review of Notation") that must be either computed exactly from the data or estimated by a process of "statistics gathering" that we discuss in Section 16.5.1. Given values for these parameters, we may make a number of reasonable estimates of relation sizes that can be used to predict the cost of a complete physical plan.

16.4.1 Estimating Sizes of Intermediate Relations

The physical plan is selected to minimize the estimated cost of evaluating the query. No matter what method is used for executing query plans, and no matter how costs of query plans are estimated, the sizes of intermediate relations of the plan have a profound influence on costs. Ideally, we want rules for estimating the number of tuples in an intermediate relation so that the rules:

1. Give accurate estimates.
2. Are easy to compute.
3. Are logically consistent; that is, the size estimate for an intermediate relation should not depend on how that relation is computed. For instance, the size estimate for a join of several relations should not depend on the order in which we join the relations.

There is no universally agreed-upon way to meet these three conditions. We shall give some simple rules that serve in most situations. Fortunately, the goal of size estimation is not to predict the exact size; it is to help select a physical query plan. Even an inaccurate size-estimation method will serve that purpose well if it errs consistently, that is, if the size estimator assigns the least cost to the best physical query plan, even if the actual cost of that plan turns out to be different from what was predicted.

16.4.2 Estimating the Size of a Projection

The extended projection of Section 5.2.5 is a bag projection and does not eliminate duplicates. We shall treat a clasical, duplicate-eliminating projection as a bag-projection followed by a δ. The extended projection of bags is different from the other operators, in that the size of the result is computable exactly. Normally, tuples shrink during a projection, as some components are eliminated. However, the extended projection allows the creation of new components that are combinations of attributes, and so there are situations where a π operator actually increases the size of the relation.

Example 16.20: Suppose $R(a, b, c)$ is a relation, where a and b are integers of four bytes each, and c is a string of 100 bytes. Let tuple headers require 12 bytes. Then each tuple of R requires 120 bytes. Let blocks be 1024 bytes long, with block headers of 24 bytes. We can thus fit 8 tuples in one block. Suppose $T(R) = 10{,}000$; i.e., there are 10,000 tuples in R. Then $B(R) = 1250$.

Consider $S = \pi_{a+b \rightarrow x, c}(R)$; that is, we replace a and b by their sum. Tuples of S require 116 bytes: 12 for header, 4 for the sum, and 100 for the string. Although tuples of S are slightly smaller than tuples of R, we can still fit only 8 tuples in a block. Thus, $T(S) = 10{,}000$ and $B(S) = 1250$.

Now consider $U = \pi_{a,b}(R)$, where we eliminate the string component. Tuples of U are only 20 bytes long. $T(U)$ is still 10,000. However, we can now pack 50 tuples of U into one block, so $B(U) = 200$. This projection thus shrinks the relation by a factor slightly more than 6. □

16.4.3 Estimating the Size of a Selection

When we perform a selection, we generally reduce the number of tuples, although the sizes of tuples remain the same. In the simplest kind of selection, where an attribute is equated to a constant, there is an easy way to estimate the size of the result, provided we know, or can estimate, the number of different values the attribute has. Let $S = \sigma_{A=c}(R)$, where A is an attribute of R and c is a constant. Then we recommend as an estimate:

- $T(S) = T(R)/V(R, A)$

This rule surely holds if the value of A is chosen randomly from among all the possible values.

The size estimate is more problematic when the selection involves an inequality comparison, for instance, $S = \sigma_{a<10}(R)$. One might think that on the average, half the tuples would satisfy the comparison and half not, so $T(R)/2$

The Zipfian Distribution

In estimating the size of a selection $\sigma_{A=c}$ it is not necessary to assume that values of A appear equally often. In fact, many attributes have values whose occurrences follow a *Zipfian distribution*, where the frequencies of the ith most common values are in proportion to $1/\sqrt{i}$. For example, if the most common value appears 1000 times, then the second most common value would be expected to appear about $1000/\sqrt{2}$ times, or 707 times, and the third most common value would appear about $1000/\sqrt{3}$ times, or 577 times. Originally postulated as a way to describe the relative frequencies of words in English sentences, this distribution has been found to appear in many sorts of data. For example, in the US, state populations follow an approximate Zipfian distribution. The three most populous states, California, Texas, and New York, have populations in ratio approximately 1:0.62:0.56, compared with the Zipfian ideal of 1:0.71:0.58. Thus, if `state` were an attribute of a relation describing US people, say a list of magazine subscribers, we would expect the values of `state` to distribute in the Zipfian, rather than uniform manner.

As long as the constant in the selection condition is chosen randomly, it doesn't matter whether the values of the attribute involved have a uniform, Zipfian, or other distribution; the *average* size of the matching set will still be $T(R)/V(R,a)$. However, if the constants are also chosen with a Zipfian distribution, then we would expect the average size of the selected set to be somewhat larger than $T(R)/V(R,a)$.

would estimate the size of S. However, there is an intuition that queries involving an inequality tend to retrieve a small fraction of the possible tuples.[3] Thus, we propose a rule that acknowledges this tendency, and assumes the typical inequality will return about one third of the tuples, rather than half the tuples. If $S = \sigma_{a<c}(R)$, then our estimate for $T(S)$ is:

- $T(S) = T(R)/3$

The case of a "not equals" comparison is rare. However, should we encounter a selection like $S = \sigma_{a\neq 10}(R)$, we recommend assuming that essentially all tuples will satisfy the condition. That is, take $T(S) = T(R)$ as an estimate. Alternatively, we may use $T(S) = T(R)\big(V(R,a)-1\big)/V(R,a)$, which is slightly less, as an estimate, acknowledging that about fraction $1/V(R,a)$ tuples of R will fail to meet the condition because their a-value *does* equal the constant.

When the selection condition C is the AND of several equalities and inequalities, we can treat the selection $\sigma_C(R)$ as a cascade of simple selections, each of

[3]For instance, if you had data about faculty salaries, would you be more likely to query for those faculty who made *less* than \$200,000 or *more* than \$200,000?

which checks for one of the conditions. Note that the order in which we place these selections doesn't matter. The effect will be that the size estimate for the result is the size of the original relation multiplied by the *selectivity* factor for each condition. That factor is 1/3 for any inequality, 1 for $\neq$, and $1/V(R, A)$ for any attribute A that is compared to a constant in the condition C.

Example 16.21: Let $R(a, b, c)$ be a relation, and $S = \sigma_{a=10 \texttt{ AND } b<20}(R)$. Also, let $T(R) = 10{,}000$, and $V(R, a) = 50$. Then our best estimate of $T(S)$ is $T(R)/(50 \times 3)$, or 67. That is, 1/50th of the tuples of R will survive the $a = 10$ filter, and 1/3 of those will survive the $b < 20$ filter.

An interesting special case where our analysis breaks down is when the condition is contradictory. For instance, consider $S = \sigma_{a=10 \texttt{ AND } a>20}(R)$. According to our rule, $T(S) = T(R)/3V(R, a)$, or 67 tuples. However, it should be clear that no tuple can have both $a = 10$ and $a > 20$, so the correct answer is $T(S) = 0$. When rewriting the logical query plan, the query optimizer can look for instances of many special-case rules. In the above instance, the optimizer can apply a rule that finds the selection condition logically equivalent to FALSE and replaces the expression for S by the empty set. □

When a selection involves an OR of conditions, say $S = \sigma_{C_1 \texttt{ OR } C_2}(R)$, then we have less certainty about the size of the result. One simple assumption is that no tuple will satisfy both conditions, so the size of the result is the sum of the number of tuples that satisfy each. That measure is generally an overestimate, and in fact can sometimes lead us to the absurd conclusion that there are more tuples in S than in the original relation R.

A less simple, but possibly more accurate estimate of the size of

$$S = \sigma_{C_1 \texttt{ OR } C_2}(R)$$

is to assume that C_1 and C_2 are independent. Then, if R has n tuples, m_1 of which satisfy C_1 and m_2 of which satisfy C_2, we would estimate the number of tuples in S as $n\big(1 - (1 - m_1/n)(1 - m_2/n)\big)$. In explanation, $1 - m_1/n$ is the fraction of tuples that do not satisfy C_1, and $1 - m_2/n$ is the fraction that do not satisfy C_2. The product of these numbers is the fraction of R's tuples that are *not* in S, and 1 minus this product is the fraction that are in S.

Example 16.22: Suppose $R(a, b)$ has $T(R) = 10{,}000$ tuples, and

$$S = \sigma_{a=10 \texttt{ OR } b<20}(R)$$

Let $V(R, a) = 50$. Then the number of tuples that satisfy $a = 10$ we estimate at 200, i.e., $T(R)/V(R, a)$. The number of tuples that satisfy $b < 20$ we estimate at $T(R)/3$, or 3333.

The simplest estimate for the size of S is the sum of these numbers, or 3533. The more complex estimate based on independence of the conditions $a = 10$ and $b < 20$ gives $10000\big(1 - (1 - 200/10000)(1 - 3333/10000)\big)$, or 3466. In this case, there is little difference between the two estimates, and it is very unlikely that choosing one over the other would change our estimate of the best physical query plan. □

The final operator that could appear in a selection condition is `NOT`. The estimated number of tuples of R that satisfy condition `NOT` C is $T(R)$ minus the estimated number that satisfy C.

16.4.4 Estimating the Size of a Join

We shall consider here only the natural join. Other joins can be handled according to the following outline:

1. The number of tuples in the result of an equijoin can be computed exactly as for a natural join, after accounting for the change in variable names. Example 16.24 will illustrate this point.

2. Other theta-joins can be estimated as if they were a selection following a product. Note that the number of tuples in a product is the product of the number of tuples in the relations involved.

We shall begin our study with the assumption that the natural join of two relations involves only the equality of two attributes. That is, we study the join $R(X,Y) \bowtie S(Y,Z)$, but initially we assume that Y is a single attribute although X and Z can represent any set of attributes.

The problem is that we don't know how the Y-values in R and S relate. For instance:

1. The two relations could have disjoint sets of Y-values, in which case the join is empty and $T(R \bowtie S) = 0$.

2. Y might be the key of S and the corresponding foreign key of R, so each tuple of R joins with exactly one tuple of S, and $T(R \bowtie S) = T(R)$.

3. Almost all the tuples of R and S could have the same Y-value, in which case $T(R \bowtie S)$ is about $T(R)T(S)$.

To focus on the most common situations, we shall make two simplifying assumptions:

- *Containment of Value Sets.* If Y is an attribute appearing in several relations, then each relation chooses its values from the front of a fixed list of values $y_1, y_2, y_3, \ldots$ and has all the values in that prefix. As a consequence, if R and S are two relations with an attribute Y, and $V(R,Y) \leq V(S,Y)$, then every Y-value of R will be a Y-value of S.

- *Preservation of Value Sets.* If we join a relation R with another relation, then an attribute A that is not a join attribute (i.e., not present in both relations) does not lose values from its set of possible values. More precisely, if A is an attribute of R but not of S, then $V(R \bowtie S, A) = V(R,A)$.

Assumption (1), containment of value sets, clearly might be violated, but it *is* satisfied when Y is a key in S and the corresponding foreign key in R. It also is approximately true in many other cases, since we would intuitively expect that if S has many Y-values, then a given Y-value that appears in R has a good chance of appearing in S.

Assumption (2), preservation of value sets, also might be violated, but it is true when the join attribute(s) of $R \bowtie S$ are a key for S and the corresponding foreign key of R. In fact, (2) can only be violated when there are "dangling tuples" in R, that is, tuples of R that join with no tuple of S; and even if there *are* dangling tuples in R, the assumption might still hold.

Under these assumptions, we can estimate the size of $R(X,Y) \bowtie S(Y,Z)$ as follows. Suppose r is a tuple in R, and S is a tuple in S. What is the probability that r and s agree on attribute Y? Suppose that $V(R,Y) \geq V(S,Y)$. Then the Y-value of s is surely one of the Y values that appear in R, by the containment-of-value-sets assumption. Hence, the chance that r has the same Y-value as s is $1/V(R,Y)$. Similarly, if $V(R,Y) < V(S,Y)$, then the value of Y in r will appear in S, and the probability is $1/V(S,Y)$ that r and s will share the same Y-value. In general, we see that the probability of agreement on the Y value is $1/\max\big(V(R,Y), V(S,Y)\big)$. Thus:

- $T(R \bowtie S) = T(R)T(S)/\max\big(V(R,Y), V(S,Y)\big)$

That is, the estimated number of tuples in $T(R \bowtie S)$ is the number of pairs of tuples — one from R and one from S, times the probability that such a pair shares a common Y value.

Example 16.23: Let us consider the following three relations and their important statistics:

$R(a,b)$	$S(b,c)$	$U(c,d)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R,b) = 20$	$V(S,b) = 50$	
	$V(S,c) = 100$	$V(U,c) = 500$

Suppose we want to compute the natural join $R \bowtie S \bowtie U$. One way is to group R and S first, as $(R \bowtie S) \bowtie U$. Our estimate for $T(R \bowtie S)$ is $T(R)T(S)/\max\big(V(R,b), V(S,b)\big)$, which is $1000 \times 2000/50$, or 40,000.

We then need to join $R \bowtie S$ with U. Our estimate for the size of the result is $T(R \bowtie S)T(U)/\max\big(V(R \bowtie S, c), V(U,c)\big)$. By our assumption that value sets are preserved, $V(R \bowtie S, c)$ is the same as $V(S,c)$, or 100; that is no values of attribute c disappeared when we performed the join. In that case, we get as our estimate for the number of tuples in $R \bowtie S \bowtie U$ the value $40{,}000 \times 5000/\max(100, 500)$, or 400,000.

We could also start by joining S and U. If we do, then we get the estimate $T(S \bowtie U) = T(S)T(U)/\max\big(V(S,c), V(U,c)\big) = 2000 \times 5000/500 = 20{,}000$.

By our assumption that value sets are preserved, $V(S \bowtie U, b) = V(S, b) = 50$, so the estimated size of the result is

$$T(R)T(S \bowtie U)/\max(V(R,b), V(S \bowtie U, b))$$

which is $1000 \times 20{,}000/50$, or 400,000. □

16.4.5 Natural Joins With Multiple Join Attributes

When the set of attributes Y in the join $R(X,Y) \bowtie S(Y,Z)$ consists of more than one attribute, the same argument as we used for a single attribute Y applies to each attribute in Y. That is:

- The estimate of the size of $R \bowtie S$ is computed by multiplying $T(R)$ by $T(S)$ and dividing by the larger of $V(R,y)$ and $V(S,y)$ for each attribute y that is common to R and S.

Example 16.24: The following example uses the rule above. It also illustrates that the analysis we have been doing for natural joins applies to any equijoin. Consider the join

$$R(a,b,c) \bowtie_{R.b=S.d \text{ AND } R.c=S.e} S(d,e,f)$$

Suppose we have the following size parameters:

$R(a,b,c)$	$S(d,e,f)$
$T(R) = 1000$	$T(S) = 2000$
$V(R,b) = 20$	$V(S,d) = 50$
$V(R,c) = 100$	$V(S,e) = 50$

We can think of this join as a natural join if we regard $R.b$ and $S.d$ as the same attribute and also regard $R.c$ and $S.e$ as the same attribute. Then the rule given above tells us the estimate for the size of $R \bowtie S$ is the product 1000×2000 divided by the larger of 20 and 50 and also divided by the larger of 100 and 50. Thus, the size estimate for the join is $1000 \times 2000/(50 \times 100) = 400$ tuples. □

Example 16.25: Let us reconsider Example 16.23, but consider the third possible order for the joins, where we first take $R(a,b) \bowtie U(c,d)$. This join is actually a product, and the number of tuples in the result is $T(R)T(U) = 1000 \times 5000 = 5{,}000{,}000$. Note that the number of different b's in the product is $V(R,b) = 20$, and the number of different c's is $V(U,c) = 500$.

When we join this product with $S(b,c)$, we multiply the numbers of tuples and divide by both $\max(V(R,b), V(S,b))$ and $\max(V(U,c), V(S,c))$. This quantity is $2000 \times 5{,}000{,}000/(50 \times 500) = 400{,}000$. Note that this third way of joining gives the same estimate for the size of the result that we found in Example 16.23. □

16.4.6 Joins of Many Relations

Finally, let us consider the general case of a natural join:

$$S = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$$

Suppose that attribute A appears in k of the R_i's, and the numbers of its sets of values in these k relations — that is, the various values of $V(R_i, A)$ for $i = 1, 2, \ldots, k$ — are $v_1 \leq v_2 \leq \cdots \leq v_k$, in order from smallest to largest. Suppose we pick a tuple from each relation. What is the probability that all tuples selected agree on attribute A?

In answer, consider the tuple t_1 chosen from the relation that has the smallest number of A-values, v_1. By the containment-of-value-sets assumption, each of these v_1 values is among the A-values found in the other relations that have attribute A. Consider the relation that has v_i values in attribute A. Its selected tuple t_i has probability $1/v_i$ of agreeing with t_1 on A. Since this claim is true for all $i = 2, 3, \ldots, k$, the probability that all k tuples agree on A is the product $1/v_2v_3 \cdots v_k$. This analysis gives us the rule for estimating the size of any join.

- Start with the product of the number of tuples in each relation. Then, for each attribute A appearing at least twice, divide by all but the least of the $V(R, A)$'s.

Likewise, we can estimate the number of values that will remain for attribute A after the join. By the preservation-of-value-sets assumption, it is the least of these $V(R, A)$'s.

Example 16.26: Consider the join $R(a, b, c) \bowtie S(b, c, d) \bowtie U(b, e)$, and suppose the important statistics are as given in Fig. 16.26. To estimate the size of this join, we begin by multiplying the relation sizes; $1000 \times 2000 \times 5000$. Next, we look at the attributes that appear more than once; these are b, which appears three times, and c, which appears twice. We divide by the two largest of $V(R, b)$, $V(S, b)$, and $V(U, b)$; these are 50 and 200. Finally, we divide by the larger of $V(R, c)$ and $V(S, c)$, which is 200. The resulting estimate is

$$1000 \times 2000 \times 5000/(50 \times 200 \times 200) = 5000$$

We can also estimate the number of values for each of the attributes in the join. Each estimate is the least value count for the attribute among all the relations in which it appears. These numbers are, for a, b, c, d, e respectively: 100, 20, 100, 400, and 500. □

Based on the two assumptions we have made — containment and preservation of value sets — we have a surprising and convenient property of the estimating rule given above.

- No matter how we group and order the terms in a natural join of n relations, the estimation rules, applied to each join individually, yield the

$R(a,b,c)$	$S(b,c,d)$	$U(b,e)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R,a) = 100$		
$V(R,b) = 20$	$V(S,b) = 50$	$V(U,b) = 200$
$V(R,c) = 200$	$V(S,c) = 100$	
	$V(S,d) = 400$	
		$V(U,e) = 500$

Figure 16.26: Parameters for Example 16.26

same estimate for the size of the result. Moreover, this estimate is the same that we get if we apply the rule for the join of all n relations as a whole.

Examples 16.23 and 16.25 form an illustration of this rule for the three groupings of a three-relation join, including the grouping where one of the "joins" is actually a product.

16.4.7 Estimating Sizes for Other Operations

We have seen two operations — selection and join — with reasonable estimating techniques. In addition, projections do not change the number of tuples in a relation, and products multiply the numbers of tuples in the argument relations. However, for the remaining operations, the size of the result is not easy to determine. We shall review the other relational-algebra operators and give some suggestions as to how this estimation could be done.

Union

If the bag union is taken, then the size is exactly the sum of the sizes of the arguments. A set union can be as large as the sum of the sizes or as small as the larger of the two arguments. We suggest that something in the middle be chosen, e.g., the larger plus half the smaller.

Intersection

The result can have as few as 0 tuples or as many as the smaller of the two arguments, regardless of whether set- or bag-intersection is taken. One approach is to take the average of the extremes, which is half the smaller.

Difference

When we compute $R - S$, the result can have between $T(R)$ and $T(R) - T(S)$ tuples. We suggest the average as an estimate: $T(R) - T(S)/2$.

Duplicate Elimination

If $R(a_1, a_2, \ldots, a_n)$ is a relation, then $V(R, [a_1, a_2, \ldots, a_n])$ is the size of $\delta(R)$. However, often we shall not have this statistic available, so it must be approximated. In the extremes, the size of $\delta(R)$ could be the same as the size of R (no duplicates) or as small as 1 (all tuples in R are the same).[4] Another upper limit on the number of tuples in $\delta(R)$ is the maximum number of distinct tuples that could exist: the product of $V(R, a_i)$ for $i = 1, 2, \ldots, n$. That number could be smaller than other estimates of $T\big(\delta(R)\big)$. There are several rules that could be used to estimate $T\big(\delta(R)\big)$. One reasonable one is to take the smaller of $T(R)/2$ and the product of all the $V(R, a_i)$'s.

Grouping and Aggregation

Suppose we have an expression $\gamma_L(R)$, the size of whose result we need to estimate. If the statistic $V(R, [g_1, g_2, \ldots, g_k])$, where the g_i's are the grouping attributes in L, is available, then that is our answer. However, that statistic may well not be obtainable, so we need another way to estimate the size of $\gamma_L(R)$. The number of tuples in $\gamma_L(R)$ is the same as the number of groups. There could be as few as one group in the result or as many groups as there are tuples in R. As with δ, we can also upper-bound the number of groups by a product of $V(R, A)$'s, but here attribute A ranges over only the grouping attributes of L. We again suggest an estimate that is the smaller of $T(R)/2$ and this product.

16.4.8 Exercises for Section 16.4

Exercise 16.4.1: Below are the vital statistics for four relations, W, X, Y, and Z:

$W(a,b)$	$X(b,c)$	$Y(c,d)$	$Z(d,e)$
$T(W) = 100$	$T(X) = 200$	$T(Y) = 300$	$T(Z) = 400$
$V(W,a) = 20$	$V(X,b) = 50$	$V(Y,c) = 50$	$V(Z,d) = 40$
$V(W,b) = 60$	$V(X,c) = 100$	$V(Y,d) = 50$	$V(Z,e) = 100$

Estimate the sizes of relations that are the results of the following expressions:

(a)	$W \bowtie X \bowtie Y \bowtie Z$	(b)	$\sigma_{a=10}(W)$	(c)	$\sigma_{c=20}(Y)$
(d)	$\sigma_{c=20}(Y) \bowtie Z$	(e)	$W \times Y$	(f)	$\sigma_{d>10}(Z)$
(g)	$\sigma_{a=1 \text{ AND } b=2}(W)$	(h)	$\sigma_{a=1 \text{ AND } b>2}(W)$	(i)	$X \bowtie_{X.c<Y.c} Y$

Exercise 16.4.2: Here are the statistics for four relations E, F, G, and H:

[4]Strictly speaking, if R is empty there are no tuples in either R or $\delta(R)$, so the lower bound is 0. However, we are rarely interested in this special case.

$E(a,b,c)$	$F(a,b,d)$	$G(a,c,d)$	$H(b,c,d)$
$T(E) = 1000$	$T(F) = 2000$	$T(G) = 3000$	$T(H) = 4000$
$V(E,a) = 1000$	$V(F,a) = 50$	$V(G,a) = 50$	$V(H,b) = 40$
$V(E,b) = 50$	$V(F,b) = 100$	$V(G,c) = 300$	$V(H,c) = 100$
$V(E,c) = 20$	$V(F,d) = 200$	$V(G,d) = 500$	$V(H,d) = 400$

How many tuples does the join of these tuples have, using the techniques for estimation from this section?

! **Exercise 16.4.3:** How would you estimate the size of a semijoin?

!! **Exercise 16.4.4:** Suppose we compute $R(a,b) \bowtie S(a,c)$, where R and S each have 1000 tuples. The a attribute of each relation has 100 different values, and they are the *same* 100 values. If the distribution of values was uniform; i.e., each a-value appeared in exactly 10 tuples of each relation, then there would be 10,000 tuples in the join. Suppose instead that the 100 a-values have the same Zipfian distribution in each relation. Precisely, let the values be $a_1, a_2, \ldots, a_{100}$. Then the number of tuples of both R and S that have a-value a_i is proportional to $1/\sqrt{i}$. Under these circumstances, how many tuples does the join have? You should ignore the fact that the number of tuples with a given a-value may not be an integer.

16.5 Introduction to Cost-Based Plan Selection

Whether selecting a logical query plan or constructing a physical query plan from a logical plan, the query optimizer needs to estimate the cost of evaluating certain expressions. We study the issues involved in cost-based plan selection here, and in Section 16.6 we consider in detail one of the most important and difficult problems in cost-based plan selection: the selection of a join order for several relations.

As before, we shall assume that the "cost" of evaluating an expression is approximated well by the number of disk I/O's performed. The number of disk I/O's, in turn, is influenced by:

1. The particular logical operators chosen to implement the query, a matter decided when we choose the logical query plan.

2. The sizes of intermediate results, whose estimation we discussed in Section 16.4.

3. The physical operators used to implement logical operators, e.g., the choice of a one-pass or two-pass join, or the choice to sort or not sort a given relation; this matter is discussed in Section 16.7.

4. The ordering of similar operations, especially joins as discussed in Section 16.6.

5. The method of passing arguments from one physical operator to the next, which is also discussed in Section 16.7.

Many issues need to be resolved in order to perform effective cost-based plan selection. In this section, we first consider how the size parameters, which were so essential for estimating relation sizes in Section 16.4, can be obtained from the database efficiently. We then revisit the algebraic laws we introduced to find the preferred logical query plan. Cost-based analysis justifies the use of many of the common heuristics for transforming logical query plans, such as pushing selections down the tree. Finally, we consider the various approaches to enumerating all the physical query plans that can be derived from the selected logical plan. Especially important are methods for reducing the number of plans that need to be evaluated, while making it likely that the least-cost plan is still considered.

16.5.1 Obtaining Estimates for Size Parameters

The formulas of Section 16.4 were predicated on knowing certain important parameters, especially $T(R)$, the number of tuples in a relation R, and $V(R, a)$, the number of different values in the column of relation R for attribute a. A modern DBMS generally allows the user or administrator explicitly to request the gathering of statistics, such as $T(R)$ and $V(R, a)$. These statistics are then used in query optimization, unchanged until the next command to gather statistics.

By scanning an entire relation R, it is straightforward to count the number of tuples $T(R)$ and also to discover the number of different values $V(R, a)$ for each attribute a. The number of blocks in which R can fit, $B(R)$, can be estimated either by counting the actual number of blocks used (if R is clustered), or by dividing $T(R)$ by the number of R's tuples that can fit in one block.

In addition, a DBMS may compute a ***histogram*** of the values for a given attribute. If $V(R, A)$ is not too large, then the histogram may consist of the number (or fraction) of the tuples having each of the values of attribute A. If there are many values of this attribute, then only the most frequent values may be recorded individually, while other values are counted in groups. The most common types of histograms are:

1. *Equal-width.* A width w is chosen, along with a constant v_0. Counts are provided of the number of tuples with values v in the ranges $v_0 \leq v < v_0 + w$, $v_0 + w \leq v < v_0 + 2w$, and so on. The value v_0 may be the lowest possible value or a lower bound on values seen so far. In the latter case, should a new, lower value be seen, we can lower the value of v_0 by w and add a new count to the histogram.

2. *Equal-height.* These are the common "percentiles." We pick some fraction p, and list the lowest value, the value that is fraction p from the lowest, the fraction $2p$ from the lowest, and so on, up to the highest value.

3. *Most-frequent-values.* We may list the most common values and their numbers of occurrences. This information may be provided along with a count of occurrences for all the other values as a group, or we may record frequent values in addition to an equal-width or equal-height histogram for the other values.

One advantage of keeping a histogram is that the sizes of joins can be estimated more accurately than by the simplified methods of Section 16.4. In particular, if a value of the join attribute appears explicitly in the histograms of both relations being joined, then we know exactly how many tuples of the result will have this value. For those values of the join attribute that do not appear explicitly in the histogram of one or both relations, we estimate their effect on the join as in Section 16.4. However, if we use an equal-width histogram, with the same bands for the join attributes of both relations, then we can estimate the size of the joins of corresponding bands, and sum those estimates. The result will be a good estimate, because only tuples in corresponding bands can join. The following examples will suggest how to carry out histogram-based estimation; we shall not use histograms in estimates subsequently.

Example 16.27: Consider histograms that mention the three most frequent values and their counts, and group the remaining values. Suppose we want to compute the join $R(a,b) \bowtie S(b,c)$. Let the histogram for $R.b$ be:

1: 200, 0: 150, 5: 100, others: 550

That is, of the 1000 tuples in R, 200 of them have b-value 1, 150 have b-value 0, and 100 have b-value 5. In addition, 550 tuples have b-values other than 0, 1, or 5, and none of these other values appears more than 100 times.

Let the histogram for $S.b$ be:

0: 100, 1: 80, 2: 70, others: 250

Suppose also that $V(R,b) = 14$ and $V(S,b) = 13$. That is, the 550 tuples of R with unknown b-values are divided among eleven values, for an average of 50 tuples each, and the 250 tuples of S with unknown b-values are divided among ten values, each with an average of 25 tuples each.

Values 0 and 1 appear explicitly in both histograms, so we can calculate that the 150 tuples of R with $b = 0$ join with the 100 tuples of S having the same b-value, to yield 15,000 tuples in the result. Likewise, the 200 tuples of R with $b = 1$ join with the 80 tuples of S having $b = 1$ to yield 16,000 more tuples in the result.

The estimate of the effect of the remaining tuples is more complex. We shall continue to make the assumption that every value appearing in the relation with the smaller set of values (S in this case) will also appear in the set of values of the other relation. Thus, among the eleven remaining b-values of S, we know one of those values is 2, and we shall assume another of the values is 5, since that is one of the most frequent values in R. We estimate that 2 appears 50 times in R, and 5 appears 25 times in S. These estimates are each obtained by assuming that the value is one of the "other" values for its relation's histogram. The number of additional tuples from b-value 2 is thus $70 \times 50 = 3500$, and the number of additional tuples from b-value 5 is $100 \times 25 = 2500$.

Finally, there are nine other b-values that appear in both relations, and we estimate that each of them appears in 50 tuples of R and 25 tuples of S. Each of the nine values thus contributes $50 \times 25 = 1250$ tuples to the result. The estimate of the output size is thus:

$$15000 + 16000 + 3500 + 2500 + 9 \times 1250$$

or 48,250 tuples. Note that the simpler estimate from Section 16.4 would be $1000 \times 500/14$, or 35,714, based on the assumptions of equal numbers of occurrences of each value in each relation. □

Example 16.28: In this example, we shall assume an equal-width histogram, and we shall demonstrate how knowing that values of two relations are almost disjoint can impact the estimate of a join size. Our relations are:

```
Jan(day, temp)
July(day, temp)
```

and the query is:

```
SELECT Jan.day, July.day
FROM Jan, July
WHERE Jan.temp = July.temp;
```

That is, find pairs of days in January and July that had the same temperature. The query plan is to equijoin `Jan` and `July` on the temperature, and project onto the two `day` attributes.

Suppose the histogram of temperatures for the relations `Jan` and `July` are as given in the table of Fig. 16.27.[5] In general, if both join attributes have equal-width histograms with the same set of bands, then we can estimate the size of the join by considering each pair of corresponding bands and summing.

If two corresponding bands have T_1 and T_2 tuples, respectively, and the number of values in a band is V, then the estimate for the number of tuples in the join of those bands is T_1T_2/V, following the principles laid out in Section 16.4.4. For the histograms of Fig. 16.27, many of these products are 0, because one or the other of T_1 and T_2 is 0. The only bands for which neither is 0 are 40–49 and 50–59. Since $V = 10$ is the width of a band, the 40–49 band contributes $10 \times 5/10 = 5$ tuples, and the 50–59 band contributes $5 \times 20/10 = 10$ tuples.

[5]Our friends south of the equator should reverse the columns for January and July, and convert to centigrade as well.

Range	Jan	July
0–9	40	0
10–19	60	0
20–29	80	0
30–39	50	0
40–49	10	5
50–59	5	20
60–69	0	50
70–79	0	100
80–89	0	60
90–99	0	10

Figure 16.27: Histograms of temperature

Thus our estimate for the size of this join is $5 + 10 = 15$ tuples. If we had no histogram, and knew only that each relation had 245 tuples distributed among 100 values from 0 to 99, then our estimate of the join size would be $245 \times 245/100 = 600$ tuples. □

16.5.2 Computation of Statistics

Statistics normally are computed only periodically, for several reasons. First, statistics tend not to change radically in a short time. Second, even somewhat inaccurate statistics are useful as long as they are applied consistently to all the plans. Third, the alternative of keeping statistics up-to-date can make the statistics themselves into a "hot-spot" in the database; because statistics are read frequently, we prefer not to update them frequently too.

The recomputation of statistics might be triggered automatically after some period of time, or after some number of updates. However, a database administrator, noticing that poor-performing query plans are being selected by the query optimizer on a regular basis, might request the recomputation of statistics in an attempt to rectify the problem.

Computing statistics for an entire relation R can be very expensive, particularly if we compute $V(R, a)$ for each attribute a in the relation (or even worse, compute histograms for each a). One common approach is to compute approximate statistics by sampling only a fraction of the data. For example, let us suppose we want to sample a small fraction of the tuples to obtain an estimate for $V(R, a)$. A statistically reliable calculation can be complex, depending on a number of assumptions, such as whether values for a are distributed uniformly, according to a Zipfian distribution, or according to some other distribution. However, the intuition is as follows. If we look at a small sample of R, say 1% of its tuples, and we find that most of the a-values we see are different, then it is likely that $V(R, a)$ is close to $T(R)$. If we find that the sample has very few different values of a, then it is likely that we have seen most of the a-values that exist in the current relation.

16.5.3 Heuristics for Reducing the Cost of Logical Query Plans

One important use of cost estimates for queries or subqueries is in the application of heuristic transformations of the query. We already have observed in Section 16.3.3 how certain heuristics, such as pushing selections down the tree, can be expected almost certainly to improve the cost of a logical query plan, regardless of relation sizes. However, there are other points in the query optimization process where estimating the cost both before and after a transformation will allow us to apply a transformation where it appears to reduce cost and avoid the transformation otherwise. In particular, when the preferred logical query plan is being generated, we may consider a number of optional transformations and the costs before and after.

Because we are estimating the cost of a *logical* query plan, and so we have not yet made decisions about the physical operators that will be used to implement the operators of relational algebra, our cost estimate cannot be based on disk I/O's. Rather, we estimate the sizes of all intermediate results using the techniques of Section 16.4, and their sum is our heuristic estimate for the cost of the entire logical plan. One example will serve to illustrate the issues and process.

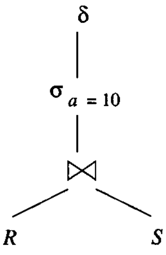

Figure 16.28: Logical query plan for Example 16.29

Example 16.29: Consider the initial logical query plan of Fig. 16.28, and let the statistics for the relations R and S be as follows:

$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$

To generate a final logical query plan from Fig. 16.28, we shall insist that the selection be pushed down as far as possible. However, we are not sure whether it makes sense to push the δ below the join or not. Thus, we generate from Fig. 16.28 the two query plans shown in Fig. 16.29; they differ in whether we have chosen to eliminate duplicates before or after the join. Notice that in plan (a) the δ is pushed down both branches of the tree. If R and/or S is known to have no duplicates, then the δ along its branch could be eliminated.

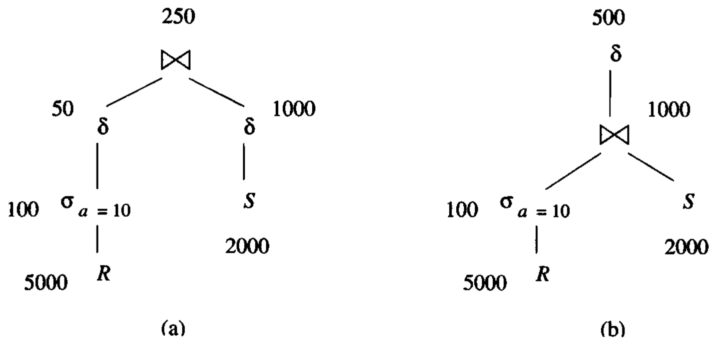

Figure 16.29: Two candidates for the best logical query plan

We know how to estimate the size of the result of the selections, from Section 16.4.3; we divide $T(R)$ by $V(R, a) = 50$. We also know how to estimate the size of the joins; we multiply the sizes of the arguments and divide by $\max\big(V(R,b), V(S,b)\big)$, which is 200. What we don't know is how to estimate the size of the relations with duplicates eliminated.

First, consider the size estimate for $\delta\big(\sigma_{a=10}(R)\big)$. Since $\sigma_{a=10}(R)$ has only one value for a and up to 100 values for b, and there are an estimated 100 tuples in this relation, the rule from Section 16.4.7 tells us that the product of the value counts for each of the attributes is not a limiting factor. Thus, we estimate the size of the result of δ as half the tuples in $\sigma_{a=10}(R)$, and Fig. 16.29(a) shows an estimate of 50 tuples for $\delta\big(\sigma_{a=10}(R)\big)$.

Now, consider the estimate of the result of the δ in Fig. 16.29(b). The join has one value for a, an estimated $\min\big(V(R,b), V(S,b)\big) = 100$ values for b, and an estimated $V(S,c) = 100$ values for c. Thus again the product of the value counts does not limit how big the result of the δ can be. We estimate this result as 500 tuples, or half the number of tuples in the join.

To compare the two plans of Fig. 16.29, we add the estimated sizes for all the nodes except the root and the leaves. We exclude the root and leaves, because these sizes are not dependent on the plan chosen. For plan (a) this cost, the sum of the estimated sizes of the interior nodes, is $100 + 50 + 1000 = 1150$, while for plan (b) the sum is $100 + 1000 = 1100$. Thus, by a small margin we conclude that deferring the duplicate elimination to the end is a better plan. We would come to the opposite conclusion if, say, R or S had fewer b-values. Then the join size would be greater, making the cost of plan (b) greater. □

Estimates for Result Sizes Need Not Be the Same

Notice that in Fig. 16.29 the estimates at the roots of the two trees are different: 250 in one case and 500 in the other. Because estimation is an inexact science, these sorts of anomalies will occur. In fact, it is the exception when we can offer a guarantee of consistency, as we did in Section 16.4.6.

Intuitively, the estimate for plan (b) is higher because if there are duplicates in both R and S, these duplicates will be multiplied in the join; e.g., for tuples that appear 3 times in R and twice in S, their join will appear six times in $R \bowtie S$. Our simple formula for estimating the size of the result of a δ does not take into account the possibility that the effect of duplicates has been amplified by previous operations.

16.5.4 Approaches to Enumerating Physical Plans

Now, let us consider the use of cost estimates in the conversion of a logical query plan to a physical query plan. The baseline approach, called *exhaustive*, is to consider all combinations of choices for each of the issues outlined at the beginning of Section 16.4 (order of joins, physical implementation of operators, and so on). Each possible physical plan is assigned an estimated cost, and the one with the smallest cost is selected.

However, there are a number of other approaches to selection of a physical plan. In this section, we shall outline various approaches that have been used, while Section 16.6 focuses on selecting a join order. Before proceeding, let us comment that there are two broad approaches to exploring the space of possible physical plans:

- *Top-down*: Here, we work down the tree of the logical query plan from the root. For each possible implementation of the operation at the root, we consider each possible way to evaluate its argument(s), and compute the cost of each combination, taking the best.[6]

- *Bottom-up*: For each subexpression of the logical-query-plan tree, we compute the costs of all possible ways to compute that subexpression. The possibilities and costs for a subexpression E are computed by considering the options for the subexpressions of E, and combining them in all possible ways with implementations for the root operator of E.

There is actually not much difference between the two approaches in their broadest interpretations, since either way, all possible combinations of ways to

[6]Remember from Section 16.3.4 that a single node of the logical-query-plan tree may represent many uses of a single commutative and associative operator, such as join. Thus, the consideration of all possible plans for a single node may itself involve enumeration of very many choices.

implement each operator in the query tree are considered. We shall concentrate on bottom-up methods in what follows.

You may, in fact, have noticed that there is an apparent simplification of the bottom-up method, where we consider only the *best* plan for each subexpression when we compute the plans for a larger subexpression. This approach, called ***dynamic programming*** in the list of methods below, is not guaranteed to yield the best overall plan, although often it does. The approach called *Selinger-style* (or *System-R-style*) optimization, also listed below, exploits additional properties that some of the plans for a subexpression may have, in order to produce optimal overall plans from plans that are not optimal for certain subexpressions.

Heuristic Selection

One option is to use the same approach to selecting a physical plan that is generally used for selecting a logical plan: make a sequence of choices based on heuristics. In Section 16.6.6, we shall discuss a "greedy" heuristic for join ordering, where we start by joining the pair of relations whose result has the smallest estimated size, then repeat the process for the result of that join and the other relations in the set to be joined. There are many other heuristics that may be applied; here are some of the most commonly used ones:

1. If the logical plan calls for a selection $\sigma_{A=c}(R)$, and stored relation R has an index on attribute A, then perform an index-scan (as in Section 15.1.1) to obtain only the tuples of R with A-value equal to c.

2. More generally, if the selection involves one condition like $A = c$ above, and other conditions as well, we can implement the selection by an index-scan followed by a further selection on the tuples, which we shall represent by the physical operator *filter*. This matter is discussed further in Section 16.7.1.

3. If an argument of a join has an index on the join attribute(s), then use an index-join with that relation in the inner loop.

4. If one argument of a join is sorted on the join attribute(s), then prefer a sort-join to a hash-join, although not necessarily to an index-join if one is possible.

5. When computing the union or intersection of three or more relations, group the smallest relations first.

Branch-and-Bound Plan Enumeration

This approach, often used in practice, begins by using heuristics to find a good physical plan for the entire logical query plan. Let the cost of this plan be C. Then as we consider other plans for subqueries, we can eliminate any plan for a subquery that has a cost greater than C, since that plan for the subquery could not possibly participate in a plan for the complete query that is better than what we already know. Likewise, if we construct a plan for the complete query that has cost less than C, we replace C by the cost of this better plan in subsequent exploration of the space of physical query plans.

An important advantage of this approach is that we can choose when to cut off the search and take the best plan found so far. For instance, if the cost C is small, then even if there are much better plans to be found, the time spent finding them may exceed C, so it does not make sense to continue the search. However, if C is large, then investing time in the hope of finding a faster plan is wise.

Hill Climbing

This approach, in which we really search for a "valley" in the space of physical plans and their costs, starts with a heuristically selected physical plan. We can then make small changes to the plan, e.g., replacing one method for executing an operator by another, or reordering joins by using the associative and/or commutative laws, to find "nearby" plans that have lower cost. When we find a plan such that no small modification yields a plan of lower cost, we make that plan our chosen physical query plan.

Dynamic Programming

In this variation of the general bottom-up strategy, we keep for each subexpression only the plan of least cost. As we work up the tree, we consider possible implementations of each node, assuming the best plan for each subexpression is also used. We examine this approach extensively in Section 16.6.

Selinger-Style Optimization

This approach improves upon the dynamic-programming approach by keeping for each subexpression not only the plan of least cost, but certain other plans that have higher cost, yet produce a result that is sorted in an order that may be useful higher up in the expression tree. Examples of such *interesting* orders are when the result of the subexpression is sorted on one of:

1. The attribute(s) specified in a sort (τ) operator at the root.
2. The grouping attribute(s) of a later group-by (γ) operator.
3. The join attribute(s) of a later join.

If we take the cost of a plan to be the sum of the sizes of the intermediate relations, then there appears to be no advantage to having an argument sorted. However, if we use the more accurate measure, disk I/O's, as the cost, then the advantage of having an argument sorted becomes clear if we can use one of the sort-based algorithms of Section 15.4, and save the work of the first pass for the argument that is sorted already.

16.5.5 Exercises for Section 16.5

Exercise 16.5.1: Estimate the size of the join $R(a,b) \bowtie S(b,c)$ using histograms for $R.b$ and $S.b$. Assume $V(R,b) = V(S,b) = 20$, and the histograms for both attributes give the frequency of the four most common values, as tabulated below:

	0	1	2	3	4	others
$R.b$	5	6	4	5		32
$S.b$	10	8	5		7	48

How does this estimate compare with the simpler estimate, assuming that all 20 values are equally likely to occur, with $T(R) = 52$ and $T(S) = 78$?

Exercise 16.5.2: Estimate the size of the join $R(a,b) \bowtie S(b,c)$ if we have the following histogram information:

	$b < 0$	$b = 0$	$b > 0$
R	500	100	400
S	300	200	500

! **Exercise 16.5.3:** In Example 16.29 we suggested that reducing the number of values that either attribute named b had could make plan (a) better than plan (b) of Fig. 16.29. For what values of:

a) $V(R,b)$

b) $V(S,b)$

will plan (a) have a lower estimated cost than plan (b)?

! **Exercise 16.5.4:** Consider four relations R, S, T, and V. Respectively, they have 200, 300, 400, and 500 tuples, chosen randomly and independently from the same pool of 1000 tuples (e.g., the probabilities of a given tuple being in R is $1/5$, in S is $3/10$, and in both is $3/50$).

a) What is the expected size of $R \cup S \cup T \cup V$?

b) What is the expected size of $R \cap S \cap T \cap V$?

c) What order of unions gives the least cost (estimated sum of the sizes of the intermediate relations)?

d) What order of intersections gives the least cost (estimated sum of the sizes of the intermediate relations)?

! **Exercise 16.5.5:** Repeat Exercise 16.5.4 if all four relations have 500 of the 1000 tuples, at random.

!! **Exercise 16.5.6:** Suppose we wish to compute the expression

$$\tau_b\big(R(a,b) \bowtie S(b,c) \bowtie T(c,d)\big)$$

That is, we join the three relations and produce the result sorted on attribute b. Let us make the simplifying assumptions:

i. We shall not "join" R and T first, because that is a product.

ii. Any other join can be performed with a two-pass sort-join or hash-join, but in no other way.

iii. Any relation, or the result of any expression, can be sorted by a two-phase, multiway merge-sort, but in no other way.

iv. The result of the first join will be passed as an argument to the last join one block at a time and not stored temporarily on disk.

v. Each relation occupies 1000 blocks, and the result of either join of two relations occupies 5000 blocks.

Answer the following based on these assumptions:

a) What are all the subexpressions and orders that a Selinger-style optimization would consider?

b) Which query plan uses the fewest disk I/O's?[7]

!! **Exercise 16.5.7:** Give an example of a logical query plan of the form $E \bowtie F$, for some expressions E and F (which you may choose), where using the best plans to evaluate E and F does not allow any choice of algorithm for the final join that minimizes the total cost of evaluating the entire expression. Make whatever assumptions you wish about the number of available main-memory buffers and the sizes of relations mentioned in E and F.

16.6 Choosing an Order for Joins

In this section we focus on a critical problem in cost-based optimization: selecting an order for the (natural) join of three or more relations. Similar ideas can be applied to other binary operations like union or intersection, but these operations are less important in practice, because they typically take less time to execute than joins, and they more rarely appear in clusters of three or more.

[7]Notice that, because we have made some very specific assumptions about the join methods to be used, we can estimate disk I/O's, instead of relying on the simpler, but less accurate, counts of tuples as our cost measure.

16.6.1 Significance of Left and Right Join Arguments

When ordering a join, we should remember that many of the join methods discussed in Chapter 15 are asymmetric. That is, the roles played by the two argument relations are different, and the cost of the join depends on which relation plays which role. Perhaps most important, the one-pass join of Section 15.2.3 reads one relation — preferably the smaller — into main memory, creating a structure such as a hash table to facilitate matching of tuples from the other relation. It then reads the other relation, one block at a time, to join its tuples with the tuples stored in memory.

For instance, suppose that when we select a physical plan we decide to use a one-pass join. Then we shall assume the left argument of the join is the smaller relation and store it in a main-memory data structure. This relation is called the ***build relation.*** The right argument of the join, called the ***probe relation,*** is read a block at a time and its tuples are matched in main memory with those of the build relation. Other join algorithms that distinguish between their arguments include:

1. Nested-loop join, where we assume the left argument is the relation of the outer loop.
2. Index-join, where we assume the right argument has the index.

16.6.2 Join Trees

When we have the join of two relations, we need to order the arguments. We shall conventionally select the one whose estimated size is the smaller as the left argument. It is quite common for there to be a significant and discernible difference in the sizes of arguments, because a query involving joins often also involves a selection on at least one attribute, and that selection reduces the estimated size of one of the relations greatly.

Example 16.30: Recall the query

```
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name AND
    birthdate LIKE '%1960';
```

from Fig. 16.4, which leads to the preferred logical query plan of Fig. 16.24, in which we take the join of relation `StarsIn` and the result of a selection on relation `MovieStar`. We have not given estimates for the sizes of relations `StarsIn` or `MovieStar`, but we can assume that selecting for stars born in a single year will produce about 1/50th of the tuples in `MovieStar`. Since there are generally several stars per movie, we expect `StarsIn` to be larger than `MovieStar` to begin with, so the second argument of the join, $\sigma_{birthdate\ \texttt{LIKE}\ \texttt{'\%1960'}}$(`MovieStar`), is much smaller than the first argument `StarsIn`. We conclude that the order of arguments in Fig. 16.24 should be reversed, so that the selection on `MovieStar` is the left argument. □

There are only two choices for a join tree when there are two relations — take either of the two relations to be the left argument. When the join involves more than two relations, the number of possible join trees grows rapidly. For example, Fig. 16.30 shows three of the five shapes of trees in which four relations R, S, T, and U, are joined. However, each of these trees has the four relations in alphabetical order from the left. Since order of arguments matters, and there are $n!$ ways to order n things, each tree represents $4! = 24$ different trees when the possible labelings of the leaves are considered.

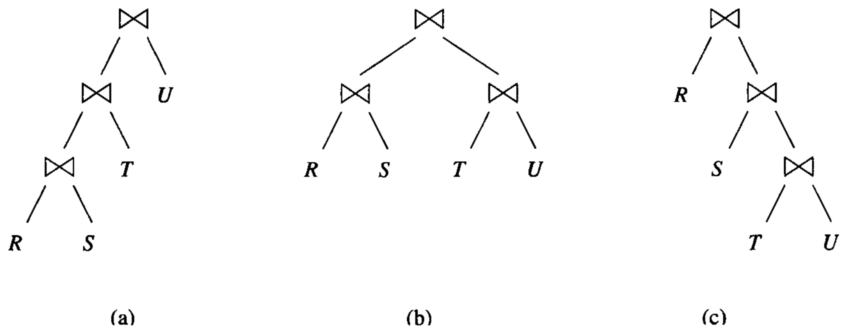

Figure 16.30: Ways to join four relations

16.6.3 Left-Deep Join Trees

Figure 16.30(a) is an example of what is called a *left-deep* tree. In general, a binary tree is left-deep if all right children are leaves. Similarly, a tree like Fig. 16.30(c), all of whose left children are leaves, is called a *right-deep* tree. A tree such as Fig. 16.30(b), that is neither left-deep nor right-deep, is called *bushy*. We shall argue below that there is a two-fold advantage to considering only left-deep trees as possible join orders.

1. The number of possible left-deep trees with a given number of leaves is large, but not nearly as large as the number of all trees. Thus, searches for query plans can be used for larger queries if we limit the search to left-deep trees.

2. Left-deep trees for joins interact well with common join algorithms — nested-loop joins and one-pass joins in particular. Query plans based on left-deep trees plus these join implementations will tend to be more efficient than the same algorithms used with non-left-deep trees.

The "leaves" in a left- or right-deep join tree can actually be interior nodes, with operators other than a join. Thus, for instance, Fig. 16.24 is technically a left-deep join tree with one join operator. The fact that a selection is applied to the right operand of the join does not take the tree out of the left-deep join class.

The number of left-deep trees does not grow nearly as fast as the number of all trees for the multiway join of a given number of relations. For n relations, there is only one left-deep tree shape, to which we may assign the relations in $n!$ ways. There are the same number of right-deep trees for n relations. However, the total number of tree shapes $T(n)$ for n relations is given by the recurrence:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= \textstyle\sum_{i=1}^{n-1} T(i)T(n-i) \end{aligned}$$

The explanation for the second equation is that we may pick any number i between 1 and $n-1$ to be the number of leaves in the left subtree of the root, and those leaves may be arranged in any of the $T(i)$ ways that trees with i leaves can be arranged. Similarly, the remaining $n-i$ leaves in the right subtree can be arranged in any of $T(n-i)$ ways.

The first few values of $T(n)$ are:

n	1	2	3	4	5	6
$T(n)$	1	1	2	5	14	42

To get the total number of trees once relations are assigned to the leaves, we multiply $T(n)$ by $n!$. Thus, for instance, the number of leaf-labeled trees of 6 leaves is $42 \times 6!$ or 30,240, of which 6!, or 720, are left-deep trees and another 720 are right-deep trees.

Now, let us consider the second advantage mentioned for left-deep join trees: their tendency to produce efficient plans. We shall give two examples:

1. If one-pass joins are used, and the build relation is on the left, then the amount of memory needed at any one time tends to be smaller than if we used a right-deep tree or a bushy tree for the same relations.

2. If we use nested-loop joins, with the relation of the outer loop on the left, then we avoid constructing any intermediate relation more than once.

Example 16.31: Consider the left-deep tree in Fig. 16.30(a), and suppose that we use a simple one-pass join for each of the three $\bowtie$ operators. As always, the left argument is the build relation; i.e., it will be held in main memory. To compute $R \bowtie S$, we need to keep R in main memory, and as we compute $R \bowtie S$ we need to keep the result in main memory as well. Thus, we need $B(R) + B(R \bowtie S)$ main-memory buffers. If we pick R to be the smallest of the relations, and a selection has made R be rather small, then there is likely to be no problem making this number of buffers available.

Having computed $R \bowtie S$, we must join this relation with T. However, the buffers used for R are no longer needed and can be reused to hold (some of) the result of $(R \bowtie S) \bowtie T$. Similarly, when we join this relation with U, the

Role of the Buffer Manager

The reader may notice a difference between our approach in the series of examples such as Example 15.4 and 15.6, where we assumed that there was a fixed limit on the number of main-memory buffers available for a join, and the more flexible assumption taken here, where we assume that as many buffers as necessary are available, but we try not to use "too many." Recall from Section 15.7 that the buffer manager has significant flexibility to allocate buffers to operations. However, if too many buffers are allocated at once, there will be thrashing, thus degrading the assumed performance of the algorithm being used.

relation $R \bowtie S$ is no longer needed, and its buffers can be reused for the result of the final join. In general, a left-deep join tree that is computed by one-pass joins requires main-memory space for at most two of the temporary relations any time.

Now, let us consider a similar implementation of the right-deep tree of Fig. 16.30(c). The first thing we need to do is load R into main-memory buffers, since left arguments are always the build relation. Then, we need to construct $S \bowtie (T \bowtie U)$ and use that as the probe relation for the join at the root. To compute $S \bowtie (T \bowtie U)$ we need to bring S into buffers and then compute $T \bowtie U$ as the probe relation for S. But $T \bowtie U$ requires that we first bring T into buffers. Now we have all three of R, S, and T in memory at the same time. In general, if we try to compute a right-deep join tree with n leaves, we shall have to bring $n - 1$ relations into memory simultaneously.

Of course it is possible that the total size $B(R) + B(S) + B(T)$ is less than the amount of space we need at either of the two intermediate stages of the computation of the left-deep tree, which are $B(R) + B(R \bowtie S)$ and $B(R \bowtie S) + B\big((R \bowtie S) \bowtie T\big)$, respectively. However, as we pointed out in Example 16.30, queries with several joins often will have a small relation with which we can start as the leftmost argument in a left-deep tree. If R is small, we might expect $R \bowtie S$ to be significantly smaller than S and $(R \bowtie S) \bowtie T$ to be smaller than T, further justifying the use of a left-deep tree. □

Example 16.32: Now, let us suppose we are going to implement the four-way join of Fig. 16.30 by nested-loop joins, and that we use an iterator (as in Section 15.1.6) for each of the three joins involved. Also, assume for simplicity that each of the relations R, S, T, and U are stored relations, rather than expressions. If we use the left-deep tree of Fig. 16.30(a), then the iterator at the root gets a main-memory-sized chunk of its left argument $(R \bowtie S) \bowtie T$. It then joins the chunk with all of U, but as long as U is a stored relation, it is only necessary to scan U, not to construct it. When the next chunk of the left argument is obtained and put in memory, U will be read again, but nested-loop join requires that repetition, which cannot be avoided if both arguments are large.

Similarly, to get a chunk of $(R \bowtie S) \bowtie T$, we get a chunk of $R \bowtie S$ into memory and scan T. Several scans of T may eventually be necessary, but cannot be avoided. Finally, to get a chunk of $R \bowtie S$ requires reading a chunk of R and comparing it with S, perhaps several times. However, in all this action, only stored relations are read multiple times, and this repeated reading is an artifact of the way nested-loop join works when the main memory is insufficient to hold an entire relation.

Now, compare the behavior of iterators on the left-deep tree with the behavior of iterators on the right-deep tree of Fig. 16.30(c). The iterator at the root starts by reading a chunk of R. It must then construct the entire relation $S \bowtie (T \bowtie U)$ and compare it with that chunk of R. When we read the next chunk of R into memory, $S \bowtie (T \bowtie U)$ must be constructed again. Each subsequent chunk of R likewise requires constructing this same relation.

Of course, we could construct $S \bowtie (T \bowtie U)$ once and store it, either in memory or on disk. If we store it on disk, we are using extra disk I/O's compared with the left-deep tree's plan, and if we store it in memory, then we run into the same problem with overuse of memory that we discussed in Example 16.31. □

16.6.4 Dynamic Programming to Select a Join Order and Grouping

To pick an order for the join of many relations we have three choices:

1. Consider them all.

2. Consider a subset.

3. Use a heuristic to pick one.

We shall here consider a sensible approach to enumeration called *dynamic programming*. It can be used either to consider all orders, or to consider certain subsets only, such as orders restricted to left-deep trees. In Section 16.6.6 we consider a heuristic for selecting a single ordering. Dynamic programming is a common algorithmic paradigm.[8] The idea behind dynamic programming is that we fill in a table of costs, remembering only the minimum information we need to proceed to a conclusion.

Suppose we want to join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$. In a dynamic programming algorithm, we construct a table with an entry for each subset of one or more of the n relations. In that table we put:

1. The estimated size of the join of these relations. For this quantity we may use the formula of Section 16.4.6.

[8]See Aho, Hopcroft and Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, for a general treatment of dynamic programming.

2. The least cost of computing the join of these relations. We shall use in our examples the sum of the sizes of the intermediate relations (not including the R_i's themselves or the join of the full set of relations associated with this table entry).

3. The expression that yields the least cost. This expression joins the set of relations in question, with some grouping. We can optionally restrict ourselves to left-deep expressions, in which case the expression is just an ordering of the relations.

The construction of this table is an induction on the subset size. There are two variations, depending on whether we wish to consider all possible tree shapes or only left-deep trees. We explain the difference when we discuss the inductive step of table construction.

BASIS: The entry for a single relation R consists of the size of R, a cost of 0, and an expression that is just R itself. The entry for a pair of relations $\{R_i, R_j\}$ is also easy to compute. The cost is 0, since there are no intermediate relations involved, and the size estimate is given by the rule of Section 16.4.6; it is the product of the sizes of R_i and R_j divided by the larger value-set size for each attribute shared by R_i and R_j, if any. The expression is either $R_i \bowtie R_j$ or $R_j \bowtie R_i$. Following the idea introduced in Section 16.6.1, we pick the smaller of R_i and R_j as the left argument.

INDUCTION: Now, we can build the table, computing entries for all subsets of size 3, 4, and so on, until we get an entry for the one subset of size n. That entry tells us the best way to compute the join of all the relations; it also gives us the estimated cost of that method, which is needed as we compute later entries. We need to see how to compute the entry for a set of k relations $\mathcal{R}$.

If we wish to consider only left-deep trees, then for each of the k relations R in $\mathcal{R}$ we consider the possibility that we compute the join for $\mathcal{R}$ by first computing the join of $\mathcal{R} - \{R\}$ and then joining it with R. The cost of the join for $\mathcal{R}$ is the cost of $\mathcal{R} - \{R\}$ plus the size of the result for $\mathcal{R} - \{R\}$. We pick whichever R yields the least cost. The expression for $\mathcal{R}$ has the best join expression for $\mathcal{R} - \{R\}$ as the left argument of a final join, and R as the right argument. The size for $\mathcal{R}$ is whatever the formula from Section 16.4.6 gives.

If we wish to consider all trees, then computing the entry for a set of relations $\mathcal{R}$ is somewhat more complex. We need to consider all ways to partition $\mathcal{R}$ into disjoint sets $\mathcal{R}_1$ and $\mathcal{R}_2$. For each such subset, we consider the sum of:

1. The best costs of $\mathcal{R}_1$ and $\mathcal{R}_2$.

2. The sizes of the results for $\mathcal{R}_1$ and $\mathcal{R}_2$.

For whichever partition gives the best cost, we use this sum as the cost for $\mathcal{R}$, and the expression for $\mathcal{R}$ is the join of the best join orders for $\mathcal{R}_1$ and $\mathcal{R}_2$.

Example 16.33: Consider the join of four relations R, S, T, and U. For simplicity, we shall assume they each have 1000 tuples. Their attributes and the estimated sizes of values sets for the attributes in each relation are summarized in Fig. 16.31.

$R(a,b)$	$S(b,c)$	$T(c,d)$	$U(d,a)$
$V(R,a) = 100$			$V(U,a) = 50$
$V(R,b) = 200$	$V(S,b) = 100$		
	$V(S,c) = 500$	$V(T,c) = 20$	
		$V(T,d) = 50$	$V(U,d) = 1000$

Figure 16.31: Parameters for Example 16.33

For the singleton sets, the sizes, costs, and best plans are as in the table of Fig. 16.32. That is, for each single relation, the size is as given, 1000 for each, the cost is 0 since there are no intermediate relations needed, and the best (and only) expression is the relation itself.

	$\{R\}$	$\{S\}$	$\{T\}$	$\{U\}$
Size	1000	1000	1000	1000
Cost	0	0	0	0
Best plan	R	S	T	U

Figure 16.32: The table for singleton sets

Now, consider the pairs of relations. The cost for each is 0, since there are still no intermediate relations in a join of two. There are two possible plans, since either of the two relations can be the left argument, but since the sizes happen to be the same for each relation we have no basis on which to choose between the plans. We shall take the first, in alphabetical order, to be the left argument in each case. The sizes of the resulting relations are computed by the usual formula. The results are summarized in Fig. 16.33.

	$\{R,S\}$	$\{R,T\}$	$\{R,U\}$	$\{S,T\}$	$\{S,U\}$	$\{T,U\}$
Size	5000	1,000,000	10,000	2000	1,000,000	1000
Cost	0	0	0	0	0	0
Best plan	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

Figure 16.33: The table for pairs of relations

Now, consider the table for joins of three out of the four relations. The only way to compute a join of three relations is to pick two to join first. The size estimate for the result is computed by the standard formula, and we omit the details of this calculation; remember that we'll get the same size regardless of which way we compute the join.

The cost estimate for each triple of relations is the size of the one intermediate relation — the join of the first two chosen. Since we want this cost to be as small as possible, we consider each pair of two out of the three relations and take the pair with the smallest size.

For the expression, we group the two chosen relations first, but these could be either the left or right argument. Let us suppose that we are only interested in left-deep trees, so we always use the join of the first two relations as the left argument. Since in all cases the estimated size for the join of two of our relations is at least 1000 (the size of each individual relation), were we to allow non-left-deep trees we would always select the single relation as the left argument in this example. The summary table for the triples is shown in Fig. 16.34.

	$\{R,S,T\}$	$\{R,S,U\}$	$\{R,T,U\}$	$\{S,T,U\}$
Size	10,000	50,000	10,000	2,000
Cost	2,000	5,000	1,000	1,000
Best plan	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

Figure 16.34: The table for triples of relations

Let us consider $\{R,S,T\}$ as an example of the calculation. We must consider each of the three pairs in turn. If we start with $R \bowtie S$, then the cost is the size of this relation, which is 5000 (see Fig. 16.33). Starting with $R \bowtie T$ gives us a cost of 1,000,000 for the intermediate relation, and starting with $S \bowtie T$ has a cost of 2000. Since the latter is the smallest cost of the three options, we choose that plan. The choice is reflected not only in the cost entry of the $\{R,S,T\}$ column, but in the best-plan row, where the plan that groups S and T first appears.

Now, we must consider the situation for the join of all four relations. There are two general ways we can compute the join of all four:

1. Pick three to join in the best possible way, and then join in the fourth.

2. Divide the four relations into two pairs of two, join the pairs and then join the results.

Of course, if we consider only left-deep trees then the second type of plan is excluded, because it yields bushy trees. The table of Fig. 16.35 summarizes the seven possible ways to group the joins, based on the preferred groupings from Figs. 16.33 and 16.34.

For instance, consider the first expression in Fig. 16.35. It represents joining R, S, and T first, and then joining that result with U. From Fig. 16.34, we know that the best way to join R, S, and T is to join S and T first. We have used the left-deep form of this expression, and joined U on the right to continue

Grouping	Cost
$((S \bowtie T) \bowtie R) \bowtie U$	12,000
$((R \bowtie S) \bowtie U) \bowtie T$	55,000
$((T \bowtie U) \bowtie R) \bowtie S$	11,000
$((T \bowtie U) \bowtie S) \bowtie R$	3,000
$(T \bowtie U) \bowtie (R \bowtie S)$	6,000
$(R \bowtie T) \bowtie (S \bowtie U)$	2,000,000
$(S \bowtie T) \bowtie (R \bowtie U)$	12,000

Figure 16.35: Join groupings and their costs

the left-deep form. If we consider only left-deep trees, then this expression and relation order is the only option. If we allowed bushy trees, we would join U on the left, since it is smaller than the join of the other three. The cost of this join is 12,000, which is the sum of the cost and size of $(S \bowtie T) \bowtie R$, which are 2000 and 10,000, respectively.

The last three expressions in Fig. 16.35 represent additional options if we include bushy trees. These are formed by joining relations first in two pairs. For example, the last line represents the strategy of joining $R \bowtie U$ and $S \bowtie T$, and then joining the result. The cost of this expression is the sum of the sizes and costs of the two pairs. The costs are 0, as must be the case for any pair, and the sizes are 10,000 and 2000, respectively. Since we generally select the smaller relation to be the left argument, we show the expression as $(S \bowtie T) \bowtie (R \bowtie U)$.

In this example, we see that the least of all costs is associated with the fourth expression: $((T \bowtie U) \bowtie S) \bowtie R$. This expression is the one we select for computing the join; its cost is 3000. Since it is a left-deep tree, it is the selected logical query plan regardless of whether our dynamic-programming strategy considers all plans or just left-deep plans. □

16.6.5 Dynamic Programming With More Detailed Cost Functions

Using relation sizes as the cost estimate simplifies the calculations in a dynamic-programming algorithm. However, a disadvantage of this simplification is that it does not involve the actual costs of the joins in the calculation. As an extreme example, if one possible join $R(a,b) \bowtie S(b,c)$ involves a relation R with one tuple and another relation S that has an index on the join attribute b, then the join takes almost no time. On the other hand, if S has no index, then we must scan it, taking $B(S)$ disk I/O's, even when R is a singleton. A cost measure that only involved the sizes of R, S, and $R \bowtie S$ cannot distinguish these two cases, so the cost of using $R \bowtie S$ in the grouping will be either overestimated or underestimated.

However, modifying the dynamic programming algorithm to take join algorithms into account is not hard. First, the cost measure we use becomes disk

I/O's. When computing the cost of $\mathcal{R}_1 \bowtie \mathcal{R}_2$, we sum the cost of $\mathcal{R}_1$, the cost of $\mathcal{R}_2$, and the least cost of joining these two relations using the best available algorithm. Since the latter cost usually depends on the sizes of $\mathcal{R}_1$ and $\mathcal{R}_2$, we must also compute estimates for these sizes as we did in Example 16.33.

An even more powerful version of dynamic programming is based on the Selinger-style optimization mentioned in Section 16.5.4. Now, for each set of relations that might be joined, we keep not only one cost, but several costs. Recall that Selinger-style optimization considers not only the least cost of producing the result of the join, but also the least cost of producing that relation sorted in any of a number of "interesting" orders. These interesting sorts include any that might be used to advantage in a later sort-join or that could be used to produce the output of the entire query in the sorted order desired by the user. When sorted relations must be produced, the use of sort-join, either one-pass or multipass, must be considered as an option, while without considering the value of sorting a result, hash-joins are always at least as good as the corresponding sort-join.

16.6.6 A Greedy Algorithm for Selecting a Join Order

As Example 16.33 suggests, even the carefully limited search of dynamic programming leads to a number of calculations that is exponential in the number of relations joined. It is reasonable to use an exhaustive method like dynamic programming or branch-and-bound search to find optimal join orders of five or six relations. However, when the number of joins grows beyond that, or if we choose not to invest the time necessary for an exhaustive search, then we can use a join-order heuristic in our query optimizer.

The most common choice of heuristic is a *greedy* algorithm, where we make one decision at a time about the order of joins and never backtrack or reconsider decisions once made. We shall consider a greedy algorithm that only selects a left-deep tree. The "greediness" is based on the idea that we want to keep the intermediate relations as small as possible at each level of the tree.

BASIS: Start with the pair of relations whose estimated join size is smallest. The join of these relations becomes the *current tree.*

INDUCTION: Find, among all those relations not yet included in the current tree, the relation that, when joined with the current tree, yields the relation of smallest estimated size. The new current tree has the old current tree as its left argument and the selected relation as its right argument.

Example 16.34: Let us apply the greedy algorithm to the relations of Example 16.33. The basis step is to find the pair of relations that have the smallest join. Consulting Fig. 16.33, we see that this honor goes to the join $T \bowtie U$, with a cost of 1000. Thus, $T \bowtie U$ is the "current tree."

We now consider whether to join R or S into the tree next. Thus we compare the sizes of $(T \bowtie U) \bowtie R$ and $(T \bowtie U) \bowtie S$. Figure 16.34 tells us that the

Join Selectivity

A useful way to view heuristics such as the greedy algorithm for selecting a left-deep join tree is that each relation R, when joined with the current tree, has a *selectivity*, which is the ratio of the size of the join result to size of the current tree's result. Since we usually do not have the exact sizes of either relation, we estimate these sizes as we have done previously. A greedy approach to join ordering is to pick that relation with the smallest selectivity.

For example, if a join attribute is a key for R, then the selectivity is at most 1, which is usually a favorable situation. Notice that, judging from the statistics of Fig. 16.31, attribute d is a key for U, and there are no keys for other relations, which suggests why joining T with U is the best way to start the join.

latter, with a size of 2000 is better than the former, with a size of 10,000. Thus, we pick as the new current tree $(T \bowtie U) \bowtie S$.

Now there is no choice; we must join R at the last step, leaving us with a total cost of 3000, the sum of the sizes of the two intermediate relations. Note that the tree resulting from the greedy algorithm is the same as that selected by the dynamic-programming algorithm in Example 16.33. However, there are examples where the greedy algorithm fails to find the best solution, while the dynamic-programming algorithm guarantees to find the best; see Exercise 16.6.4. □

16.6.7 Exercises for Section 16.6

Exercise 16.6.1: For the relations of Exercise 16.4.1, give the dynamic-programming table entries that evaluates all possible join orders allowing: a) All trees b) Left-deep trees only. What is the best choice in each case?

Exercise 16.6.2: Repeat Exercise 16.6.1 with the following modifications:

i. The schema for Z is changed to $Z(d, a)$.

ii. $V(Z, a) = 100$.

Exercise 16.6.3: Repeat Exercise 16.6.1 with the relations of Exercise 16.4.2.

Exercise 16.6.4: Consider the join of relations $R(a, b)$, $S(b, c)$, $T(c, d)$, and $U(a, d)$, where R and U each have 1000 tuples, while S and T each have 100 tuples. Further, there are 100 values of all attributes of all relations, except for attribute c, where $V(S, c) = V(T, c) = 10$.

a) What is the order selected by the greedy algorithm? What is its cost?

b) What is the optimum join ordering and its cost?

Exercise 16.6.5: How many trees are there for the join of (a) seven (b) eight relations? How many of these are neither left-deep nor right-deep?

! **Exercise 16.6.6:** Suppose we wish to join the relations R, S, T, and U in one of the tree structures of Fig. 16.30, and we want to keep all intermediate relations in memory until they are no longer needed. Following our usual assumption, the result of the join of all four will be consumed by some other process as it is generated, so no memory is needed for that relation. In terms of the number of blocks required for the stored relations and the intermediate relations [e.g., $B(R)$ or $B(R \bowtie S)$], give a lower bound on M, the number of blocks of memory needed, for each of the trees in Fig. 16.30? What assumptions let us conclude that one tree is certain to use less memory than another?

! **Exercise 16.6.7:** If we use dynamic programming to select an order for the join of k relations, how many entries of the table do we have to fill?

16.7 Completing the Physical-Query-Plan

We have parsed the query, converted it to an initial logical query plan, and improved that logical query plan with transformations described in Section 16.3. Part of the process of selecting the physical query plan is enumeration and cost-estimation for all of our options, which we discussed in Section 16.5. Section 16.6 focused on the question of enumeration, cost estimation, and ordering for joins of several relations. By extension, we can use similar techniques to order groups of unions, intersections, or any associative/commutative operation.

There are still several steps needed to turn the logical plan into a complete physical query plan. The principal issues that we must yet cover are:

1. Selection of algorithms to implement the operations of the query plan, when algorithm-selection was not done as part of some earlier step such as selection of a join order by dynamic programming.

2. Decisions regarding when intermediate results will be *materialized* (created whole and stored on disk), and when they will be *pipelined* (created only in main memory, and not necessarily kept in their entirety at any one time).

3. Notation for physical-query-plan operators, which must include details regarding access methods for stored relations and algorithms for implementation of relational-algebra operators.

We shall not discuss the subject of selection of algorithms for operators in its entirety. Rather, we sample the issues by discussing two of the most important operators: selection in Section 16.7.1 and joins in Section 16.7.2.

Then, we consider the choice between pipelining and materialization in Sections 16.7.3 through 16.7.5. A notation for physical query plans is presented in Section 16.7.6.

16.7.1 Choosing a Selection Method

One of the important steps in choosing a physical query plan is to pick algorithms for each selection operator. In Section 15.2.1 we mentioned the obvious implementation of a $\sigma_C(R)$ operator, where we access the entire relation R and see which tuples satisfy condition C. Then in Section 15.6.2 we considered the possibility that C was of the form "attribute equals constant," and we had an index on that attribute. If so, then we can find the tuples that satisfy condition C without looking at all of R. Now, let us consider the generalization of this problem, where we have a selection condition that is the AND of several conditions. Assume at least one condition is of the form $A\theta c$, where A is an attribute with an index, c is a constant, and θ is a comparison operator such as $=$ or $<$.

Each physical plan uses some number of attributes that each:

a) Have an index, and

b) Are compared to a constant in one of the terms of the selection.

We then use these indexes to identify the sets of tuples that satisfy each of the conditions. Sections 14.1.7 and 14.4.3 discussed how we could use pointers obtained from these indexes to find only the tuples that satisfied all the conditions before we read these tuples from disk.

For simplicity, we shall not consider the use of several indexes in this way. Rather, we limit our discussion to physical plans that:

1. Retrieve all tuples that satisfy a comparison for which an index exists, using the index-scan physical operator discussed in Section 15.1.1.

2. Consider each tuple selected in (1) to decide whether it satisfies the rest of the selection condition. The physical operator that performs this step is callled `Filter`.

In addition to physical plans of this form, we must also consider the plan that uses no index but reads the entire relation (using the table-scan physical operator) and passes each tuple to the `Filter` operator to check for satisfaction of the selection condition.

We decide among the possible physical plans for a selection by estimating the cost of reading data with each plan. To compare costs of alternative plans we cannot continue using the simplified cost estimate of intermediate-relation size. The reason is that we are now considering implementations of a single step of the logical query plan, and intermediate relations are independent of implementation.

Thus, we shall refocus our attention and resume counting disk I/O's, as we did when we discussed algorithms and their costs in Chapter 15. To simplify as before, we shall count only the cost of accessing the data blocks, not the index blocks. Recall that the number of index blocks needed is generally much smaller than the number of data blocks needed, so this approximation to disk I/O cost is usually accurate enough.

The following is an outline of how costs for the various plans are estimated. We assume that the operation is $\sigma_C(R)$, where condition C is the AND of one or more terms.

1. The cost of the table-scan algorithm coupled with a filter step is:
   (a) $B(R)$ if R is clustered, and
   (b) $T(R)$ if R is not clustered.

2. The cost of a plan that picks an equality term such as $a = 10$ for which an index on attribute a exists, uses index-scan to find the matching tuples, and then filters the retrieved tuples to see if they satisfy the full condition C is:
   (a) $B(R)/V(R,a)$ if the index is clustering, and
   (b) $T(R)/V(R,a)$ if the index is not clustering.

3. The cost of a plan that picks an inequality term such as $b < 20$ for which an index on attribute b exists, uses index-scan to retrieve the matching tuples, and then filters the retrieved tuples to see if they satisfy the full condition C is:
   (a) $B(R)/3$ if the index is clustering,[9] and
   (b) $T(R)/3$ if the index is not clustering.

Example 16.35: Consider selection $\sigma_{x=1 \texttt{ AND } y=2 \texttt{ AND } z<5}(R)$, where $R(x,y,z)$ has the following parameters: $T(R) = 5000$, $B(R) = 200$, $V(R,x) = 100$, and $V(R,y) = 500$. Further, suppose R is clustered, and there are indexes on all of x, y, and z, but only the index on z is clustering. The following are the options for implementing this selection:

1. Table-scan followed by filter. The cost is $B(R)$, or 200 disk I/O's, since R is clustered.

2. Use the index on x and the index-scan operator to find those tuples with $x = 1$, then use the filter operator to check that $y = 2$ and $z < 5$. Since there are about $T(R)/V(R,x) = 50$ tuples with $x = 1$, and the index is not clustering, we require about 50 disk I/O's.

[9]Recall that we assume the typical inequality retrieves only 1/3 the tuples, for reasons discussed in Section 16.4.3.

3. Use the index on y and index-scan to find those tuples with $y = 2$, then filter these tuples to see that $x = 1$ and $z < 5$. The cost for using this nonclustering index is about $T(R)/V(R,y)$, or 10 disk I/O's.

4. Use the clustering index on z and index-scan to find those tuples with $z < 5$, then filter these tuples to see that $x = 1$ and $y = 2$. The number of disk I/O's is about $B(R)/3 = 67$.

We see that the least cost plan is the third, with an estimated cost of 10 disk I/O's. Thus, the preferred physical plan for this selection retrieves all tuples with $y = 2$ and then filters for the other two conditions. □

16.7.2 Choosing a Join Method

We saw in Chapter 15 the costs associated with the various join algorithms. On the assumption that we know (or can estimate) how many buffers are available to perform the join, we can apply the formulas in Section 15.4.9 for sort-joins, Section 15.5.7 for hash-joins, and Sections 15.6.3 and 15.6.4 for indexed joins.

However, if we are not sure of, or cannot know, the number of buffers that will be available during the execution of this query (because we do not know what else the DBMS is doing at the same time), or if we do not have estimates of important size parameters such as the $V(R,a)$'s, then there are still some principles we can apply to choosing a join method. Similar ideas apply to other binary operations such as unions, and to the full-relation, unary operators, γ and δ.

- One approach is to call for the one-pass join, hoping that the buffer manager can devote enough buffers to the join, or that the buffer manager can come close, so thrashing is not a major cost. An alternative (for joins only, not for other binary operators) is to choose a nested-loop join, hoping that if the left argument cannot be granted enough buffers to fit in memory at once, then that argument will not have to be divided into too many pieces, and the resulting join will still be reasonably efficient.

- A sort-join is a good choice when either:

  1. One or both arguments are already sorted on their join attribute(s), or

  2. There are two or more joins on the same attribute, such as

  $$\big(R(a,b) \bowtie S(a,c)\big) \bowtie T(a,d)$$

  where sorting R and S on a will cause the result of $R \bowtie S$ to be sorted on a and used directly in a second sort-join.

- If there is an index opportunity such as a join $R(a,b) \bowtie S(b,c)$, where R is expected to be small (perhaps the result of a selection on a key that must yield only one tuple), and there is an index on the join attribute $S.b$, then we should choose an index-join.

- If there is no opportunity to use already-sorted relations or indexes, and a multipass join is needed, then hashing is probably the best choice, because the number of passes it requires depends on the size of the smaller argument rather than on both arguments.

16.7.3 Pipelining Versus Materialization

The last major issue we shall discuss in connection with choice of a physical query plan is pipelining of results. The naive way to execute a query plan is to order the operations appropriately (so an operation is not performed until the argument(s) below it have been performed), and store the result of each operation on disk until it is needed by another operation. This strategy is called *materialization*, since each intermediate relation is materialized on disk.

A more subtle, and generally more efficient, way to execute a query plan is to interleave the execution of several operations. The tuples produced by one operation are passed directly to the operation that uses it, without ever storing the intermediate tuples on disk. This approach is called *pipelining*, and it typically is implemented by a network of iterators (see Section 15.1.6), whose methods call each other at appropriate times. Since it saves disk I/O's, there is an obvious advantage to pipelining, but there is a corresponding disadvantage. Since several operations must share main memory at any time, there is a chance that algorithms with higher disk-I/O requirements must be chosen, or thrashing will occur, thus giving back all the disk-I/O savings that were gained by pipelining, and possibly more.

16.7.4 Pipelining Unary Operations

Unary operations — selection and projection — are excellent candidates for pipelining. Since these operations are tuple-at-a-time, we never need to have more than one block for input, and one block for the output. This mode of operation was suggested by Fig. 15.5.

We may implement a pipelined unary operation by iterators, as discussed in Section 15.1.6. The consumer of the pipelined result calls `GetNext()` each time another tuple is needed. In the case of a projection, it is only necessary to call `GetNext()` once on the source of tuples, project that tuple appropriately, and return the result to the consumer. For a selection σ_C (technically, the physical operator `Filter(`C`)`), it may be necessary to call `GetNext()` several times at the source, until one tuple that satisfies condition C is found. Figure 16.36 illustrates this process.

16.7.5 Pipelining Binary Operations

The results of binary operations can also be pipelined. We use one buffer to pass the result to its consumer, one block at a time. However, the number of other buffers needed to compute the result and to consume the result varies,

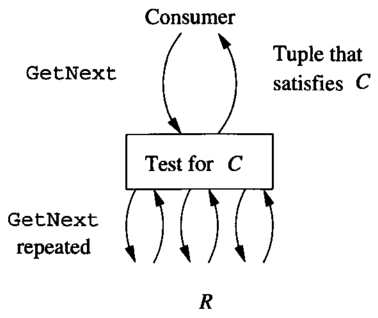

Figure 16.36: Execution of a pipelined selection using iterators

Materialization in Memory

One might imagine that there is an intermediate approach, between pipelining and materialization, where the entire result of one operation is stored in main-memory buffers (not on disk) before being passed to the consuming operation. We regard this possible mode of operation as pipelining, where the first thing that the consuming operation does is organize the entire relation, or a large portion of it, in memory. An example of this sort of behavior is a selection whose result becomes the left (build) argument to one of several join algorithms, including the simple one-pass join, multipass hash-join, or sort-join.

depending on the size of the result and the sizes of the arguments. We shall use an extended example to illustrate the tradeoffs and opportunities.

Example 16.36: Let us consider physical query plans for the expression

$$\big(R(w,x) \bowtie S(x,y)\big) \bowtie U(y,z)$$

We make the following assumptions:

1. R occupies 5000 blocks; S and U each occupy 10,000 blocks.
2. The intermediate result $R \bowtie S$ occupies k blocks for some k.
3. Both joins will be implemented as hash-joins, either one-pass or two-pass, depending on k.
4. There are 101 buffers available. This number, as usual, is set artificially low.