Chapter 6

Query Execution

Previous chapters gave us data structures that help support basic database operations such as finding tuples given a search key. We are now ready to use these structures to support efficient algorithms for answering queries. The broad topic of query processing will be covered in this chapter and Chapter 7. The query processor is the group of components of a DBMS that turns user queries and data-modification commands into a sequence of operations on the database and executes those operations. Since SQL lets us express queries at a very high level, the query processor must supply a lot of detail regarding how the query is to be executed. Moreover, a naive execution strategy for a query may lead to an algorithm for executing the query that takes far more time than necessary.

Figure 6.1 suggests the division of topics between Chapters 6 and 7. In this chapter, we concentrate on query execution, that is, the algorithms that manipulate the data of the database. We shall begin with a review of relational algebra. This notation, or something similar, is used by most relational database systems to represent internally the queries that the user expresses in SQL. This algebra involves operations on relations, such as join and union, with which you may already be familiar. However, SQL uses a bag (multiset) model of data, rather than a set model. Also, there are operations in SQL, such as aggregation, grouping, and ordering (sorting), that are not part of the classical relational algebra. Thus, we need to reconsider this algebra in the light of its role as a representation for SQL queries.

One advantage of using relational algebra is that it makes alternative forms of a query easy to explore. The different algebraic expressions for a query are called logical query plans. Often, these plans are represented as expression trees, as we shall do in this book.

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

**Preview of Query Compilation**

Query compilation can be divided into three major steps, as sketched in Fig. 6.2.

a) *Parsing*, in which a *parse tree*, representing the query and its structure, is constructed.

b) *Query rewrite*, in which the parse tree is converted to an initial query plan, which is usually an algebraic representation of the query. This initial plan is then transformed into an equivalent plan that is expected to require less time to execute.

c) *Physical plan generation*, where the abstract query plan from (b), often called a *logical query plan*, is turned into a *physical query plan* by selecting algorithms to implement each of the operators of the logical plan, and by selecting an order of execution for these operators. The physical plan, like the result of parsing and the logical plan, is represented by an expression tree. The physical plan also includes details such as how the queried relations are accessed, and when and if a relation should be sorted.
Parts (b) and (c) are often called the *query optimizer*, and these are the hard parts of query compilation. Chapter 7 is devoted to query optimization; we shall learn there how to select a "query plan" that takes as little time as possible. To select the best query plan we need to decide:

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

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

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

Each of these choices depends on the metadata about the database. Typical metadata that is available to the query optimizer includes: the size of each relation; statistics such as the approximate number and frequency of different values for an attribute; the existence of certain indexes; and the layout of data on disk.
CHAPTER 6. QUERY EXECUTION

6.1 An Algebra for Queries

In order to talk about good algorithms for executing queries, we first need to develop a notation for the elementary actions of which queries are constructed. Many SQL queries are expressed with a few operators that form the classical "relational algebra," and even object-oriented query languages perform essentially the same operations as are found in relational algebra. However, there are also features of SQL and other query languages that are not expressed in the classical relational algebra, so after describing that algebra we shall add operators for SQL features such as group-by, order-by, and select-distinct.

Moreover, relational algebra was originally designed as if relations were sets. Yet relations in SQL are really bags, or multisets; that is, the same tuple can appear more than once in an SQL relation. Thus, we shall introduce relational algebra as an algebra on bags. The relational algebra operators are:

- **Union, intersection, and difference.** On sets, these are the usual set operators. On bags, there are some differences that we shall discuss in Section 6.1.1. These operators correspond to the SQL operators UNION, INTERSECT, and EXCEPT.

- **Selection.** This operator produces a new relation from an old one by selecting some of the rows of the old relation, based on some condition or predicate. It corresponds roughly to the WHERE clause of an SQL query.

- **Projection.** This operator produces a new relation from an old one by choosing some of the columns, like the SELECT clause of an SQL query. We shall extend this operator beyond classical relational algebra to allow renaming of attributes and construction of attributes by calculation with constants and attributes of the old relation, just as the SQL SELECT clause does.

- **Product.** This operator is the set-theoretic Cartesian product (or cross-product), which constructs tuples by pairing the tuples of two relations in all possible ways. It corresponds in SQL to the list of relations in a FROM clause, whose product forms the relation to which the condition of the WHERE clause and the projection of the SELECT clause are applied.

- **Join.** There are various types of join operators, which correspond to the operators such as JOIN, NATURAL JOIN, and OUTER JOIN in the SQL2 standard. We shall discuss these in Section 6.1.5.

In addition, we shall extend the relational algebra with the following operators, introduced for the purpose of discussing the optimization of the full range of possible SQL queries:

- **Duplicate elimination.** This operator turns a bag into a set, like the keyword DISTINCT in an SQL SELECT clause.
6.1. AN ALGEBRA FOR QUERIES

• **Grouping.** This operator is designed to mimic the effect of an SQL GROUP BY, as well as the aggregation operators (sum, average, and so on) that may appear in SQL SELECT clauses.

• **Sorting.** This operator represents the effect of the SQL ORDER BY clause. It is also used as part of certain sort-based algorithms for other operators such as join.

6.1.1 Union, Intersection, and Difference

If relations were sets, then the operators U, ∩, and — on relations would be the familiar operations. However, there are two major differences between SQL relations and sets:

a) Relations are bags.

b) Relations have schemas, i.e., sets of attributes that name their columns.

Problem (b) is easy to deal with. For union, intersection, and difference, we require that the schemas of the two argument relations must be the same. The schema of the resulting relation is thus the same as that of either argument.

However, (a) requires some new definitions, because the way union, intersection, and difference work on bags is somewhat different from the way they work on sets. The rules for constructing the results change to the following:

1. For \( R \cup S \), a tuple \( t \) is in the result as many times as the number of times it is in \( R \) plus the number of times it is in \( S \).

2. For \( R \cap S \), a tuple \( t \) is in the result the minimum of the number of times it is in \( R \) and \( S \).

3. For \( R - S \), a tuple \( t \) is in the result the number of times it is in \( R \) minus the number of times it is in \( S \), but not fewer than zero times.

Notice that if \( R \) and \( S \) happen to be sets, i.e., no element appears more than once in either, then the result of \( R \cap S \) or \( R - S \) will be exactly what we would expect from the definition of these operators on sets. However, even if \( R \) and \( S \) are sets, the bag-union \( R \cup S \) can have a result that is not a set. Specifically, an element that appears in both \( R \) and \( S \) will appear twice in \( R \cup S \), but only once if the set-union were taken.

**Example 6.1:** Let \( R = \{A, B, B\} \) and \( S = \{C, A, B, C\} \) be two bags. Then:

- \( R \cup S = \{A, A, B, B, B, C, C\} \).
- \( R \cap S = \{A, B\} \).
- \( R - S = \{B\} \).
The union contains \( A \) twice because each of \( R \) and \( S \) has one copy of \( A \); it contains three \( B \)'s because there are two in \( R \) and one in \( S \). We have shown the members of the union in sorted order, but you should remember that order doesn’t matter and we could have permuted the seven members of the bag in any way we wished.

The intersection has one \( A \) because each of \( R \) and \( S \) contains one \( A \); so the minimum number of times \( A \) appears is 1. There is also one \( B \), because while there are two \( B \)'s in \( S \), there is only one in \( R \). \( C \) does not appear at all, because although it appears twice in \( S \), it appears zero times in \( R \).

Finally, the difference does not have an \( A \), because although \( A \) appears in \( R \), it appears at least as many times in \( S \). The difference has one \( B \), because it appears twice in \( R \) and once in \( S \), and \( 2 - 1 = 1 \). There is no \( C \), because \( C \) does not appear in \( R \). The fact that \( C \) appears in \( S \) is thus irrelevant as far as the difference \( R - S \) is concerned.

The rules above for operations on bags do not depend on whether the members of \( R \) and \( S \) are tuples, objects, or something else. However, for the relational algebra, we assume that \( R \) and \( S \) are relations, and therefore that they have schemas (i.e., lists of attributes labeling the columns of the relations). We require that the schemas of the two relations be the same, if we are to take their union, intersection, or difference. Moreover, we assign the same schema to the result, so the result too is a relation.

By default, the SQL operators UNION, INTERSECT, and EXCEPT eliminate duplicates from the result, even if the argument relations have duplicates. We can take the bag versions of those operations in SQL by using the keyword ALL, as in UNION ALL for example. Note that the default versions of these operators in SQL are not precisely the set operators; rather they are the bag versions of the operators followed by the duplicate-elimination operator \( \text{SELECT DISTINCT} \) introduced in Section 6.1.6.

In this chapter we shall offer algorithms for both the set and bag versions of the union, intersection, and difference operators. To avoid confusion, we distinguish these two kinds of operators by subscript \( S \) or \( B \), for "set" or "bag," respectively. Thus, for example, \( U \) is set-union, and \( B \) is bag-difference. An unsubscripted operator is the bag version by default. As an exception, when we write algebraic laws (see Section 7.2), the intent is that the law holds for both versions of the operator if there is no subscript.

### 6.1.2 The Selection Operator

The selection \( \sigma_C(R) \) takes a relation \( R \) and a condition \( C \). The condition \( C \) may involve:

1. Arithmetic (e.g., \( + \), \( \times \) or string operators (e.g., concatenation or LIKE on constants and/or attributes),
2. Comparisons between terms constructed in (1), e.g., \( a < b \) or \( a + b = 10 \) and
6.1. AN ALGEBRA FOR QUERIES

Subqueries in WHERE Clauses

The \( \text{a} \) operator introduced here is more powerful than the conventional selection operator of relational algebra, since we allow logical operators AND, OR, and NOT in the subscript of the \( \sigma \). However, even this \( \text{a} \) operator is not as powerful as a WHERE clause in SQL, because there we can have subqueries and certain logical operators on relations, such as EXISTS. A condition involving a subquery must be expressed by an operator that works on whole relations, while the subscript of the \( \text{a} \) operator is intended to be a test applied to individual tuples.

In relational algebra, all relations involved in an operation are explicit arguments of the operator, not parameters appearing in a subscript. Thus, it is necessary in relational algebra to handle subqueries by operators such as \( \text{ix} \) (join), in which the relation of the subquery and relation of the outer query are connected. We defer the matter to Section 7.1. We also discuss in Section 7.3.2 a variant of the selection operator that allows subqueries as explicit arguments.

3. Boolean connectives AND, OR, and NOT applied to the terms constructed in (2).

Selection \( \sigma_{C}(R) \) produces the bag of those tuples in \( R \) that satisfy the condition \( C \). The schema of the resulting relation is the same as the schema of \( R \).

**Example 6.2**: Let \( R(a, b) \) be the relation

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

The result of \( \sigma_{a \geq 1}(R) \) is

<table>
<thead>
<tr>
<th>( a )</th>
<th>( b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Notice that a tuple meeting the condition appears as many times in the output as it does in the input, while a tuple such as \((0, 1)\) that does not meet the condition does not appear at all.

The result of \( \sigma_{b > 3 \text{ AND } a+b > 6}(R) \) is
CHAPTER 6. QUERY EXECUTION

6.1.3 The Projection Operator

If \( R \) is a relation, then \( \pi_L(R) \) is the projection of \( R \) onto the list \( L \). In the classical relational algebra, \( L \) is a list of (some of the) attributes of \( R \). We extend the projection operator to make it resemble the SELECT clause in SQL. Our projection lists can have the following kinds of elements:

1. A single attribute of \( R \).

2. An expression \( x \rightarrow y \), where \( x \) and \( y \) are names for attributes. The element \( x \rightarrow y \) in the list \( L \) asks that we take the attribute \( x \) of \( R \) and rename it \( y \); i.e., the name of this attribute in the schema of the result relation is \( y \).

3. An expression \( E \rightarrow z \), where \( E \) is an expression involving attributes of \( R \), constants, arithmetic operators, and string operators, and \( z \) is a new name for the attribute that results from the calculation implied by \( E \). For example, \( a + b \rightarrow x \) as a list element represents the sum of the attributes \( a \) and \( b \), renamed \( x \). Element \( c \& d \rightarrow e \) means concatenate the (presumably string-valued) attributes \( c \) and \( d \) and call the result \( e \).

The result of the projection is computed by considering each tuple of \( R \) in turn. We evaluate the list \( L \) by substituting the tuple’s components for the corresponding attributes mentioned in \( L \) and applying any operators indicated by \( L \) to these values. The result is a relation whose schema is the names of the attributes on list \( L \), with whatever renaming the list specifies. Each tuple of \( R \) yields one tuple of the result. Duplicate tuples in \( R \) surely yield duplicate tuples in the result, but the result can have duplicates even if \( R \) does not.

Example 6.3: Let \( R \) be the relation

\[
\begin{array}{ccc}
  a & b & c \\
  0 & 1 & 2 \\
  0 & 1 & 2 \\
  3 & 4 & 5 \\
\end{array}
\]

Then the result of \( \pi_{a,b+c\rightarrow x}(R) \) is

\[
\begin{array}{c}
  a & x \\
  0 & 3 \\
  0 & 3 \\
  3 & 9 \\
\end{array}
\]
6.1. AN ALGEBRA FOR QUERIES

The result’s schema has two attributes. One is \( a \), the first attribute of \( R \), not renamed. The second is the sum of the second and third attributes of \( R \), with the name \( x \).

For another example, \( \pi_{b \rightarrow a \rightarrow x, c \rightarrow y}(R) \) is

<table>
<thead>
<tr>
<th></th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Notice that the calculation required by this projection list happens to turn different tuples \((0, 1, 2)\) and \((3, 4, 5)\) into the same tuple \((1, 1)\). Thus, the latter tuple appears three times in the result.

6.1.4 The Product of Relations

If \( R \) and \( S \) are relations, the product \( R \times S \) is a relation whose schema consists of the attributes of \( R \) and the attributes of \( S \). Should there be an attribute name, say \( a \), found in both schemas, then we use \( R.a \) and \( S.a \) as the names of the two attributes in the product schema.

The tuples of the product are all those that can be formed by taking a tuple of \( R \) and following its components by the components of any one tuple of \( S \). If a tuple \( r \) appears \( n \) times in \( R \), and a tuple \( s \) appears \( m \) times in \( S \), then in the product, the tuple \( rs \) appears \( nm \) times.

Example 6.4: Let \( R(a, b) \) be the relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

and let relation \( S(b, c) \) be

<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Then \( R \times S \) is the relation shown in Fig. 6.3. Note that each tuple of \( R \) is paired with each tuple of \( S \), regardless of duplication. Thus, for example, tuple \((2, 3, 1, 4)\) appears four times, since it is composed of duplicated tuples from both \( R \) and \( S \). Also notice that the schema of the product has two attributes called \( R.b \) and \( 5.6 \) corresponding to the attributes called \( b \) from the two relations.
6.1.5 Joins

There are a number of useful "join" operators that are built from a product followed by a selection and projection. These operators are found explicitly in the SQL2 standard, as ways to combine relations in a FROM clause. However, joins also represent the effect of many common SQL queries whose FROM clause is a list of two or more relations and whose WHERE clause applies equalities or other comparisons to some attributes of these relations.

The simplest and most common is the natural join; we denote the natural join of relations $R$ and $S$ by $R \bowtie S$. This expression is a shorthand for $\pi_L(\sigma_C(R \times S))$, where:

1. $C$ is a condition that equates all pairs of attributes of $R$ and $S$ that have the same name.

2. $L$ is a list of all the attributes of $R$ and $S$, except that one copy of each pair of equated attributes is omitted. If $R.x$ and $S.x$ are equated attributes, then since in the result of the projection there is only one attribute $x$, we conventionally rename as $x$ whichever of $R.x$ and $S.x$ is chosen.

Example 6.5: If $R(a, b)$ and $S(b, c)$ are the relations introduced in Example 6.4, then $R \bowtie S$ stands for $\pi_{a, b \rightarrow b, c}(\sigma_{R.b=S.b}(R \times S))$. That is, since $b$ is the only common attribute name of $R$ and $S$, the selection equates these two attributes named $b$ and no other pair of attributes. The projection list consists of one copy of each attribute name. We have chosen to get $b$ from $R.b$ and renamed it $6$; we could have chosen $S.b$ had we wished.

The result of a natural join could be computed by applying the three operators $\times$, $\sigma$, and $\pi$, in turn. However, it is easier to compute the natural join "all at once." Several methods exist for finding those pairs of tuples, one from $R$ and one from $S$, that agree on all pairs of attributes with the same name. For each such pair of tuples, we produce the result tuple that agrees with these
tuples on all attributes. For instance, using the relations R and S from Example 6.4, we find that the only pairs of tuples that match on their \( b \)-values are (0,1) from R and the two (1,4) tuples from S. The result of \( R \bowtie S \) is thus:

\[
\begin{array}{ccc}
  a & b & c \\
  0 & 1 & 4 \\
  0 & 1 & 4 \\
\end{array}
\]

Note that there are two pairs of joining tuples; they happen to have identical tuples from S, which is why the tuple (0,1,4) appears twice in the result. D

Another form of join is the theta-join. If \( R \) and S are relations, then \( R \bowtie C S \) is shorthand for \( \pi_C(R \times S) \). If the condition \( C \) is a single term of the form \( x = y \), where \( x \) is an attribute of \( R \) and \( y \) is an attribute of \( S \), then we call the join an equijoin. Notice that, unlike the natural join, an equijoin does not involve projecting out any attributes, even if the result has two or more identical columns.

**Example 6.6**: Let \( R(a, b) \) and \( S(b, c) \) be the relations from Examples 6.4 and 6.5. Then \( R \bowtie R S(a + R b < c + S b) \) is the same as \( \pi_{a+R b, c+S b}(R \times S) \). That is, the joining condition requires the sum of the components from the tuple of \( R \) to be less than the sum of the components of the tuple from \( S \). The result has all the tuples of \( R \times S \), except those where the \( R \)-tuple is (2,3) and the \( S \)-tuple is (1,4), because here the sum from \( R \) is not less than the sum from \( S \). The result relation is shown in Fig. 6.4.

\[
\begin{array}{ccc}
  a & R, b & S, b & c \\
  0 & 1 & 1 & 4 \\
  0 & 1 & 1 & 4 \\
  0 & 1 & 2 & 5 \\
  2 & 3 & 2 & 5 \\
  2 & 3 & 2 & 5 \\
\end{array}
\]

**Figure 6.4**: The result of a theta-join

For another example, consider the equijoin \( R \bowtie S \). By convention, the first of the equated attributes in a theta-join comes from the left argument and the second comes from the right argument, so this expression is the same as \( R \bowtie b = c \ S \). The result is the same as that of \( R \bowtie S \), except the two equated \( b \) attributes remain. That is,

\[
\begin{array}{ccc}
  a & R, b & S, b & c \\
  0 & 1 & 1 & 4 \\
  0 & 1 & 1 & 4 \\
\end{array}
\]

is the result of this equijoin. D
The Meaning of “Theta-Join”

Historically, all joins involved a simple condition that compared two attributes, one from each of the two argument relations. The generic form of such a join was written $R \bowtie S$, where $x$ and $y$ are attributes from $R$ and $S$, respectively, and $\theta$ stands for any of the six arithmetic comparisons: =, \neq, \leq, <, \geq, >, \text{ or } >$. Since the comparison was represented by symbol $\theta$, the operation came to be known as a “theta-join.” Today, we retain the terminology, although the condition of our theta-join is no longer restricted to be a simple comparison between attributes; it can be any condition that could appear in a selection. Nevertheless, the theta-join that compares two attributes, especially the equijoin version that equates two attributes, is undoubtedly the predominant form of join in practice.

6.1.6 Duplicate Elimination

We need an operator that converts a bag to a set, corresponding to the keyword DISTINCT in SQL. For that purpose, we use $\delta(R)$ to return the set consisting of one copy of every tuple that appears one or more times in relation $R$.

Example 6.7: If $R$ is the relation of that name from Example 6.4, then $\delta(R)$ is

$$
\begin{array}{c|c}
  a & b \\
  \hline
  0 & 1 \\
  2 & 3 \\
\end{array}
$$

Note that the tuple (2, 3), which appeared twice in $R$, appeals only once in $\delta(R)$.

Recall that the UNION, INTERSECT, and EXCEPT operators of SQL eliminate duplicates by default, but we have defined the operators $U$, $\cap$, and $-$ to follow the bag definitions of these operators by default. Thus, if we want to translate an SQL expression like $R \cup S$ to our algebra, we have to write $S(R \cup S')$.

6.1.7 Grouping and Aggregation

There is a family of features in SQL that work together to allow queries involving "grouping and aggregation":

1. Aggregation operators. The five operators $\text{AVG}$, $\text{SUM}$, $\text{COUNT}$, $\text{MIN}$, and $\text{MAX}$ produce the average, sum, count, minimum, or maximum values, respectively, of the attribute to which they are applied. These operators appear in SELECT clauses.
2. **Grouping.** A GROUP BY clause in an SQL query causes the relation constructed by the FROM and WHERE clauses to be grouped according to the value of the attribute or attributes mentioned in the GROUP BY clause. Aggregations are then applied on a per-group basis.

3. **“Having”**. A HAVING clause must follow a GROUP BY clause and provides a condition (which may involve aggregations and the attributes in the group-by) that a group must satisfy to contribute to the result of the query.

Grouping and aggregation generally need to be implemented and optimized together. We shall thus introduce into our extended relational algebra a single operator, \( \gamma \), that represents the effect of grouping and aggregation. The \( \gamma \) operator also helps implement a HAVING clause, which is represented by following the \( \gamma \) with selection and projection.

The subscript used with the \( \gamma \) operator is a list \( L \) of elements, each of which is either:

a) An attribute of the relation to which the \( \gamma \) is applied; this attribute is one of the attributes of the GROUP BY list of the query. This element is said to be a **grouping** attribute.

b) An aggregation operator applied to an attribute of the relation. To provide a name for the attribute corresponding to this aggregation in the result, an arrow and new name are appended to the aggregation. This element represents one of the aggregations in the SELECT clause of the query. The underlying attribute is said to be an **aggregated** attribute.

The relation returned by the expression \( \gamma_L(R) \) is constructed as follows:

1. Partition the tuples of \( R \) into groups. Each group consists of all tuples having one particular assignment of values to the grouping attributes in the list \( L \). If there are no grouping attributes, the entire relation \( R \) is one group.

2. For each group, produce one tuple consisting of:
   i. The grouping attributes' values for that group and
   ii. The aggregations, over all tuples of that group, that are specified by the aggregated attributes on list \( L \).

**Example 6.8**: Suppose we have the relation

\[
\text{StarsIn(title, year, starName)}
\]

and we wish to find, for each star who has appeared in at least three movies, the earliest year in which they appeared. The following SQL query does the job:
5 is a Special Case of $\gamma$

Technically, the $\delta$ operator is redundant. If $R(a_1, a_2, \ldots, a_n)$ is a relation, then $\delta(R)$ is equivalent to $\gamma_{a_1, a_2, \ldots, a_n}(R)$. That is, to eliminate duplicates, we group on all the attributes of the relation and do no aggregation. Then each group corresponds to a tuple that is found one or more times in $R$. Since the result of $\gamma$ contains exactly one tuple from each group, the effect of this "grouping" is to eliminate duplicates. However, because $\delta$ is such a common and important operator, we shall continue to consider it separately when we study algebraic laws and algorithms for implementing the operators.

```
SELECT starName, MIN(year) AS minYear
FROM StarsIn
GROUP BY starName
HAVING COUNT(title) >= 3;
```

The equivalent algebraic expression will involve grouping, using starName as a grouping attribute. We clearly must compute for each group the MIN(year) aggregate. However, in order to decide which groups satisfy the HAVING clause, we must also compute the COUNT(title) aggregate for each group.

We begin with the grouping expression

```
\gamma_{starName, MIN(year)} \rightarrow \text{minYear}, COUNT(title) \rightarrow \text{ctTitle}(\text{StarsIn})
```

The first two columns of the result of this expression are needed for the query result. The third column is an auxiliary attribute, which we have named ctTitle; it is needed to apply to each tuple the test of the HAVING clause. That is, we continue the algebraic expression for the query by selecting for $\text{ctTitle} \geq 3$ and then projecting onto the first two columns. A representation for the query is shown in Fig. 6.5; it is a simple form of an expression tree (see Section 6.1.9), where four operators in cascade are shown, each above the previous operator.

Even without a HAVING clause, there are SQL group-by queries that cannot be expressed entirely as a single $\gamma$ operation. For example, the FROM clause may contain several relations, and these must first be combined with a product operator. If the query has a WHERE clause, then the condition of this clause must be expressed with a $\sigma$ operation or possibly by turning the product of relations into a join. Additionally, there is the possibility that an attribute mentioned in the GROUP BY clause does not appear in the SELECT list. For instance, we could have omitted starName from the SELECT clause of Example 6.8, although the effect would have been strange: we'd get a list of years without an indication of which year corresponded to which star. In this situation, we mention all
attributes of the GROUP BY in the list of the 7 and follow that operator by a projection that removes the grouping attributes that do not appear in the SELECT clause.

6.1.8 The Sorting Operator

We shall use the operator \( r \) to sort a relation. This operator can be used to implement an SQL ORDER BY clause. Sorting also plays a role as a physical-query-plan operator, since many of the other operators of relational algebra can be facilitated by first sorting one or more of the argument relations.

To be precise, the expression \( r_L(R) \), where \( R \) is a relation and \( L \) a list of some of \( R \)'s attributes, is the relation \( R \), but with the tuples of \( R \) sorted in the order indicated by \( L \). If \( L \) is the list \( a_1, a_2, \ldots, a_n \), then the tuples of \( R \) are sorted first by their value of attribute \( a_1 \). Ties are broken according to the value of \( a_2 \); tuples that agree on both \( a_1 \) and \( a_2 \) are ordered according to their value of \( a_3 \), and so on. Ties that remain after attribute \( a_n \) is considered may be ordered arbitrarily. As with SQL, we shall assume that by default sorting is in ascending order but can be changed to descending order by following the attribute by DESC.

**Example 6.9:** If \( R \) is a relation with schema \( R(a, b, r) \), then \( r_{<,a}(R) \) orders the tuples of \( R \) by their value of \( c \), and tuples with the same \( c \)-value are ordered by their \( b \)-value. Tuples that agree on both \( b \) and \( c \) may be ordered arbitrarily.

The operator \( r \) is anomalous, in that it is the only operator in our relational algebra whose result is a list of tuples, rather than a set. Thus, it only makes sense to talk about \( T \) as the final operator in an algebraic expression. If another operator of relational algebra is applied after \( T \), the result of the \( T \) is treated as a set or bag, and no ordering of the tuples is implied. However, we shall often use \( T \) in physical query plans, whose operators are not the same as the
relational-algebra operators. Many of the latter operators take advantage of an argument or arguments that are sorted and may themselves produce sorted results.

6.1.9 Expression Trees

We can combine several relational-algebra operators into one expression by applying one operator to the result(s) of one or more other operators. Thus, as for any algebra, we can picture the application of several operators as an expression tree. The leaves of this tree are names of relations, and the interior nodes are each labeled with an operator that makes sense when applied to the relation(s) represented by its child or children. Figure 6.5 was an example of a simple expression tree representing the successive application of three unary operators. However, many expression trees involve binary operators and have several branches.

Example 6.10: Suppose we have available the relations

\[
\text{MovieStar}(\text{name, addr, gender, birthdate}) \\
\text{StarsIn}(\text{title, year, starName})
\]

from which we ask for the birthdate and movie title for those female stars who appeared in movies in 1996:

\[
\text{SELECT title, birthdate} \\
\text{FROM MovieStar, StarsIn} \\
\text{WHERE year = 1996 AND} \\
\text{gender = 'F' AND} \\
\text{starName = name;}
\]

That is, we join the two relations MovieStar and StarsIn, using the condition that the name of the star in both relations is the same, and we select for the year of the movie being 1996 and the gender of the star being female.

A simple SQL query such as the one above will be translated by the parser (as in Fig. 6.2) into a logical query plan whose first step is to combine the relations in the FROM list using product operators. The next step is to perform a selection that represents the WHERE clause, and the last step is to project onto the list in the SELECT clause. The algebraic expression for the above query is shown as a tree in Fig. 6.6.

There are many other expressions equivalent to that of Fig. 6.6, in the sense that whatever instances of relations MovieStar and StarsIn one takes, the results of the two expressions are the same. We give one example of an equivalent expression in Fig. 6.7. This expression uses a significantly different plan from Fig. 6.6. First, we recognize that the condition starName = name in the WHERE clause, applied to the product of the two relations, is the same as an equijoin. In Fig. 6.7 this combination of a selection and a product into a join has been
performed. Generally, joins produce fewer tuples than products, so joins are preferable to products when we have a choice.

Second, the other two conditions of the WHERE clause have been separated into two \( \sigma \) operations, and these operations have each been “pushed” down the tree to the appropriate relation. For instance, the selection \( \sigma_{\text{year}=1996} \) applies directly to the relation \( \$\text{StarsIn} \), because that relation is the one that brought attribute year to the product in Fig. 6.6. There is a general rule that (usually) it makes sense to do selections as soon as possible. Since products and joins are typically more time-consuming than selections, cutting down the sizes of the relations quickly, by performing selections as far down the tree as we can as in Fig. 6.7, will tend to reduce the join time more than it increases the time required for selection. The general topic of improving logical query plans is taken up again in Section 7.2.
6.1.10 Exercises for Section 6.1

Exercise 6.1.1: Here are two relations:

\[ R(a, b): \{(0, 1), (2, 3), (0, 1), (2, 4), (3, 4)\} \]
\[ S(a, b): \{(0, 1), (2, 4), (2, 5), (3, 4), (0, 2), (3, 4)\} \]

Compute the following:

* a) \( R \cup_S S \).
  
b) \( R \cup_B S \).
  
c) \( R \cap_S S \).
  
d) \( R \cap_B S \).
  
e) \( R - S \).
  
f) \( R - B \).
  
g) \( S - S \).
  
h) \( S - B \).

* i) \( \pi_{a+b, a^2, b^2}(R) \).
  
j) \( \pi_{a+1, a-1}(S) \).

* k) \( \sigma_{a<b \land \text{AND} \ (a+b)>a \land b \land a+b<6}(R) \).
  
l) \( \sigma_{a<b \land \text{AND} \ (a+b)>a \land b \land a+b<6}(S) \).

m) \( \sigma_{a>1 \lor b>4 \lor b=2}(R) \).

\( \sigma_{a>1 \lor b>4 \lor b=2}(S) \).

Exercise 6.1.2: Here are three relations:

\[ R(a, b): \{(0, 1), (2, 3), (0, 1), (2, 4), (3, 4)\} \]
\[ S(b, c): \{(1, 2), (1, 2), (2, 5), (3, 5), (4, 5)\} \]
\[ T(c, d): \{(2, 3), (3, 4), (4, 5), (5, 6)\} \]

Compute the following:

* a) \( R \bowtie S \).
  
b) \( S \bowtie T \).
  
c) \( R \bowtie T \).
6.1. AN ALGEBRA FOR QUERIES

Join-Like Operators

There are several operators that are commonly viewed as varieties of join. Here are their symbols and definitions:

1. The semijoin of relations $R$ and $S$, written $R \bowtie S$, is the bag of tuples $t$ in $R$ such that there is at least one tuple in $S$ that agrees with $t$ in all attributes that $R$ and $S$ have in common.

2. The antisemijoin $R \bar{\bowtie} S$ is the bag of tuples $t$ in $R$ that do not agree with any tuple of $S$ in the attributes common to $R$ and $S$.

3. The outerjoin $R \bowtie S$ is formed by starting with $R \bowtie S$, and adding any dangling tuples from $R$ or $S$ (a tuple is dangling if it did not join with any tuple from the other relation). The added tuples must be padded with a special null symbol (NULL in SQL, but $\bot$ will serve in these exercises) for all the attributes that they do not possess, but that appear in the join result.

4. The left outerjoin $R \bowtie L S$ is like the outerjoin, but only dangling tuples of the left argument $R$ are padded with $\bot$ and added to the result.

5. The right outerjoin $R \bowtie R S$ is like the outerjoin, but only the dangling tuples of the right argument $S$ are padded with $\bot$ and added to the result.

\[ d) \quad R \bowtie \overline{R \bowtie S, b} S. \]

* e) \[ R \bowtie R a \neq a S. \]

f) \[ R \bowtie R b \neq c S. \]

* g) \[ \gamma_{a, SUM(b)}(R). \]

h) \[ \gamma_{c, MIN(b)}(S). \]

i) \[ \delta(R). \]

j) \[ \tau_{b,c}(R). \]

Exercise 6.1.3: Give expressions for the five operators defined in the box "Join-Like Operators" using only the standard operators of relational algebra defined in this section. For the outerjoin variants, you may use special "null relations" $N(a_1, a_2, \ldots, a_k)$ that consist of one tuple, with $\bot$ in each component.
* a) Semijoin.
   b) Antisemijoin.
* c) Left outerjoin.
   d) Right outerjoin.
  e) Outerjoin.

**Exercise 6.1.4:** Write the following joins as expressions involving selection, projection, and product.

a) \( R(a, b, c, d) \bowtie S(b, d, e) \).

b) \( R(a, b, c) \mid a + d = 10 \text{ OR } b = S(c, d) \).

**Exercise 6.1.5:** A unary operator / is said to be **idempotent** if for all relations \( R, f(f(R)) = f(R) \). That is, applying / more than once is the same as applying it once. Which of the following operators are idempotent? Either explain why or give a counterexample.

* a) \( \delta \).
* b) \( \pi_L \).
   
   c) \( \sigma_C \).
   
   d) \( \gamma_L \).
   
   e) \( \tau \).

**Exercise 6.1.6:** Using the following "movie" relations:

- Movie(title, year, length, studioName)
- MovieStar(name, address, gender, birthdate)
- StarsIn(title, year, starName)
- Studio(name, address)

turn the following queries into expression trees using the algebraic operators of this section.

a) SELECT address
   FROM Movie, Studio
   WHERE studioName = name AND title = 'Gone With the Wind';

b) (SELECT name FROM MovieStar)
   UNION
   (SELECT starName FROM StarsIn);
6.2. INTRODUCTION TO PHYSICAL-QUERY-PLAN OPERATORS

6.2.1 Scanning Tables

Perhaps the most basic thing we can do in a physical query plan is to read the entire contents of a relation \( R \). This step is necessary when, for example, we take the union or join of \( R \) with another relation. A variation of this operator involves a simple predicate, where we read only those tuples of the relation \( R \) that satisfy the predicate. There are two basic approaches to locating the tuples of a relation \( R \):

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

2. If there is an index on any attribute of \( R \), we may be able to use this index to get all the tuples of \( R \). For example, a sparse index on \( R \), as discussed in Section 4.1.3, can be used to lead us to all the blocks holding \( R \), even if we don’t know otherwise which blocks these are. This operation is called index-scan.
We shall take up index-scan again in Section 6.7.2 when we talk about implementation of the $a$ operator. However, the important observation for now is that we can use the index not only to get all the tuples of the relation it indexes, but to get only those tuples that have a particular value (or sometimes a particular range of values) in the attribute or attributes that form the search key for the index.

6.2.2 Sorting While Scanning Tables

There are a number of reasons why we might want to sort a relation as we read its tuples. For one, the query could include an ORDER BY clause, requiring that a relation be sorted. For another, various algorithms for relational-algebra operations require one or both of their arguments to be sorted relations. These algorithms appear in Section 6.5 and elsewhere.

The physical-query-plan operator \textit{sort-scan} takes a relation $R$ and a specification of the attributes on which the sort is to be made, and produces $R$ in that sorted order. There are several ways that sort-scan can be implemented:

a) If we are to produce a relation $R$ sorted by attribute $a$, and there is a B-tree index on $a$, or $R$ is stored as an indexed-sequential file ordered by $a$, then a scan of the index allows us to produce $R$ in the desired order.

b) If the relation $R$ that we wish to retrieve in sorted order is small enough to fit in main memory, then we can retrieve its tuples using a table scan or index scan, and then use one of many possible efficient, main-memory sorting algorithms. Main-memory sorting is covered in many books on the subject, and we shall not consider the matter here.

c) If $R$ is too large to fit in main memory, then the multiway merging approach covered in Section 2.3.3 is a good choice. However, instead of storing the final sorted $R$ back on disk, we produce one block of the sorted $R$ at a time, as its tuples are needed.

6.2.3 The Model of Computation for Physical Operators

A query generally consists of several operations of relational algebra, and the corresponding physical query plan is composed of several physical operators. Often, a physical operator is an implementation of a relational-algebra operator, but as we saw in Section 6.2.1, other physical plan operators correspond to operations like scanning that may be invisible in relational algebra.

Since choosing physical plan operators wisely is an essential of a good query processor, we must be able to estimate the "cost" of each operator we use. We shall use the number of disk I/O's as our measure of cost for an operation. This measure is consistent with our view (see Section 2.3.1) that it takes longer to get data from disk than to do anything useful with it once the data is in main memory. The one major exception is when answering a query involves
communicating data across a network. We discuss costs for distributed query processing in Sections 6.10 and 10.4.4.

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

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

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

Similarly, the result of an operator that forms part of a query (rather than the whole query) often is not written to disk. In Section 7.7.3 we shall discuss "pipelining," where the result of one operator is constructed in main memory, perhaps a small piece at a time, and passed as an argument to another operator. In this situation, we never have to write the result to disk, and moreover, we save the cost of reading from disk this argument of the operator that uses the result. This saving is an excellent opportunity for the query optimizer.

6.2.4 Parameters for Measuring Costs

Now, let us introduce the parameters that we use to express the cost of an operator. Estimates of cost are essential if the optimizer is to determine which of the many query plans is likely to execute fastest. Section 7.5 introduces the exploitation of these cost estimates.

We need a parameter to represent the portion of main memory that the operator uses, and we require other parameters to measure the size of its arguments. Assume that main memory is divided into buffers, whose size is the same as the size of disk blocks. Then $M$ will denote the number of main-memory buffers available to an execution of a particular operator. Remember that when evaluating the cost of an operator, we do not count the cost — either memory used or disk I/O’s — of producing the output; thus $M$ includes only the space used to hold the input and any intermediate results of the operator.

Sometimes, we can think of $M$ as the entire main memory, or most of the main memory, as we did in Section 2.3.4. However, we shall also see situations where several operations share the main memory, so $M$ could be much smaller than the total main memory. In fact, as we shall discuss in Section 6.8, the number of buffers available to an operation may not be a predictable constant, but may be decided during execution, based on what other processes are executing at the same time. If so, $M$ is really an estimate of the number of buffers available to the operation. If the estimate is wrong, then the actual execution time will differ from the predicted time used by the optimizer. We could
even find that the chosen physical query plan would have been different, had
the query optimizer known what the true buffer availability would be during
execution.

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

We shall make the simplifying assumption that data is accessed one block
at a time from disk. In practice, one of the techniques discussed in Section 2.4
might be able to speed up the algorithm if we are able to read many blocks of
the relation at once, and they can be read from consecutive blocks on a track.

There are three parameter families, $B$, $T$, and $V$:

- When describing the size of a relation $R$, we most often are concerned with
  the number of blocks that are needed to hold all the tuples of $R$. This
  number of blocks will be denoted $B(R)$, or just $B$ if we know that relation
  $R$ is meant. Usually, we assume that $R$ is clustered; that is, it is stored in
  $B$ blocks or in approximately $B$ blocks. As discussed in Section 4.1.6, we
  may in fact wish to keep a small fraction of each block holding $R$ empty
  for future insertions into $R$. Nevertheless, $B$ will often be a good-enough
  approximation to the number of blocks that we must read from disk to
  see all of $R$, and we shall use $B$ as that estimate uniformly.

- Sometimes, we also need to know the number of tuples in $R$, and we
denote this quantity by $T(R)$, or just $T$ if $R$ is understood. If we need the
  number of tuples of $R$ that can fit in one block, we can use the ratio $T/B$.
  Further, there are some instances where a relation is stored distributed
  among blocks that are also occupied by tuples of other relations. If so,
  then a simplifying assumption is that each tuple of $R$ requires a separate
disk read, and we shall use $T$ as an estimate of the disk I/O's needed to
  read $R$ in this situation.

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

### 6.2.5 I/O Cost for Scan Operators

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

If $R$ is clustered but requires a two-phase multiway merge sort, then, as discussed in Section 2.3.4, we require about $3B$ disk I/O's, divided equally among the operations of reading $R$ in sublists, writing out the sublists, and rereading the sublists. Remember that we do not charge for the final writing of the result. Neither do we charge memory space for accumulated output. Rather, we assume each output block is immediately consumed by some other operation; possibly it is simply written to disk.

However, if $R$ is not clustered, then the number of required disk I/O's is generally much higher. If $R$ is distributed among tuples of other relations, then a table-scan for $R$ may require reading as many blocks as there are tuples of $R$; that is, the I/O cost is $T$. Similarly, if we want to sort $R$, but $R$ fits in memory, then $T$ disk I/O's are what we need to get all of $R$ into memory. Finally, if $R$ is not clustered and requires a two-phase sort, then it takes $T$ disk I/O's to read the subgroups initially. However, we may store and reread the sublists in clustered form, so these steps require only $2B$ disk I/O's. The total cost for performing sort-scan on a large, unclustered relation is thus $T + 2B$.

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

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

However, if we only want part of $R$, we often are able to avoid looking at the entire index and the entire $R$. We shall defer analysis of these uses of indexes to Section 6.7.2.

### 6.2.6 Iterators for Implementation of Physical Operators

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

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

2. **GetNext.** This function returns the next tuple in the result and adjusts data structures as necessary to allow subsequent tuples to be obtained. In getting the next tuple of its result, it typically calls GetNext one or more times on its `argument(s)`. This function also sets a signal that tells
CHAPTER 6. QUERY EXECUTION

Why Iterators?

We shall see in Section 7.7 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 function, which is tantamount to materialization.

whether a tuple was produced, or whether there were no more tuples to be produced. We shall use Found as a boolean variable that is true if and only if a new tuple has been returned.

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

When describing iterators and their functions, we shall regard Open, GetNext, and Close as overloaded names of methods. That is, these methods have many different implementations, depending on the "class" to which the method is applied. In particular, assume that for each physical operator there is a class whose objects are the relations that can be produced by this operator. If $R$ is a member of such a class, then we use $R$.Open(), $R$.GetNext(), and $R$.Close() to apply the functions of the iterator for $R$.

Example 6.11: Perhaps the simplest iterator is the one that implements the table-scan operator. Let us suppose that we want to perform TableScan($R$), where $R$ is a relation clustered in some list of blocks, which we can access in a convenient way. Thus, we shall assume that the notion of "get the next block of $R$" is implemented by the storage system and need not be described in detail. Further, we assume that within a block there is a directory of records (tuples) so that it is easy to get the next tuple of a block or tell that the last tuple has been reached.

Figure 6.8 sketches the three functions for this iterator. We imagine a block pointer $b$ and a tuple pointer $t$ that points to a tuple within block 6. We assume that both pointers can point "beyond" the last block or last tuple of a block, respectively, and that it is possible to identify when these conditions occur. Notice that Close in this case does nothing. In practice, a Close function for an iterator might clean up the internal structure of the DBMS in various ways. It might inform the buffer manager that certain buffers are no longer needed.
INTRODUCTION TO PHYSICAL-QUERY-PLAN OPERATORS

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

GetNext(R) {
    IF (t is past the last tuple on block b) {
        increment b to the next block;
        IF (there is no next block) {
            Found := FALSE;
            RETURN;
        }
        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(R) {
}

Figure 6.8: An iterator for the table-scan operator

or inform the concurrency manager that the read of a relation has completed.

Example 6.12: Now, let us consider an example where the iterator does most of the work in its Open function. The operator is sort-scan, where we read the tuples of a relation R but return them in sorted order. Further, let us suppose that R is so large that we need to use a two-phase, multiway merge-sort, as in Section 2.3.4.

We cannot return even the first tuple until we have examined each tuple of R. Thus, Open must do at least the following:

1. Read all the tuples of R in main-memory-sized chunks, sort them, and store them on disk.

2. Initialize the data structure for the second (merge) phase, and load the first block of each sublist into the main-memory structure.

Then, GetNext can run a competition for the first remaining tuple at the heads of all the sublists. If the block from the winning sublist is exhausted, GetNext
Example 6.13: Finally, let us consider a simple example of how iterators can be combined by calling other iterators. It is not a good example of how many iterators can be active simultaneously, but that will have to wait until we have considered algorithms for physical operators like selection and join, which exploit this capability of iterators better.

Our 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. We assume that there are functions \( R\text{-Open}, R\text{-GetNext}, \) and \( R\text{-Close} \), that form the iterator for \( R \), and analogous functions for relation \( S \). These functions could be the functions for table-scan applied to \( R \) and \( S \), if these are stored relations, or they could be iterators that call a network of other iterators to compute \( R \) and \( S \). The iterator functions for this union are sketched in Fig. 6.9. One subtle point is that the functions use a shared variable \( \text{CurRel} \) that is either \( R \) or \( S \), depending on which relation is being read from currently.

### 6.3 One-Pass Algorithms for Database Operations

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 an 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. These are covered primarily in Section 6.5.
2. Hash-based methods. These are mentioned in Section 6.6 and Section 6.10, among other places.
3. Index-based methods. These are emphasized in Section 6.7.

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 work only when at least one of the arguments of the operation fits in main memory, although there are exceptions, especially for selection and projection as discussed in Section 6.3.1.

b) Some methods work for data that is too large to fit in available main memory but not for the largest imaginable data sets. An example of
Open(R,S) {
    R.Open();
    CurRel := R;
    >
    GetNext(R,S) {
        IF (CurRel = R) {
            t := R.GetNext();
            IF (Found) /* 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, Found will be set to FALSE by S.GetNext, which is the correct action for GetNext as well */
    }
    Close(R,S) {
        R.Close();
        S.Close();
    }
}

Figure 6.9: Building a union iterator from its components

such an algorithm is the two-phase, multiway merge sort of Section 2.3.4. 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 6.5 and 6.6.

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 6.9.

In this section, we shall concentrate on the one-pass methods. However, both in this section and subsequently, we shall classify operators into three broad groups:
1. **Tuple-at-a-time, unary operations.** Those 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 $M$ or less. The operations of this class that we consider here are 7 and 6.

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

### 6.3.1 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. 6.10. 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 $M > f$ for the input buffer, regardless of $B$.

![Figure 6.10: A selection or projection being performed on a relation R](image)

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 6.2.5; typically it is $B$ if $R$ is clustered and $T$ if it is not clustered. However, we should remind the reader again of the important exception when
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. 6.10, we can often speed up processing if we allocate more input buffers. The idea appeared first in Section 2.4.1. 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.

the operation being performed is a selection, and the condition compares a constant to an 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.

6.3.2 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 (6) and grouping (7).

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. 6.11. 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 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 was significant. Thus, we need a main-memory structure that allows each of the operations:

1. Add a new tuple, and
2. Tell whether a given tuple is already there
to be done in time that is close to a constant, independent of the number of tuples \( n \) that we currently have in memory. There are many such structures known. For example, we could use a hash table with a large number of buckets, or some form of balanced binary search tree. Each of these structures requires some 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 extra space needed tends to be small compared with the space needed to store the tuples. We shall thus make the simplifying assumption that no extra space is needed and concentrate on the space needed to store the tuples in main memory.

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 \( B(\delta(R)) \) 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:

\[ O(n) \]

---

1See Aho, A. V., J. E. Hopcroft, and J. D. Ullman *Data Structures and Algorithms*, Addison-Wesley, 1984 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.
Note that we cannot in general compute the size of $\delta(R)$ without computing $S(R)$ itself. Should we underestimate that size, so $B(\delta(R))$ 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.

**Grouping**

A grouping operation $\gamma_L$ 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. The accumulated value is, except in one case, obvious:

- For a $\text{MIN}(a)$ or $\text{MAX}(a)$ 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 $\text{COUNT}$ aggregation, add one for each tuple of the group that is seen.
- For $\text{SUM}(a)$, add the value of attribute $a$ to the accumulated sum seen so far.
- $\text{AVG}(a)$ 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 $\text{COUNT}$ and $\text{SUM}$ 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 7 operation. A consequence of this observation is that this algorithm does not fit the iterator framework very well; the entire grouping has to be done by the Open function 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 $S$ 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.
CHAPTER 6. QUERY EXECUTION

Operations on Nonclustered Data

Remember that 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.

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 \( D \) 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.

6.3.3 One-Pass Algorithms for Binary Operations

Let us now take up the binary operations: union, intersection, difference, product, and join. 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.

There is one exceptional operation — bag union — that 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 6.13. 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 6.3.2. As before, a hash table or balanced tree suffices. The structure requires a small amount of space (in addition to the space for the tuples themselves), which we shall neglect. Thus, the approximate requirement for a binary operation on relations \( R \) and \( S \) to be performed in one pass is:

\[ \min(B(R), B(S)) \leq M \]
This rule assumes that one buffer will be 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 where the search key is the entire tuple. All these tuples are also copied to the output. We then read each block of $R$ into the $M$th 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 the difference is not a commutative operator, we must distinguish between $R - S$ and $S - S$, continuing to assume that $R$ is the larger relation. In each case, we read $S$ into $M - 1$ buffers and build a search structure with full tuples as the search key.

To compute $R - S$, 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 $S - S$, 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 *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 $B(S) < M - 1$ 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
CHAPTER 6. QUERY EXECUTION

$t$ 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.

Notice that 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 we would expect the time needed to write the result to disk or otherwise process the output to exceed the processor time needed to create the output.

**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. As usual, a hash table or balanced tree are good examples of such structures. 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
What if M is not Known?

While we present algorithms as if M, the number of available memory blocks, were fixed and known in advance, remember that the available M is often unknown, except within some obvious limits like the total memory of the machine. Thus, a query optimizer, when choosing between a one-pass and a two-pass algorithm, might estimate M and make the choice based on this estimate. If the optimizer is wrong, the penalty is either thrashing of buffers between disk and memory (if the guess of M was too high), or unnecessary passes if M was underestimated.

There are also some algorithms that degrade gracefully when there is less memory than expected. For example, we can behave like a one-pass algorithm, unless we run out of space, and then start behaving like a two-pass algorithm. Sections 6.6.6 and 6.8.3 discuss some of these approaches.

6.3.4 Exercises for Section 6.3

Exercise 6.3.1: For each of the operations below, write an iterator that uses the algorithm described in this section.

* a) Projection.
* b) Distinct \( \{b\} \).
* c) Grouping \( \gamma_L \).
* d) Set union.
* e) Set intersection.
CHAPTER 6. QUERY EXECUTION

f) Set difference.
g) Bag intersection.
h) Bag difference.
i) Product.
j) Natural join.

**Exercise 6.3.2:** For each of the operators in Exercise 6.3.1, 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 6.3.3:** Show what the entries for groups would be if we implemented the 7 operator for the query of Exercise 6.1.6(d).

**Exercise 6.3.4:** Figure 6.14 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 6.3.5:** In Exercise 6.1.3 we defined five join-like operators. Give one-pass algorithms for each of them:

* a) \( R \bowtie S \), assuming \( R \) fits in memory.
* b) \( R \bowtie S \), assuming \( S \) fits in memory.
* c) \( R \bowtie S \), assuming \( R \) fits in memory.
* d) \( R \bowtie S \), assuming \( S \) fits in memory.
* e) \( R \bowtie_L S \), assuming \( R \) fits in memory.
* f) \( R \bowtie_L S \), assuming \( S \) fits in memory.
* g) \( R \bowtie_R S \), assuming \( R \) fits in memory.
* h) \( R \bowtie_R S \), assuming \( S \) fits in memory.
* i) \( R \bowtie S \), assuming \( R \) fits in memory.

### 6.4 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.
6.4. NESTED-LOOP JOINS

6.4.1 Tuple-Based Nested-Loop Join

We shall begin with the simplest variation of the nested-loop theme, where the loops 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 6.7.3. 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 6.4.3.

6.4.2 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 7.7.3, 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 we denote by \( R, \text{Open} \), and so on, as in Section 6.2.6. The code for the three iterator functions for nested-loop join is in Fig. 6.12. It makes the assumption that neither relation \( R \) nor \( S \) is empty.

6.4.3 A Block-Based Nested-Loop Join Algorithm

We can improve on the tuple-based nested-loop join of Section 6.4.1 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
OPEN(R,S) {
  R.Open();
  S.Open();
  s := S.GetNext();
}

GETNEXT(R,S) {
  REPEAT {
    r := R.GetNext();
    IF (NOT Found) { /* R is exhausted for
                   the current s */
      R.Close();
      s := S.GetNext();
      IF (NOT Found) RETURN; /* 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,S) {
  R.Close();
  S.Close();
}

Figure 6.12: Iterator functions for tuple-based nested-loop 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.

As in Section 6.3.3, let us assume $B(S) < B(R)$, but now let us also assume
that $B(S) > M$; i.e., neither relation fits entirely in main memory. We repeat-
edly read $M - 1$ blocks of 5 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. 6.13.
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 6.13: The nested-loop join algorithm

The program of Fig. 6.13 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 6.14: Assume \( B(R) = 1000 \) and \( B(S) = 500 \), and let \( M = 101 \). we shall use 100 blocks of memory to buffer 5 in 100-block chunks, so the outer loop of Fig. 6.13 iterates five times. At each iteration, we do 100 disk I/O's to read the chunk of 5, 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. □

The algorithm of Fig. 6.13 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. If necessary to distinguish it from the tuple-based nested-loop join of Section 6.4.1, we can call Fig. 6.13 "block-based nested-loop join."
6.4.4 Analysis of Nested-Loop Join

The analysis of Example 6.14 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

$$\frac{B(S)}{M-1}(M - 1 + B(R))$$

or

$$B(S) + \frac{B(S)B(R)}{M-1}$$

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 when both relations are large, although we should notice that for reasonably small examples such as Example 6.14, the cost of the nested-loop join is not much greater than the cost of a one-pass join, which would be 1500 disk I/O's. In fact, if $B(S) < M - 1$, the nested-loop join becomes identical to the one-pass join algorithm of Section 6.3.3.

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 6.5.5.

6.4.5 Summary of Algorithms so Far

The main-memory and disk I/O requirements for the algorithms we have discussed in Sections 6.3 and 6.4 are shown in Fig. 6.14. The memory requirements for 7 and 5 are actually more complex than shown, and $M = B$ is only a loose approximation. For 7, $M$ grows with the number of groups, and for 5, $M$ grows with the number of distinct tuples.

6.4.6 Exercises for Section 6.4

Exercise 6.4.1: Give the three iterator functions for the block-based version of nested-loop join.

* Exercise 6.4.2: Suppose $B(R) = B(S) = 10,000$, and $M = 1000$. Calculate the disk I/O cost of a nested-loop join.

Exercise 6.4.3: For the relations of Exercise 6.4.2, what value of $M$ would we need to compute $R \times M \times S$ using the nested-loop algorithm with no more than
6.5. TWO-PASS ALGORITHMS BASED ON SORTING

<table>
<thead>
<tr>
<th>Operators</th>
<th>Approximate M required</th>
<th>Disk I/O</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>σ, π</td>
<td>1</td>
<td>B</td>
<td>6.3.1</td>
</tr>
<tr>
<td>γ, δ</td>
<td>B</td>
<td>B</td>
<td>6.3.2</td>
</tr>
<tr>
<td>∪, ∩, −, ×, π</td>
<td>min(B(R), B(S))</td>
<td>B(R) + B(S)</td>
<td>6.3.3</td>
</tr>
<tr>
<td>⊙</td>
<td>any M ≥ 2</td>
<td>B(R)B(S)/M</td>
<td>6.4.3</td>
</tr>
</tbody>
</table>

Figure 6.14: Main memory and disk I/O requirements for one-pass and nested-loop algorithms

a) 100,000

! b) 25,000

! c) 15,000

disk I/O’s?

Exercise 6.4.4: If R and S are both unclustered, it seems that nested-loop join would require about \(\frac{T(R)T(S)}{M}\) 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 6.4.5: The iterator of Fig. 6.12 will not work properly if either R or S is empty. Rewrite the functions so they will work, even if one or both relations are empty.

6.5 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 6.3 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 6.9.

In this section, we consider sorting as a tool for implementing relational operations. The basic idea is as follows. If we have a large relation \( R \), where \( B(R) \) is larger than \( M \), the number of memory buffers we have available, then we can repeatedly:

1. Read \( M \) blocks of \( R \) into main memory.
2. Sort these \( M \) blocks in main memory, using an efficient sorting algorithm. Such an algorithm will take an amount of processor time that is just slightly more than linear in the number of tuples in main memory, so we expect that the time to sort will not exceed the disk I/O time for step (1).
3. Write the sorted list into \( M \) blocks of disk. We shall refer to the contents of these blocks as one of the sorted sublists of \( R \).

All the algorithms we shall discuss then use a second pass to "merge" the sorted sublists in some way to execute the desired operator.

### 6.5.1 Duplicate Elimination Using Sorting

To perform the \( \delta(R) \) operation in two passes, we sort the tuples of \( R \) in sublists as described above. We then use the available main memory to hold one block from each sorted sublist, as we did for the multiway merge sort of Section 2.3.4. However, instead of sorting the tuples from these sublists, we repeatedly copy one to the output and ignore all tuples identical to it. The process is suggested by Fig. 6.15.

More precisely, we look at the first unconsidered tuple from each block, and we find among them the first in sorted order, say \( t \). We make one copy of \( t \) in the output, and we remove from the fronts of the various input blocks all copies of \( t \). If a block is exhausted, we bring into its buffer the next block from the same sublist, and if there are \( t \)'s on that block we remove them as well.

**Example 6.15:** Suppose for simplicity that tuples are integers, and only two tuples fit on a block. Also, \( M = 3 \); i.e., there are three blocks in main memory. The relation \( R \) consists of 17 tuples:

\[
2, 5, 2, 1, 2, 2, 4, 5, 4, 3, 4, 2, 1, 5, 2, 1, 3
\]

We read the first six tuples into the three blocks of main memory, sort them, and write them out as the sublist \( R_1 \). Similarly, tuples seven through twelve are then read in, sorted and written as the sublist \( R_2 \). The last five tuples are likewise sorted and become the sublist \( R_3 \).

To start the second pass, we can bring into main memory the first block (two tuples) from each of the three sublists. The situation is now:
Looking at the first tuples of the three blocks in main memory, we find that 1 is the first tuple in sorted order. We therefore make one copy of 1 to the output, and we remove all 1's from the blocks in memory. When we do so, the block from \( R_1 \) is exhausted, so we bring in the next block, with tuples 2 and 3, from that sublist. Had there been more 1's on this block, we would eliminate them. The situation is now:

<table>
<thead>
<tr>
<th>Sublist ( R_i )</th>
<th>In Memory</th>
<th>Waiting on Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>1 2</td>
<td>2 2, 2 5</td>
</tr>
<tr>
<td>( R_2 )</td>
<td>2 3</td>
<td>4 4, 4 5</td>
</tr>
<tr>
<td>( R_3 )</td>
<td>1 1</td>
<td>2 3, 5</td>
</tr>
</tbody>
</table>

Now, 2 is the least tuple at the fronts of the lists, and in fact it happens to appear on each list. We write one copy of 2 to the output and eliminate 2's from the in-memory blocks. The block from \( R_1 \) is exhausted and the next block from that sublist is brought to memory. That block has 2's, which are eliminated, again exhausting the block from \( R_1 \). The third block from that sublist is brought to memory, and its 2 is eliminated. The present situation is:
CHAPTER 6. QUERY EXECUTION

To complete the example, 4 is next selected, consuming most of list \( R_2 \). At the final step, each list happens to consist of a single 5, which is output once and eliminated from the input buffers.

The number of disk I/O’s performed by this algorithm, as always ignoring the handling of the output, is:

1. \( B(R) \) to read each block of \( R \) when creating the sorted sublists.
2. \( B(R) \) to write each of the sorted sublists to disk.
3. \( B(R) \) to read each block from the sublists at the appropriate time.

Thus, the total cost of this algorithm is \( 3B(R) \), compared with \( B(R) \) for the single-pass algorithm of Section 6.3.2.

On the other hand, we can handle much larger files using the two-pass algorithm than with the one-pass algorithm. Assuming M blocks of memory are available, we create sorted sublists of M blocks each. For the second pass, we need one block from each sublist in main memory, so there can be no more than M sublists, each M blocks long. Thus, \( B < M/2 \) is required for the two-pass algorithm to be feasible, compared with \( B < M \) for the one-pass algorithm.

Put another way, to compute \( 6(R) \) with the two-pass algorithm requires only \( \sqrt{B(R)} \) blocks of memory, rather than \( B(R) \) blocks of memory.

6.5.2 Grouping and Aggregation Using Sorting

The two-pass algorithm for \( \gamma_L(R) \) is quite similar to the algorithm of Section 6.5.1 for \( 6(R) \). We summarize it as follows:

1. Read the tuples of \( R \) into memory, M blocks at a time. Sort each 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 6.3.2, 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 6 algorithm, this two-pass algorithm for 7 takes \( 3B(R) \) disk I/O’s, and will work as long as \( B(R) < M^2 \).

### 6.5.3 A Sort-Based Union Algorithm

When bag-union is wanted, the one-pass algorithm of Section 6.3.3, 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 should 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 6.5.4. To compute \( R \cup_S S \), we do the following:

1. Repeatedly bring \( M \) blocks of \( R \) into main memory, sort their tuples, and write the resulting sorted sublist back to disk.
2. Do the same for \( S \), to create sorted sublists for relation \( S \).
3. Use one main-memory buffer for each sublist of \( R \) and \( S \). Initialize each with the first block from the corresponding sublist.
4. 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). If a buffer becomes empty, reload it with the next block from its sublist.

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 \( 3(B(R) + B(S)) \).
The algorithm works as long as the total number of sublists among the two relations does not exceed M, because we need one buffer for each sublist. Since each sublist is M blocks long, that says the sizes of the two relations must not exceed $M^2$; that is, $B(R) + B(S) < M^2$.

### 6.5.4 Sort-Based Algorithms for Intersection and Difference

Whether the set version or the bag version is wanted, the algorithms are essentially the same as that of Section 6.5.3, except that the way we handle the copies of a tuple $t$ at the fronts of the sorted sublists differs. In general we create the sorted sublists of M blocks each for both argument relations $R$ and $S$. We use one main-memory buffer for each sublist, initially loaded with the first block of the sublist.

We then repeatedly consider the least tuple $t$ among the remaining tuples in all the buffers. We count the number of tuples of $R$ that are identical to $t$ and we also count the number of tuples of $S$ that are identical to $t$. Doing so requires that we reload buffers from any sublists whose currently buffered block is exhausted. The following indicates how we determine whether $t$ is output, and if so, how many times:

- If the operation is set intersection, output $t$ if it appears in both $R$ and $S$.
- If the operation is 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 $t$ does not appear in both of the relations.
- If the operation is set difference, $R - S$, output $t$ if and only if it appears in $R$ but not in $S$.
- If the operation is 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.

**Example 6.16**: Let us make the same assumptions as in Example 6.15: $M = 3$, tuples are integers, and two tuples fit in a block. The data will be almost the same as in that example as well. However, here we need two arguments, so we shall assume that $R$ has 12 tuples and $S$ has 5 tuples. Since main memory can fit six tuples, in the first pass we get two sublists from $R$, which we shall call $R_1$ and $R_2$, and only one sorted sublist from $S$, which we refer to as $S_1$. After creating the sorted sublists (from unsorted relations similar to the data from Example 6.15), the situation is:

---

2 Since $S$ fits in main memory, we could actually use the one-pass algorithms of Section 6.3.3, but we shall use the two-pass approach for illustration.
6.5. TWO-PASS ALGORITHMS BASED ON SORTING

Suppose we want to take the bag difference \( R - B \cdot S \). We find that the least tuple among the main-memory buffers is 1, so we count the number of 1’s among the sublists of \( R \) and among the sublists of \( S \). We find that 1 appears once in \( R \) and twice in \( S \). Since 1 does not appear more times in \( R \) than in \( S \), we do not output any copies of tuple 1. Since the first block of \( S_i \) was exhausted counting 1’s, we loaded the next block of \( S_i \), leaving the following situation:

<table>
<thead>
<tr>
<th>Sublist</th>
<th>In memory</th>
<th>Waiting on disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 ):</td>
<td>1 2</td>
<td>2 2, 2 5</td>
</tr>
<tr>
<td>( R_2 ):</td>
<td>2 3</td>
<td>4 4, 4 5</td>
</tr>
<tr>
<td>( S_1 ):</td>
<td>1 1</td>
<td>2 3, 5</td>
</tr>
</tbody>
</table>

We now find that 2 is the least remaining tuple, so we count the number of its occurrences in \( R \), which is five occurrences, and we count the number of its occurrences in \( S \), which is one. We thus output tuple 2 four times. As we perform the counts, we must reload the buffer for \( R_1 \) twice, which leaves:

<table>
<thead>
<tr>
<th>Sublist</th>
<th>In memory</th>
<th>Waiting on disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 ):</td>
<td>2</td>
<td>2 2, 2 5</td>
</tr>
<tr>
<td>( R_2 ):</td>
<td>2 3</td>
<td>4 4, 4 5</td>
</tr>
<tr>
<td>( S_1 ):</td>
<td>2 3</td>
<td>5</td>
</tr>
</tbody>
</table>

Next, we consider tuple 3, and find it appears once in \( R \) and once in \( S \). We therefore do not output 3 and remove its copies from the buffers, leaving:

<table>
<thead>
<tr>
<th>Sublist</th>
<th>In memory</th>
<th>Waiting on disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 ):</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>( R_2 ):</td>
<td>3</td>
<td>4 4, 4 5</td>
</tr>
<tr>
<td>( S_1 ):</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Tuple 4 occurs three times in \( R \) and not at all in \( S \), so we output three copies of 4. Last, 5 appears twice in \( R \) and once in \( S \), so we output 5 once. The complete output is 2, 2, 2, 2, 4, 4, 4, 5. □

The analysis of this family of algorithms is the same as for the set-union algorithm described in Section 6.5.3:

- \( 3(B(R) + B(S)) \) disk I/O’s.
- Approximately \( B(R) + B(S) \leq M^2 \) for the algorithm to work.
6.5.5 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 6.5.7 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 a two-phase, multiway merge sort, with \( Y \) as the sort key.
2. Sort \( S \) similarly.
3. Merge the sorted \( R \) and \( S \). We generally 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 \( S \) with a common \( Y \)-value.
   (e) If either relation has no more unconsidered tuples in main memory,
   reload the buffer for that relation.

Example 6.17: Let us consider the relations \( R \) and \( S \) from Example 6.14. Recall these relations occupy 1000 and 500 blocks, respectively, and there are \( M = 101 \) main-memory buffers. When we use two-phase, multiway merge sort
on a relation, 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 6.14. 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. Moreover, we shall see in Section 6.5.7 that it is usually possible to perform a sort-join in $3(B(R) + B(S))$ disk I/O's, which would be 4500 in this example and which is below the cost of nested-loop join.

If there is a $y$-value $y$ for which the number of tuples with this $Y$-value $y$ does not fit in $M$ buffers, then we need to modify the above algorithm.

1. If the tuples from one of the relations, say $R$, that have $Y$-value $y$ fit in $M - 1$ buffers, then load these blocks of $R$ into buffers, and read the blocks of $S$ that hold tuples with $y$, one at a time, into the remaining buffer. In effect, we do the one-pass join of Section 6.3.3 on only the tuples with $Y$-value $y$.

2. If neither relation has sufficiently few tuples with $Y$-value $y$ that they all fit in $M - 1$ buffers, then use the $M$ buffers to perform a nested-loop join on the tuples with $Y$-value $y$ from both relations.

Note that in either case, it may be necessary to read blocks from one relation and then ignore them, having to read them later. For example, in case (1), we might first read the blocks of $S$ that have tuples with $y$-value $y$ and find that there are too many to fit in $M - 1$ buffers. However, if we then read the tuples of $R$ with that $y$-value we find that they do fit in $M - 1$ buffers.

### 6.5.6 Analysis of Simple Sort-Join

As we noted in Example 6.17, our algorithm performs five disk I/O's for every block of the argument relation. The exception would be if there were so many tuples with a common $y$-value that we needed to do one of the specialized joins on these tuples. In that case, the number of extra disk I/O's depends on whether one or both relations have so many tuples with a common $y$-value that they require more than $M - 1$ buffers by themselves. We shall not go into all the detailed cases here; the exercises contain some examples to work out.
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 2.3.4, we need $B(R) < M^2$ and $B(S) < M^2$ to perform these sorts. Once done, we shall not run out of buffers, although as discussed before, we may have to deviate from the simple merge if the tuples with a common $Y$-value cannot fit in $M$ buffers. In summary, assuming no such deviations are necessary:

- 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.

### 6.5.7 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 we:

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 6.18:** Let us again consider the problem of Example 6.14: 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. 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, but if there are more tuples than that we must use a special algorithm such as was discussed at the end of Section 6.5.5.

Assuming that we do not need to modify the algorithm for large groups of tuples with the same $Y$-value, then we perform three disk I/O's per block of

---

3 Technically, we could have arranged for the sublists to have length 10 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.
6.5. TWO-PASS ALGORITHMS BASED ON SORTING 289

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 6.5.5
when it can be used. As we observed in Example 6.18, the number of disk
I/O's is 3(B(R) + B(S)). We can perform the algorithm on data that is almost
as large as that of the previous algorithm. The sizes of the sorted sublists are
M blocks, and there can be at most M of them among the two lists. Thus,
B(R) + B(S) \leq M^2 is sufficient.

We might wonder whether we can avoid the trouble that arises when there
are many tuples with a common Y-value. Some important considerations are:

1. Sometimes we can be sure the problem will not arise. For example, if Y
   is a key for R, then a given Y-value y can appear only once among all the
   blocks of the sublists for R. When it is y's turn, we can leave the tuple
   from R in place and join it with all the tuples of S that match. If blocks of
   S's sublists are exhausted during this process, they can have their buffers
   reloaded with the next block, and there is never any need for additional
   space, no matter how many tuples of S have Y-value y. Of course, if Y
   is a key for S rather than R, the same argument applies with R and S
   switched.

2. If B(R) + B(S) is much less than M^2, we shall have many unused buffers
   for storing tuples with a common Y-value, as we suggested in Exam-
   ple 6.18.

3. If all else fails, we can use a nested-loop join on just the tuples with a
   common Y-value, using extra disk I/O's but getting the job done correctly.
   This option was discussed in Section 6.5.5.

6.5.8 Summary of Sort-Based Algorithms

In Fig. 6.16 is a table of the analysis of the algorithms we have discussed in
Section 6.5. As discussed in Sections 6.5.5 and 6.5.7, modifications to the time
and memory requirements are necessary if we join two relations that have many
tuples with the same value in the join attribute(s).

6.5.9 Exercises for Section 6.5

Exercise 6.5.1: Using the assumptions of Example 6.15 (two tuples per block,
etc.),

a) Show the behavior of the two-pass duplicate-elimination algorithm on the
sequence of thirty one-component tuples in which the sequence 0, 1, 2, 3,
4 repeats six times.
### Figure 6.16: Main memory and disk I/O requirements for sort-based algorithms

<table>
<thead>
<tr>
<th>Operators</th>
<th>Approximate $M$ required</th>
<th>Disk I/O</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma, \delta$</td>
<td>$\sqrt{B}$</td>
<td>$3B$</td>
<td>6.5.1, 6.5.2</td>
</tr>
<tr>
<td>$\cup, \cap, -$</td>
<td>$\sqrt{B(R) + B(S)}$</td>
<td>$3{B(R) + B(S)}$</td>
<td>6.5.3, 6.5.4</td>
</tr>
<tr>
<td>$\bowtie$</td>
<td>$\sqrt{\max{B(R), B(S)}}$</td>
<td>$5{B(R) + B(S)}$</td>
<td>6.5.5</td>
</tr>
<tr>
<td>$\bowtie$</td>
<td>$\sqrt{B(R) + B(S)}$</td>
<td>$3{B(R) + B(S)}$</td>
<td>6.5.7</td>
</tr>
</tbody>
</table>

b) Show the behavior of the two-pass grouping algorithm computing the relation $\gamma_{a,AVG(b)}(R)$. Relation $R(a,b)$ consists of the thirty tuples $t_0$ through $t_{29}$, and the tuple $t_i$ has $i$ modulo 5 as its grouping component $a$, and $i$ as its second component $b$.

**Exercise 6.5.2:** For each of the operations below, write an iterator that uses the algorithm described in this section.

* a) Distinct ($\delta$).
* b) Grouping ($\gamma_L$).
* c) Set intersection.
* d) Bag difference.
* e) Natural join.

**Exercise 6.5.3:** If $B(R) = B(S) = 10000$ 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 6.5.7.

**Exercise 6.5.4:** 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 6.5.5: In Example 6.17 we discussed the join of two relations \( R \) and \( S \), with 1000 and 500 blocks, respectively, and \( M = 101 \). However, we pointed out that there would be additional disk I/O’s if there were 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 6.5.6: Repeat Exercise 6.5.5 for the more efficient sort-join of Section 6.5.7.

Exercise 6.5.7: 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) \( S \).
- b) \( \gamma \).
- c) A binary operation such as join or union.

Exercise 6.5.8: Describe a two-pass, sort-based algorithm for each of the five join-like operators of Exercise 6.1.3.

Exercise 6.5.9: Suppose records could be larger than blocks, i.e., we could have spanned records. How would the memory requirements of two-pass, sort-based algorithms change?

Exercise 6.5.10: Sometimes, it is possible to save some disk I/O’s if we leave the last sublist in memory. It may even make sense to use sublists of fewer than \( M \) blocks to take advantage of this effect. How many disk I/O’s can be saved this way?

Exercise 6.5.11: OQL allows grouping of objects according to arbitrary, user-specified functions of the objects. For example, one could group tuples according to the sum of two attributes. How would we perform a sort-based grouping operation of this type on a set of objects?

6.6 Two-Pass Algorithms Based on Hashing

There is a family of hash-based algorithms that attack the same problems as in Section 6.5. 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 have the same hash value.

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. If there are M buffers available, we can pick M as the number of buckets, thus gaining a factor of M in the size of the relations we can handle. Notice that the sort-based algorithms of Section 6.5 also gain a factor of M by preprocessing, although the sorting and hashing approaches achieve their similar gains by rather different means.

### 6.6.1 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 bucket of each block if it is not empty. The algorithm is given in more detail in Fig. 6.17. Note that it assumes that tuples, while they may be variable-length, are never too large to fit in an empty buffer.

```plaintext
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 6.17: Partitioning a relation $R$ into M — 1 buckets
6.6. TWO-PASS ALGORITHMS BASED ON HASHING

6.6.2 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. 6.17. Note that two copies of the same tuple \( t \) will hash to the same bucket. Thus, \( \delta \) has the essential property we need: we can examine one bucket at a time, perform \( \delta \) on that bucket in isolation, and take as the answer the union of \( \delta(R_i) \), where \( R_i \) is the portion of \( R \) that hashes to the \( i \)th bucket. The one-pass algorithm of Section 6.3.2 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 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., \( B(R) < M(M-1) \), then the two-pass, hash-based algorithm will work. In fact, as we discussed in Section 6.3.2, it is only necessary that the number of distinct tuples in one bucket fit in \( M \) buffers, but we cannot be sure that there are any duplicates at all. Thus, a conservative estimate, with a simple form in which \( M \) and \( M - 1 \) are considered the same, is \( B(R) < M^2 \), exactly as for the sort-based, two-pass algorithm for \( \delta \).

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 \( 3B(R) \).

6.6.3 A Hash-Based Algorithm for 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 \( \delta \) from Section 6.3.2 to process each bucket in turn. As we discussed for \( \delta \) in Section 6.6.2, we can process each bucket in main memory provided \( B(R) < M^2 \).

However, on the second pass, we only need 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. Normally, a group's record will be no larger than a tuple of \( R \). If so, then a better upper bound on \( B(R) \) is \( M^2 \) times the average number of tuples per group.

As a consequence, if there are are few groups, then we may actually be able
6.6.4 Hash-Based Algorithms for 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 $R \cup S$, 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 $\cap$, the simple bag-union algorithm of Section 6.3.3 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 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 $3B(R) + B(S)$ 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 $\min(B(R), B(S)) < M^2$, approximately.

6.6.5 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 6.6.4. 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, $t_{XY}$ 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.

Example 6.19: Let us renew our discussion of the two relations $R$ and $S$ from Example 6.14, 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 6.5.7.

We may generalize Example 6.19 to conclude that:

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

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.

### 6.6.6 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 \textit{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 \textsc{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:

\[
\frac{mB(S)}{k} + k - m \leq M
\]  

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. The result of any successful joins is immediately output. 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 5-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 \( 2(m/k)(B(R) + B(S)) \). We must thus ask how to maximize \( m/k \), subject to the constraint of equation (6.1). While the solution of this problem can be done formally, there is an intuition that gives the surprising but correct answer: \( m = 1 \), while \( k \) is as small as possible.

The argument 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

\[
\frac{2M}{B(S)} (B(R) + B(S))
\]

and the total cost is

\[
3 - \frac{2M}{B(S)} (B(R) + B(S))
\]

**Example 6.20:** Consider the problem of Example 6.14, 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.
6.6. **TWO-PASS ALGORITHMS BASED ON HASHING**

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.

### 6.6.7 Summary of Hash-Based Algorithms

Figure 6.18 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 $\gamma$ and $\delta$ 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.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Approximate $M$ required</th>
<th>Disk I/O</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma, \delta$</td>
<td>$\sqrt{B}$</td>
<td>$3B$</td>
<td>6.6.2, 6.6.3</td>
</tr>
<tr>
<td>$\cup, \cap, -$</td>
<td>$\sqrt{B(S)}$</td>
<td>$3(B(R) + B(S))$</td>
<td>6.6.4</td>
</tr>
<tr>
<td>$\bowtie$</td>
<td>$\sqrt{B(S)}$</td>
<td>$3(B(R) + B(S))$</td>
<td>6.6.5</td>
</tr>
<tr>
<td>$\bowtie$</td>
<td>$\sqrt{B(S)}$</td>
<td>$(3 - 2M/B(S))(B(R) + B(S))$</td>
<td>6.6.6</td>
</tr>
</tbody>
</table>

*Figure 6.18: Main memory and disk I/O requirements for hash-based algorithms; for binary operations, assume $B(S) < B(R)$*

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, as for sort-based algorithms.
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 later, 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 somewhat 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.

6.6.8 Exercises for Section 6.6

Exercise 6.6.1: 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) \( \delta \).
* b) \( \gamma \).
* c) \( \cap_B \).
* d) \( -s \).

Exercise 6.6.2: 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 6.6.3: Write iterators that implement the two-pass, hash-based algorithms for (a) \( \delta \) (b) \( \gamma \) (c) \( \cap_B \) (d) \( -s \) (e) \( \cup \).
Exercise 6.6.4: Suppose we are performing a two-pass, hash-based grouping operation on a relation $R$ of the appropriate size; i.e., $B(R) < M^2$. 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 6.6.5: 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 $R \Join M \Join S$, 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 6.20?

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

6.7 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 6.2.1. To appreciate many of the issues, we first need to digress and consider "clustering" indexes.

6.7.1 Clustering and Nonclustering Indexes

Recall from Section 6.2.3 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, but even a clustered relation can have nonclustering indexes.

**Example 6.21**: 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. 6.19. 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 6.19: A clustering index has all tuples with a fixed value packed into (close to) the minimum possible number of blocks

### 6.7.2 Index-Based Selection

In Section 6.2.1 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. 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 clustering, then the number of disk I/O’s to retrieve the set $\sigma_{a=v}(R)$ will be about $B(R)/V(R.a)$. The actual number may be somewhat higher, because:

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

---

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.

6 Recall from Section 6.2.3 the notation we developed: $T(R)$ for the number of tuples in $R$ and $V(R, L)$ for the number of distinct tuples in $\pi_L(R)$. 
6.7. INDEX-BASED ALGORITHMS

Notions of Clustering

We have seen three different, although related, concepts called “clustering” or “clustered.”

1. In Section 4.2.2 we spoke of the “clustered-file organization,” where tuples of one relation $R$ are placed with a tuple of some other relation $S$ with which they share a common value; the example was grouping movie tuples with the tuple of the studio that made the movie.

2. In Section 6.2.3 we spoke of a “clustered relation,” meaning that the tuples of the relation are stored in blocks that are exclusively, or at least predominantly, devoted to storing that relation.

3. Here, we have introduced the notion of a clustering index — an index in which the tuples having a given value of the search key appear in blocks that are largely devoted to storing tuples with that search-key value. Typically, the tuples with a fixed value will be stored consecutively, and only the first and last blocks with tuples of that value will also have tuples of another search-key value.

The clustered-file organization is one example of a way to have a clustered relation that is not packed into blocks which are exclusively its own. Suppose that one tuple of the relation $S$ is associated with many $R$-tuples in a clustered file. Then, while the tuples of $R$ are not packed in blocks exclusively devoted to $R$, these blocks are “predominantly” devoted to $R$, and we call $R$ clustered. On the other hand, $S$ will typically not be a clustered relation, since its tuples are usually on blocks devoted predominantly to $R$-tuples rather than $S$-tuples.

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.

3. Although the index is clustering, the tuples with $a = v$ may be spread over several extra blocks. Two reasons why that situation might occur are:

   (a) We might not pack blocks of $R$ as tightly as possible because we want to leave room for growth of $R$, as discussed in Section 4.1.6.

   (b) $R$ might be stored with some other tuples that do not belong to $R$, say in a clustered-file organization.

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 appeal on the same block, and that block remains buffered in memory.

**Example 6.22**: Suppose $B(R) = 1000$, and $T(R) = 20,000$. That is, $R$ has 20,000 tuples that are packed 20 to a block. Let $a$ be one of the attributes of $R$, suppose there is an index on $a$, and consider the operation $\sigma_{a=q}(R)$. 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.

4. If $V(R,a) = 100$ and the index is nonclustering, then the index-based algorithm uses $20,000/100 = 200$ 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 $\sigma_{a>10 \text{ AND } a<20}(R)$.

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 $a = v \text{ AND } C'$, 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 7.7.1.

### 6.7.3 Joining by Using an Index

All the binary operations we have considered, and the unary full-relation operations of \( \sigma \) and \( \delta \) 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 adequate 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 \( ty \) 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 \( ty \) 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 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, so we may take \( T(R)T(S) / V(S,Y) \) or \( T(R) \) \( \max(1, B(S) / V(S,Y)) \) as the cost of this join method, for the cases of nonclustered and clustered indexes on \( S \), respectively.

**Example 6.23:** 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 \( R \) (neglected in the formulas above) plus \( 10,000 \times 500 / 100 = 50,000 \) disk I/O’s. This number is considerably above

\(^7\) But remember that \( B(S) / V(S,Y) \) must be replaced by 1 if it is less, as discussed in Section 6.7.2.
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 6.23 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 6.7.5 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.

### 6.7.4 Joins Using a Sorted Index

When the index is a B-tree or other structure from which we can easily extract the tuples of a relation in sorted order, then 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 a sorted 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 6.5.5. 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 6.24:** 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 5 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 5, 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
6.7. INDEX-BASED ALGORITHMS

structure, as suggested in Fig. 6.20. 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 6.5.5. 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 6.20: A zig-zag join using two indexes](image)

**Example 6.25**: Let us continue with Example 6.23, 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 \( 5 \) that allows us to retrieve the tuples of \( 5 \) 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 \( 5 \)'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. □

6.7.5 Exercises for Section 6.7

Exercise 6.7.1: 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) \( R \cup S \) (assume that \( R \) and \( S \) have no duplicates, although they may have tuples in common).

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

c) \( \delta(R) \).

Exercise 6.7.2: Suppose \( B(R) = 10,000 \) and \( T(R) = 500,000 \). Let there be an index on \( R.a \), and \( |\pi R.a| = k \) for some number \( k \). Give the cost of \( \sigma_{a=\theta}(R) \), 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 6.7.3: Repeat Exercise 6.7.2 if the operation is the range query \( \sigma_{C<a \text{ AND } C<D(R)} \). You may assume that \( C \) and \( D \) are constants such that \( k/10 \) of the values are in the range.

! Exercise 6.7.4: 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 6.7.2.

b) Exercise 6.7.3.

* Exercise 6.7.5: Consider the SQL query:

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

This query uses the "movie" relations:
StarsIn(title, year, starName)
MovieStar(name, address, gender, birthdate)

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

\[ \sigma_{title='King Kong'}(\text{StarsIn}) \]

and MovieStar, which can be implemented much as a natural join \( R \bowtie S \). Since there were only two 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 \( R \bowtie S \) with the cost of a sort- or hash-based join.

Exercise 6.7.6: In Example 6.25 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?

### 6.8 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. 6.21.

#### 6.8.1 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.

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
Figure 6.21: The buffer manager responds to requests for main-memory access to disk blocks

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.

6.8.2 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:

- **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.** This algorithm is a commonly implemented, efficient approximation to LRU. Think of the buffers as arranged in a circle, as suggested by Fig. 6.22. 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. 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. 6.22, 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 3.3.5 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
CHAPTER 6. QUERY EXECUTION

Figure 6.22: The clock algorithm visits buffers in a round-robin fashion and replaces the first one with a 0 flag

More Tricks Using the Clock Algorithm

The "clock" algorithm for choosing buffers to free is not limited to the scheme described in Section 6.8.2, 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.

The 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.

6.8.3 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 6.26:** As an example of the issues, let us consider the block-based nested-loop join of Fig. 6.13. 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 6.4.4 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 6.3.3.

If we pin the M — 1 blocks we use for S on one iteration of the outer loop, then we shall not lose their buffers during the round. On the other hand, more buffers may become available during that iteration. These buffers allow more than one block of R to be kept in memory at the same time, but unless we are careful, the extra buffers will not improve the running time of the nested-loop join.

For instance, 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 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 will visit the blocks of R in an order that alternates: first-to-last and then last-to-first. 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. □
CHAPTER 6. QUERY EXECUTION

Other algorithms are also 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.

• The main-memory sorting of sublists can be performed by a number of different algorithms. Since algorithms like merge-sort and quicksort are recursive, most of the time is spent on rather small regions of memory. Thus, either LRU or FIFO will perform well for this part of a sort-based algorithm.

• 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.

6.8.4 Exercises for Section 6.8

Exercise 6.8.1: 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 6.8.2: 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 6.8.3: In Example 6.26, 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 5. Which strategy yields the fewest disk I/O's?

6.9 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 6.5 and 6.6 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.

6.9.1 Multipass Sort-Based Algorithms

In Section 2.3.5 we alluded to how the two-phase multiway merge sort 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 particular \( n \).

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., \( B(R) < M \)), then read \( R \) into main memory, sort it using your favorite 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 2.3.4.

If we are not merely sorting \( R \), but performing a unary operation such as 7 or 5 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 6, output one copy of each distinct tuple, and skip over copies of the tuple.
- For a 7, 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 6.5.2.

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 6.5.

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 \( M \times B(R)/(B(R)+B(S)) \) of the buffers, and \( S \) gets the rest.

### 6.9.2 Performance of Multipass, Sort-Based Algorithms

Now, let us explore the relationship between the number of disk 1/0'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 \( B(R) < M \). 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

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

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 \( B(R) < s(M, k) \), which says that \( B(R) < M^k \). Put another way, if we want to sort \( R \) in \( k \) passes, then the minimum number of buffers we can use is \( M = \lceil B(R) \rceil \).

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 \( B(R) + B(S) < j(M, k) \).

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, \( B(R) + B(S) \) can be no larger than \( M^k \), or put another way, \( j(M, k) = M^k \).

Reversing the role of the parameters, we can also state that to compute the join in \( k \) passes requires \( \lceil B(R)+ B(S) \rceil \) 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 \( 2(k-1) (B(R) + B(S)) \) disk I/O's to sort the sublists, and another \( B(R)+ B(S) \)
6.9. ALGORITHMS USING MORE THAN TWO PASSES

6.9.3 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. For the common relational operations we have considered — duplicate-elimination, grouping, union, intersection, difference, natural join, and equijoin — the result of the operation on the entire relation(s) will be the union of the results on the bucket(s). 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 $M$th buffer.

**INDUCTION:** If no relation fits in main memory, then hash each relation into $M - 1$ buckets, as discussed in Section 6.6.1. Recursively perform the operation on each bucket or corresponding pair of buckets, and accumulate the output from each bucket or pair.

6.9.4 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 $T$ or $S$ 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., $B(R) < M$.

**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 $u(M, k) = (M - 1)u(M, k - 1)$. 

The result is a total of $(2k - 1)(B(R) + B(S))$ disk I/O's.
CHAPTER 6. QUERY EXECUTION

If we expand the recurrence above, we find that

\[ u(M, k) = M(M - 1)^{k-1} \]

or approximately, assuming \( M \) is large, \( u(M, fc) \approx M^k \). Equivalently, we can perform one of the unary relational operations on relation \( R \) in \( k \) passes with \( M \) buffers, provided \( M < \left( B(R) \right)^{1/k} \).

We may perform a similar analysis for binary operations. As in Section 6.9.2, 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 \( R(X, Y) \bowtie M S(Y, Z) \). 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 6.3.3.

**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 \( j(M, fc) = M^k \). That is, we can join \( R(X, Y) \bowtie S(Y, Z) \) using \( fc \) passes and \( M \) buffers provided \( M^k \geq \min(B(R), B(S)) \).

### 6.9.5 Exercises for Section 6.9

**Exercise 6.9.1:** 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 6.9.2:** 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 6.6.6.

b) Improving a sort-based algorithm by storing blocks consecutively on disk (Section 6.6.7).

c) Improving a hash-based algorithm by storing blocks consecutively on disk (Section 6.6.7).
6.10 Parallel Algorithms for Relational Operations

Database operations, frequently being time-consuming and involving a lot of data, can generally profit from parallel processing. In this section, we shall review the principal architectures for parallel machines. We then concentrate on the "shared-nothing" architecture, which appeals to be the most cost effective for database operations, although it may not be superior for other parallel applications. There are simple modifications of the standard algorithms for most relational operations that will exploit parallelism almost perfectly. That is, the time to complete an operation on a p-processor machine is about \( \frac{1}{p} \) of the time it takes to complete the operation on a uniprocessor.

6.10.1 Models of Parallelism

At the heart of all parallel machines is a collection of processors. Often the number of processors \( p \) is large, in the hundreds or thousands. We shall assume that each processor has its own local cache, which we do not show explicitly in our diagrams. In most organizations, each processor also has local memory, which we do show. Of great importance to database processing is the fact that along with these processors are many disks, perhaps one or more per processor, or in some architectures a large collection of disks accessible to all processors directly.

![Figure 6.23: A shared-memory machine](image)

Additionally, parallel computers all have some communications facility for passing information among processors. In our diagrams, we show the com-
munication as if there were a shared bus for all the elements of the machine. However, in practice a bus cannot interconnect as many processors or other elements as are found in the largest machines, so the interconnection system is in many architectures a powerful switch, perhaps augmented by busses that connect subsets of the processors in local clusters.

The three most important classes of parallel machines are:

1. **Shared Memory.** In this architecture, illustrated in Fig. 6.23, each processor has access to all the memory of all the processors. That is, there is a single physical address space for the entire machine, rather than one address space for each processor. The diagram of Fig. 6.23 is actually too extreme, suggesting that processors have no private memory at all. Rather, each processor has some local memory, which it typically uses whenever it can. However, it has direct access to the memory of other processors when it needs to. Large machines of this class are of the NUMA (nonuniform memory access) type, meaning that it takes somewhat more time for a processor to access data in a memory that "belongs" to some other processor than it does to access its "own" memory, or the memory of processors in its local cluster. However, the difference in memory-access times are not great in current architectures. Rather, all memory accesses, no matter where the data is, take much more time than a cache access, so the critical issue is whether or not the data a processor needs is in its own cache.

![Figure 6.24: A shared-disk machine](image)

2. **Shared Disk.** In this architecture, suggested by Fig. 6.24, every processor has its own memory, which is not accessible directly from other processors.
However, the disks are accessible from any of the processors through the communication network. Disk controllers manage the potentially competing requests from different processors. The number of disks and processors need not be identical, as it might appear from Fig. 6.24.

3. **Shared Nothing.** Here, all processors have their own memory and their own disk or disks, as in Fig. 6.25. All communication is via the communication network, from processor to processor. For example, if one processor \( P \) wants to read tuples from the disk of another processor \( Q \), then processor \( P \) sends a message to \( Q \) asking for the data. Then, \( Q \) obtains the tuples from its disk and ships them over the network in another message, which is received by \( P \).

As we mentioned in the introduction to this section, the shared-nothing architecture is the most commonly used architecture for “database machines,” that is, parallel computers designed specifically for supporting a database. Shared-nothing machines are relatively inexpensive to build, but when we design algorithms for these machines we must be aware that it is costly to send data from one processor to another.

Normally, data must be sent between processors in a message, which has considerable overhead associated with it. Both processors must execute a program that supports the message transfer, and there may be contention or delays associated with the communication network as well. Typically, the cost of a message can be broken into a large fixed overhead plus a small amount of time per byte transmitted. Thus, there is a significant advantage to designing a parallel algorithm so that communications between processors involve large amounts of data sent at once. For instance, we might buffer several blocks of data at processor \( P \), all bound for processor \( Q \). If \( P \) does not need the data immediately, it may be much more efficient to wait until we have a long message at \( P \) and then send it to \( Q \).
Algorithms on Other Parallel Architectures

The shared-disk machine favors long messages, just like the shared-nothing machine does. If all communication is via a disk, then we need to move data in block-sized chunks, and if we can organize the data to be moved so it is together on one track or cylinder, then we can save much of the latency, as discussed in Section 2.4.1.

On the other hand, a shared-memory machine allows communication to occur between any two processors via the memory. There is no extensive software needed to send a message, and the cost of reading or writing main memory is proportional to the number of bytes involved. Thus, shared-memory machines would be able to take advantage of algorithms that required fast, frequent, and short communications between processors. It is interesting that, while such algorithms are known in other domains, database processing does not seem to require such algorithms.

6.10.2 Tuple-at-a-Time Operations in Parallel

Let us begin our discussion of parallel algorithms for a shared-nothing machine by considering the selection operator. First, we must consider how data is best stored. As first suggested by Section 2.4.2, it is useful to distribute our data across as many disks as possible. For convenience, we shall assume there is one disk per processor. Then if there are $p$ processors, divide any relation $R$'s tuples evenly among the $p$ processor’s disks.

Suppose we want to perform $\sigma_{C}(R)$. We may use each processor to examine the tuples of $R$ present on its own disk. For each, it finds those tuples satisfying condition $C$ and copies those to the output. To avoid communication among processors, we store those tuples $t$ in $\sigma_{C}(R)$ at the same processor that has $t$ on its disk. Thus, the result relation $\sigma_{C}(R)$ is divided among the processors, just like $R$ is.

Since $\sigma_{C}(R)$ may be the input relation to another operation, and since we want to minimize the elapsed time and keep all the processors busy all the time, we would like $\sigma_{C}(R)$ to be divided evenly among the processors. If we were doing a projection, rather than a selection, then the number of tuples in $\pi_{L}(R)$ at each processor would be the same as the number of tuples of $R$ at that processor. Thus, if $R$ is distributed evenly, so would its projection. However, a selection could radically change the distribution of tuples in the result, compared to the distribution of $R$.

Example 6.27: Suppose the selection is $\sigma_{a=10}(R)$, that is, find all the tuples of $R$ whose value in the attribute $a$ (assumed to be one of $R$'s attributes) is 10. Suppose also that we have divided $R$ according to the value of the attribute $a$. Then all the tuples of $R$ with $a = 10$ are at one of the processors, and the entire
To avoid the problem suggested by Example 6.27, we need to think carefully about the policy for partitioning our stored relations among the processors. Probably the best we can do is to use a hash function \( h \) that involves all the components of a tuple in such a way that changing any component of a tuple \( t \) can change \( h(t) \) to be any possible bucket number. For example, if we want \( B \) buckets, we might convert each component somehow to an integer between 0 and \( B - 1 \), add the integers for each component, divide the result by \( B \), and take the remainder as the bucket number. If \( B \) is also the number of processors, then we can associate each processor with a bucket and give that processor the contents of its bucket.

### 6.10.3 Parallel Algorithms for Full-Relation Operations

First, let us consider the operation \( \delta(R) \), which is somewhat atypical of the full-relation operations. If we use a hash function to distribute the tuples of \( R \) as suggested in Section 6.10.2, then we shall place duplicate tuples of \( R \) at the same processor. If so, then we can produce \( \delta(R) \) in parallel by applying a standard, uniprocessor algorithm (as in Section 6.5.1 or 6.6.2, e.g.) to the portion of \( R \) at each processor. Likewise, if we use the same hash functions to distribute the tuples of both \( R \) and \( S \), then we can take the union, intersection, or difference of \( R \) and \( S \) by working in parallel on the portions of \( R \) and \( S \) at each processor.

However, suppose that \( R \) and \( S \) are not distributed using the same hash function, and we wish to take their union. In this case, we must first make copies of all the tuples of \( R \) and \( S \) and distribute them according to a single hash function \( h \).

In parallel, we hash the tuples of \( R \) and \( S \) at each processor, using hash function \( h \). The hashing proceeds as described in Section 6.6.1, but when the buffer corresponding to a bucket \( i \) at one processor \( j \) is filled, instead of moving it to the disk at \( j \), we ship the contents of the buffer to processor \( i \). If we have room for several blocks per bucket in main memory, then we may wait to fill several buffers with tuples of bucket \( i \) before shipping them to processor \( i \).

Thus, processor \( i \) receives all the tuples of \( R \) and \( S \) that belong in bucket \( i \). In the second stage, each processor performs the union of the tuples from \( R \) and \( S \) belonging to its bucket. As a result, the relation \( R \cup S \) will be distributed over all the processors. If hash function \( h \) truly randomizes the placement of tuples, then it is unlikely that a particular attribute value will be represented in only a small subset of the buckets.

In general, the union operation could be either a set- or bag-union. But the simple bag-union technique from Section 6.3.3 of copying all the tuples from both arguments works in parallel, so we probably would not want to use the algorithm described here for a bag-union.
tuples in buckets, then we expect approximately the same number of tuples of \( R \cup S \) to be at each processor.

The operations of intersection and difference may be performed just like a union; it does not matter whether these are set or bag versions of these operations. Moreover:

- To take a join \( R(X, Y) \bowtie S(Y, Z) \), we hash the tuples of \( R \) and \( S \) to a number of buckets equal to the number of processors. However, the hash function \( h \) we use must depend only on the attributes of \( Y \), not all the attributes, so that joining tuples are always sent to the same bucket. As with union, we ship tuples of bucket \( i \) to processor \( i \). We may then perform the join at each processor using any of the uniprocessor join algorithms we have discussed in this chapter.

- To perform grouping and aggregation \( \gamma_L(R) \), we distribute the tuples of \( R \) using a hash function \( h \) that depends only on the grouping attributes in list \( L \). If each processor has all the tuples corresponding to one of the buckets of \( h \), then we can perform the \( \gamma_L \) operation on these tuples locally, using any uniprocessor algorithm.

### 6.10.4 Performance of Parallel Algorithms

Now, let us consider how the running time of a parallel algorithm on a \( p \)-processor machine compares with the time to execute an algorithm for the same operation on the same data, using a uniprocessor. The total work — disk I/O’s and processor cycles — cannot be smaller for a parallel machine than a uniprocessor. However, because there are \( p \) processors working with \( p \) disks, we can expect the elapsed, or wall-clock, time to be much smaller for the multiprocessor than for the uniprocessor.

A unary operation such as \( \sigma_C(R) \) can be completed in \( 1/p \) of the time it would take to perform the operation at a single processor, provided relation \( R \) is distributed evenly, as was supposed in Section 6.10.2. The number of disk I/O’s is essentially the same as for a uniprocessor selection. The only difference is that there will, on average, be \( p \) half-full blocks of \( R \), one at each processor, rather than a single half-full block of \( R \) had we stored all of \( R \) on one processor’s disk.

Now, consider a binary operation, such as join. We use a hash function on the join attributes that sends each tuple to one of \( p \) buckets, where \( p \) is the number of processors. To send the tuples of bucket \( i \) to processor \( i \), for all \( i \), we must read each tuple from disk to memory, compute the hash function, and ship all tuples except the one out of \( p \) tuples that happens to belong to the bucket at its own processor. If we are computing \( R(X, Y) \bowtie S(Y, Z) \), then we need to do \( B(R) + B(S) \) disk I/O’s to read all the tuples of \( R \) and \( S \) and determine their buckets.

We then must ship \((p-1)/p)(B(R)+B(S)) \) blocks of data across the network to their proper processors; only the \((1/p)B\) of the tuples already at the right
PARALLEL ALGORITHMS FOR RELATIONAL OPERATIONS

323

processor need not be shipped. The cost of shipment can be greater or less than the cost of the same number of disk I/O’s, depending on the architecture of the machine. However, we shall assume that shipment across the network is significantly cheaper than movement of data between disk and memory, because no physical motion is involved in shipment across a network, while it is for disk I/O.

In principle, we might suppose that the receiving processor has to store the data on its own disk, then execute a local join on the tuples received. For example, if we used a two-pass sort-join at each processor, a naive parallel algorithm would use $3(B(R) + B(S))/p$ disk I/O’s at each processor, since the sizes of the relations in each bucket would be approximately $B(R)/p$ and $B(S)/p$, and this type of join takes three disk I/O’s per block occupied by each of the argument relations. To this cost we would add another $2(B(R) + B(S))/p$ disk I/O’s per processor, to account for the first read of each tuple and the storing away of each tuple by the processor receiving the tuple during the hash and distribution of tuples. We should also add the cost of shipping the data, but we have elected to consider that cost negligible compared with the cost of disk I/O for the same data.

The above comparison demonstrates the value of the multiprocessor. While we do more disk I/O in total — five disk I/O’s per block of data, rather than three — the elapsed time, as measured by the number of disk I/O’s performed at each processor has gone down from $3(B(R) + B(S))/p$ to $5(B(R) + B(S))/p$, a significant win for large $p$.

Moreover, there are ways to improve the speed of the parallel algorithm so that the total number of disk I/O’s is not greater than what is required for a uniprocessor algorithm. In fact, since we operate on smaller relations at each processor, we may be able to use a local join algorithm that uses fewer disk I/O’s per block of data. For instance, even if $R$ and $S$ were so large that we need a two-pass algorithm on a uniprocessor, we may be able to use a one-pass algorithm on $(1/p)$th of the data.

We can avoid two disk I/O’s per block if, when we ship a block to the processor of its bucket, that processor can use the block immediately as part of its join algorithm. Most of the algorithms known for join and the other relational operators allow this use, in which case the parallel algorithm looks just like a multipass algorithm in which the first pass uses the hashing technique of Section 6.9.3.

Example 6.28: Consider our running example $R(X, Y) \bowtie S(Y, Z)$, where $R$ and $S$ occupy 1000 and 500 blocks, respectively. Now, let there be 10 buffers at each processor of a 10-processor machine. Also, assume that $R$ and $S$ are distributed uniformly among these 10 processors.

We begin by hashing each tuple of $R$ and $S$ to one of 10 “buckets,” using a hash function $h$ that depends only on the join attributes $Y$. These 10 “buckets” represent the 10 processors, and tuples are shipped to the processor corresponding to their “bucket.” The total number of disk I/O’s needed to read
**Biiig Mistake**

When using hash-based algorithms to distribute relations among processors and to execute operations, as in Example 6.28, we must be careful not to overuse one hash function. For instance, suppose we used a hash function \( h \) to hash the tuples of relations \( R \) and \( S \) among processors, in order to take their join. We might be tempted to use \( h \) to hash the tuples of \( S \) locally into buckets as we perform a one-pass hash-join at each processor. But if we do so, all those tuples will go to the same bucket, and the main-memory join suggested in Example 6.28 will be extremely inefficient.

The tuples of \( R \) and \( S \) is 1500, or 150 per processor. Each processor will have about 15 blocks worth of data for each other processor, so it ships 135 blocks to the other nine processors. The total communication is thus 1350 blocks.

We shall arrange that the processors ship the tuples of \( S \) before the tuples of \( R \). Since each processor receives about 50 blocks of tuples from 5, it can store those tuples in a main-memory data structure, using 50 of its 101 buffers. Then, when processors start sending \( R \)-tuples, each one is compared with the local \( S \)-tuples, and any resulting joined tuples are output.

In this way, the only cost of the join is 1500 disk I/O's, much less than for any other method discussed in this chapter. Moreover, the elapsed time is primarily the 150 disk I/O's performed at each processor, plus the time to ship tuples between processors and perform the main-memory computations. Note that 150 disk I/O's is less than 1/10th of the time to perform the same algorithm on a uniprocessor; we have not only gained because we had 10 processors working for us, but the fact that there are a total of 1010 buffers among those 10 processors gives us additional efficiency.

Of course, one might argue that had there been 1010 buffers at a single processor, then our example join could have been done in one pass, using 1500 disk I/O's. However, since multiprocessors usually have memory in proportion to the number of processors, we have only exploited two advantages of multi-processing simultaneously to get two independent speedups: one in proportion to the number of processors and one because the extra memory allows us to use a more efficient algorithm.

### 6.10.5 Exercises for Section 6.10

**Exercise 6.10.1**: Suppose that a disk I/O takes 100 milliseconds. Let \( B(R) = 100 \), so the disk I/O's for computing \( \sigma_C(R) \) on a uniprocessor machine will take about 10 seconds. What is the speedup if this selection is executed on a parallel machine with \( p \) processors, where:
* a) $p = 8.$  
b) $p = 100.$  
c) $p = 1000.$

**Exercise 6.10.2:** In Example 6.28 we described an algorithm that computed the join $R \times S$ in parallel by first hash-distributing the tuples among the processors and then performing a one-pass join at the processors. In terms of $B(R)$ and $B(S)$, the sizes of the relations involved, $p$ (the number of processors), and $M$ (the number of blocks of main memory at each processor), give the condition under which this algorithm can be executed successfully.

### 6.11 Summary of Chapter 6

- **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 7.

- **Extended Relational Algebra:** The usual operators of relational algebra — union, intersection, difference, selection, projection, product, and various forms of join — must be modified for a relational query processor by using bag (rather than set) forms of these operators. We must additionally add operators corresponding to the SQL operations of duplicate elimination, grouping and aggregation, and ordering.

- **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 functions, to open the construction of a relation, to get 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.

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.
Parallel Machines: Today's parallel machines can be characterized as shared-memory, shared-disk, or shared-nothing. For database applications, the shared-nothing architecture is generally the most cost-effective.

Parallel Algorithms: The operations of relational algebra can generally be sped up on a parallel machine by a factor close to the number of processors. The preferred algorithms start by hashing the data to buckets that correspond to the processors, and shipping data to the appropriate processor. Each processor then performs the operation on its local data.

6.12 References for Chapter 6

Two surveys of query optimization are [7] and [2]. An early study of join methods is in [6]. Buffer-pool management was analyzed, surveyed, and improved by [3].

The relational algebra dates from Codd's original paper on the relational model [4]. Our extension to the grouping operator and the generalization of Codd's projection are from [8].

The use of sort-based techniques was pioneered by [1]. The advantage of hash-based algorithms for join was expressed by [9] and [5]; the latter is the origin of the hybrid hash-join. The use of hashing in parallel join and other operations has been proposed several times. The earliest source we know of is [10].

