CHAPTER 15

Query Processing

Query processing refers to the range of activities involved in extracting data from a database. The activities include translation of queries in high-level database languages into expressions that can be used at the physical level of the file system, a variety of query-optimizing transformations, and actual evaluation of queries.

15.1 Overview

The steps involved in processing a query appear in Figure 15.1. The basic steps are:

1. Parsing and translation.
2. Optimization.
3. Evaluation.

Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use, but it is ill suited to be the system's internal representation of a query. A more useful internal representation is one based on the extended relational algebra.

Thus, the first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.[1] Most compiler texts cover parsing in detail.

[1] For materialized views, the expression defining the view has already been evaluated and stored. Therefore, the stored relation can be used, instead of uses of the view being replaced by the expression defining the view. Recursive views are handled differently, via a fixed-point procedure, as discussed in Section 5.4 and Section 27.4.7.

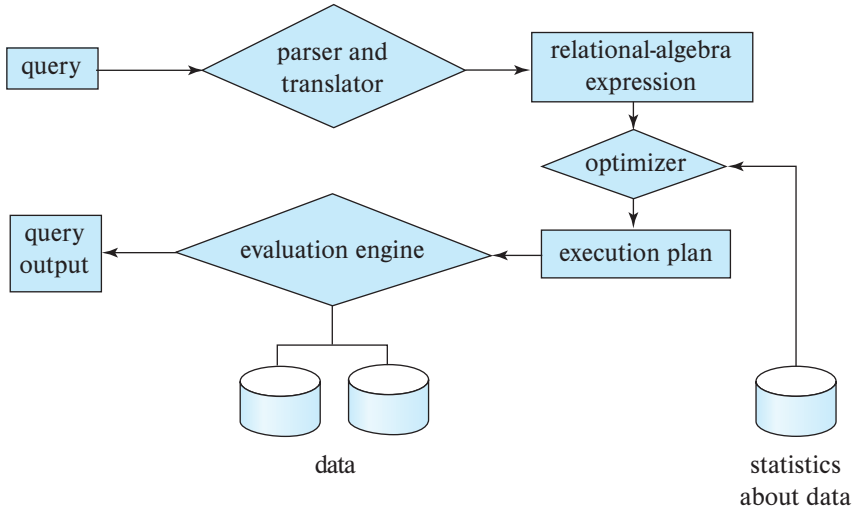

Figure 15.1 Steps in query processing.

Given a query, there are generally a variety of methods for computing the answer. For example, we have seen that, in SQL, a query could be expressed in several different ways. Each SQL query can itself be translated into a relational-algebra expression in one of several ways. Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions. As an illustration, consider the query:

select *salary*
from *instructor*
where *salary* < 75000;

This query can be translated into either of the following relational-algebra expressions:

- $\sigma_{salary<75000}\ (\Pi_{salary}\ (instructor))$
- $\Pi_{salary}\ (\sigma_{salary<75000}\ (instructor))$

Further, we can execute each relational-algebra operation by one of several different algorithms. For example, to implement the preceding selection, we can search every tuple in *instructor* to find tuples with salary less than 75000. If a B^+-tree index is available on the attribute *salary*, we can use the index instead to locate the tuples.

To specify fully how to evaluate a query, we need not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to evaluate each operation. Annotations may state the algorithm to be used for a specific operation or the particular index or indices to use. A relational-algebra operation annotated

with instructions on how to evaluate it is called an **evaluation primitive**. A sequence of primitive operations that can be used to evaluate a query is a **query-execution plan** or **query-evaluation plan**. Figure 15.2 illustrates an evaluation plan for our example query, in which a particular index (denoted in the figure as "index 1") is specified for the selection operation. The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.

The different evaluation plans for a given query can have different costs. We do not expect users to write their queries in a way that suggests the most efficient evaluation plan. Rather, it is the responsibility of the system to construct a query-evaluation plan that minimizes the cost of query evaluation; this task is called *query optimization*. Chapter 16 describes query optimization in detail.

Once the query plan is chosen, the query is evaluated with that plan, and the result of the query is output.

The sequence of steps already described for processing a query is representative; not all databases exactly follow those steps. For instance, instead of using the relational-algebra representation, several databases use an annotated parse-tree representation based on the structure of the given SQL query. However, the concepts that we describe here form the basis of query processing in databases.

In order to optimize a query, a query optimizer must know the cost of each operation. Although the exact cost is hard to compute, since it depends on many parameters such as actual memory available to the operation, it is possible to get a rough estimate of execution cost for each operation.

In this chapter, we study how to evaluate individual operations in a query plan and how to estimate their cost; we return to query optimization in Chapter 16. Section 15.2 outlines how we measure the cost of a query. Section 15.3 through Section 15.6 cover the evaluation of individual relational-algebra operations. Several operations may be grouped together into a **pipeline**, in which each of the operations starts working on its input tuples even as they are being generated by another operation. In Section 15.7, we examine how to coordinate the execution of multiple operations in a query evaluation

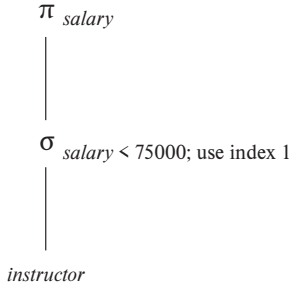

Figure 15.2 A query-evaluation plan.

plan, in particular, how to use pipelined operations to avoid writing intermediate results to disk.

15.2 Measures of Query Cost

There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan. To do so, we must estimate the cost of individual operations and combine them to get the cost of a query evaluation plan. Thus, as we study evaluation algorithms for each operation later in this chapter, we also outline how to estimate the cost of the operation.

The cost of query evaluation can be measured in terms of a number of different resources, including disk accesses, CPU time to execute a query, and, in parallel and distributed database systems, the cost of communication. (We discuss parallel and distributed database systems in Chapter 21 through Chapter 23.)

For large databases resident on magnetic disk, the I/O cost to access data from disk usually dominates the other costs; thus, early cost models focused on the I/O cost when estimating the cost of query operations. However, with flash storage becoming larger and less expensive, most organizational data today can be stored on solid-state drives (SSDs) in a cost effective manner. In addition, main memory sizes have increased significantly, and the cost of main memory has decreased enough in recent years that for many organizations, organizational data can be stored cost-effectively in main memory for querying, although it must of course be stored on flash or magnetic storage to ensure persistence.

With data resident in-memory or on SSDs, I/O cost does not dominate the overall cost, and we must include CPU costs when computing the cost of query evaluation. We do not include CPU costs in our model to simplify our presentation, but note that they can be approximated by simple estimators. For example, the cost model used by PostgreSQL (as of 2018) includes (i) a CPU cost per tuple, (ii) a CPU cost for processing each index entry (in addition to the I/O cost), and (iii) a CPU cost per operator or function (such as arithmetic operators, comparison operators, and related functions). The database has default values for each of these costs, which are multiplied by the number of tuples processed, the number of index entries processed, and the number of operators and functions executed, respectively. The defaults can be changed as a configuration parameter.

We use the *number of blocks transferred* from storage and the *number of random I/O accesses*, each of which will require a disk seek on magnetic storage, as two important factors in estimating the cost of a query-evaluation plan. If the disk subsystem takes an average of t_T seconds to transfer a block of data and has an average block-access time (disk seek time plus rotational latency) of t_S seconds, then an operation that transfers b blocks and performs S random I/O accesses would take $b * t_T + S * t_S$ seconds.

The values of t_T and t_S must be calibrated for the disk system used. We summarize performance data here; see Chapter 12 for full details on storage systems. Typical values for high-end magnetic disks in the year 2018 would be $t_S = 4$ milliseconds and $t_T = 0.1$

milliseconds, assuming a 4-kilobyte block size and a transfer rate of 40 megabytes per second.[2]

Although SSDs do not perform a physical seek operation, they have an overhead for initiating an I/O operation; we refer to the latency from the time an I/O request is made to the time when the first byte of data is returned as t_S. For mid-range SSDs in 2018 using the SATA interface, t_S is around 90 microseconds, while the transfer time t_T is about 10 microseconds for a 4-kilobyte block. Thus, SSDs can support about 10,000 random 4-kilobyte reads per second, and they support 400 megabytes/second throughput on sequential reads using the standard SATA interface. SSDs using the PCIe 3.0x4 interface have smaller t_S, of 20 to 60 microseconds, and much higher transfer rates of around 2 gigabytes/second, corresponding to t_T of 2 microseconds, allowing around 50,000 to 15,000 random 4-kilobyte block reads per second, depending on the model.[3]

For data that are already present in main memory, reads happen at the unit of cache lines, instead of disk blocks. But assuming entire blocks of data are read, the time to transfer t_T for a 4-kilobyte block is less than 1 microsecond for data in memory. The latency to fetch data from memory, t_S, is less than 100 nanoseconds.

Given the wide diversity of speeds of different storage devices, database systems must ideally perform test seeks and block transfers to estimate t_S and t_T for specific systems/storage devices, as part of the software installation process. Databases that do not automatically infer these numbers often allow users to specify the numbers as part of configuration files.

We can refine our cost estimates further by distinguishing block reads from block writes. Block writes are typically about twice as expensive as reads on magnetic disks, since disk systems read sectors back after they are written to verify that the write was successful. On PCIe flash, write throughput may be about 50 percent less than read throughput, but the difference is almost completely masked by the limited speed of SATA interfaces, leading to write throughput matching read throughput. However, the throughput numbers do not reflect the cost of erases that are required if blocks are overwritten. For simplicity, we ignore this detail.

The cost estimates we give do not include the cost of writing the final result of an operation back to disk. These are taken into account separately where required.

---

[2]Storage device specifications often mention the transfer rate, and the number of random I/O operations that can be carried out in 1 second. The values t_T can be computed as block size divided by transfer rate, while t_S can be computed as $(1/N) - t_T$, where N is the number of random I/O operations per second that the device supports, since a random I/O operation performs a random I/O access, followed by data transfer of 1 block.

[3]The I/O operations per second number used here are for the case of sequential I/O requests, usually denoted as QD-1 in the SSD specifications. SSDs can support multiple random requests in parallel, with 32 to 64 parallel requests being commonly supported; an SSD with SATA interface supports nearly 100,000 random 4-kilobyte block reads in a second if multiple requests are sent in parallel, while PCIe disks can support over 350,000 random 4-kilobyte block reads per second; these numbers are referred to as the QD-32 or QD-64 numbers depending on how many requests are sent in parallel. We do not explore parallel requests in our cost model, since we only consider sequential query processing algorithms in this chapter. Shared-memory parallel query processing techniques, discussed in Section 22.6, can be used to exploit the parallel request capabilities of SSDs.

The costs of all the algorithms that we consider depend on the size of the buffer in main memory. In the best case, if data fits in the buffer, the data can be read into the buffers, and the disk does not need to be accessed again. In the worst case, we may assume that the buffer can hold only a few blocks of data—approximately one block per relation. However, with large main memories available today, such worst-case assumptions are overly pessimistic. In fact, a good deal of main memory is typically available for processing a query, and our cost estimates use the amount of memory available to an operator, M, as a parameter. In PostgreSQL the total memory available to a query, called the effective cache size, is assumed by default to be 4 gigabytes, for the purpose of cost estimation; if a query has several operators that run concurrently, the available memory has to be divided amongst the operators.

In addition, although we assume that data must be read from disk initially, it is possible that a block that is accessed is already present in the in-memory buffer. Again, for simplicity, we ignore this effect; as a result, the actual disk-access cost during the execution of a plan may be less than the estimated cost. To account (at least partially) for buffer residence, PostgreSQL uses the following "hack": the cost of a random page read is assumed to be $1/10^{th}$ of the actual random page read cost, to model the situation that 90% of reads are found to be resident in cache. Further, to model the situation that internal nodes of B^+-tree indices are traversed often, most database systems assume that all internal nodes are present in the in-memory buffer, and assume that a traversal of an index only incurs a single random I/O cost for the leaf node.

The **response time** for a query-evaluation plan (that is, the wall-clock time required to execute the plan), assuming no other activity is going on in the computer, would account for all these costs, and could be used as a measure of the cost of the plan. Unfortunately, the response time of a plan is very hard to estimate without actually executing the plan, for the following two reasons:

1. The response time depends on the contents of the buffer when the query begins execution; this information is not available when the query is optimized and is hard to account for even if it were available.
2. In a system with multiple disks, the response time depends on how accesses are distributed among disks, which is hard to estimate without detailed knowledge of data layout on disk.

Interestingly, a plan may get a better response time at the cost of extra resource consumption. For example, if a system has multiple disks, a plan A that requires extra disk reads, but performs the reads in parallel across multiple disks may, finish faster than another plan B that has fewer disk reads, but performs reads from only one disk at a time. However, if many instances of a query using plan A run concurrently, the overall response time may actually be more than if the same instances are executed using plan B, since plan A generates more load on the disks.

As a result, instead of trying to minimize the response time, optimizers generally try to minimize the total **resource consumption** of a query plan. Our model of estimating

the total disk access time (including seek and data transfer) is an example of such a resource consumption–based model of query cost.

15.3 Selection Operation

In query processing, the **file scan** is the lowest-level operator to access data. File scans are search algorithms that locate and retrieve records that fulfill a selection condition. In relational systems, a file scan allows an entire relation to be read in those cases where the relation is stored in a single, dedicated file.

15.3.1 Selections Using File Scans and Indices

Consider a selection operation on a relation whose tuples are stored together in one file. The most straightforward way of performing a selection is as follows:

- **A1** (**linear search**). In a linear search, the system scans each file block and tests all records to see whether they satisfy the selection condition. An initial seek is required to access the first block of the file. In case blocks of the file are not stored contiguously, extra seeks may be required, but we ignore this effect for simplicity.

  Although it may be slower than other algorithms for implementing selection, the linear-search algorithm can be applied to any file, regardless of the ordering of the file, or the availability of indices, or the nature of the selection operation. The other algorithms that we shall study are not applicable in all cases, but when applicable they are generally faster than linear search.

Cost estimates for linear scan, as well as for other selection algorithms, are shown in Figure 15.3. In the figure, we use h_i to represent the height of the B^+-tree, and assume a random I/O operation is required for each node in the path from the root to a leaf. Most real-life optimizers assume that the internal nodes of the tree are present in the in-memory buffer since they are frequently accessed, and usually less than 1 percent of the nodes of a B^+-tree are nonleaf nodes. The cost formulae can be correspondingly simplified, charging only one random I/O cost for a traversal from the root to a leaf, by setting $h_i = 1$.

Index structures are referred to as **access paths**, since they provide a path through which data can be located and accessed. In Chapter 14, we pointed out that it is efficient to read the records of a file in an order corresponding closely to physical order. Recall that a *clustering index* (also referred to as a *primary index*) is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file. An index that is not a clustering index is called a *secondary index* or a *nonclustering index*.

Search algorithms that use an index are referred to as **index scans**. We use the selection predicate to guide us in the choice of the index to use in processing the query. Search algorithms that use an index are:

	Algorithm	Cost	Reason
A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies the condition, scan can be terminated as soon as the required record is found. In the worst case, b_r block transfers are still required.
A2	Clustering B^+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.
A3	Clustering B^+-tree Index, Equality on Non-key	$h_i * (t_T + t_S) + t_S + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a clustering index) and don't require additional seeks.
A4	Secondary B^+-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	This case is similar to clustering index.
A4	Secondary B^+-tree Index, Equality on Non-key	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5	Clustering B^+-tree Index, Comparison	$h_i * (t_T + t_S) + t_S + b * t_T$	Identical to the case of A3, equality on non-key.
A6	Secondary B^+-tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on non-key.

Figure 15.3 Cost estimates for selection algorithms.

- **A2 (clustering index, equality on key)**. For an equality comparison on a key attribute with a clustering index, we can use the index to retrieve a single record that satisfies the corresponding equality condition. Cost estimates are shown in Figure 15.3. To model the common situation that the internal nodes of the index are in the in-memory buffer, h_i can be set to 1.
- **A3 (clustering index, equality on non-key)**. We can retrieve multiple records by using a clustering index when the selection condition specifies an equality comparison on a non-key attribute, A. The only difference from the previous case is that multiple records may need to be fetched. However, the records must be stored consecutively in the file since the file is sorted on the search key. Cost estimates are shown in Figure 15.3.
- **A4 (secondary index, equality)**. Selections specifying an equality condition can use a secondary index. This strategy can retrieve a single record if the equality condition is on a key; multiple records may be retrieved if the indexing field is not a key.

  In the first case, only one record is retrieved. The cost in this case is the same as that for a clustering index (case A2).

  In the second case, each record may be resident on a different block, which may result in one I/O operation per retrieved record, with each I/O operation requiring a seek and a block transfer. The worst-case cost in this case is $(h_i + n) * (t_S + t_T)$, where n is the number of records fetched, if each record is in a different disk block, and the block fetches are randomly ordered. The worst-case cost could become even worse than that of linear search if a large number of records are retrieved.

  If the in-memory buffer is large, the block containing the record may already be in the buffer. It is possible to construct an estimate of the *average* or *expected* cost of the selection by taking into account the probability of the block containing the record already being in the buffer. For large buffers, that estimate will be much less than the worst-case estimate.

In certain algorithms, including A2, the use of a B^+-tree file organization can save one access since records are stored at the leaf level of the tree.

As described in Section 14.4.2, when records are stored in a B^+-tree file organization or other file organizations that may require relocation of records, secondary indices usually do not store pointers to the records.[4] Instead, secondary indices store the values of the attributes used as the search key in a B^+-tree file organization. Accessing a record through such a secondary index is then more expensive: First the secondary index is searched to find the B^+-tree file organization search-key values, then the B^+-tree file organization is looked up to find the records. The cost formulae described for secondary indices have to be modified appropriately if such indices are used.

[4]Recall that if B^+-tree file organizations are used to store relations, records may be moved between blocks when leaf nodes are split or merged, and when records are redistributed.

15.3.2 Selections Involving Comparisons

Consider a selection of the form $\sigma_{A \leq v}(r)$. We can implement the selection either by using linear search or by using indices in one of the following ways:

- **A5 (clustering index, comparison)**. A clustering ordered index (for example, a clustering B^+-tree index) can be used when the selection condition is a comparison. For comparison conditions of the form $A > v$ or $A \geq v$, a clustering index on A can be used to direct the retrieval of tuples, as follows: For $A \geq v$, we look up the value v in the index to find the first tuple in the file that has a value of $A \geq v$. A file scan starting from that tuple up to the end of the file returns all tuples that satisfy the condition. For $A > v$, the file scan starts with the first tuple such that $A > v$. The cost estimate for this case is identical to that for case A3.

  For comparisons of the form $A < v$ or $A \leq v$, an index lookup is not required. For $A < v$, we use a simple file scan starting from the beginning of the file, and continuing up to (but not including) the first tuple with attribute $A = v$. The case of $A \leq v$ is similar, except that the scan continues up to (but not including) the first tuple with attribute $A > v$. In either case, the index is not useful.

- **A6 (secondary index, comparison)**. We can use a secondary ordered index to guide retrieval for comparison conditions involving $<$, $\leq$, $\geq$, or $>$. The lowest-level index blocks are scanned, either from the smallest value up to v (for $<$ and $\leq$), or from v up to the maximum value (for $>$ and $\geq$).

  The secondary index provides pointers to the records, but to get the actual records we have to fetch the records by using the pointers. This step may require an I/O operation for each record fetched, since consecutive records may be on different disk blocks; as before, each I/O operation requires a disk seek and a block transfer. If the number of retrieved records is large, using the secondary index may be even more expensive than using linear search. Therefore, the secondary index should be used only if very few records are selected.

As long as the number of matching tuples is known ahead of time, a query optimizer can choose between using a secondary index or using a linear scan based on the cost estimates. However, if the number of matching tuples is not known accurately at compilation time, either choice may lead to bad performance, depending on the actual number of matching tuples.

To deal with the above situation, PostgreSQL uses a hybrid algorithm that it calls a *bitmap index scan*,[5] when a secondary index is available, but the number of matching records is not known precisely. The bitmap index scan algorithm first creates a bitmap with as many bits as the number of blocks in the relation, with all bits initialized to 0. The algorithm then uses the secondary index to find index entries for matching tuples, but instead of fetching the tuples immediately, it does the following. As each index

[5]This algorithm should not be confused with a scan using a bitmap index.

entry is found, the algorithm gets the block number from the index entry, and sets the corresponding bit in the bitmap to 1.

Once all index entries have been processed, the bitmap is scanned to find all blocks whose bit is set to 1. These are exactly the blocks containing matching records. The relation is then scanned linearly, but blocks whose bit is not set to 1 are skipped; only blocks whose bit is set to 1 are fetched, and then a scan within each block is used to retrieve all matching records in the block.

In the worst case, this algorithm is only slightly more expensive than linear scan, but in the best case it is much cheaper than linear scan. Similarly, in the worst case it is only slightly more expensive than using a secondary index scan to directly fetch tuples, but in the best case it is much cheaper than a secondary index scan. Thus, this hybrid algorithm ensures that performance is never much worse than the best plan for that database instance.

A variant of this algorithm collects all the index entries, and sorts them (using sorting algorithms which we study later in this chapter), and then performs a relation scan that skips blocks that do not have any matching entries. Using a bitmap as above can be cheaper than sorting the index entries.

15.3.3 Implementation of Complex Selections

So far, we have considered only simple selection conditions of the form *A op B*, where *op* is an equality or comparison operation. We now consider more complex selection predicates.

- **Conjunction:** A **conjunctive selection** is a selection of the form:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

- **Disjunction:** A **disjunctive selection** is a selection of the form:

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

  A disjunctive condition is satisfied by the union of all records satisfying the individual, simple conditions θ_i.

- **Negation:** The result of a selection $\sigma_{\neg\theta}(r)$ is the set of tuples of r for which the condition θ evaluates to false. In the absence of nulls, this set is simply the set of tuples in r that are not in $\sigma_\theta(r)$.

We can implement a selection operation involving either a conjunction or a disjunction of simple conditions by using one of the following algorithms:

- **A7 (conjunctive selection using one index).** We first determine whether an access path is available for an attribute in one of the simple conditions. If one is, one of the

selection algorithms A2 through A6 can retrieve records satisfying that condition. We complete the operation by testing, in the memory buffer, whether or not each retrieved record satisfies the remaining simple conditions.

To reduce the cost, we choose a θ_i and one of algorithms A1 through A6 for which the combination results in the least cost for $\sigma_{\theta_i}(r)$. The cost of algorithm A7 is given by the cost of the chosen algorithm.

- **A8 (conjunctive selection using composite index)**. An appropriate **composite index** (that is, an index on multiple attributes) may be available for some conjunctive selections. If the selection specifies an equality condition on two or more attributes, and a composite index exists on these combined attribute fields, then the index can be searched directly. The type of index determines which of algorithms A2, A3, or A4 will be used.

- **A9 (conjunctive selection by intersection of identifiers)**. Another alternative for implementing conjunctive selection operations involves the use of record pointers or record identifiers. This algorithm requires indices with record pointers, on the fields involved in the individual conditions. The algorithm scans each index for pointers to tuples that satisfy an individual condition. The intersection of all the retrieved pointers is the set of pointers to tuples that satisfy the conjunctive condition. The algorithm then uses the pointers to retrieve the actual records. If indices are not available on all the individual conditions, then the algorithm tests the retrieved records against the remaining conditions.

  The cost of algorithm A9 is the sum of the costs of the individual index scans, plus the cost of retrieving the records in the intersection of the retrieved lists of pointers. This cost can be reduced by sorting the list of pointers and retrieving records in the sorted order. Thereby, (1) all pointers to records in a block come together, hence all selected records in the block can be retrieved using a single I/O operation, and (2) blocks are read in sorted order, minimizing disk-arm movement. Section 15.4 describes sorting algorithms.

- **A10 (disjunctive selection by union of identifiers)**. If access paths are available on all the conditions of a disjunctive selection, each index is scanned for pointers to tuples that satisfy the individual condition. The union of all the retrieved pointers yields the set of pointers to all tuples that satisfy the disjunctive condition. We then use the pointers to retrieve the actual records.

  However, if even one of the conditions does not have an access path, we have to perform a linear scan of the relation to find tuples that satisfy the condition. Therefore, if there is even one such condition in the disjunct, the most efficient access method is a linear scan, with the disjunctive condition tested on each tuple during the scan.

The implementation of selections with negation conditions is left to you as an exercise (Practice Exercise 15.6).