APPENDIX D

Storage Structures and Access Methods

D.1 INTRODUCTION

In this appendix we present a tutorial survey of techniques typically used in today's systems for physically representing and accessing the database on the disk. (*Note:* We use the term *disk* throughout to stand generically for all direct-access media, including, for example, RAID arrays, mass storage, optical disks, and so forth, as well as conventional moving-head magnetic disks *per se*.) You are assumed to have a basic familiarity with disk architecture and to understand what is meant by the terms *seek time, rotational delay, cylinder, track, read/write head,* and so on. Good tutorials on such material can be found in many places; see, for example, reference [D.4].

The basic point motivating all storage-structure and access-method technology is that disk access times are *much* slower than main-memory access times. Typical seek times and rotational delays are both on the order of 5 or 6 milliseconds or so, and typical data transfer rates are somewhere in the range 5 to 10 million bytes per second; main-memory access is thus likely to be at least four or five orders of magnitude faster than disk access

on any given system. An overriding performance objective is thus to **minimize the number of disk accesses** (or disk I/O's). This appendix is concerned with techniques for achieving that objective—that is, techniques for arranging data on the disk so that any required piece of data, say some specific record, can be located in as few I/O's as possible. *Note:* Throughout this appendix, since our discussions are all at the storage level, we use storage-level terminology (in particular, *files, records,* and *fields,* meaning *stored* files, records, and fields, respectively) in place of relational terminology.

As already suggested, any given arrangement of data on the disk is referred to as a **storage structure**. Many different storage structures can be and have been devised, and of course different structures have different performance characteristics; some are good for some applications, others are good for others. There is probably no single structure that is optimal for all possible applications. It follows that a good system should support a variety of different structures, so that different portions of the database can be stored in different ways, and the storage structure for a given portion can be changed as performance requirements change or become better understood.

The structure of the appendix is as follows. Following this introductory section, Section D.2 explains in outline what is involved in the overall process of locating and accessing some particular record, and identifies the major software components involved in that process. Section D.3 then goes into a little more detail on two of those components, the *file manager* and the *disk manager.* Those two sections (D.2 and D.3) need only be skimmed on a first reading; a lot of the detail they contain is not really required for an understanding of the subsequent material. The next four sections (D.4–D.7) should not just be skimmed, however, since they represent the most important part of the entire discussion; they describe some of the most commonly occurring storage structures found in present-day systems, under the headings "Indexing," "Hashing," "Pointer Chains," and "Compression Techniques," respectively. Finally, Section D.8 presents a summary and a brief conclusion.

Note: The emphasis throughout is on *concepts,* not detail. The objective is to explain the general idea behind such notions as indexing, hashing, and so on without getting too bogged down in the specifics of any one particular system or technique. If you want such detail, see the books and papers listed in the "References and Bibliography" section at the end of the appendix.

D.2 DATABASE ACCESS: AN OVERVIEW

Before we get into our discussion of storage structures *per se,* we first need to consider what is involved in the overall process of data access in general. Locating a specific piece of data in the database and presenting it to the user involves several distinct layers of software. Of course, the details of those layers vary considerably from system to system, as does the terminology, but the principles are fairly standard and can be explained in outline as follows (refer to Fig. D.1).

1. First, the DBMS determines what record is required, and asks the **file manager** to retrieve that record. (We assume for the purposes of this simple explanation that the

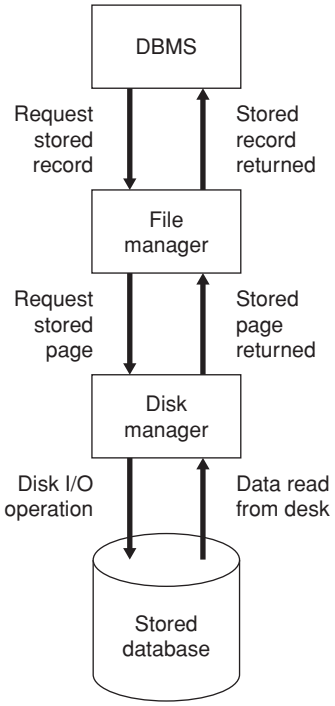

Fig. D.1 The DBMS, file manager, and disk manager

DBMS is able to pinpoint the exact record desired ahead of time. In practice it might need to retrieve a set of several records and search through those records in main memory to find the specific one desired. In principle, however, this only means that the sequence of steps 1–3 must be repeated for each record in that set.)

2. The file manager in turn determines what page contains the desired record, and asks the **disk manager** to retrieve that page.
3. Finally, the disk manager determines the physical location of the desired page on the disk, and issues the necessary disk I/O request. *Note:* Sometimes, of course, the required page will already be in a buffer in main memory as the result of a previous retrieval, in which case it obviously should not be necessary to retrieve it again.

Loosely speaking, therefore, the DBMS has a view of the database as a collection of records, and that view is supported by the file manager; the file manager, in turn, has a view of the database as a collection of pages, and that view is supported by the disk manager; and the disk manager has a view of the disk "as it really is." The next three subsections amplify these ideas somewhat. Section D.3 then goes into more detail on the same topics.

Disk Manager

The disk manager is a component of the underlying operating system. It is the component responsible for all physical I/O operations (in some systems it is referred to as the "basic I/O services" component). As such, it clearly needs to be aware of **physical disk addresses**. For example, when the file manager asks to retrieve some specific page *p,* the disk manager needs to know exactly where page *p* is on the disk. However, the user of the disk manager—namely, the file manager—does *not* need to know that information. Instead, the file manager regards the disk simply as a logical collection of **page sets**, each consisting of a collection of fixed-size pages. Each page set is identified by a unique **page-set ID**. Each page, in turn, is identified by a **page number** that is unique within the disk; distinct page sets are disjoint (i.e., do not have any pages in common). The mapping between page numbers and physical disk addresses is understood and maintained by the disk manager. The major advantage of this arrangement (not the only one) is that all device-specific code can be isolated within a single system component (namely, the disk manager), and all higher-level components—in particular, the file manager—can thus be *device-independent.*

As just explained, the complete set of pages on the disk is divided into a collection of disjoint subsets called page sets. One of those page sets, the **free-space** page set, serves as a pool of available (i.e., currently unused) pages; the others are all considered to contain significant data. The allocation and deallocation of pages to and from page sets is performed by the disk manager in response to requests from the file manager. The operations supported by the disk manager on page sets—that is, the operations the file manager is able to request—include the following:

- Retrieve page *p* from page set *s*.
- Replace page *p* within page set *s*.
- Add a new page to page set *s* (i.e., acquire an empty page from the free-space page set and return the new page number *p*).
- Remove page *p* from page set *s* (i.e., return page *p* to the free-space page set).

The first two of these operations are of course the basic page-level I/O operations the file manager needs. The other two allow page sets to grow and shrink as necessary.

File Manager

The file manager uses the disk manager facilities just described in such a way as to permit its user—that is, the DBMS—to regard the disk as a collection of **files**. Each page set will contain zero or more files. *Note:* The DBMS might need to be aware of the existence of page sets, even though it is not responsible for managing them in detail, for reasons indicated in the next subsection. In particular, the DBMS might need to know when two files share the same page set or when two records share the same page.

Each file is identified by a **file name** or **file ID**, unique at least within its containing page set; each record in turn is identified by a **record number** or **record ID** (RID), unique at least within its containing file. (In practice, record IDs are usually unique not just within their containing file but actually within the entire disk, since they typically

consist of the combination of a page number and some value that is unique within that page. See Section D.3.)

The operations supported by the file manager on files—that is, the operations the DBMS is able to request—include the following:

- Retrieve record *r* from file *f*.
- Replace record *r* within file *f*.
- Add a new record to file *f* and return the new record ID *r*.
- Remove record *r* from file *f*.
- Create a new file *f*.
- Destroy file *f*.

Using these primitive file management operations, the DBMS is able to build and manipulate the storage structures that are the principal concern of this appendix (see Sections D.4–D.7).

Note: In some systems the file manager is a component of the underlying operating system; in others it is packaged with the DBMS. For our purposes the distinction is not important. However, we remark in passing that although operating systems do invariably provide such a component, it is often the case that the general-purpose file manager provided by the operating system is not ideally suited to the requirements of the special-purpose "application" that is the DBMS. For more discussion of this topic, see reference [D.44].

Clustering

We should not leave this overview discussion without a brief mention of the subject of *data clustering*. The basic idea behind clustering is to try to store records that are logically related (and therefore frequently used together) physically close together on the disk. Physical data clustering is an extremely important factor in performance, as can easily be seen from the following. Suppose the record most recently accessed is *r1,* and suppose the next record required is *r2*. Suppose also that *r1* is stored on page *p1* and *r2* is stored on page *p2*. Then:

1. If *p1* and *p2* are one and the same, then access to *r2* will not require any physical I/O at all, because the desired page *p2* will already be in a buffer in main memory.
2. If *p1* and *p2* are distinct but physically close together—in particular, if they are physically adjacent—then access to *r2* will require a physical I/O (unless, of course, *p2* also happens to be in a buffer in main memory), but the seek time involved in that I/O will be small, because the read/write heads will already be close to the desired position. In particular, the seek time will be *zero* if *p1* and *p2* are in the same cylinder.

   As an example of clustering, we consider the usual suppliers-and-parts database:[1]

[1] We assume for simplicity throughout this appendix that each individual supplier, part, or shipment tuple maps to a single record on the disk, as would typically be the case in most commercial systems today. See Appendix A for further discussion.

- If sequential access to all suppliers in supplier number order is a frequent application requirement, then the supplier records should be clustered such that the supplier S1 record is physically close to the supplier S2 record, the supplier S2 record is physically close to the supplier S3 record, and so on. This is an example of **intra-file** clustering: The clustering is applied within a single file.
- If, on the other hand, access to some specific supplier together with all shipments for that supplier is a frequent application requirement, then supplier and shipment records should be stored interleaved, with the shipment records for supplier S1 physically close to the supplier S1 record, the shipment records for supplier S2 physically close to the supplier S2 record, and so on. This is an example of **inter-file** clustering: The clustering is applied across more than one file.

Of course, a given file or set of files can be physically clustered in at most one way at any given time.

The DBMS can support clustering, both intra- and inter-file, by storing logically related records on the same page where possible and on adjacent pages where not (this is why the DBMS might need to know about pages as well as files). When the DBMS creates a new record, the file manager must allow it to specify that the new record be stored "near"—that is, on the same page as, or at least on a page logically near—some existing record. The disk manager, in turn, will do its best to ensure that two pages that are logically adjacent are physically adjacent on the disk. See Section D.3.

In general, of course, the DBMS can only know what clustering is required if the database administrator is able to tell it. A good DBMS should allow the DBA to specify different kinds of clustering for different files. It should also allow the clustering for a given file or set of files to be changed if the performance requirements change. Furthermore, of course, any such change in physical clustering should obviously not require any concomitant changes in application programs, if data independence is to be achieved.

D.3 PAGE SETS AND FILES

As explained in the previous section, a major function of the disk manager is to allow the file manager to ignore all details of physical disk I/O and to think in terms of (logical) "page I/O" instead. This function of the disk manager is referred to as **page management**. We present a very simple example to show how page management is typically handled.

Consider the suppliers-and-parts database once again. Suppose the desired logical ordering of records is (loosely) *primary key sequence*—that is, suppliers are required to be in supplier number order, parts in part number order, and shipments in part number order within supplier number order.[2] To keep matters simple, suppose too that each file is stored in a page set of its own, and that each record requires an entire page of its own. Now consider the following sequence of events.

[2] We say "loosely" because the term *primary key sequence* is not well-defined if the primary key is composite. In the case of shipments, for example, it might mean either part number order within supplier number order or the other way about. (In any case, primary keys are a relational concept, not a file-level concept, so we really ought not to be talking about primary keys at all at the storage level.)

1. Initially the database contains no data at all. There is only one page set, the free-space page set, which contains all pages on the disk—except for page zero, which is special (see later). The remaining pages are numbered sequentially from one.
2. The file manager requests the creation of a page set for supplier records, and inserts the five supplier records for suppliers S1–S5. The disk manager removes pages 1–5 from the free-space page set and labels them "the suppliers page set."
3. Similarly for parts and shipments. Now there are four page sets: the suppliers page set (pages 1–5), the parts page set (pages 6–11), the shipments page set (pages 12–23), and the free-space page set (pages 24, 25, 26, and so on). The situation at this point is as shown in Fig. D.2.

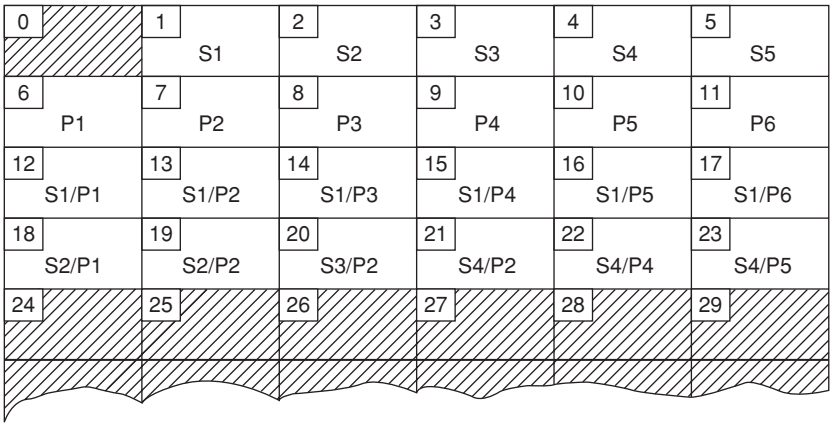

Fig. D.2 Disk layout after creation and initial loading of the suppliers-and-parts database

To continue with the example:

4. Next, the file manager inserts a new supplier record (for a new supplier, supplier S6). The disk manager locates the first free page in the free-space page set—namely, page 24—and adds it to the suppliers page set.
5. The file manager deletes the record for supplier S2. The disk manager returns the page for supplier S2 (page 2) to the free-space page set.
6. The file manager inserts a new part record (for part P7). The disk manager locates the first free page in the free-space page set—namely, page 2—and adds it to the parts page set.
7. The file manager deletes the record for supplier S4. The disk manager returns the page for supplier S4 (page 4) to the free-space page set.

And so on. The situation at this juncture is as illustrated in Fig. D.3. The key point is the following: After the system has been running for a while, it can no longer be guaranteed that pages that are logically adjacent are still physically adjacent, even if they started

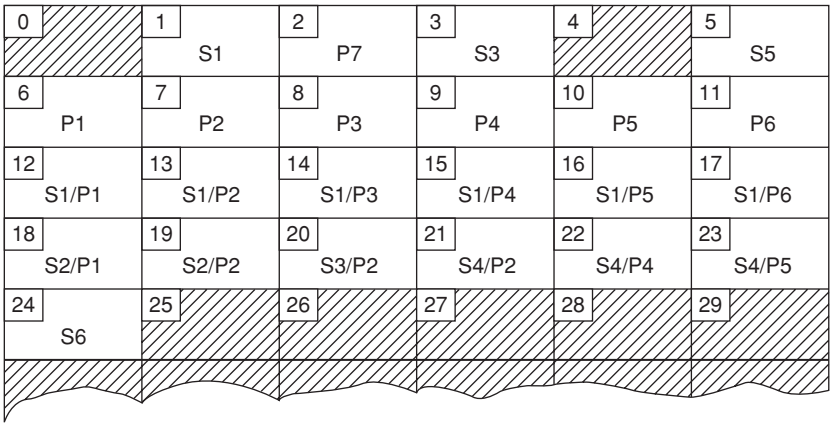

Fig. D.3 Disk layout after inserting supplier S6, deleting supplier S2, inserting part P7, and deleting supplier S4

out that way. For this reason, the logical sequence of pages in a given page set must be represented not by physical adjacency but by *pointers*. Each page will contain a **page header**—that is, a set of control information that includes (among other things) the physical disk address of the page that immediately follows that page in logical sequence. See Fig. D.4.

Points arising from the foregoing example:

- The page headers—in particular, the "next page" pointers—are managed by the disk manager; they should be completely invisible to the file manager.

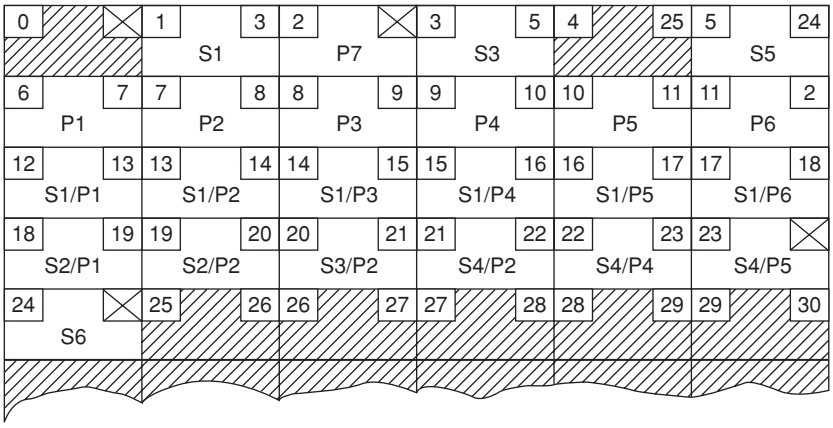

Fig. D.4 Fig. D.3 revised to show "next page" pointers (top right corner of each page)

- As explained in the subsection on clustering at the end of Section D.2, it is desirable to store logically adjacent pages in physically adjacent locations on the disk (as far as possible). For this reason, the disk manager normally allocates and deallocates pages to and from page sets, not one at a time as suggested in the example, but rather in physically contiguous groups or **extents** of (say) 64 pages at a time.
- The question arises: How does the disk manager know where the various page sets are located?—or, more precisely, how does it know, for each page set, where the (logically) first page of that page set is located? (It is sufficient to locate the first page, of course, because the second and subsequent pages can then be located by following the pointers in the page headers.) The answer is that some fixed location on the disk—typically cylinder zero, track zero—is used to store a page that gives precisely that information. That page (variously referred to as the *disk table of contents,* the *disk directory,* the *page set directory,* or simply *page zero*) thus typically contains a list of the page sets currently in existence on the disk, together with a pointer to the first page of each such page set. See Fig. D.5.

Now we turn to the file manager. Just as the disk manager allows the file manager to ignore details of physical disk I/O and to think for the most part in terms of logical pages, so the file manager allows the DBMS to ignore details of page I/O and to think for the most part in terms of files and records. This function of the file manager is referred to as **record management**. We discuss that function very briefly here, once again taking the suppliers-and-parts database as the basis for our examples.

Suppose, then (rather more realistically now), that a single page can accommodate several records, instead of just one as in the page management example. Suppose too that

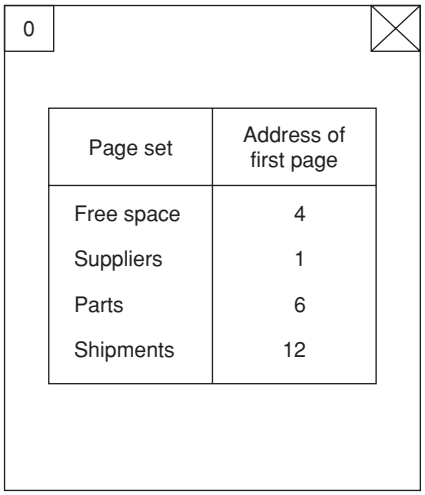

Fig. D.5 The disk directory ("page zero")

the desired logical order for supplier records is supplier number order, as before. Consider the following sequence of events:

1. First, the five records for suppliers S1–S5 are inserted and are stored together on some page *p,* as shown in Fig. D.6. Note that page *p* still contains a considerable amount of free space.

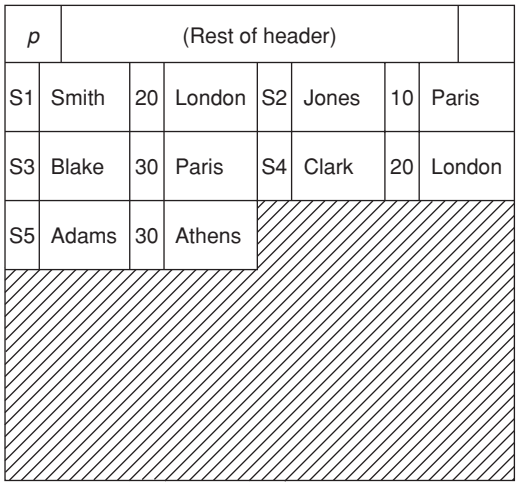

Fig. D.6 Layout of page *p* after initial loading of the five supplier records for S1–S5

2. Now suppose the DBMS inserts a new supplier record (for a new supplier, say supplier S9). The file manager stores this record on page *p* (because there is still space), immediately following the record for supplier S5.
3. The DBMS deletes the record for supplier S2. The file manager erases the S2 record from page *p,* and shifts the records for suppliers S3, S4, S5, and S9 up to fill the gap.
4. The DBMS inserts a new supplier record for another new supplier, supplier S7. Again the file manager stores this record on page *p* (because there is still space); it places the new record immediately following that for supplier S5, shifting the record for supplier S9 down to make room. The situation at this juncture is illustrated in Fig. D.7.

And so on. The key point here is that the logical sequence of records within any given page can be represented by *physical* sequence within that page; the file manager will shift records up and down to achieve this effect, keeping all data records together at the top of the page and all free space together at the bottom. (The logical sequence of records *across* pages is of course represented by the sequence of those pages within their containing page set, as described in the page management example earlier.)

As explained in Section D.2, records are identified internally by *record ID* (RID). Fig. D.8 shows how RIDs are typically implemented. The RID for a record *r* consists of

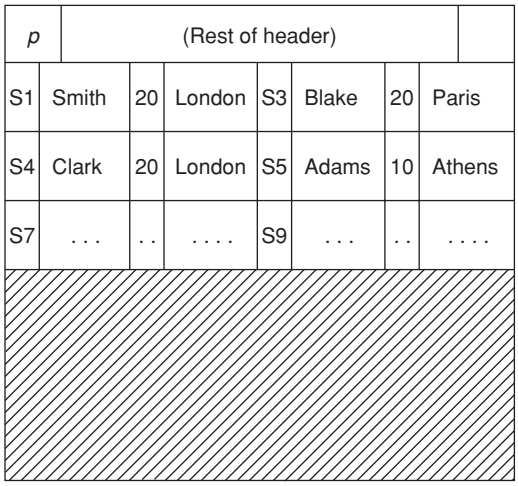

Fig. D.7 Layout of page *p* after inserting supplier S9, deleting supplier S2, and inserting supplier S7

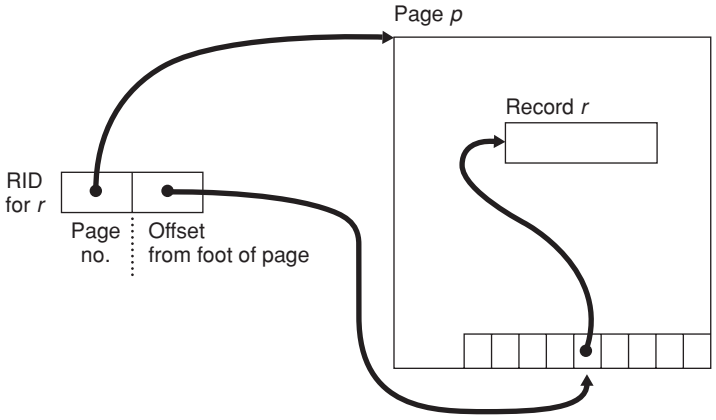

Fig. D.8 Implementation of record IDs (RIDs)

two parts: (a) the page number of the page *p* containing *r* and (b) a byte offset from the foot of *p* identifying a slot that contains, in turn, the byte offset of *r* from the top of *p*. This scheme represents a good compromise between the speed of direct addressing and the flexibility of indirect addressing: Records can be shifted up and down within their containing page, as illustrated in Figs. D.7 and D.8, without having to change RIDs (only the local offsets at the foot of the page have to change); yet access to a given record given its

RID is fast, involving only a single page access. (It is desirable that RIDs not change, because they are typically used elsewhere in the database as pointers to the records in question—for example, in indexes. If the RID of some record did in fact change, then all such pointer references elsewhere would have to be changed too.)

Note: Access to a specific record under the foregoing scheme might in rare cases involve two page accesses (but never more than two). Two accesses will be required if a varying-length record is updated in such a way that it is now longer than it was before, and there is not enough free space on the page to accommodate the increase. In such a situation, the updated record will be placed on another page (an **overflow** page), and the original record will then be replaced by a pointer (another RID) to the new location. If the same thing happens again, so that the updated record has to be moved to still a third page, then the pointer in the original page will be changed to point to this newest location.

We are now almost ready to move on to our discussion of storage structures. From this point forward, we will assume for the most part (just as the DBMS normally assumes) that a given file is simply a collection of records, each uniquely identified by a record ID that never changes as long as that record remains in existence. A few final points to conclude this section:

- Note that one consequence of the preceding discussion is that, for any given file, it is *always* possible to access all of the records in that file sequentially—where by "sequentially" we mean "record sequence within page sequence within page set" (typically, ascending RID sequence). This sequence is often referred to loosely as **physical** sequence, though it should be clear that it does not necessarily correspond to any obvious physical sequence on the disk. For convenience, however, we will adopt the same term in what follows.
- Observe that access to a file in physical sequence is possible even if several files share the same page set (i.e., if files are interleaved). Records that do not belong to the file in question can simply be skipped during the sequential scan.
- It should be stressed that physical sequence is often at least adequate as an access path to a given file. Sometimes it might even be optimal (especially if the file is small). However, it is frequently the case that something better is needed. And, as indicated in Section D.1, an enormous variety of techniques exist for achieving such a "something better."
- For the remainder of this appendix, we will usually assume for simplicity that the (unique) "physical" sequence for any given file is *primary key sequence* (as defined in Section D.3), barring explicit statements to the contrary. Please note, however, that this assumption is made purely for the purpose of simplifying subsequent discussion; we recognize that there might be good reasons in practice for physically sequencing a given file in some other manner: for example, by the value(s) of some other field(s), or simply by time of arrival (**chronological** sequence).
- For various reasons, a record will probably contain certain control information in addition to its regular data fields. That information is typically collected together at the front of the record in the form of a **record prefix**. Examples of the kind of information found in such prefixes are the ID of the containing file (necessary if one page

can contain records from several files), the record length (necessary for varying-length records), a delete flag (necessary if records are not physically erased at the time of a logical delete operation), pointers (necessary if records are chained together in any way), and so on. But of course all such control information will normally be concealed from the user.

- Finally, note that the regular data fields in a given record will be of interest to the DBMS *but not to the file manager* (and not to the disk manager). The DBMS needs to be aware of those fields because it will use them as the basis for building indexes, responding to queries, and so forth. The file manager, however, has no need to be aware of them at all. As noted in Chapter 2, therefore, another distinction between the DBMS and the file manager is that a given record has an internal structure that is known to the DBMS but not to the file manager (to the file manager, a record is basically just a byte string).

In the remainder of this appendix we describe some of the more important techniques for achieving that desired "something better" (i.e., an access path that is better than physical sequence). The techniques are discussed under the general headings "Indexing," "Hashing," "Pointer Chains," and "Compression Techniques." One final general remark: The various techniques should not be seen as mutually exclusive. For example, it is perfectly feasible to have a file with (say) both hashed and indexed access to that file based on the same field, or with hashed access based on one field and pointer-chain access based on another.

D.4 INDEXING

Consider suppliers once again. Suppose the query "Get all suppliers in city *c*" (where *c* is a parameter) is an important one—that is, one that is frequently executed and is therefore required to perform well. Given such a requirement, the DBA might choose the stored representation shown in Fig. D.9. In that representation, there are two files, a supplier file and a city file (probably in different page sets); the city file, which we assume to be stored in city sequence (because CITY is the primary key), includes pointers (RIDs) into the supplier file. To get all suppliers in London (say), the DBMS now has two possible strategies available to it:

1. Search the entire supplier file, looking for all records with city value equal to London.
2. Search the city file for the London entries, and for each such entry follow the pointer to the corresponding record in the supplier file.

If the ratio of London suppliers to others is small, the second of these strategies is likely to be more efficient than the first, because (a) the DBMS is aware of the physical sequencing of the city file (it can stop its search of that file as soon as it finds a city that comes later than London in alphabetic order), and (b) even if it did have to search the entire city file, that search would still probably require fewer I/O's overall, because the city file is physically smaller than the supplier file (because the records are smaller).

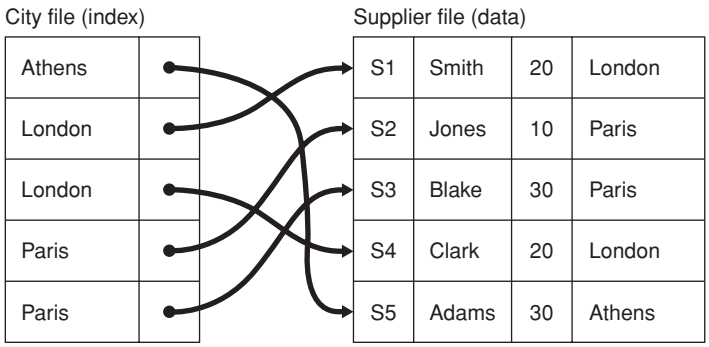

Fig. D.9 Indexing the supplier file on CITY

In this example, the city file is said to be an **index** ("the CITY index") to the supplier file; equivalently, the supplier file is said to be **indexed by** the city file. An index is thus a special kind of file. To be specific, it is a file in which each entry—that is, each record—consists of precisely two values, a data value and a pointer (RID); the data value is a value for some field of the indexed file, and the pointer identifies a record of that file that has that value for that field. The relevant field of the indexed file is called the **indexed field**, or sometimes the *index key* (we will not use this latter term, however).

Note: Indexes are so called by analogy with conventional book indexes, which also consist of entries containing "pointers" (page numbers) to facilitate the retrieval of information from an "indexed file" (i.e., the body of the book). Note, however, that, unlike the CITY index of Fig. D.9, book indexes are *hierarchically compressed*—that is, entries typically contain several page numbers, not just one. See Section D.7.

More terminology: An index on a primary key—for example, an index on field S#, in the case of suppliers—is sometimes called a *primary* index. An index on any other field—that is, the CITY index in the example—is sometimes called a *secondary* index. Also, an index on a primary key, or more generally on any candidate key, is frequently called a *unique* index.

How Indexes Are Used

The fundamental advantage of an index is that it speeds up retrieval. But there is a disadvantage too: It slows down updates. For instance, every time a new record is added to the indexed file, a new entry will also have to be added to the index. As a more specific example, consider what the DBMS must do to the CITY index of Fig. D.9 if supplier S2 moves from Paris to London. In general, therefore, the question that must be answered when some field is being considered as a candidate for indexing is: Which is more important, efficient retrieval based on values of the field in question, or the update overhead involved in providing that efficient retrieval?

For the remainder of this section we concentrate on retrieval operations specifically.

Indexes can be used in essentially two different ways. First, they can be used for **sequential** access to the indexed file—where *sequential* means "in the sequence defined by values of the indexed field." For instance, the CITY index of Fig. D.9 will allow records in the supplier file to be accessed in city sequence. Second, indexes can be used for **direct** access to individual records in the indexed file on the basis of a given value for the indexed field. The query "Get suppliers in London," discussed at the start of the section, illustrates this second case.

In fact, the two basic ideas just outlined can each be generalized slightly:

- *Sequential access:* The index can also help with *range* queries—for instance, "Get suppliers whose city is in some specified alphabetic range" (e.g., begins with a letter in the range L–R). Two important special cases are (a) "Get all suppliers whose city alphabetically precedes (or follows) some specified value," and (b) "Get all suppliers whose city is alphabetically first (or last)."
- *Direct access:* The index can also help with *list* queries—for instance, "Get suppliers whose city is in some specified list" (e.g., London, Paris, and New York).

In addition, there are certain queries—for example, *existence tests*—that can be answered from the index alone, without any access to the indexed file at all. For example, consider the query "Are there any suppliers in Athens?" The response to this query is clearly "Yes" if and only if an entry for Athens exists in the CITY index. Similar remarks apply to queries involving certain aggregate operators—for example, the query "Get the first supplier city in alphabetic order," which makes use of the aggregate operator MIN (this possibility was mentioned in Chapter 18, as you might recall).

A given file can have any number of indexes. For example, the supplier file might have both a CITY index and a STATUS index (see Fig. D.10). Those indexes could then be used to provide efficient access to supplier records on the basis of given values for either *or both* of CITY and STATUS. As an illustration of the "both" case, consider the query "Get suppliers in Paris with status 30." The CITY index gives the RIDs—*r2* and *r3,* say—for the

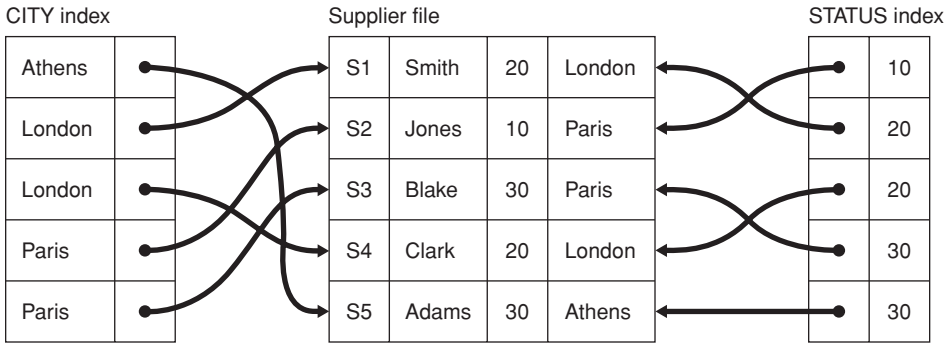

Fig. D.10 Indexing the supplier file on both CITY and STATUS

suppliers in Paris; likewise, the STATUS index gives the RIDs—*r3* and *r5,* say—for suppliers with status 30. From these two sets of RIDs it is clear that the only supplier satisfying the original query is the supplier with RID equal to *r3* (namely, supplier S3). Only then does the DBMS have to access the supplier file itself, in order to retrieve the desired record.

More terminology: Indexes are sometimes referred to as **inverted lists**, for the following reason. First, a "regular" file—the supplier file of Figs. D.9 and D.10 can be taken as a typical "regular file" in this sense—lists, for each record, the values of the fields in that record. By contrast, an index lists, for each value of the indexed field, the records that contain that value. (The inverted-list database systems mentioned briefly in Chapter 1, Section 1.6, draw their name from this terminology.) And one more term: A file with an index on every field is sometimes said to be *fully inverted.*

Indexing on Field Combinations

It is also possible to construct an index on the basis of values of two or more fields in combination. For example, Fig. D.11 shows an index to the supplier file on the combination of fields CITY and STATUS, in that order. Given such an index, the DBMS can respond to the query "Get suppliers in Paris with status 30" in a single scan *of a single index*. If the combined index were replaced by two separate indexes, then that query would involve two separate index scans (as described earlier). Furthermore, it might be difficult in that case to decide which of those two scans should be done first; since the two possible sequences might have very different performance characteristics, the choice could be significant.

Note that the combined CITY/STATUS index can also serve as an index on the CITY field alone, since all the entries for a given city are at least still consecutive within the combined index. (Another, separate index will have to be provided if indexing on STATUS is also required, however.) In general, an index on the combination of fields *F1, F2, F3, ..., Fn* (in that order) will also serve as an index on *F1* alone, as an index on the combination *F1F2* (or *F2F1*), as an index on the combination *F1F2F3* (in any order), and so on. Thus the total number of indexes required to provide complete indexing in this way is not as large as might appear at first glance (see Exercise D.9 at the end of this appendix).

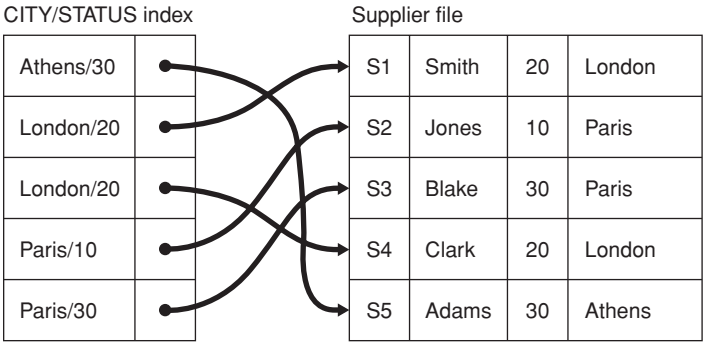

Fig. D.11 Indexing the supplier file on the combination of CITY and STATUS

Dense *vs.* Nondense Indexing

As stated several times already, the fundamental purpose of an index is to speed up retrieval—more specifically, to reduce the number of disk I/O's needed to retrieve some given record. Basically, this purpose is achieved by means of *pointers;* and up to this point we have assumed that all such pointers are *record* pointers (i.e., RIDs). In fact, however, it is sufficient for the stated purpose if those pointers are simply *page* pointers (i.e., page numbers). It is true that to find the desired record within a given page, the system will then have to do some additional work to search through the page in main memory, but the number of I/O's will remain unchanged. *Note:* As a matter of fact, the book index analogy mentioned earlier provides an example of an index in which the pointers are page pointers rather than "record" pointers.

We can take this idea further. Recall that any given file has a single "physical" sequence, represented by the combination of (a) the sequence of records within each page and (b) the sequence of pages within the containing page set. Suppose the supplier file is stored such that its physical sequence corresponds to the logical sequence as defined by the values of some field, say the supplier number field; in other words, the supplier file is *clustered* on that field (see the discussion of intra-file clustering at the end of Section D.2). Suppose also that an index is required on that field. Then there is no need for that index to include an entry for every record in the indexed file (i.e., the supplier file, in the example); all that is needed is an entry for each *page,* giving the highest supplier number on the page and the corresponding page number. See Fig. D.12 (where we assume for simplicity that a given page can hold a maximum of two supplier records).

As an example, consider what is involved in retrieving supplier S3 using this index. First the system must scan the index, looking for the first entry with supplier number greater than or equal to S3. It finds the index entry for supplier S4, which points to page *p* (say). It then retrieves page *p* and scans it in main memory, looking for the required record (which in this example, of course, will be found very quickly).

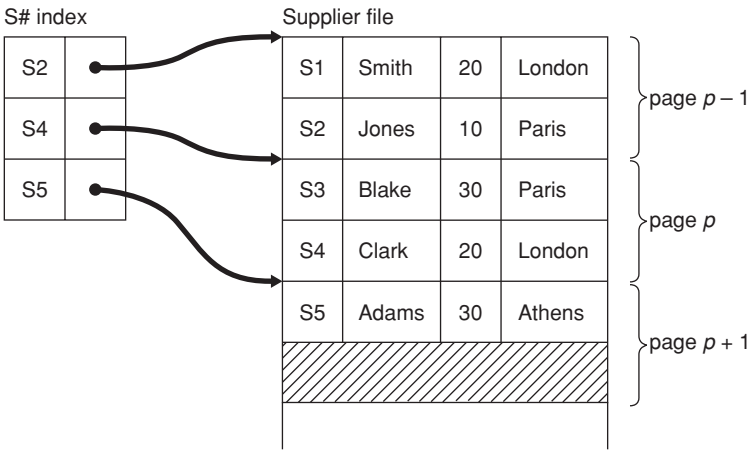

Fig. D.12 Example of a nondense index

An index such as that of Fig. D.12 is said to be **nondense** (or sometimes **sparse**), because it does not contain an entry for every record in the indexed file. (By contrast, all indexes discussed in this appendix prior to this point have been **dense**.) One advantage of a nondense index is that it will occupy less space than a corresponding dense index, for the obvious reason that it will contain fewer entries. As a result, it will probably be quicker to scan also. A disadvantage is that it might no longer be possible to perform existence tests on the basis of the index alone (see the brief note on performing existence tests in the subsection "How Indexes Are Used" earlier in this section).

Note that in general a given file can have at most one nondense index, because such an index relies on the (unique) physical sequence of the file in question. All other indexes must necessarily be dense.

B-Trees

A particularly common and important kind of index is the **B-tree** (in fact, most relational systems support B-trees as their principal form of storage structure, and some support no other). Before we can explain what a B-tree is, however, we must first introduce one more preliminary notion: namely, the notion of a **multi-level** or **tree-structured** index.

The reason for providing an index in the first place is to remove the need for physical sequential scanning of the indexed file. However, physical sequential scanning is still needed in the *index.* If the indexed file is very large, then the index can itself get to be quite sizable, and sequentially scanning the index can itself get to be quite time consuming. The solution to this problem is the same as before: We treat the index simply as a regular file, and build an index to it (an index to the index). This idea can be carried to as many levels as desired (three are common in practice; a file would have to be very large indeed to require more than three levels of indexing). Each level of the index acts as a nondense index to the level below (it *must* be nondense, of course, for otherwise nothing would be achieved—level n would contain the same number of entries as level n+1, and would therefore take just as long to scan).

Now we can discuss B-trees. A B-tree is a particular type of tree-structured index. B-trees as such were first proposed by Bayer and McCreight in 1972 [D.16]. Since that time, numerous variations on the basic idea have been investigated, by Bayer himself and by many other researchers; as already suggested, B-trees of one kind or another are now probably the most common storage structure of all in today's database systems. Here we describe the variation discussed by Knuth [D.1]. (We remark in passing that the index structure of IBM's Virtual Storage Access Method, VSAM [D.18], is very similar to Knuth's structure; however, the VSAM version was invented independently and includes additional features of its own, such as the use of compression techniques. In fact, a precursor of the VSAM structure was described as early as 1969 [D.19].)

In Knuth's variation, the index consists of two parts, the *sequence set* and the *index set* (to use VSAM terminology):

- The **sequence set** consists of a single-level index to the actual data; that index is normally dense, but could be nondense if the indexed file were clustered on the indexed field. The entries in the index are (of course) grouped into pages, and the pages are (of course) chained together, such that the logical ordering represented by the index is

obtained by taking the entries in physical order in the first page on the chain, followed by the entries in physical order in the second page on the chain, and so on. Thus the sequence set provides fast *sequential* access to the indexed data.

- The **index set**, in turn, provides fast *direct* access to the sequence set (and hence to the data too). The index set is actually a tree-structured index to the sequence set; in fact, it is the index set that is the real B-tree, strictly speaking. The combination of index set and sequence set is sometimes called a "B^+-tree." The top level of the index set consists of a single node (i.e., a single page, but of course containing many index entries, like all the other nodes). That top node is called the **root**.

A simple example is shown in Fig. D.13. We explain that figure as follows. First, the values 6, 8, 12, ..., 97, 99 are values of the indexed field, *F* say. Consider the top node, which consists of two *F* values (50 and 82) and three pointers (actually page numbers). Data records with *F* less than or equal to 50 can be found (eventually) by following the left pointer from this node; similarly, records with *F* greater than 50 and less than or equal to 82 can be found by following the middle pointer; and records with *F* greater than 82 can be found by following the right pointer. The other nodes of the index set are interpreted analogously; note that (for example) following the right pointer from the first node at the second level takes us to all records with *F* greater than 32 *and also less than or equal to 50* (by virtue of the fact that we have already followed the left pointer from the next higher node).

The B-tree (i.e., index set) of Fig. D.13 is somewhat unrealistic, however, for the following reasons:

- First, the nodes of a B-tree do not normally all contain the same number of data values.
- Second, they normally do contain a certain amount of free space.

In general, a "B-tree of order *n*" has at least *n* but not more than 2*n* data values at any given node (and if it has *k* data values, then it also has *k*+1 pointers). No data value appears in the tree more than once. We give the algorithm for searching for a particular value *V* in the structure of Fig. D.13; the algorithm for the general B-tree of order *n* is a simple generalization.

```
set N to the root node ;
do until N is a sequence-set node ;
   let X, Y (X < Y) be the data values in node N ;
   if V ≤ X     then set N to the left   lower node of N ;
   if X < V ≤ Y then set N to the middle lower node of N ;
   if V > Y     then set N to the right  lower node of N ;
end ;
if V occurs in node N then exit /* found */ ;
if V does not occur in node N then exit /* not found */ ;
```

A problem with tree structures in general is that insertions and deletions can cause the tree to become *unbalanced.* A tree is unbalanced if the leaf nodes are not all at the same level—that is, if different leaf nodes are at different distances from the root node. Since searching the tree involves a disk access for every node visited, search times can become very unpredictable in an unbalanced tree. *Note:* In practice, the top level of the index—probably portions of other levels too—will typically be kept in main memory most of the

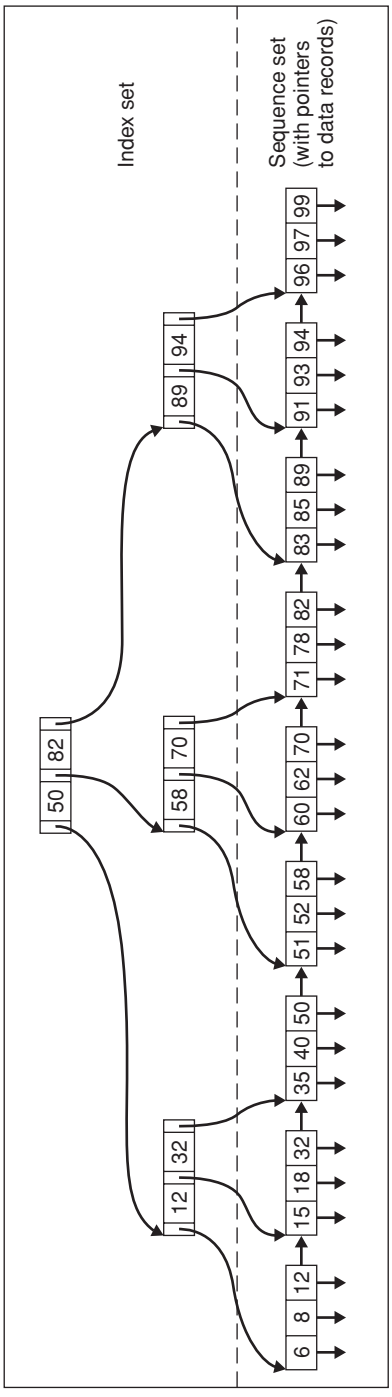

Fig. D.13 Part of a simple B-tree (Knuth's variation)

time, which will have the effect of reducing the average number of disk accesses. The overall point remains valid, however.

The notable advantage of B-trees, by contrast, is that the insertion and deletion algorithms guarantee that the tree will always be balanced. (The "B" in "B-tree" is sometimes said to stand for "balanced" for this reason.) We briefly consider insertion of a new value, *V* say, into a B-tree of order *n*. The algorithm caters to the index set only, since (as explained earlier) it is the index set that is the B-tree proper; a trivial extension is needed to deal with the sequence set also.

- First, the search algorithm is executed to locate not the sequence-set node but that node (*N* say) at the lowest level of the index set in which *V* logically belongs. If *N* contains free space, *V* is inserted into *N* and the process terminates.
- Otherwise, node *N* (which must therefore contain 2*n* values) is *split* into two nodes *N1* and *N2*. Let *S* be the original *2n* values plus the new value *V,* in their logical sequence. The lowest *n* values in *S* are placed in the left node *N1,* the highest *n* values in *S* are placed in the right node *N2,* and the middle value, *W* say, is promoted to the parent node of *N*, *P* say, to serve as a separator value for nodes *N1* and *N2*. Future searches for a value *U,* on reaching node *P,* will be directed to node *N1* if $U \leq W$ and to node *N2* if $U > W$.
- An attempt is now made to insert *W* into *P,* and the process is repeated.

In the worst case, splitting will occur all the way to the top of the tree; a new root node (parent to the old root, which will now have been split into two) will be created, and the tree will increase in height by one level (but even so will still remain balanced).

The deletion algorithm is of course essentially the inverse of the insertion algorithm just described. Changing a value is handled by deleting the old value and inserting the new one.

D.5 HASHING

Hashing—also called **hash addressing**, and sometimes, a little confusingly, **hash indexing**—is a technique for providing fast *direct* access to a specific record on the basis of a given value for some field. The field in question is usually but not necessarily the primary key. In outline, the technique works as follows:

- Each record is placed in the database at a location whose address—that is, RID, or perhaps just page number—is computed as some function (the **hash function**) of some field of that record (the *hash field,* or sometimes *hash key;* we will not use this latter term, however). The computed address is called the **hash address**.
- To store the record initially, the DBMS computes the hash address for the new record and instructs the file manager to place the record at that position.

- To retrieve the record subsequently given the hash field value, the DBMS performs the same computation as before and instructs the file manager to fetch the record at the computed position.

As a simple illustration, suppose that (a) supplier number values are S100, S200, S300, S400, S500 (instead of S1, S2, S3, S4, S5), and (b) each supplier record requires an entire page to itself, and consider the following hash function:

```
hash address (i.e., page number) =
    remainder after dividing numeric part of S# value by 13
```

This is a trivial example of a very common class of hash function called **division/remainder**. (For reasons that are beyond the scope of this appendix, the divisor in a division/remainder hash is usually chosen to be prime, as in our example.) The page numbers for the five suppliers are then 9, 5, 1, 10, 6, respectively, giving us the representation shown in Fig. D.14.

It should be clear from the foregoing description that hashing differs from indexing inasmuch as, while a given file can have any number of indexes, it can have *at most one* hash structure. To state this differently: A file can have any number of indexed fields, but only one hash field. (These remarks assume the hash is *direct*. By contrast, a file can have any number of *indirect* hashes. See reference [D.24].)

In addition to showing how hashing works, the example also shows why the hash function is necessary. It would theoretically be possible to use an "identity" hash function—in other words, to take the primary key value directly as the hash address (assuming, of course, that the primary key is numeric). Such a technique would generally be inadequate in practice, however, because the range of possible primary key values will usually be much wider than the range of available addresses. For instance, suppose that supplier numbers are in fact in the range S000–S999, as in the foregoing example. Then there would be 1,000 possible distinct supplier numbers, whereas there might in fact be only 10 or so actual suppliers. In order to avoid a considerable waste of storage space, therefore, we would ideally like to find a hash function that will reduce any value in the range 000–999 to one in the range 0–9 (say). To allow a little room for future growth, it is usual to extend the target range by 20 percent or so; that was why we chose a function that generated values in the range 0–12 rather than 0–9 in our example.

The example also illustrates one of the disadvantages of hashing: The "physical sequence" of records within the file will almost certainly not be the primary key sequence, nor indeed any other sequence that has any sensible logical interpretation. (In addition, there will be gaps of arbitrary size between consecutive records.) In fact, the physical sequence of a file with a hashed structure is usually—not invariably—considered to represent no particular logical sequence.

Note: Of course, it is always possible to *impose* any desired logical sequence on a hashed file by means of an index; indeed, it is possible to impose several such sequences by means of several indexes, one for each such sequence. See also references [D.35] and [D.37], which discuss the possibility of hashing schemes that do preserve logical sequence in the file as stored.

Another disadvantage of hashing in general is that there is always the possibility of **collisions**: that is, two or more distinct records ("synonyms") that hash to the same

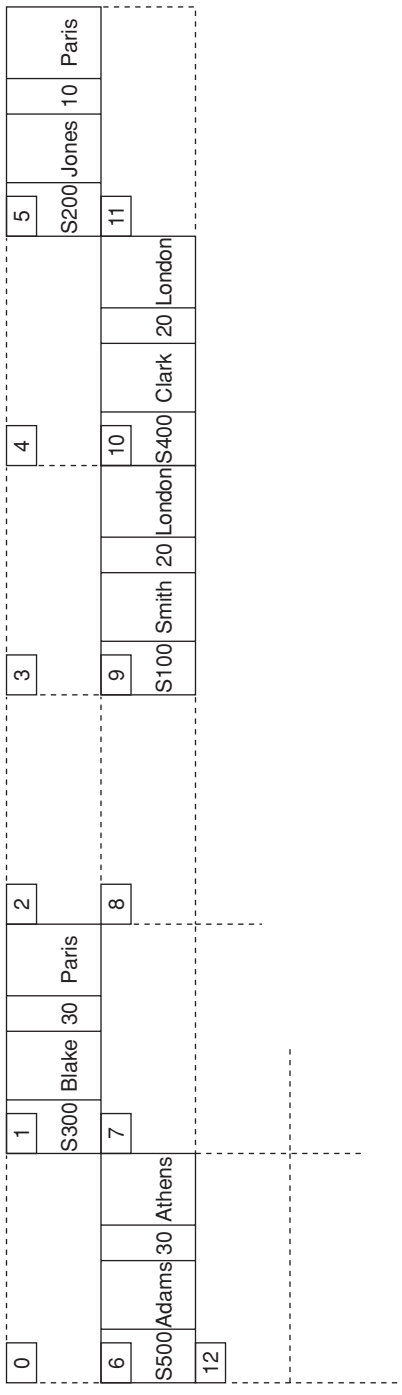

Fig. D.14 Example of a hashed structure

address. For example, suppose the supplier file (with suppliers S100, S200, and so on) also includes a supplier with supplier number S1400. Given the hash function in our example ("divide by 13 and take the remainder"), that supplier will collide with supplier S100 at hash address 9. The hash function as it stands is thus clearly inadequate—it needs to be extended somehow to deal with the collision problem.

In terms of our original example, one possible extension is to treat the remainder after division by 13, not as the hash address *per se,* but rather as the start point for a sequential scan. Thus, to insert supplier S1400 (assuming that suppliers S100–S500 already exist), we go to page 9 and search forward from that position for the first free page. The new supplier will be stored on page 11. To retrieve that supplier subsequently, we go through a similar procedure. This **linear search** method might well be adequate if (as is likely in practice) several records are stored on each page. Suppose each page can hold *n* records. Then the first *n* collisions at some hash address *p* will all be stored on page *p,* and a linear search through those collisions will be totally contained within that page. However, the next—that is, (*n*+1)st—collision will of course have to be stored on some distinct **overflow** page, and another I/O will be needed.

Another approach to the collision problem, perhaps more frequently encountered in real systems, is to treat the result from the hash function, *a* say, as the storage address, not for a data record, but rather for an **anchor point**. The anchor point at storage address *a* is then taken as the head of a chain of pointers (a **collision chain**) linking together all records—or all pages of records—that collide at *a.* Within any given collision chain, the collisions will typically be kept in hash field sequence, to simplify subsequent searching.

Extendable Hashing

Yet another disadvantage of hashing as described so far is that as the size of the hashed file increases, so the number of collisions also tends to increase, and hence the average access time increases correspondingly (because more and more time is spent searching through sets of collisions). Eventually a point might be reached where it becomes desirable to reorganize the file—in other words, to unload the existing file and reload it, using a new hashing function.

Extendable hashing [D.28] is an elegant variation on the basic hashing idea that alleviates the foregoing problems.[3] In fact, extendable hashing guarantees that the number of disk accesses needed to locate a specific record is never more than two, and will usually be just one, no matter what the file size might be. (It therefore also guarantees that file reorganization will never be required.) *Note:* Values of the hash field must be unique in the extendable hashing scheme, which of course they will be if that field is in fact the primary key as suggested at the start of this section.

In outline, the scheme works as follows:

1. Let the basic hash function be *h,* and let the primary key value of some specific record *r* be *k.* Hashing *k*—that is, evaluating *h*(*k*)—yields a value *s* called the *pseudo-*

[3] Reference [D.28] spells "extendable" with an *i,* thus: extendible.

key of *r.* Pseudokeys are not interpreted directly as addresses but instead lead to storage locations in an indirect fashion, as described in what follows.

2. The file has a *directory* associated with it, also stored on the disk. The directory consists of a header, containing a value d (the *depth* of the directory), together with 2^d pointers. The pointers are pointers to data pages, which contain the actual records (many records per page). A directory of depth d can thus handle a maximum file size of 2^d distinct data pages.

3. If we consider the leading d bits of a pseudokey as an unsigned binary integer b, then the ith pointer in the directory ($1 \le i \le 2^d$) points to a page that contains all records for which b takes the value i–1. In other words, the first pointer points to the page containing all records for which b is all zeros, the second pointer points to the page for which b is 0 . . . 01, and so on. (These 2^d pointers are typically not all distinct; that is, there will typically be fewer than 2^d distinct data pages. See Fig. D.15.) Thus, to find the record having primary key value k, we hash k to find the pseudokey s and take the first d bits of that pseudokey; if those bits have the numeric value i–1, we go to the ith pointer in the directory (first disk access) and follow it to the page containing the required record (second disk access).

   Note: In practice the directory will usually be sufficiently small that it can be kept in main memory most of the time. The "two" disk accesses will thus usually reduce to one in practice.

4. Each data page also has a header giving the *local depth* p of that page ($p \le d$). Suppose, for example, that d is 3, and that the first pointer in the directory (the 000 pointer) points to a page for which the local depth p is 2. Local depth 2 here means that, not only does this page contain all records with pseudokeys starting with 000, it contains *all* records with pseudokeys starting with 00 (i.e., those starting with 000 and also those starting with 001). In other words, the 001 directory pointer also points to this page. Again, see Fig. D.15.

5. Suppose now that the 000 data page is full and we wish to insert a new record having a pseudokey that starts with 000 (or 001). At this point the page is split in two; that is, a new, empty page is acquired, and all 001 records are moved out of the old page and into the new one. The 001 pointer in the directory is changed to point to the new page (the 000 pointer still points to the old one). The local depth p for each of the two pages will now be 3, not 2.

6. Now suppose that the data page for 000 becomes full again and has to split again. The existing directory cannot handle such a split, because the local depth of the page to be split is already equal to the directory depth. Therefore we *double the directory;* that is, we increase d by one and replace each pointer by a pair of adjacent, identical pointers. The data page can now be split; 0000 records are left in the old page and 0001 records go in the new page, the first pointer in the directory is left unchanged (i.e., it still points to the old page), and the second pointer is changed to point to the new page. Note that doubling the directory is a fairly inexpensive operation, since it does not involve access to any of the data pages.

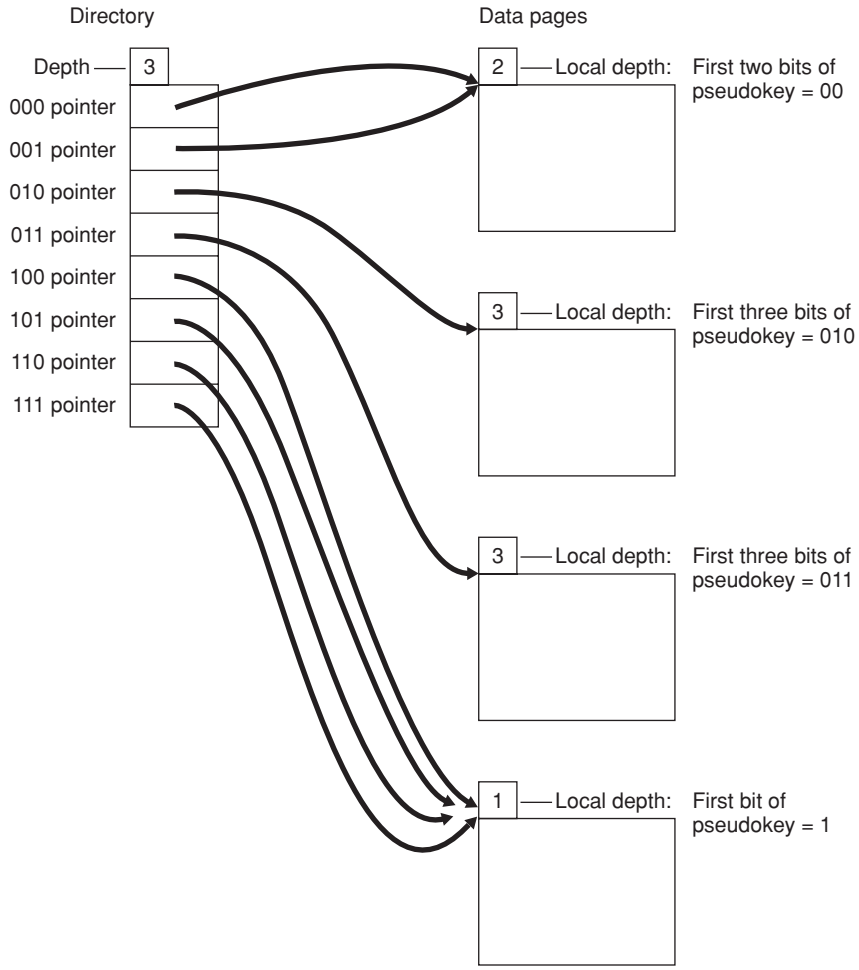

Fig. D.15 Example of extendable hashing

So much for our discussion of extendable hashing. Numerous further variations on the basic hashing idea have been devised; see, for example, references [D.29–D.36].

D.6 POINTER CHAINS

Suppose again, as at the beginning of Section D.4, that the query "Get all suppliers in city *c*" is an important one. Another stored representation that can handle that query reasonably well—possibly better than an index, though not necessarily so—uses *pointer chains.* Such a representation is illustrated in Fig. D.16. As can be seen, it involves two files, a supplier

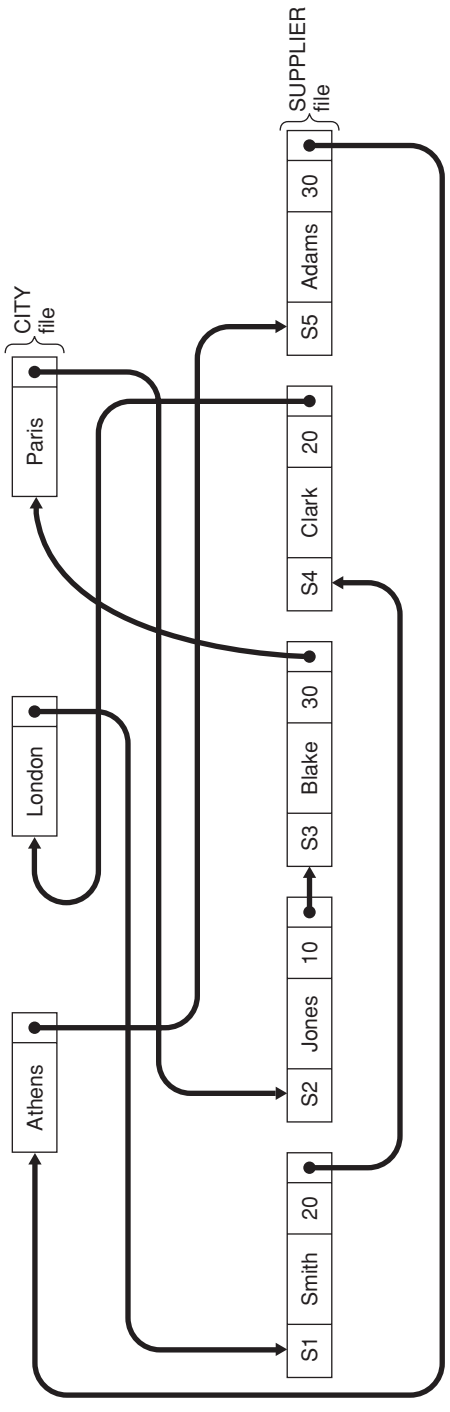

Fig. D.16 Example of a parent/child structure

file and a city file, much as in the index representation of Fig. D.9 (this time both files are probably in the same page set, for reasons to be explained in Section D.7). In the pointer-chain representation of Fig. D.16, however, the city file is not an index but what is sometimes referred to as a *parent* file. The supplier file is accordingly referred to as the *child* file, and the overall structure is an example of **parent/child organization**.

In the example, the parent/child structure is based on supplier city values. The parent (city) file contains one record for each distinct supplier city, giving the city value and acting as the head of a *chain* or *ring* of pointers linking together all child (supplier) records for suppliers in that city. Note that the city field has been removed from the supplier file; to get all suppliers in London (say), the DBMS can search the city file for the London entry and then follow the corresponding pointer chain.

The principal advantage of the parent/child structure is that the insertion and deletion algorithms are somewhat simpler, and might conceivably be more efficient, than the corresponding algorithms for an index; also, the structure will probably occupy less storage than the corresponding index structure, because each city value appears exactly once instead of many times. The principal disadvantages are as follows:

- For a given city, the only way to access the *n*th supplier is to follow the chain and access the 1st, 2nd, ..., (*n*–1)st supplier too. Unless the supplier records are appropriately clustered, each access will involve a separate seek operation, and the time taken to access the *n*th supplier could be considerable.
- Although the structure might be suitable for the query "Get suppliers in a given city," it is of no help—in fact, it is a positive hindrance—for the inverse query "Get the city for a given supplier" (where the given supplier is identified by a given supplier number). For this latter query, either a hash or an index on the supplier file is probably desirable; note that a parent/child structure based on supplier numbers would not make much sense (why not?). And even when the given supplier record has been located, it is still necessary to follow the chain to the parent record to discover the desired city (the need for this extra step is our justification for claiming that the parent/child structure is actually a hindrance for this class of query).

  Note, moreover, that the parent (city) file will probably require index or hash access too if it is of any significant size. Hence pointer chains alone are not really an adequate basis for a storage structure—other mechanisms, such as indexes, will almost certainly be needed as well.
- Because (a) the pointer chains actually run through the records—that is, the record prefixes physically include the relevant pointers—and (b) values of the relevant field are factored out of the child records and placed in the parent records instead, it is a nontrivial task to create a parent/child structure over an existing set of records. In fact, such an operation will typically require a database reorganization, at least for the relevant portion of the database. By contrast, it is a comparatively straightforward matter to create a new index over an existing set of records. *Note:* Creating a new hash will also typically require a reorganization, incidentally, unless the hash is indirect [D.24].

Several variations are possible on the basic parent/child structure. For example:

- The pointers could be made two-way. One advantage of this variation is that it simplifies the process of pointer adjustment necessitated by the operation of deleting a child record.
- Another extension would be to include a pointer (a "parent pointer") from each child record direct to the corresponding parent. This extension would reduce the amount of chain traversing involved in answering the query "Get the city for a given supplier" (note, however, that it does not eliminate the need for a hash or index to help with that query).
- Yet another variation would be *not* to remove the city field from the supplier file but to repeat the field in the supplier records (a simple form of controlled redundancy). Certain retrievals—for example, "Get the city for supplier S4"—would then become more efficient. Note, however, that that increased efficiency has nothing to do with the pointer-chain structure as such; note too that a hash or index on supplier numbers will probably still be required.

Finally, of course, just as it is possible to have any number of indexes over a given file, so it is possible to have any number of pointer chains running through a given file. (It is also possible, though perhaps unusual, to have both.) Fig. D.17 shows a representation

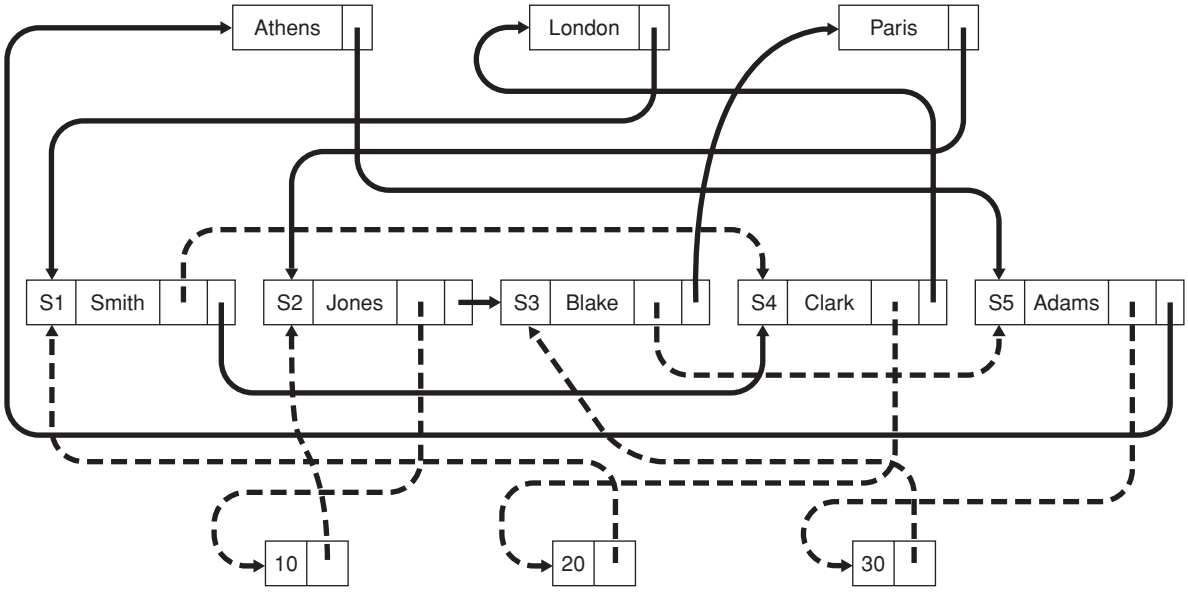

Fig. D.17 Another example of a parent/child structure

for the supplier file that involves two distinct pointer chains, and therefore two distinct parent/child structures, one with a city file as parent (as in Fig. D.16) and one with a status file as parent. The supplier file is the child file for both of these structures.

D.7 COMPRESSION TECHNIQUES

Compression techniques are ways of reducing the amount of storage required for a given collection of data. Quite frequently the result of such compression will be not only to save on storage space but also (and probably more significantly) to save on disk I/O; for if the data occupies less space, then fewer I/O operations will be needed to access it. On the other hand, extra processing will be needed to decompress the data after it has been retrieved. On balance, however, the I/O savings will probably outweigh the disadvantage of that additional processing.

Compression techniques are designed to exploit the fact that data values are almost never completely random but instead display a certain amount of predictability. As a trivial example, if a given person's name in a name-and-address file starts with the letter R, then it is extremely likely that the next person's name will start with the letter R also—assuming, of course, that the file is in alphabetic order by name.

A common compression technique is thus to replace each individual data value by some representation of the difference between it and the value that immediately precedes it: **differential compression**. Note, however, that such a technique requires that the data in question be accessed sequentially, because to decompress any given stored value requires knowledge of the immediately preceding stored value. Differential compression thus has its main applicability in situations in which the data must be accessed sequentially anyway, as in the case of (for example) the entries in a single-level index. Note, however, that in the case of an index specifically, the pointers can be compressed as well as the data—for if the logical data ordering imposed by the index is the same as, or close to, the physical ordering of the underlying file, then successive pointer values in the index will be quite similar to one another, and pointer compression is likely to be beneficial. In fact, indexes almost always stand to gain from the use of compression, at least for the data if not for the pointers.

To illustrate differential compression, we depart for a moment from suppliers and parts and consider a page of entries from an "employee name" index. Suppose the first four entries on that page are for the following employees:

```
Roberton
Robertson
Robertstone
Robinson
```

Suppose also that employee names are 12 characters long, so that each of these names should be considered (in its uncompressed form) as padded at the right with an appropriate number of blanks. One way to apply differential compression to this set of values is by replacing those characters at the front of each entry that are the same as those in the previous entry by a corresponding count: **front compression**. This approach yields:

```
0 — Roberton++++
6 — son+++
7 — tone+
3 — inson++++
```

(trailing blanks now shown explicitly as "+").

Another possible compression technique for this set of data is simply to eliminate all trailing blanks (again, replacing them by an appropriate count): an example of **rear compression**. Further rear compression can be achieved by dropping all characters to the right of the one required to distinguish the entry in question from its two immediate neighbors, as follows:

```
0 — 7 — Roberto
6 — 2 — so
7 — 1 — t
3 — 1 — i
```

The first of the two counts in each entry here is as in the previous example, the second is a count of the number of characters recorded (we have assumed that the next entry does not have "Robi" as its first four characters when decompressed). Note, however, that we have actually lost some information from this index. That is, when decompressed, it looks like this:

```
Roberto?????
Robertso????
Robertst????
Robi????????
```

(where "?" represents an unknown character). Such a loss of information is obviously permissible only if the data is recorded in full *somewhere:* in the example, in the underlying employee file.

Hierarchic Compression

Suppose a given file is physically sequenced—that is, clustered—by values of some field *F,* and suppose also that each distinct value of *F* occurs in several consecutive records of that file. For example, the supplier file might be clustered by values of the city field, in which case all London suppliers would be stored together, all Paris suppliers would be stored together, and so on. In such a situation, the set of all supplier records for a given city might profitably be compressed into a single **hierarchic** record, in which the city value in question appears exactly once, followed by supplier number, name, and status information for each supplier that happens to be located in that city. See Fig. D.18.

The records in Fig. D.18 consist of two parts: a fixed part (namely, the city field) and a varying part (namely, the set of supplier entries). *Note:* The latter part is varying in the sense that the number of entries it contains—i.e., the number of suppliers in the city in question—varies from one occurrence of the record to another. As mentioned in Chapter 6, such a varying set of entries within a record is sometimes referred to as a *repeating group.* Thus, we might say that the hierarchic records of Fig. D.18 consist of a single city

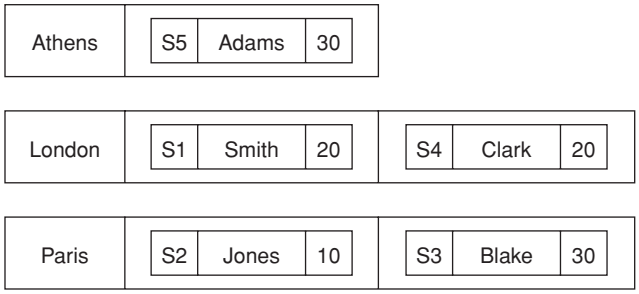

Fig. D.18 Example of hierarchic compression (intra-file)

field and a repeating group of supplier information, and the supplier information in turn consists of a supplier number field, a supplier name field, and a supplier status field (one such group of fields for each supplier in the relevant city).

Hierarchic compression of the type just described is often particularly appropriate in an index, where it is commonly the case that several successive entries all have the same data value (but of course different pointer values).

It follows from the foregoing that hierarchic compression of the kind illustrated is feasible only if intra-file clustering is in effect. As you might already have realized, however, a similar kind of compression can be applied with *inter*-file clustering also. Suppose that suppliers and shipments are clustered as suggested at the end of Section D.2—that is, shipments for supplier S1 immediately follow the supplier record for S1, shipments for supplier S2 immediately follow the supplier record for S2, and so on. More specifically, suppose that supplier S1 and the shipments for supplier S1 are stored on page *p1,* supplier S2 and the shipments for supplier S2 are stored on page *p2,* and so on. Then an inter-file compression technique can be applied as shown in Fig. D.19.

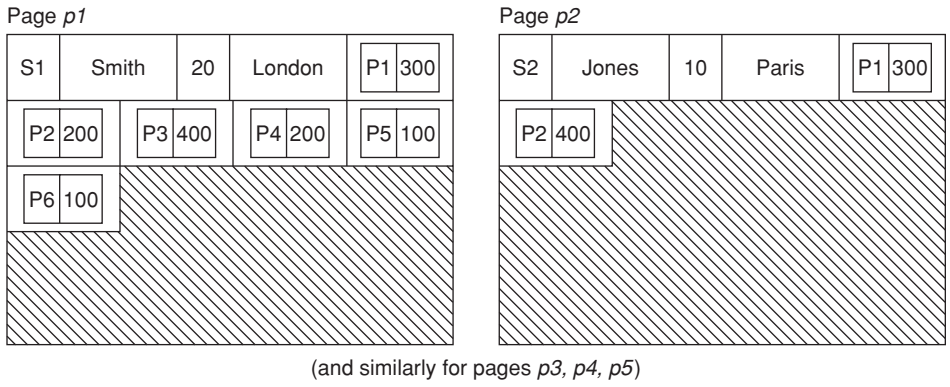

Fig. D.19 Example of hierarchic compression (inter-file)

Note: Although we describe this example as "inter-file," it really amounts to combining the supplier and shipment files into a single file and then applying *intra*-file compression to that single file. Thus, this case is not really different in kind from the case already illustrated in Fig. D.18.

We conclude this subsection by remarking that the pointer-chain structure of Fig. D.16 can be regarded as a kind of inter-file compression that does not require any corresponding inter-file clustering (or, rather, the pointers provide the logical effect of such a clustering—so that compression is possible—but do not necessarily provide the corresponding physical performance advantage at the same time—so that compression, though possible, might not be a good idea).

Huffman Coding

"Huffman coding" [D.39] is a character-coding technique that, though little used in current systems, can in principle result in significant data compression if different characters occur in the data with different frequencies (which is the normal situation, of course). The basic idea is as follows: Bit-string codings are assigned to represent characters in such a way that different characters are represented by bit strings of different lengths, and the most commonly occurring characters are represented by the shortest strings. Also, no character has a coding (of *n* bits, say) such that those *n* bits are identical to the first *n* bits of some other character coding.

As a simple example, suppose the data to be represented involves only the characters A, B, C, D, and E, and suppose also that the relative frequency of occurrence of those five characters is as indicated in the following table:

Character	*Frequency*	*Code*
E	35%	1
A	30%	01
D	20%	001
C	10%	0001
B	5%	0000

Character E has the highest frequency and is therefore assigned the shortest code, a single bit, say a 1-bit. All other codes must then start with a 0-bit and must be at least 2 bits long (a lone 0-bit would not be valid, since it would be indistinguishable from the leading portion of other codes). Character A is assigned the next shortest code, say 01; all other codes must therefore start with 00. Similarly, characters D, C, and B are assigned codes 001, 0001, and 0000, respectively. *Exercise:* What English words do the following strings represent?

```
00110001010011
010001000110011
```

Given the codings shown, the expected average length of a coded character, in bits, is

```
0.35 * 1 + 0.30 * 2 + 0.20 * 3 + 0.10 * 4 + 0.05 * 4 = 2.15 bits,
```

whereas if every character were assigned the same number of bits, as in a conventional character-coding scheme, we would need 3 bits per character (to allow for the five possibilities).

D.8 SUMMARY

In this appendix we have taken a lengthy—by no means exhaustive!—look at some of the storage structures most commonly encountered in current practice. We have also described in outline how the data access software typically functions, and have sketched the ways in which responsibility is divided among the **DBMS**, the **file manager**, and the **disk manager**. Our purpose throughout has been to explain overall concepts, not to describe in fine detail how the various system components and storage structures actually work; indeed, we have tried hard not to get bogged down in too much detail, though of course a certain amount of detail is unavoidable.

By way of summary, here is a brief review of some of the major topics we have touched on. We described **clustering**, the basic idea of which is that records that are used together should be stored physically close together. We also explained how records are identified internally by **record IDs** (**RIDs**). Then we considered some of the most important storage structures encountered in practice:

- **Indexes** (several variations thereof, including in particular **B-trees**) and their use for both **sequential** and **direct** access
- **Hashing** (including in particular **extendable** hashing) and its use for **direct** access
- **Pointer chains** (also known as **parent/child structures**) and numerous variations thereof

We also examined a variety of **compression** techniques.

We conclude by stressing the point that most users are (or should be) unconcerned with most of this material most of the time. The only "user" who needs to understand these ideas in detail is the DBA, who is responsible for the physical design of the database and for performance monitoring and tuning. For other users, such considerations should preferably all be under the covers, though it is probably true to say that those users will perform their job better if they have some idea of the way the system functions internally. For DBMS implementers, on the other hand, a working knowledge of this material (indeed, an understanding that goes much deeper than this introductory survey does) is clearly desirable, if not mandatory.

EXERCISES

Exercises D.1–D.8 might prove suitable as a basis for group discussion; they are intended to lead to a deeper understanding of various physical database design considerations. Exercises D.9 and D.10 have rather a mathematical flavor.

D.1 Investigate any database systems (the larger the better) that might be available to you. For each such system, identify the components that perform the functions ascribed in the body of this appendix to, respectively, the disk manager, the file manager, and the DBMS proper. What kind of disks or other storage media does the system support? What page sizes? What are the disk capacities, both theoretical (in bytes) and actual (in pages)? What are the data rates? The access times? How do those access times compare with the speed of main memory? Are there any limits on file size or database size? If so, what are they? Which of the storage structures described in this appendix does the system support? Does it support any others? If so, what are they?

D.2 A company's personnel database is to contain information about the divisions, departments, and employees of that company. Each employee works in one department; each department is part of one division. Invent some sample data and sketch some possible corresponding storage structures. Where possible, state the relative advantages of each of those structures—that is, consider how typical retrieval and update operations would be handled in each case. *Hint:* The constraints "each employee works in one department" and "each department is part of one division" are structurally similar to the constraint "each supplier is located in one city" (they are all examples of many-to-one relationships). A difference is that we would probably like to record more information in the database for departments and divisions than we did for cities.

D.3 Repeat Exercise D.2 for a database that is to contain information about customers and items. Each customer can order any number of items; each item can be ordered by any number of customers. *Hint:* There is a many-to-many relationship here between customers and items. One way to represent such a relationship is by means of a *double index*. A double index is an index that is used to index two data files simultaneously; a given entry in such an index corresponds to a pair of related data records, one from each of the two files, and contains two data values and two pointers. Can you think of any other ways of representing many-to-many(-to-many- . . .) relationships?

D.4 Repeat Exercise D.2 for a database that is to contain information about parts and components, where a component is itself a part and can have lower-level components. *Hint:* How does this problem differ from that of Exercise D.3?

D.5 A file of data records with no additional access structure is sometimes called a **heap**. New records are inserted into a heap wherever there happens to be room. For small files—certainly for any file not requiring more than (say) nine or ten pages of storage—a heap is probably the most efficient structure of all. Most files are bigger than that, however, and in practice all but the smallest files should have some additional access structure, say (at least) an index on the primary key. State the relative advantages and disadvantages of an indexed structure compared with a heap structure.

D.6 We referred several times in the body of this appendix to physical clustering. For example, it might be advantageous to store the supplier records such that their physical sequence is the same as or close to their logical sequence as defined by values of the supplier number field (the *clustering field*). How can the DBMS provide such physical clustering?

D.7 In Section D.5 we suggested that one method of handling hash collisions would be to treat the output from the hash function as the start point for a sequential scan (the *linear search* technique). Can you see any difficulties with that scheme?

D.8 What are the relative advantages and disadvantages of the multiple parent/child organization? (See the end of Section D.6. It might help to review the advantages and disadvantages of the multiple index organization. What are the similarities? What are the differences?)

D.9 Let us define "complete indexing" to mean that an index exists for every distinct (and distinctly ordered) field combination in the indexed file. For example, complete indexing for a file with two fields *A* and *B* would require two indexes: one on the combination *AB* (in that order) and one on the combination *BA* (in that order). How many indexes are needed to provide complete indexing for a file defined on (a) 3 fields, (b) 4 fields, (c) *N* fields?

D.10 Consider a simplified B-tree (index set plus sequence set) in which the sequence set contains a pointer to each of *N* data records, and each level above the sequence set (i.e., each level of the index set) contains a pointer to every page in the level below. At the top (root) level, of course, there is a single page. Suppose also that each page of the index set contains *n* index entries. Derive expressions for the number of *levels* and the number of *pages* in the entire B-tree.

D.11 The first 10 values of the indexed field in a particular indexed file are as follows:

```
Abrahams,GK
Ackermann,LZ
Ackroyd,S
Adams,T
Adams,TR
Adamson,CR
Allen,S
Ayres,ST
Bailey,TE
Baileyman,D
```

Each is padded with blanks at the right to a total length of 15 characters. Show the values actually recorded in the *index* if the front and rear compression techniques described in Section D.7 are applied. What is the percentage savings in space? Show the steps involved in retrieving (or attempting to retrieve) the records for "Ackroyd,S" and "Adams,V". Show also the steps involved in inserting a new record for "Allingham,M".

REFERENCES AND BIBLIOGRAPHY

The following references are organized into groups, as follows. References [D.1–D.10] are textbooks that are devoted entirely to the topic of this appendix or at least include a detailed treatment of it. References [D.11–D.15] describe some formal approaches to the subject. References [D.16–D.23] are concerned specifically with indexing, especially B-trees; references [D.24–D.38] represent a selection from the very extensive literature on hashing; references [D.39–D.40] discuss compression techniques; and, finally, references [D.41–D.59] address some miscellaneous storage structures and related issues (in particular, references [D.48–D.59] discuss certain newer storage media and newer kinds of applications, and storage structures for such media and applications).

D.1 Donald E. Knuth: *The Art of Computer Programming. Volume III: Sorting and Searching* (2d ed.). Reading, Mass.: Addison-Wesley (1998).

Volume III of Knuth's classic series of volumes contains a comprehensive analysis of search algorithms. For *database* searching, where the data resides in secondary storage, the most directly applicable sections are 6.2.4 (Multi-way Trees), 6.4 (Hashing), and 6.5 (Retrieval on Secondary Keys).

D.2 James Martin: *Computer Data-Base Organization* (2d ed.). Englewood Cliffs, N.J.: Prentice-Hall (1977).

This book is divided into two major parts, "Logical Organization" and "Physical Organization." The latter part consists of an extensive description (well over 300 pages) of storage structures and corresponding access techniques.

D.3 (Same as reference [14.45].) Toby J. Teorey and James P. Fry: *Design of Database Structures*. Englewood Cliffs, N.J.: Prentice-Hall (1982).

A tutorial and handbook on database design, both physical and logical. Over 200 pages are devoted to physical design.

D.4 Gio Wiederhold: *Database Design* (2d ed.). New York, N.Y.: McGraw-Hill (1983).

This book of 15 chapters includes a good survey of secondary storage devices and their performance parameters (one chapter, nearly 50 pages), and an extensive analysis of secondary storage structures (four chapters, over 250 pages).

D.5 T. H. Merrett: *Relational Information Systems.* Reston, Va.: Reston Publishing Company, Inc. (1984).

Includes a lengthy introduction to, and analysis of, a variety of storage structures (about 100 pages), covering not only the structures described in the present appendix but several others as well.

D.6 Jeffrey D. Ullman: *Principles of Database and Knowledge-Base Systems: Volume I.* Rockville, Md.: Computer Science Press (1988).

Includes a treatment of storage structures that is rather more theoretical than that of the present appendix.

D.7 (Same as reference [16.21].) Abraham Silberschatz, Henry F. Korth, and S. Sudarshan: *Database System Concepts* (4th ed.). New York, N.Y.: McGraw-Hill (2002).

D.8 Peter D. Smith and G. Michael Barnes: *Files and Databases: An Introduction.* Reading, Mass.: Addison-Wesley (1987).

D.9 (Same as reference [14.18].) Ramez Elmasri and Shamkant B. Navathe: *Fundamentals of Database Systems* (3d ed.). Redwood City, Calif.: Benjamin/Cummings (2000).

References [D.7], [D.8], and [D.9] are all textbooks on database systems. Each includes material on storage structures that goes beyond the treatment in the present appendix in certain respects (extensively so, in the case of reference [D.8]).

D.10 Sakti P. Ghosh: *Data Base Organization for Data Management* (2d ed.). Orlando, Fla.: Academic Press (1986).

The primary emphasis of this book is on storage structures and associated access methods (of the book's ten chapters, at least six are devoted to these topics). The treatment is fairly abstract.

D.11 David K. Hsiao and Frank Harary: "A Formal System for Information Retrieval from Files," *CACM 13,* No. 2 (February 1970).

This paper represents what was probably the earliest attempt to unify the ideas of different storage structures—principally indexes and pointer chains—into a general model, thereby providing a basis for a formal theory of such structures. A generalized retrieval algorithm is presented for retrieving records from the general structure that satisfy an arbitrary boolean combination of "*field* = *value*" conditions.

D.12 Dennis G. Severance: "Identifier Search Mechanisms: A Survey and Generalized Model," *ACM Comp. Surv. 6,* No. 3 (September 1974).

This paper falls into two parts. The first part provides a tutorial on certain storage structures (basically hashing and indexing). The second part has points in common with reference [D.11]; like that paper, it defines a unified structure, here called a *trie-tree* structure, that combines and generalizes ideas from the structures discussed in the first part. (The term *trie,* pronounced *try,* derives from a paper by Fredkin [D.42].) The resulting structure provides a general model that can represent a wide variety of different structures in terms of a small number of parameters; it can therefore be used (and in fact has been used) to help in choosing a particular structure during the process of physical database design.

A difference between this paper and reference [D.11] is that the trie-tree structure handles hashes but not pointer chains, whereas the proposal of reference [D.11] handles pointer chains but not hashes.

D.13 M. E. Senko, E. B. Altman, M. M. Astrahan, and P. L. Fehder: "Data Structures and Accessing in Data-Base Systems," *IBM Sys. J. 12,* No. 1 (1973).

This paper is in three parts:

1. Evolution of Information Systems
2. Information Organization
3. Data Representations and the Data Independent Accessing Model

The first part consists of a short historical survey of the development of database systems prior to 1973. The second part describes "the entity set model," which provides a basis for describing a given enterprise in terms of entities and entity sets (it corresponds to the conceptual level of the ANSI/SPARC architecture). The third part is the most original and significant part of the paper; it forms an introduction to the *Data Independent Accessing Model* (DIAM), which is an attempt to describe a database in terms of four successive levels of abstraction: the entity set (highest), string, encoding, and physical device levels. These four levels can be thought of as a more detailed, but still abstract, definition of the conceptual and internal portions of the ANSI/SPARC architecture. They can be briefly described as follows:

- *Entity set level:* Analogous to the ANSI/SPARC conceptual level.
- *String level:* Access paths to data are defined as ordered sets or "strings" of data objects. Three types of strings are described: atomic strings (e.g., a string connecting field occurrences to form a part record occurrence), entity strings (e.g., a string connecting part record occurrences for red parts), and link strings (e.g., a string connecting a supplier record occurrence to part record occurrences for parts supplied by that supplier).
- *Encoding level:* Data objects and strings are mapped into linear address spaces, using a simple representation primitive known as a *basic encoding unit.*
- *Physical device level:* Linear address spaces are allocated to formatted physical subdivisions of real recording media.

The aim of DIAM, like that of references [D.11–D.12], is (in part) to provide a basis for a systematic theory of storage structures and access methods. One criticism—which applies to the formalisms of references [D.11] and [D.12] also, incidentally—is that sometimes the best method of dealing with some given access request is simply to sort the data, and sorting is of course dynamic, whereas the structures described by DIAM (and the models of references [D.11] and [D.12]) are by definition always static.

D.14 S. B. Yao: "An Attribute Based Model for Database Access Cost Analysis," *ACM TODS 2,* No. 1 (March 1977).

The purpose of this paper is similar to that of references [D.11] and [D.12]; in some respects, in fact, it can be regarded as a sequel to those earlier papers, in that it presents a generalized model of storage structures that can be seen as a combination and extension of the proposals of those papers. It also presents a set of generalized access algorithms and cost equations for that generalized model. References are given to a number of other papers that report on experiments with an implemented physical file design analyzer based on the ideas of this paper.

D.15 D. S. Batory: "Modeling the Storage Architectures of Commercial Database Systems," *ACM TODS 10,* No. 4 (December 1985).

Presents a set of primitive operations, called *elementary transformations,* by which the mapping from the conceptual schema to the corresponding internal schema (i.e., the conceptual/internal mapping—see Chapter 2) can be made explicit, and hence properly studied. The elementary transformations include *augmentation* (extending a record by the inclusion of prefix data as well as user data), *encoding* (converting data to an internal form by, e.g., compression), *segmentation* (splitting a record into several pieces for storage purposes), and several others. The paper claims that any conceptual/internal mapping can be represented by an appropriate sequence of such elementary transformations, and hence that the transformations could form the basis of an approach to automating the development of data management software. By way of illustration, the paper applies the ideas to the analysis of three commercial systems: INQUIRE, ADABAS, and System 2000. *Note:* Compare and contrast the (much later) GMAP proposals of reference [2.5]. See also Appendix A for some possible counterexamples.

D.16 R. Bayer and C. McCreight: "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica 1,* No. 3 (1972).

D.17 Douglas Comer: "The Ubiquitous B-Tree," *ACM Comp. Surv. 11,* No. 2 (June 1979).

A good tutorial on B-trees.

D.18 R. E. Wagner: "Indexing Design Considerations," *IBM Sys. J. 12,* No. 4 (1973).

Describes basic indexing concepts, with details of the techniques—including compression techniques—used in IBM's Virtual Storage Access Method (VSAM).

D.19 H. K. Chang: "Compressed Indexing Method," *IBM Technical Disclosure Bulletin II,* No. 11 (April 1969).

D.20 Gopal K. Gupta: "A Self-Assessment Procedure Dealing with Binary Search Trees and B-Trees," *CACM 27,* No. 5 (May 1984).

D.21 Vincent Y. Lum: "Multi-attribute Retrieval with Combined Indexes," *CACM 13,* No. 11 (November 1970).

The paper that introduced the technique of indexing on field combinations.

D.22 James K. Mullin: "Retrieval-Update Speed Tradeoffs Using Combined Indices," *CACM 14,* No. 12 (December 1971).

A sequel to reference [D.21] that gives performance statistics for the combined index scheme for various retrieval/update ratios.

D.23 Ben Shneiderman: "Reduced Combined Indexes for Efficient Multiple Attribute Retrieval," *Information Systems 2,* No. 4 (1976).

Proposes a refinement of Lum's combined indexing technique [D.21] that considerably reduces the storage space and search time overheads. For example, the index combination *ABCD,*

BCDA, CDAB, DABC, ACBD, BDAC—see the answer to Exercise D.9(b)—could be replaced by the combination *ABCD, BCD, CDA, DAB, AC, BD.* If each of *A, B, C, D* can assume 10 distinct values, then in the worst case the original combination would involve 60,000 index entries, the reduced combination only 13,200 entries.

D.24 R. Morris: "Scatter Storage Techniques," *CACM 11,* No. 1 (January 1968).

This paper is concerned primarily with hashing as it applies to the symbol table of an assembler or compiler. Its main purpose is to describe an **indirect** hashing scheme based on *scatter tables*. A scatter table is a table of record addresses, somewhat akin to the directory used in extendable hashing [D.28]. As with extendable hashing, the hash function hashes into the scatter table, not directly to the records themselves; the records themselves can be stored anywhere that seems convenient. The scatter table can thus be thought of as a single-level *index* to the underlying data, but an index that can be accessed directly via a hash instead of having to be sequentially searched. Note that a given data file could conceivably have several distinct scatter tables, thus in effect providing hash access to the data on several distinct hash fields (at the cost of an extra I/O for any given hash access).

Despite its programming language orientation, the paper provides a good introduction to hashing techniques in general, and most of the material is applicable to database hashing also.

D.25 W. D. Maurer and T. G. Lewis: "Hash Table Methods," *ACM Comp. Surv. 7,* No. 1 (March 1975).

A good tutorial, though now somewhat dated (it does not discuss any of the newer approaches, such as extendable hashing). The topics covered include basic hashing techniques (not just division/remainder but also random, midsquare, radix, algebraic coding, folding, and digit analysis techniques); collision and bucket-overflow handling; some theoretical analysis of the various techniques; and alternatives to hashing (techniques to be used when hashing either cannot or should not be used). *Note:* A **bucket** in hashing terminology is the unit of storage—typically a page—whose address is computed by the hash function. A bucket normally contains several records.

D.26 V. Y. Lum, P.S.T. Yuen, and M. Dodd: "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files," *CACM 14,* No. 4 (April 1971).

An investigation into the performance of several different "basic" (i.e., nonextendable) hashing algorithms. The conclusion is that the division/remainder method seems to be the best all-around performer.

D.27 M. V. Ramakrishna: "Hashing in Practice: Analysis of Hashing and Universal Hashing," Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, Ill. (June 1988).

As this paper points out (following Knuth [D.1]), any system that implements hashing has to solve two problems that are almost independent of one another: It has to choose, out of the wide variety of hash functions available, one that is effective, and it also has to provide an effective technique for dealing with collisions. The author claims that while much research has been devoted to the second of these problems, very little has been done on the first, and few attempts have been made to compare the performance of hashing in practice with the performance that is theoretically achievable (reference [D.26] is an exception). How then does a system implementer choose an appropriate hash function? This paper claims that it is possible to choose a hash function that in practice does yield performance close to that predicted by theory, and presents a set of theoretical results in support of this claim.

D.28 Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong: "Extendible Hashing—A Fast Access Method for Dynamic Files," *ACM TODS 4,* No. 3 (September 1979).

D.29 G. D. Knott: "Expandable Open Addressing Hash Table Storage and Retrieval," Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif. (November 1971).

D.30 P.-Å. Larson: "Dynamic Hashing," *BIT 18* (1978).

D.31 Witold Litwin: "Virtual Hashing: A Dynamically Changing Hashing," Proc. 4th Int. Conf. on Very Large Data Bases, Berlin, FDR (September 1978).

D.32 Witold Litwin: "Linear Hashing: A New Tool for File and Table Addressing," Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada (October 1980).

D.33 Per-Åke Larson: "Linear Hashing with Overflow-Handling by Linear Probing," *ACM TODS 10,* No. 1 (March 1985).

D.34 Per-Åke Larson: "Linear Hashing with Separators—A Dynamic Hashing Scheme Achieving One-Access Retrieval," *ACM TODS 13,* No. 3 (September 1988).

References [D.28–D.34] all present extendable hashing schemes of one kind or another. The proposals of [D.29] for "expandable" hashing are earlier than (and therefore of course quite independent of) all of the others. Nevertheless, expandable hashing is fairly similar to extendable hashing as defined in reference [D.28], and so too is "dynamic" hashing [D.30], except that both schemes use a tree-structured directory instead of the simple contiguous directory proposed in reference [D.28]. "Virtual" hashing [D.31] is somewhat different; see the paper for details. "Linear" hashing, introduced in [D.32] and refined in [D.33] and [D.34], is an improvement on virtual hashing.

D.35 Witold Litwin: "Trie Hashing," Proc. 1981 ACM SIGMOD Int. Conf. on Management of Data, Ann Arbor, Mich. (April 1981).

Presents an extendable hashing scheme with a number of desirable properties:

- It is *order-preserving* (that is, the "physical" sequence of records corresponds to the logical sequence of those records as defined by values of the hash field).
- It avoids the problems of complexity and so on usually encountered with order-preserving hashes.
- An arbitrary record can be accessed (or shown not to exist) in a single disk access, even if the file contains many millions of records.
- The file can be arbitrarily volatile (by contrast, many hash schemes, at least of the nonextendable variety, tend to work rather poorly in the face of high insert volumes).

The hash function itself (which changes with time, as in all extendable hashing algorithms) is represented by a trie structure [D.42], which is kept in main memory whenever the file is in use and grows gracefully as the data file grows. The data file itself is, as already mentioned, kept in "physical" sequence on values of the hash field; and the logical sequence of leaf entries in the trie structure corresponds, precisely, to that "physical" sequence of the data records. Overflow in the data file is handled via a page-splitting technique, basically like the page-splitting technique used in a B-tree.

Trie hashing looks very interesting. Like other hash schemes, it provides better performance than indexing for direct access (one I/O *vs.* typically two or three for a B-tree); and it is preferable to most other hash schemes in that it is order-preserving, which means that sequential access will also be fast. No B-tree or other additional structure is required to provide that fast sequential access. However, note the assumption that the trie will fit into main memory (probably realistic enough). If that assumption is invalid—that is, if the data file is too large—

or if the order-preserving property is not required, then linear hashing [D.32] or some other technique might provide a preferable alternative.

D.36 David B. Lomet: "Bounded Index Exponential Hashing," *ACM TODS 8,* No. 1 (March 1983).

Another extendable hashing scheme. The paper claims that:

- The scheme provides direct access to any record in close to one I/O on average (and never more than two).
- It yields performance that is independent of the file size. (By contrast, most extendable hashing schemes suffer from temporary performance degradation at the time a directory page split occurs, because typically all such pages need to be split at approximately the same time.)
- It makes efficient use of the available disk space (i.e., space utilization can be very good).
- It is straightforward to implement.

D.37 Anil K. Garg and C. C. Gotlieb: "Order-Preserving Key Transformations," *ACM TODS 11,* No. 2 (June 1986).

As explained in the annotation to reference [D.35], an order-preserving hash function (or "key transformation") is one in which the physical sequence of records corresponds to the logical sequence of those records as defined by values of the hash field. Order-preserving hashes are desirable for obvious reasons. One simple function that is clearly order-preserving is the following:

```
hash address = quotient after dividing hash field value
               by some constant (say 10,000)
```

However, an obvious problem with a function such as this one is that it performs very poorly if values of the hash field are nonuniformly distributed (which is the usual case, of course). Hence some researchers have proposed the idea of *distribution-dependent* (but order-preserving) hash functions, or in other words functions that transform nonuniformly distributed hash field values into uniformly distributed hash addresses while maintaining the order-preserving property. (*Note:* Trie hashing [D.35] is an example of such an approach.) The present paper gives a method for constructing such hash functions for real-world data files and demonstrates the practical feasibility of those functions.

D.38 M. V. Ramakrishna and Per-Åke Larson: "File Organization Using Composite Perfect Hashing," *ACM TODS 14,* No. 2 (June 1989).

A hash function is called *perfect* if it produces no overflows. (*Note:* "No overflows" does not mean "no collisions." For example, if we assume that the hash function generates page addresses, not record addresses—see the remark on "buckets" in the annotation to reference [D.25]—and if each page can hold *n* records, then the hash function will be perfect if it never maps more than *n* records to the same page.) A perfect hash function has the property that any record can be retrieved in a single disk I/O. This paper presents a practical method for finding and using such perfect functions.

D.39 D. A. Huffman: "A Method for the Construction of Minimum Redundancy Codes," Proc. IRE 40 (September 1952).

D.40 B. A. Marron and P. A. D. de Maine: "Automatic Data Compression," *CACM 10,* No. 11 (November 1967).

Gives two compression/decompression algorithms: NUPAK, which operates on numeric data, and ANPAK, which operates on alphanumeric or "any" data (i.e., any string of bits).

D.41 Dennis G. Severance and Guy M. Lohman: "Differential Files: Their Application to the Maintenance of Large Databases," *ACM TODS 1,* No. 3 (September 1976).

Discusses "differential files" and their advantages. The basic idea is that updates are not made directly to the database itself, but instead are recorded in a physically distinct file—the differential file—and are merged with the actual database at some suitable subsequent time. The following advantages are claimed for such an approach:

- Database dumping costs are reduced.
- Incremental dumping is facilitated.
- Dumping and reorganization can both be performed concurrently with updating operations.
- Recovery after an application program failure is fast.
- Recovery after a hardware failure is fast.
- The risk of a serious data loss is reduced.
- Memo files are supported efficiently (see subsequent explanation).
- Software development is simplified.
- The main file software is simplified.
- Future storage costs might be reduced.

Note: A "memo file" is a kind of scratchpad copy of some portion of the database, used to provide quick access to data that is probably up to date and correct but is not guaranteed to be so. See the discussion of *snapshots* in Chapter 10.

One problem not discussed is that of supporting efficient sequential access to the data—for example, via an index—when some of the records are in the real database and some are in the differential file.

D.42 E. Fredkin: "TRIE Memory," *CACM 3,* No. 9 (September 1960).

A **trie** is a tree-structured data file (rather than a tree-structured access path to such a file; that is, the data is represented *by* the tree, it is not pointed to *from* the tree—unless the "data file" is really an index to some other file, as it effectively is in trie hashing [D.35]). Each node in a trie logically consists of n entries, where n is the number of distinct symbols available for representing data values. For example, if each data item is a decimal integer, then each node will have exactly 10 entries, corresponding to the decimal digits 0, 1, 2, ..., 9. Consider the data item "4285." The (unique) node at the top of the tree will include a pointer in the "4" entry. That pointer will point to a node corresponding to all existing data items having "4" as their first digit. That node in turn (the "4 node") will include a pointer in its "2" entry to a node corresponding to all data items having "42" as their first two digits (the "42 node"). The "42" node will have a pointer in its "8" entry to the "428" node, and so on. And if (for example) there are no data items beginning "429," then the "9" entry in the "42" node will be empty (there will be no pointer); in other words, the tree is pruned to contain only nodes that are nonempty. (A trie is thus generally not a balanced tree.)

Note: The term *trie* derives from "retrieval," but is nevertheless usually pronounced *try.* Tries are also known as *radix search trees* or *digital search trees*.

D.43 Eugene Wong and T. C. Chiang: "Canonical Structure in Attribute Based File Organization," *CACM 14,* No. 9 (September 1971).

Proposes a novel storage structure based on boolean algebra. It is assumed that all access requests are expressed as a boolean combination of elementary "*field* = *value*" conditions, and

that those elementary conditions are all known. Then the file can be partitioned into disjoint subsets for storage purposes. The subsets are the "atoms" of the boolean algebra consisting of the set of all sets of records retrievable via the original boolean access requests. The advantages of such an arrangement include the following:

- Set intersection of atoms is never necessary.
- An arbitrary boolean request can easily be converted into a request for the union of one or more atoms.
- Such a union never requires the elimination of duplicates.

D.44 Michael Stonebraker: "Operating System Support for Database Management," *CACM 24,* No. 7 (July 1981).

Discusses reasons why various operating system facilities—in particular, the operating system file management services—frequently do not provide the kind of services required by the DBMS, and suggests some improvements to those facilities.

D.45 M. Schkolnick: "A Survey of Physical Database Design Methodology and Techniques," Proc. 4th Int. Conf. on Very Large Data Bases, Berlin, FDR (September 1978).

D.46 S. Finkelstein, M. Schkolnick, and P. Tiberio: "Physical Database Design for Relational Databases," *ACM TODS 13,* No. 1 (March 1988).

In some respects, the problem of physical database design is more difficult in relational systems than it is in other kinds. This is because it is the system, not the user, that decides how to "navigate" through the storage structure; thus, the system will only have a chance of performing well if the storage structures chosen by the database designer are a good fit with what the system actually needs—which implies that the designer has to understand in some detail how the system works internally. And designers typically will not have such knowledge (nor is it desirable that they should, or should have to). Hence some kind of automated physical design tool is highly desirable. This paper reports on such a tool, called DBDSGN, which was developed to work with System R [4.1–4.3, 4.12–4.14]. DBDSGN takes as input a workload definition (i.e., a set of user requests and their corresponding execution frequencies) and produces as output a suggested physical design (i.e., a set of indexes for each file, typically including a "clustering index"—see Exercise D.6—in each case). It interacts with the system optimizer (see Chapter 18) to obtain information such as the optimizer's understanding of the database (with respect to file sizes, for example) and the cost formulas the optimizer uses.

DBDSGN was used as the basis for an IBM product called RDT, which was a design tool for SQL/DS [4.14].

D.47 Kenneth C. Sevcik: "Data Base System Performance Prediction Using an Analytical Model," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

As its title implies, the scope of this paper is broader than that of the present appendix—it is concerned with overall system performance issues, not just with storage structures as such. The author proposes a layered framework in which various design decisions, and the interactions among those decisions, can be systematically studied. The layers of the framework represent the system at increasingly detailed levels of description; thus, each layer is more specific (i.e., at a lower level of abstraction) than the previous one. The names of the layers give some idea of the corresponding levels of detail: abstract world, logical database, physical database, data unit access, physical I/O access, and device loadings. The author claims that an analytical model based on this framework could be used to predict numerous performance characteristics, including device utilization, transaction throughput, and response times.

The paper includes an extensive annotated bibliography on system performance. In particular, it includes a brief survey of work on the performance of different storage structures.

D.48 Hanan Samet: "The Quadtree and Related Hierarchical Data Structures," *ACM Comp. Surv. 16,* No. 2 (June 1984).

The storage structures described in the present appendix work well enough for traditional commercial databases. However, as the field of database technology expands to include new kinds of data—for example, spatial data, such as might be found in image-processing or cartographic applications—so new methods of data representation at the storage level are needed also. This paper is a tutorial introduction to some of those new methods. See Samet's book [D.49] for further discussion, also references [D.50–D.59].

D.49 (Same as reference [26.37].) Hanan Samet: *The Design and Analysis of Spatial Data Structures*. Reading, Mass.: Addison-Wesley (1990).

See the annotation to the previous reference.

D.50 Stavros Christodoulakis and Daniel Alexander Ford: "Retrieval Performance Versus Disc Space Utilization on WORM Optical Discs," Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. (May/June 1989).

See the annotation to reference [D.51].

D.51 David Lomet and Betty Salzberg: "Access Methods for Multiversion Data," Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. (May/June 1989).

A *WORM disk* is an optical disk with the property that once a record has been written to the disk, it can never be rewritten—that is, it cannot be updated in place (WORM is an acronym, standing for "Write Once, Read Many times"). Such disks have obvious advantages for certain applications, particularly those involving some kind of archival requirement. Traditional storage structures such as B-trees are not adequate for such disks, however, precisely because of the fact that rewriting is impossible. Hence (as explained in the annotation to reference [D.48], though for different reasons), new storage structures are needed. References [D.50] and [D.51] propose and analyze some such structures; reference [D.51] in particular concerns itself with structures that are appropriate for applications in which a complete historical record is to be kept—that is, applications in which data is only added to the database, never deleted (see Chapter 23).

D.52 J. Encarnaçao and F.-L. Krause (eds.): *File Structures and Data Bases for CAD*. New York, N.Y.: North-Holland (1982).

Computer-aided design (CAD) applications are a major driving force behind the research into new storage structures. This book consists of the proceedings of a workshop on the subject, with major sections as follows:

1. Data modeling for CAD
2. Data models for geometric modeling
3. Databases for geometric modeling
4. Hardware structures
5. CAD database research issues
6. Implementation problems in CAD database systems
7. Industrial applications

D.53 R. A. Finkel and J. L. Bentley: "Quad-Trees—A Data Structure for Retrieval on Composite Keys," *Acta Informatica 4* (1974).

D.54 J. Nievergelt, H. Hinterberger, and K. C. Sevcik: "The Grid File: An Adaptable, Symmetric, Multikey File Structure," *ACM TODS 9,* No. 1 (March 1984).

D.55 Antonin Guttman: "R-Trees: A Dynamic Index Structure for Spatial Searching," Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (June 1984).

D.56 Nick Roussopoulos and Daniel Leifker: "Direct Spatial Search on Pictorial Databases Using Packed R-Trees," Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Tex. (May 1985).

D.57 D. A. Beckley, M. W. Evens, and V. K. Raman: "Multikey Retrieval from K-D Trees and Quad-Trees," Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Tex. (May 1985).

D.58 Michael Freeston: "The BANG File: A New Kind of Grid File," Proc. ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

D.59 Michael F. Barnsley and Alan D. Sloan: "A Better Way to Compress Images," *BYTE 13,* No. 1 (January 1988).

Describes a novel compression technique for images called *fractal compression.* The technique works by not storing the image itself at all, but rather storing code values that can be used to re-create the desired image as and when needed by means of appropriate fractal equations. In this way, "compression ratios of 10,000 to 1—or even higher" can be achieved.