Database Processing

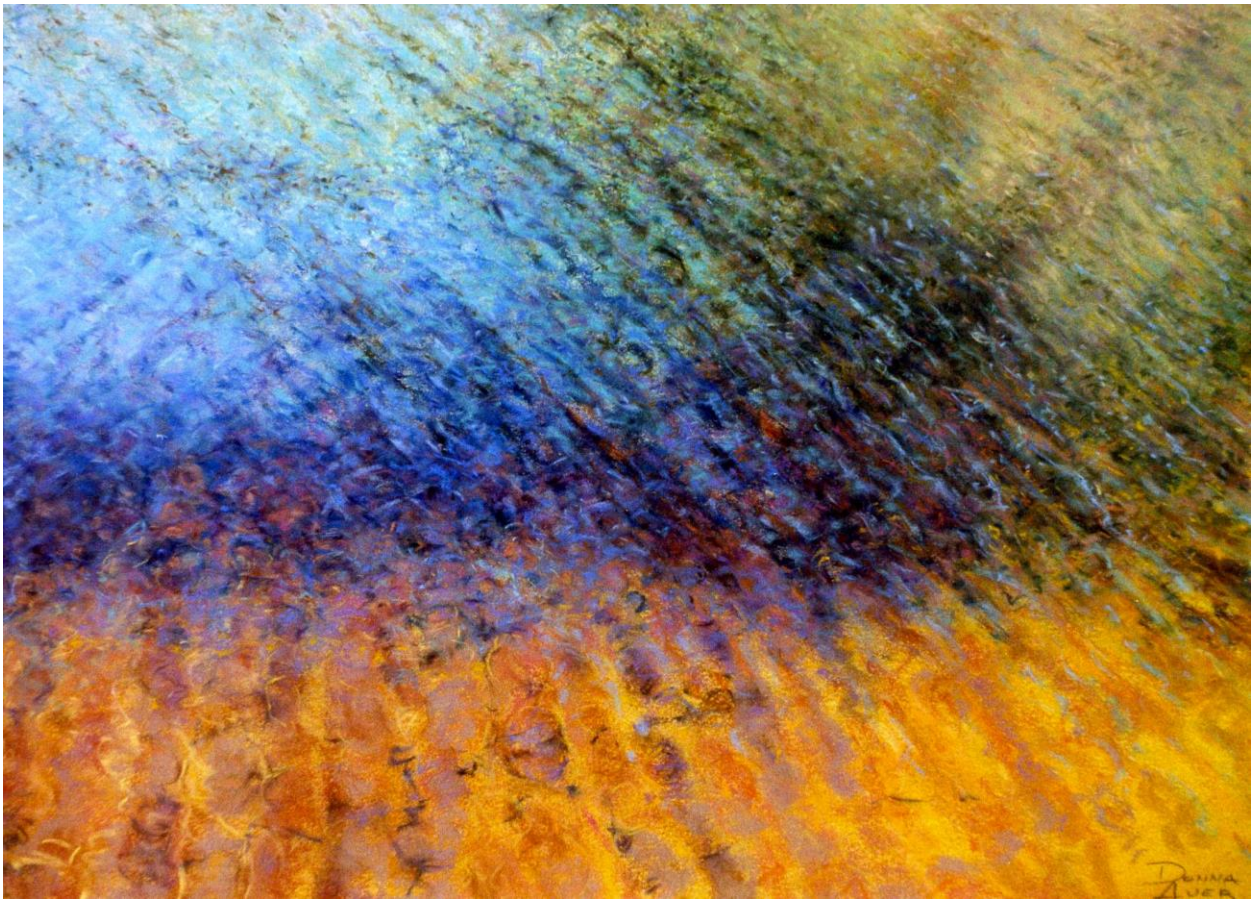
Fundamentals, Design, and Implementation

14th Edition

David M. Kroenke • David J. Auer

Online Appendix G

Data Structures for Database Processing



Vice President, Business Publishing: Donna Battista
Editor in Chief: Stephanie Wall
Acquisitions Editor: Nicole Sam
Program Manager Team Lead: Ashley Santora
Program Manager: Denise Weiss
Editorial Assistant: Olivia Vignone
Vice President, Product Marketing: Maggie Moylan
Director of Marketing, Digital Services and Products: Jeanette Koskinas
Executive Product Marketing Manager: Anne Fahlgren
Field Marketing Manager: Lenny Ann Raper
Senior Strategic Marketing Manager: Erin Gardner
Product Marketing Assistant: Jessica Quazza
Project Manager Team Lead: Jeff Holcomb
Project Manager: Ilene Kahn
Operations Specialist: Diane Peirano
Senior Art Director: Janet Slowik
Text Designer: Integra Software Services Pvt. Ltd.
Cover Designer: Integra Software Services Pvt. Ltd.
Cover Art: Donna Auer
Vice President, Director of Digital Strategy & Assessment: Paul Gentile
Manager of Learning Applications: Paul Deluca
Digital Editor: Brian Surette
Digital Studio Manager: Diane Lombardo
Digital Studio Project Manager: Robin Lazrus
Digital Studio Project Manager: Alana Coles
Digital Studio Project Manager: Monique Lawrence
Digital Studio Project Manager: Regina DaSilva
Full-Service Project Management and Composition: Integra Software Services Pvt. Ltd.
Printer/Binder: RRD Willard
Cover Printer: Phoenix Color/Hagerstown
Text Font: 10/12 Mentor Std Light

Library of Congress Cataloging-in-Publication Data

Kroenke, David M.
Database processing: fundamentals, design, and implementation/David M. Kroenke, David J. Auer.–Fourteenth edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-13-387670-3 (student edition)–ISBN 978-0-13-387676-5
(instructor's review copy)
1. Database management. I. Auer, David J. II. Title.
QA76.9.D3K76 2016
005.74–dc23

2015005632

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN 10: 0-13-387670-5
ISBN 13: 978-0-13-387670-3

Appendix G – 10 9 8 7 6 5 4 3 2 1

Chapter Objectives

- To define the term *data structures*.
- To define and illustrate the terms *flat file*, *sequential list*, *linked list*, and *index*.
- To define and illustrate B-tree multilevel indexes.
- To demonstrate how binary relationships are represented using trees, simple networks, and complex networks.
- To define and illustrate primary and secondary keys.
- To define and illustrate unique and nonunique secondary keys.

What Is the Purpose of This Appendix?

All operating systems provide data management services. These services, however, are generally not sufficient for the specialized needs of a DBMS. Therefore, to enhance performance, DBMS products build and maintain specialized data structures, which are the topic of this appendix.

What Will This Appendix Teach Me?

We begin by discussing flat files and some of the problems that can occur when such files need to be processed in different orders. Then, we turn to three specialized data structures: sequential lists, linked lists, and indexes (or inverted lists). Next, we illustrate how each of three special structures—trees, simple networks, and complex networks—are represented using various data structures. Finally, we explore how to represent and process multiple keys.

Although a thorough knowledge of data structures is not required to use most DBMS products, this background is essential for database administrators and systems programmers working with a DBMS. Being familiar with the data structures also helps you evaluate and compare database products.

What Are Flat Files?

A **flat file** is a file that has no repeating groups. Figure G-1(a) shows a flat file, and Figure G-1(b) shows a file that is not flat because of the repeating field Item. A flat file can be stored in any common physical file organization, such as sequential, indexed sequential, or direct. A **sequential file organization** stores records physically in a specific order and the records must be processed from the beginning, or using binary search if the ordering is based on a column value, in order to search for data. **Direct access file organization** allows access to specific blocks of data using the numeric addresses of those blocks.

[0] Scott Vandenberg of Siena College contributed material to this appendix.

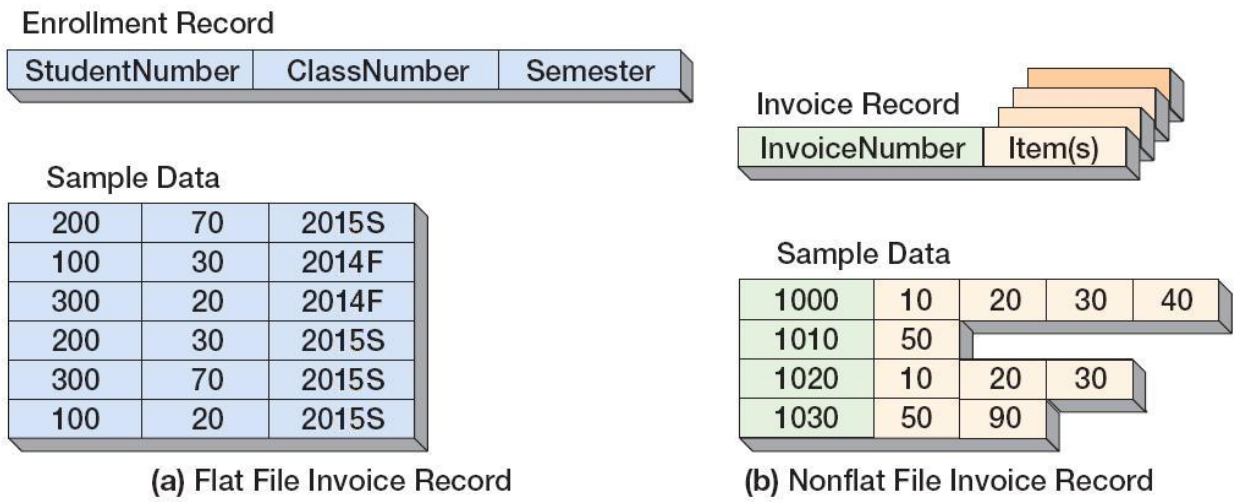


Figure G-1: Examples of Flat File and Nonflat File Records

Indexed sequential file organization allows access both via the direct address of the block and via a separate index file based on data values. Most relational DBMSs use direct file organizations for all files, with the indexing logic, if any, at a higher level (see Figure G-11). Flat files have been used for many years in commercial processing. They are usually stored and processed in some predetermined order—for example, in an ascending sequence on a key field.

Processing Flat Files in Multiple Orders

Sometimes users may want to process flat files in ways that are not readily supported by the file organization. Consider, for example, the ENROLLMENT records shown in Figure G-1(a). To produce student schedules, they must be processed in StudentNumber sequence. But to produce class rosters, the records need to be processed in ClassNumber sequence. The records, of course, can be stored in only one physical sequence. For example, they can be in order of StudentNumber or of ClassNumber, but not both at the same time. The traditional solution to the problem of processing records in different orders is to sort them in student order and process the student schedules and then sort the records in class order and produce class rosters.

For some applications, such as batch-mode systems, this solution is effective, although cumbersome. But suppose that both orders need to exist simultaneously because two concurrent users with different needs have different views of the ENROLLMENT records. What do we do then?

One solution is to create two copies of the ENROLLMENT file and sort them, as shown in Figure G-2. Because the data are listed in sequential order, this data structure is sometimes called a **sequential list**. Sequential lists can be readily stored as sequential files. This solution, however, is not generally done by DBMS products because sequentially reading a file is a slow process. Further, sequential files cannot be updated in the middle without rewriting the entire file. Also, maintaining several orders by keeping multiple copies of the same sequential list is usually not effective because the duplicated sequential list can

Student-Number	Class-Number	Semester
100	30	2014F
100	20	2015S
200	70	2015S
200	30	2015S
300	20	2014F
300	70	2015S

(a) Stored by StudentNumber

Student-Number	Class-Number	Semester
300	20	2014F
100	20	2015S
100	30	2014F
200	30	2015S
200	70	2015S
300	70	2015S

(b) Stored by ClassNumber

Figure G-2: ENROLLMENT Data Stored as Sequential Lists by Student Number and Class Number

create data integrity problems and waste space. Another solution is to maintain one copy of the file in some sorted order and, when another sort order is needed, produce another (temporary) copy of the file in a different sort order for use by a specific query. This process is called **external sorting** (since for large files that won't fit in main memory, disk space external to memory is used to complete the sorting process). Fortunately, other data structures allow us to process records in different orders and do not require the duplication of data or sorting a large file. These data structures include *linked lists* and *indexes.*

A Note on Record Addressing

Usually, the DBMS creates large **physical records**, or **blocks**, on its direct access files. These records are used as containers for logical records. Typically, each physical record contains many logical records. Here, we assume that each physical record is addressed by its **relative record number (RRN)**. Thus, a logical record might be assigned to physical record number 7, 77, or 10,000. The relative record number is thus the logical record's physical address. If each physical record has more than one logical record, the address must also specify where the logical record is within the physical record. Thus, the complete address for a logical record might be relative record number 77, byte location 100. This means the record begins in byte 100 of physical record 77.

To simplify the illustrations in this text, we assume that there is only one logical record per physical record, so we need not be concerned with byte offsets within physical records. Although this is unrealistic, it limits our discussion to the essential points.

How Can Linked Lists Be Used to Maintain Logical Record Order?

Linked lists can be used to keep records in logical order that are not necessarily in physical order. To create a linked list, we add a field to each data record. The **link field** holds the address (in our illustrations, the relative record number) of the *next* record in the logical sequence. For example, Figure G-3 shows the ENROLLMENT records expanded to include a linked list; this list maintains the records in StudentNumber order. Notice that the link for the numerically last student in the list is zero, and the ordering among records with the same StudentNumber is arbitrary.

Relative Record Number	Student-Number	Class-Number	Semester	Link
1	200	70	2015S	4
2	100	30	2014F	6
3	300	20	2014F	5
4	200	30	2015S	3
5	300	70	2015S	0
6	100	20	2015S	1

Start of list = 2

Figure G-3: ENROLLMENT Data in StudentNumber Order Using a Linked List

Relative Record Number	Student-Number	Class-Number	Semester	Student Link	Class Link
1	200	70	2015S	4	5
2	100	30	2014F	6	1
3	300	20	2014F	5	4
4	200	30	2015S	3	2
5	300	70	2015S	0	0
6	100	20	2015S	1	3

Start of student list = 2
Start of class list = 6

Figure G-4: ENROLLMENT Data in Two Orders Using Linked Lists

Figure G-4 shows ENROLLMENT records with two linked lists: One list maintains the StudentNumber order and the other list maintains the ClassNumber order. Two link fields have been added to the records, one for each list. Note that the file itself is not physically stored in *either* of these orders.

When insertions and deletions are made, linked lists have a great advantage over sequential lists. For example, to insert the ENROLLMENT record for Student 200 and Class 45, both of the lists shown in Figure G-2 would need to be rewritten. For the linked lists in G-4, however, the new record could be added to the physical end of the list, and only the values of two link fields would need to be changed to place the new record in the correct sequences. These changes are shown in Figure G-5.

Relative Record Number	Student-Number	Class-Number	Semester	Student Link	Class Link
1	200	70	2015S	4	5
2	100	30	2014F	6	7
3	300	20	2014F	5	4
4	200	30	2015S	7	2
5	300	70	2015S	0	0
6	100	20	2015S	1	3
7	200	45	2015S	3	1

Start of student list = 2
Start of class list = 6

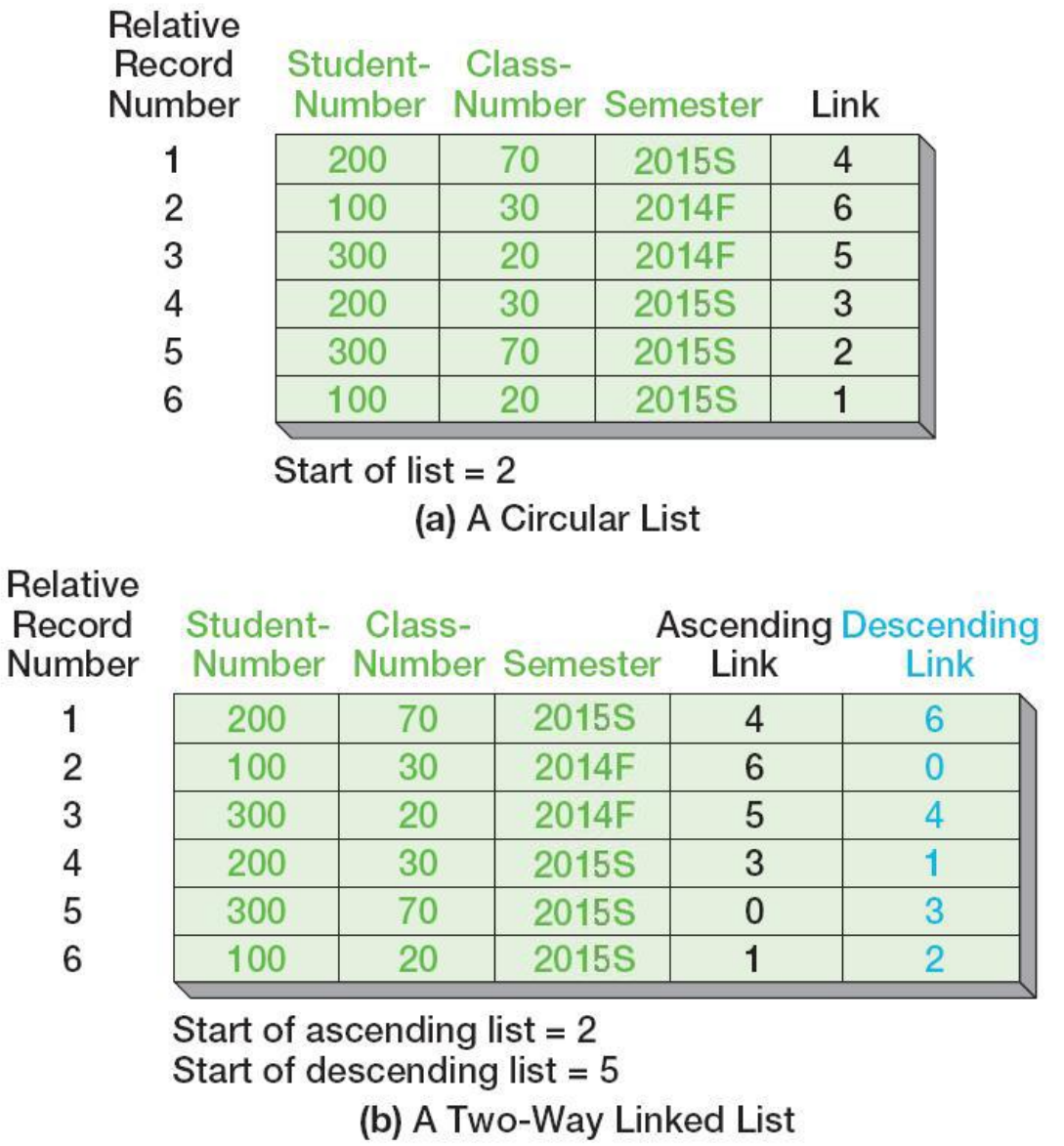
Figure G-5: ENROLLMENT Data after Inserting New Record (in Two Orders Using Linked Lists)

Relative Record Number	Student-Number	Class-Number	Semester	Student Link	Class Link
1	200	70	2015S	7	5
2	100	30	2014F	6	7
3	300	20	2014F	5	2
4	200	30	2015S	7	2
5	300	70	2015S	0	0
6	100	20	2015S	1	3
7	200	45	2015S	3	1

Start of student list = 2
Start of class list = 6

Figure G-6: ENROLLMENT Data after Deleting Student 200, Class 30 (in Two Orders Using Linked Lists)

When a record is deleted from a sequential list, a gap is created. But in a linked list, a record can be deleted simply by changing the values of the link, or the **pointer fields**. In Figure G-6, the ENROLLMENT record for Student 200, Class 30, has been logically deleted. No other record points to its address, so it has been effectively removed from the chain, even though it still exists physically. Note that in either approach (sequential or linked lists), there will be a gap that, eventually, should be reclaimed by DBMS somehow, otherwise we will eventually run out of space. In a sequential list, this is often done by periodically reorganizing (compacting) the file to remove gaps. In a linked list scenario, the DBMS can maintain a list of available spaces that can be reused.



Relative Record Number	Student-Number	Class-Number	Semester	Link
1	200	70	2015S	4
2	100	30	2014F	6
3	300	20	2014F	5
4	200	30	2015S	3
5	300	70	2015S	2
6	100	20	2015S	1

Start of list = 2

(a) A Circular List

Relative Record Number	Student-Number	Class-Number	Semester	Ascending Link	Descending Link
1	200	70	2015S	4	6
2	100	30	2014F	6	0
3	300	20	2014F	5	4
4	200	30	2015S	3	1
5	300	70	2015S	0	3
6	100	20	2015S	1	2

Start of ascending list = 2
Start of descending list = 5

(b) A Two-Way Linked List

Figure G-7: ENROLLMENT Data Sorted by StudentNumber Using a Circular and a Two-way Linked List

There are many variations of linked lists. We can make the list into a **circular list**, or **ring**, by changing the link of the last record from zero to the address of the first record in the list. Now we can reach every item in the list starting at any item in the list. Figure G-7(a) shows a circular list for the StudentNumber order. A **two-way linked list** has links in both directions. In Figure G-7(b), a two-way linked list has been created for both ascending and descending student orders.

Records ordered using linked lists cannot be stored on a sequential file because some type of direct-access file organization is needed to use the link values. Thus, either indexed sequential or direct file organization is required for linked-list processing.

How Are Indexes Used to Maintain Logical Record Order?

A logical record order can also be maintained using an **index**, or **inverted list**, as they are sometimes called. An index is simply a table that cross-references record addresses with some field value. For example, Figure G-8(a) shows the ENROLLMENT records stored in no particular order, and Figure G-8(b) shows an index on StudentNumber. In this index, the StudentNumbers are arranged in sequence, with each entry in the list pointing to a corresponding record in the original data.

As you can see, the index is simply a sorted list of StudentNumbers. To process ENROLLMENT sequentially on StudentNumber, we simply process the index sequentially, obtaining ENROLLMENT data by reading the records indicated by the pointers. Figure G-8(c) shows another index for ENROLLMENT—one that maintains ClassNumber order.

To use an index, the data to be ordered (here, ENROLLMENT) must reside on an indexed sequential or direct file, although the indexes can reside on any type of file. In practice, almost all DBMS products keep both the data and the indexes on direct files.

If you compare the linked list with the index, you will notice the essential difference between them. In a linked list, the pointers are stored along with the data. Each record contains a link field containing a pointer to the address of the next related record. But in an index, the pointers are stored in index files, separate from the data. Thus, the data records themselves contain no pointers. Both techniques are used by commercial DBMS products.

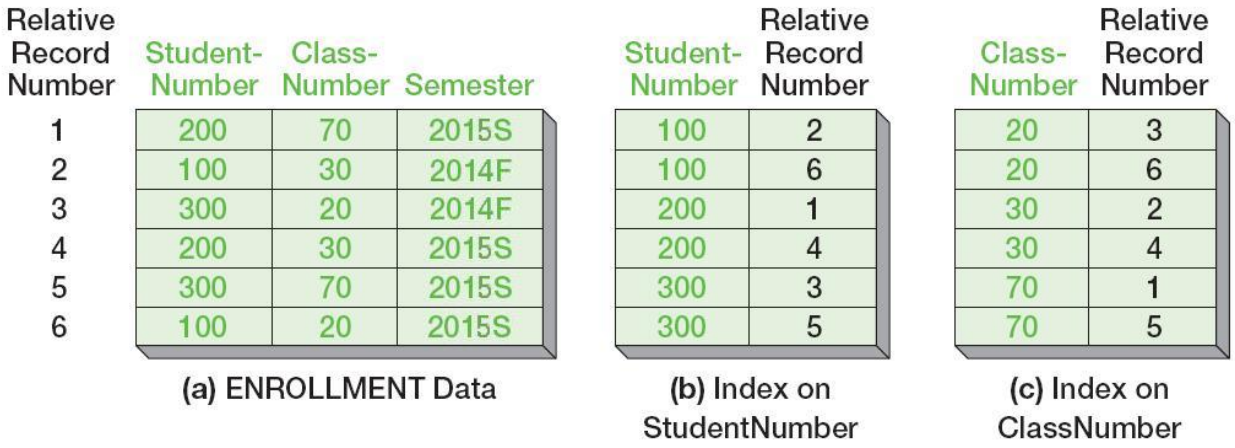


Figure G-8: ENROLLMENT Data and Corresponding Indexes:

(a) ENROLLMENT Data, (b) Index on StudentNumber, and (c) Index on ClassNumber

B-Trees

A special application of the concept of indexes, or inverted lists, is a **B-Tree**, which is a multilevel index that allows both sequential and direct processing of data records. It also ensures a certain level of efficiency in processing because of the way that the indexes are structured. B-Trees (and their variants, such as the **B+-Tree** and **B*-Tree**) are the most common form of index used in modern DBMSs. Oracle Database, for example, automatically create a B*-Tree index on every primary key.

A B-tree is an index that is made up of two parts: the sequence set and the index set. (These terms are used by IBM's VSAM file organization documentation. You may encounter other synonymous terms.) The **sequence set** is an index containing an entry for every record in the file. This index is in physical sequence, usually by primary key value. This arrangement allows sequential access to the data records, as follows: Process the sequence set in order, read the address of each record, and then read the record. In tree terminology, the sequence set is the leaf level of the B-Tree.

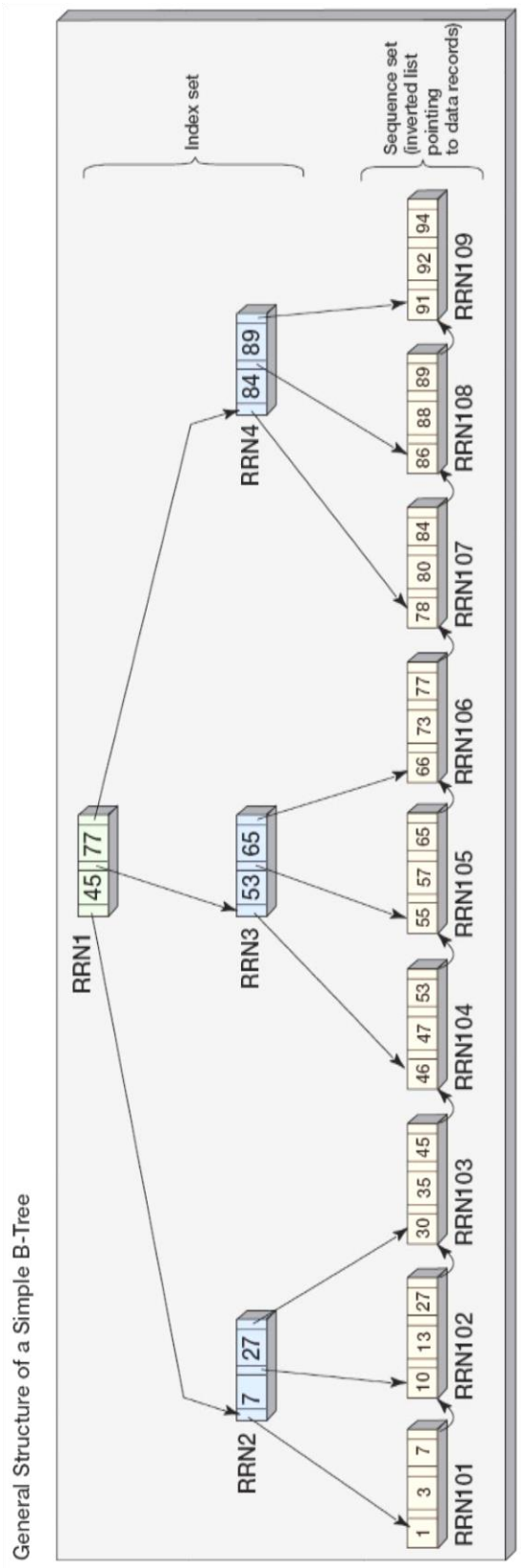
The **index set** is an index pointing to groups of entries in the sequence set index. This arrangement provides rapid direct access to records in the file, and it is the index set that makes B-trees unique. In tree terminology, the index set is made up of all the levels of the B-Tree above the leaf level.

An example of a B-tree appears in Figure G-9, and an occurrence of this structure presented in a standard file format (rather than a tree structure) can be seen in Figure G-10. Notice that the bottom row in Figure G-9, the sequence set, is simply an index similar to the examples we have already seen. It contains an entry for every record in the file (although for brevity, both the data records and their addresses have been omitted). Also, notice that the sequence set entries are in groups of three. The entries in each group are physically in sequence, and each group is chained to the next one by means of a linked list, as can be seen in Figure G-10.

Examine the index set in Figure G-9. The top entry contains two values: 45 and 77. By following the leftmost link (to RRN2), we can access all the records whose key field values are less than or equal to 45; by following the middle pointer (to RRN3) we can access all the records whose key field values are greater than 45 and less than or equal to 77; and by following the rightmost pointer (to RRN4) we can access all the records whose key field values are greater than 77.

[On Next Page]

Figure G-9: General Structure of a Simple B-Tree



General Structure of a Simple B-Tree

RRN	Link1	Value1	Link2	Value2	Link3
1	2	45	3	77	4
2	101	7	102	27	103
3	104	53	105	65	106
4	107	84	108	89	109

Index Set

.
.
.

	R1	Addr1	R2	Addr2	R3	Addr3	Link
101	1	Data or pointer	3	Data or pointer	7	Data or pointer	102
102	10	· · ·	13	· · ·	27	· · ·	103
103	30	· · ·	35	· · ·	45	· · ·	104
104	46	· · ·	47	· · ·	53	· · ·	105
105	55	· · ·	57	· · ·	65	· · ·	106
106	66	· · ·	73	· · ·	77	· · ·	107
107	78	· · ·	80	· · ·	84	· · ·	108
108	86	· · ·	88	· · ·	89	· · ·	109
109	91	· · ·	92	· · ·	94	· · ·	0

Sequence Set (Addresses of data records are omitted)

Figure G-10: Occurrence of B-Tree in Figure G-9

Similarly, at the next level there are two values and three pointers in each index entry. Each time we drop to another level, we narrow our search for a particular record. For example, if we continue to follow the leftmost pointer from the top entry and then follow the rightmost pointer from there, we can access all the records whose key field value is greater than 27 and less than or equal to 45. We have eliminated all that were greater than 45 at the first level.

B-trees are, by definition, balanced. That is, all of the data records are exactly the same distance from the top entry in the index set. This aspect of B-trees ensures performance efficiency, although the algorithms for inserting and deleting records are more complex than those for ordinary trees (which can be unbalanced), because several index entries may need to be modified when records are added or deleted to keep all records the same distance from the top index entry.

Summary of Data Structures

Figure G-11 summarizes the techniques for maintaining ordered flat files. Three supporting data structures are possible. Sequential lists can be used, but the data must be duplicated in order to maintain several orders. Because sequential lists are not used in database processing, we will not consider them further. Both linked lists and indexes can be used without data duplication. B-trees are a specific kind of index.

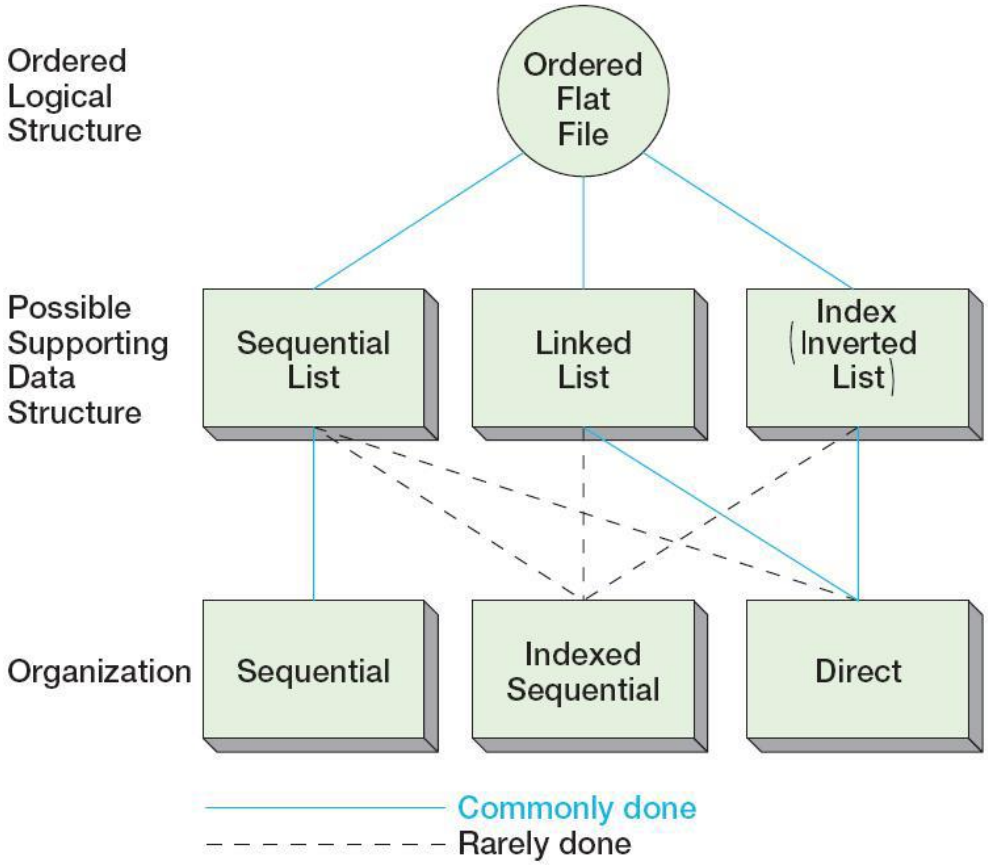


Figure G-11: Summary of Data Structures and Data Organizations Used for Ordering Flat Files

As shown in Figure G-11, sequential lists can be stored using any of three file organizations. In practice, however, they are usually kept on sequential files. In addition, although both linked lists and indexes can be stored using either indexed sequential or direct files, DBMS products almost always store them on direct files.

How Can We Represent Binary Relationships?

In this section, we examine how each of the specialized record relationships—trees, simple networks, and complex networks—can be represented using linked lists and indexes. Note that most of this discussion applies to pre-relational data models such as the network and hierarchical data models. In those models, querying the database often involved writing lots of code to follow pointers around in the database (see Review Questions G.21-G.24). With the advent of relational databases, foreign key values (instead of RRNs) are typically used to store relationships. For example, a foreign key value relating a student to an advisor (e.g. a FacultyID value) would appear in a student record. That value is then used to look up the advisor in the file containing the faculty table, using an index if one is available or a scan of the file if one is not available.

A Review of Record Relationships

Records can be related in three ways. A **tree** relationship has one or more one-to-many relationships, but each child record has at most one parent. The occurrence of faculty data shown in Figure G-12 illustrates a tree. There are several 1:N relationships, but any child record has only one parent, as shown in Figure G-13.

A **simple network** is a collection of records and the 1:N relationships among them. What distinguishes a simple network from a tree is the fact that in a simple network, a child can have more than one parent as long as the parents are different record types. The occurrence of a simple network of students, advisers, and major fields of study shown in Figure G-14 is represented schematically in Figure G-15.

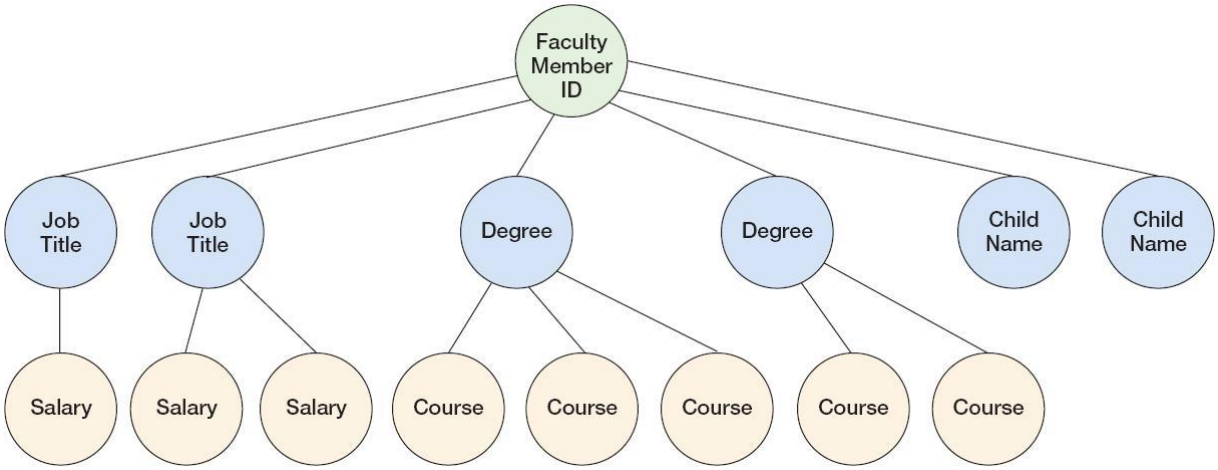


Figure G-12: Occurrence of a Faculty Member Record

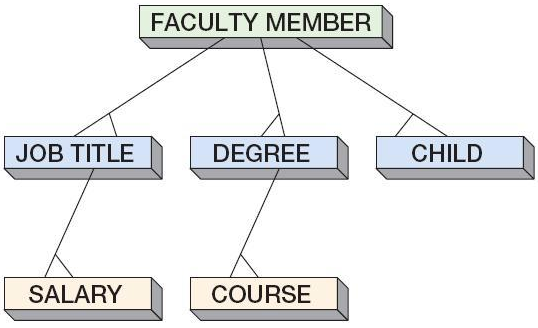


Figure G-13: Schematic of Faculty Member Tree Structure

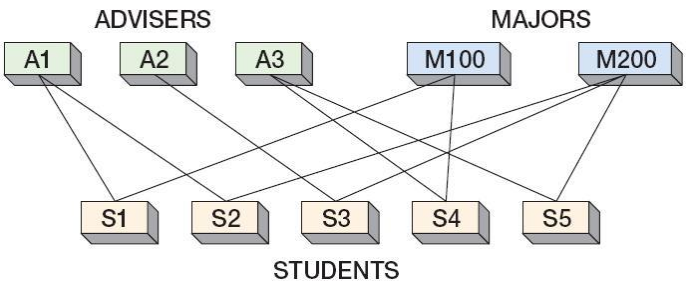


Figure G-14: Occurrence of a Simple Network

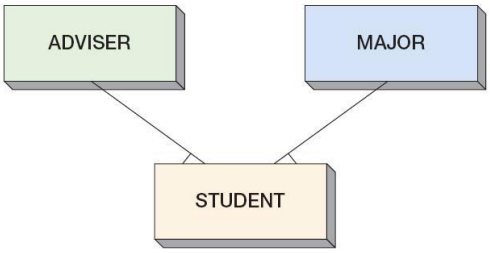


Figure G-15: General Structure of a Simple Network

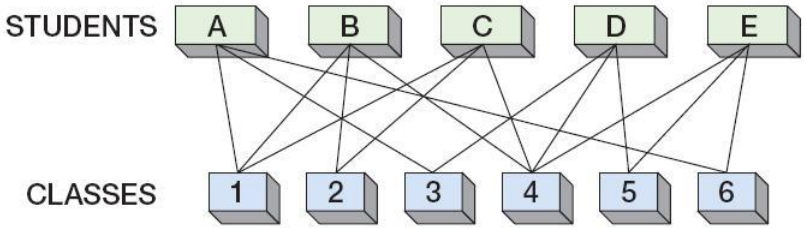


Figure G-16: Occurrence of a Complex Network

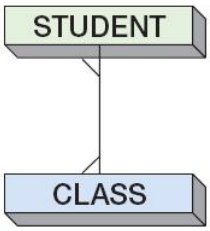


Figure G-17: Schematic of a Complex Network

A **complex network** is also a collection of records and relationships, but the relationships are many-to-many instead of one-to-many. The relationship between students and classes is a complex network. An occurrence of this relationship can be seen in Figure G-16, and the general schematic is in Figure G-17.

We saw earlier that we can use linked lists and indexes to process records in orders different from the one in which they are physically stored. We can also use those same data structures to store and process the relationships among records.

How Can We Represent Trees?

We can use sequential lists, linked lists, and indexes to represent trees. When using sequential lists, we duplicate many data. Furthermore, sequential lists are not used by DBMS products to represent trees. Therefore, we describe only linked lists and indexes.

Linked-List Representation of Trees

Figure G-18 shows a tree structure in which the VENDOR records are parents and the INVOICE records are children. Figure G-19 shows two occurrences of this structure, and all of the VENDOR and INVOICE records have been written to a direct access file in Figure G-20. VENDOR AA is in relative record number 1 (RRN1), and VENDOR BB is in relative record number 2. The INVOICE records have been stored in subsequent records, as illustrated. Note that these records are not stored in any particular order and that they do not need to be.

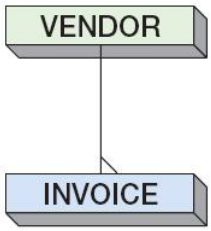


Figure G-18: Sample Tree Relating VENDOR and INVOICE Records

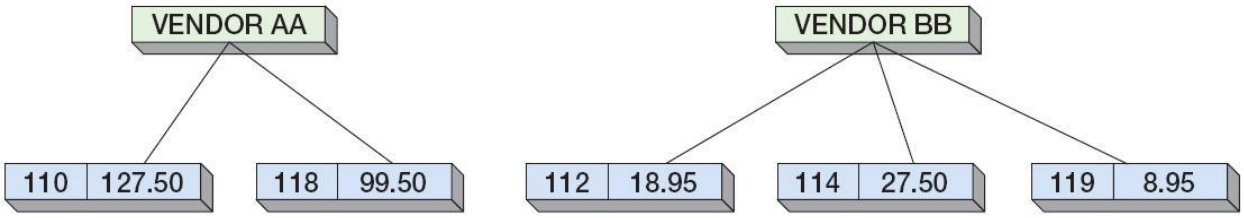


Figure G-19: Two Occurrences of the VENDOR-INVOICE Tree

Record Number	Record Contents	
1	VENDOR AA	
2	VENDOR BB	
3	118	99.50
4	119	8.95
5	112	18.95
6	114	27.50
7	110	127.50

Figure G-20: File Representation of the VENDOR-INVOICES Trees in Figure G-19

Our problem is that we cannot tell which invoices belong to which vendors from this file. To solve this problem with a linked list, we add a pointer field to every record. In this field, we store the address of some other related record. For example, we place in VENDOR AA's link field the address of the first invoice belonging to it. This is RRN7, which is Invoice 110. Then, we make Invoice 110 point to the next invoice belonging to VENDOR AA, in this case RRN3. This slot holds Invoice 118. To indicate that there are no more children in the chain, we insert a 0 in the link field for RRN3.

This technique is shown in Figure G-21. If you examine the figure, you will see that a similar set of links has been used to represent the relationship between VENDOR BB and its invoices. The structure in Figure G-21 is much easier to modify than is a sequential list of the records. For example, suppose that we add a new invoice, say number 111, to VENDOR AA. To do this, we just add the record to the file and insert it into the linked list. Physically, the record can be placed anywhere. But where should it be placed logically? Usually, the application will have a requirement; for example, children are to be kept in

Relative Record Number	Record Contents		Link Field
1	VENDOR AA		7
2	VENDOR BB		5
3	118	99.50	0
4	119	8.95	0
5	112	18.95	6
6	114	27.50	4
7	110	127.50	3

Figure G-21: Tree Occurrences Represented by Linked Lists

Relative Record Number	Record Contents		Link Field	
1	VENDOR AA		7	
2	VENDOR BB		5	
3	118	99.50	0	
4	119	8.95	0	
5	112	18.95	6	
6	114	27.50	4	
7	110	127.50	8	
8	111	19.95	3	← Inserted Record

Figure G-22: Inserting Invoice 111 into the File in Figure G-21

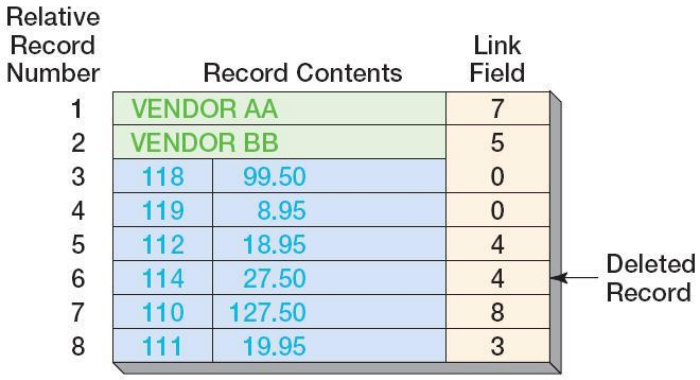


Figure G-23: Deleting Invoice 114 into the File in Figure G-22

Figure G-24: Index Representation of the VENDOR-INVOICE Relationship

ascending order on invoice number. If so, we need to make Invoice 110 point to Invoice 111 (at RRN8), and we need to make Invoice 111, the new invoice, point to Invoice 118 (at RRN3). This modification is shown in Figure G-22.

Similarly, deleting an invoice is easy. If Invoice 114 is deleted, we simply modify the pointer in the invoice that is now pointing to Invoice 114. In this case, it is Invoice 112 at RRN5. We give Invoice 112 the pointer that Invoice 114 had before deletion. In this way, Invoice 112 points to Invoice 119, as shown in Figure G-23. We have effectively cut one link out of the chain and welded together the ones it once connected.

Index Representation of Trees

A tree structure can readily be represented using indexes. The technique is to store each one-to-many relationship as an index. These lists are then used to match parents and children.

Using the VENDOR and INVOICE records in Figure G-19, we see that VENDOR AA (in RRN1) owns INVOICEs 110 (RRN7) and 118 (RRN3). Thus, RRN1 is the parent of RRN7 and RRN3. We can represent this fact with the index in Figure G-24. The list simply associates a parent's address with the addresses of each of its children.

If the tree has several 1:N relationships, several indexes will be required—one for each relationship. For example, for the structure in Figure G-13, five indexes are needed.

How Can We Represent Simple Networks?

As with trees, simple networks can also be represented using linked lists and indexes.

Linked-List Representation of Simple Networks

Consider the simple network shown in Figure G-25. It is a simple network because all of the relationships are 1:N, and the SHIPMENT records have two parents of different types. Each SHIPMENT has a CUSTOMER parent and a TRUCK parent. The relationship between CUSTOMER and SHIPMENT is 1:N because a customer can have several shipments, and the relationship from TRUCK to SHIPMENT is 1:N because one truck can hold many shipments (assuming that the shipments are small enough to fit in one truck or less). An occurrence of this network is shown in Figure G-26.

To represent this simple network with linked lists, we need to establish one set of pointers for each 1:N relationship. In this example, that means one set of pointers to connect CUSTOMERs with their SHIPMENTs and another set of pointers to connect TRUCKs with their SHIPMENTs. Thus, a CUSTOMER record will contain one pointer (to the first SHIPMENT it owns); a TRUCK record will contain one pointer (to the first SHIPMENT it owns); and a SHIPMENT record will have two pointers, one for the next SHIPMENT owned by the same CUSTOMER and one for the next SHIPMENT owned by the same TRUCK. This scheme is illustrated in Figure G-27.

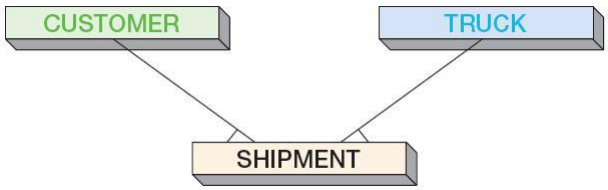


Figure G-25: Simple Network Structure

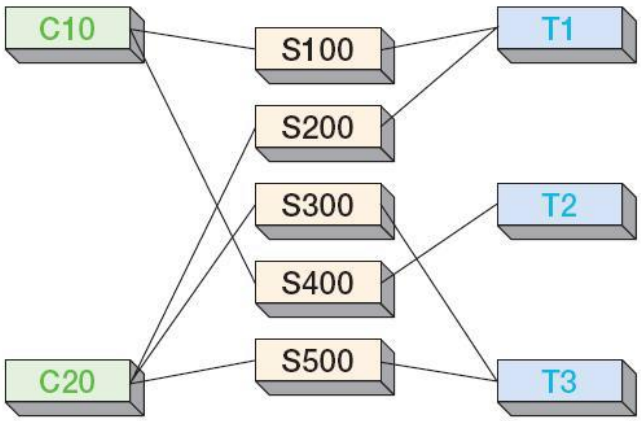


Figure G-26: Occurrence of the Simple Network in Figure G-25

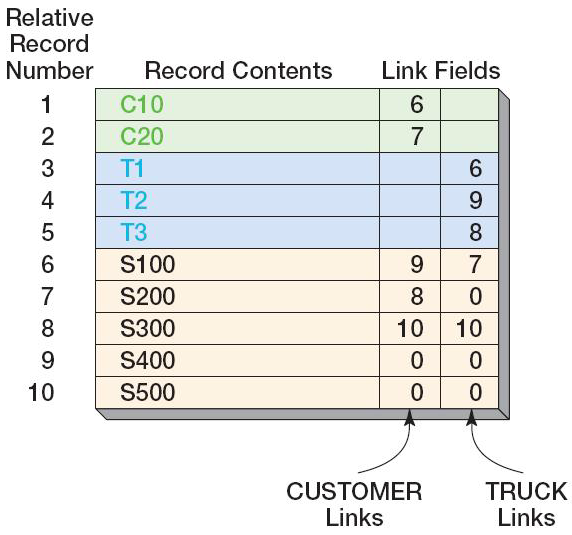


Figure G-27: Representation of a Simple Network with Linked Lists

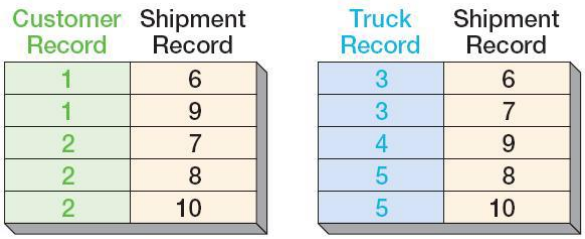


Figure G-28: Representation of a Simple Network with an Index

A simple network has at least two 1:N relationships, each of which can be represented using an index, as we explained in our discussion of trees. For example, consider the simple network shown in Figure G-25.

It has two 1:N relationships, one between TRUCK and SHIPMENT and one between CUSTOMER and SHIPMENT. We can store each of these relationships in an index. Figure G-28 shows the two indexes needed to represent the example in Figure G-26. Assume that the records are located in the same positions as shown in Figure G-27.

How Can We Represent Complex Networks?

Complex networks can be represented in a variety of ways. They can be decomposed into trees or simple networks, and these simpler structures can then be represented using one of the techniques just described. Alternatively, they can be represented directly using indexes. Linked lists are not used by any DBMS product to represent complex networks directly. In practice, complex networks are nearly always decomposed into simpler structures, so we consider only those representations using decomposition.

A common approach to representing complex networks is to reduce them to simple networks and then to represent the simple networks with linked lists or indexes. Note, however, that a complex network involves a relationship between two records, whereas a simple network involves relationships among

three records. Thus, in order to decompose a complex network into a simple one, we need to create a third record type.

The record that is created when a complex network is decomposed into a simple one is called an **intersection record**. Consider the StudentClass complex network. An intersection record contains a unique key from a STUDENT record and a unique key from a corresponding CLASS record. It will contain no other application data, although it might contain link fields. The general structure of this relationship is shown in Figure G-29. Assuming that the record names are unique (such as S1, S2, and C1), an instance of the STUDENT-CLASS relationship is illustrated in Figure G-30.

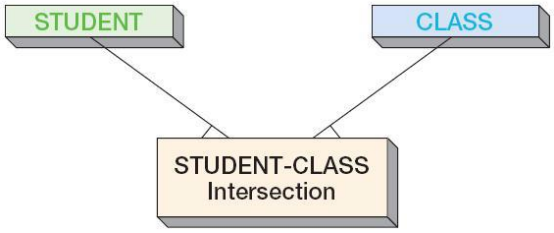


Figure G-29: Decomposition of Complex Network into Simple Network

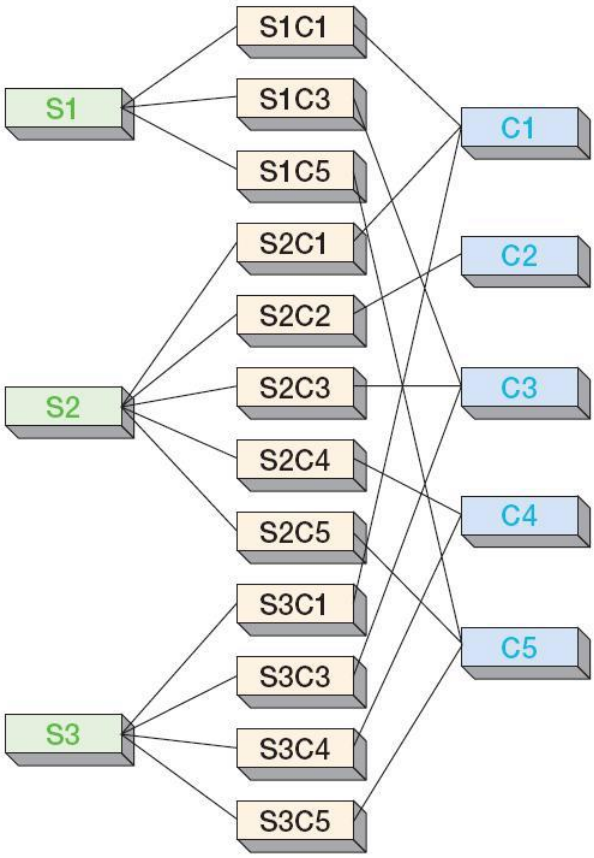


Figure G-30: Instance of the STUDENT-CLASS Relationship Showing Intersection Records

Notice that the relationship between STUDENT and the intersection record and that between CLASS and the intersection record both are 1:N. Thus, we have created a simple network that can now be represented with the linked-list or index techniques shown earlier. A file of this occurrence using the linked-list technique is shown in Figure G-31.

Summary of Relationship Representations

Figure G-32 summarizes the representations of record relationships. Trees can be represented using sequential lists (although we did not discuss this approach), linked lists, or indexes. Sequential lists are not used in DBMS products. A simple network can be decomposed into trees and then represented or it can be represented directly using either linked lists or indexes. Finally, a complex network can be decomposed into a tree or a simple network (using intersection records) or it can be represented directly using indexes.

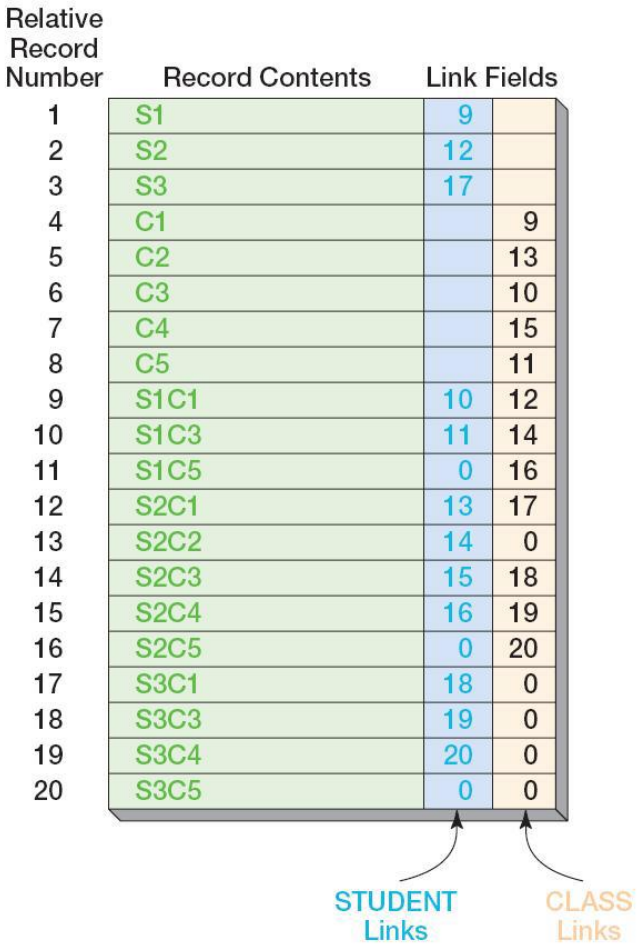


Figure G-31: Occurrence of the Network Instance in Figure G-30

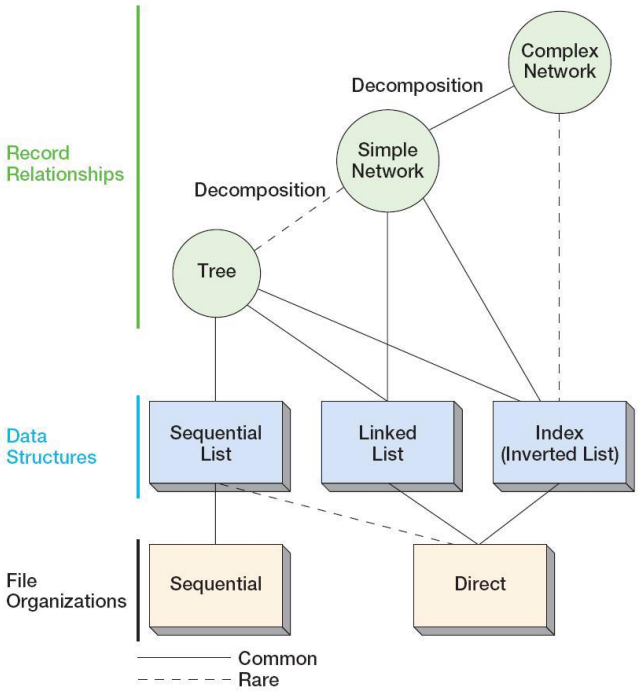


Figure G-32: Record Relationships, Data Structures, and File Organizations

How Can We Represent Secondary Keys?

In many cases, the word **key** indicates a field (or fields) whose value uniquely identifies a record. This is usually called the **primary key**. Sometimes, however, applications need to access and process records by means of a **secondary key**, one that is different from the primary key. Secondary keys may be a **unique secondary key** (such as a professor's email address) or a **nonunique secondary key** (such as a customer's zip code). In this section, we use the term **set** to refer to all records having the same value of a nonunique secondary key; for example, a set of records having zip code 98040.

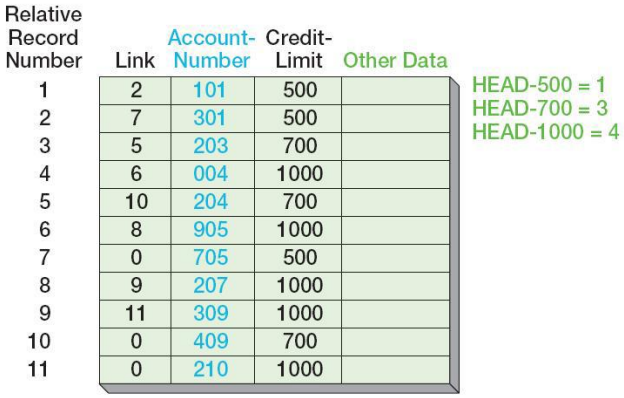
Both linked lists and indexes are used to represent secondary keys, but linked lists are practical only for nonunique keys. Indexes, however, can be used for both unique and nonunique key representations.

How Can We Represent Secondary Keys with Linked-Lists?

Consider the example of CUSTOMER records shown in Figure G-33. The primary key is AccountNumber, and there is a secondary key on CreditLimit. Possible CreditLimit values are 500, 700, and 1000. Thus, there will be a set of records for the limit of 500, a set for 700, and a set for 1000.

To represent this key using linked lists, we add a link field to the CUSTOMER records. Inside this link field, we create a linked list for each set of records. Figure G-34 shows a database of 11 customers; but, for brevity, only AccountNumber and CreditLimit are shown. A link field has been attached to the records. Assume that one database record occupies one physical record on a direct file using relative record addressing.

Figure G-33: CUSTOMER Record



Relative Record Number	Link	Account-Number	Credit-Limit	Other Data
1	2	101	500	
2	7	301	500	
3	5	203	700	
4	6	004	1000	
5	10	204	700	
6	8	905	1000	
7	0	705	500	
8	9	207	1000	
9	11	309	1000	
10	0	409	700	
11	0	210	1000	

HEAD-500 = 1
HEAD-700 = 3
HEAD-1000 = 4

Figure G-34: Representing CreditLimit Secondary Key Using Linked List

Three **pointers** need to be established so that we know where to begin each linked list. These pointers, called **heads**, are stored separately from the data. The head of the $500 linked list is RRN1. Record 1 links to record 2, which in turn links to record 7. Record 7 has a zero in the link position, indicating that it is the end of the list. Consequently, the $500 credit limit set consists of records 1, 2, and 7. Similarly, the $700 set contains records 3, 5, and 10; the $1000 set contains relative records 4, 6, 8, 9, and 11.

The $1000-set linked list can be used to answer a query such as "How many accounts in the $1000 set have a balance in excess of 900?" In this way, only those records in the $1000 set need to be read from the file and examined. Although the advantage of this approach is not readily apparent in this small example, suppose that there are 100,000 CUSTOMER records, and only 100 of them are in the $1000 set. If there is no linked list, all 100,000 records must be examined; but with the linked list, only 100 records need to be examined—namely, the ones in the $1000 set. Using the linked list, therefore, saves 99,900 reads.

Using linked lists is not an effective technique for every secondary-key application. In particular, if the records are processed nonsequentially in a set, linked lists are inefficient. For example, if it is often necessary to find the 10th or 120th or n^{th} record in the $500 CreditLimit set, processing will be slow. Linked lists are inefficient for direct access.

In addition, if the application requires that secondary keys be created or destroyed dynamically, the linked-list approach is undesirable. Whenever a new key is created, a link field must be added to every record, which often requires reorganizing the database, which is a time-consuming and expensive process.

Finally, if the secondary keys are unique, each list will have a length of one, and a separate linked list will exist for every record in the database. Because this situation is unworkable, linked lists cannot be used for unique keys. For example, suppose that the CUSTOMER records contain another unique field, say, Social Security Number. If we attempt to represent this unique secondary key using a linked list, every Social Security Number will be a separate linked list. Furthermore, each linked list will have just one item in it: the single record having the indicated Social Security Number.

How Can We Represent Secondary Keys with Indexes?

A second technique for representing secondary keys uses an index; one is established for each secondary key. The approach varies, depending on whether the key values are unique or nonunique.

Unique Secondary Keys

Suppose that the CUSTOMER records in Figure G-33 contain Social Security Numbers (SSNs) as well as the fields shown. To provide key access to the CUSTOMER records using SSN, we simply build an index on the SSN field. Sample CUSTOMER data are shown in Figure G-35(a), and a corresponding index is illustrated in Figure G-35(b). This index uses relative record numbers as addresses. It is possible to use AccountNumbers instead, in which case the DBMS locates the desired SSN in the index, obtains the matching AccountNumber, and then converts the AccountNumber to a relative record address.

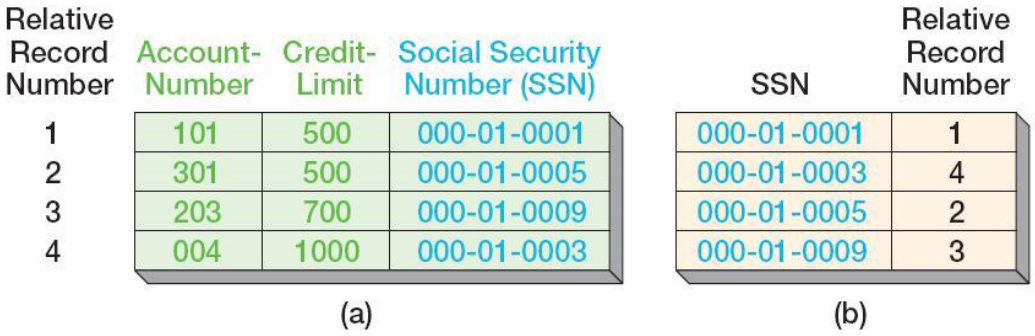


Figure G-35: Representing a Unique Secondary Key with Indexes:
(a) Sample CUSTOMER Data (with SSN) and (b) Index for SSN Secondary Key

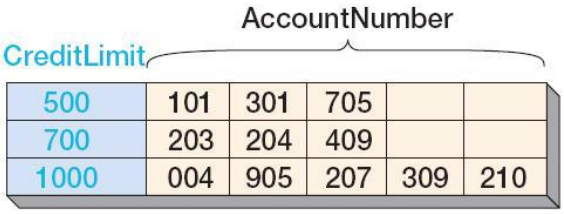


Figure G-36: Index for the Credit Limit key in Figure G-33

Nonunique Secondary Keys

Indexes can also be used to represent nonunique secondary keys, but because each set of related records can contain an unknown number of members the entries in the index are of variable length. For example, Figure G-36 shows the index for the CreditLimit sets for the CUSTOMER data. The $500 set and the $700 set both have three members, so there are three account numbers in each entry. The $1000 set has five members, so there are five account numbers in that entry.

In reality, representing and processing nonunique secondary keys are complex tasks. Several different schemes are used by commercial DBMS products. One common method uses a values table and an occurrence table. Each values table entry consists of two fields, the first of which has a key value. For the CUSTOMER CreditLimit key, the values are 500, 700, and 1000.

The second field of the values table entry is a pointer into the occurrence table. The occurrence table contains record addresses, and those having a common value in the secondary-key field appear together in the table. Figure G-37 shows the values and occurrence tables for the CreditLimit key.

To locate records having a given value of the secondary key, the values table is searched for the desired value. After the given key value is located in the values table, the pointer is followed to the occurrence

table to obtain the addresses of those records having that key value. These addresses are then used to obtain the desired records. When a new record is inserted into the file, the DBMS must modify the indexes for each secondary-key field. For nonunique keys, it must make sure that the new record key value is in the values table; if it is, it adds the new record address to the appropriate entry in the occurrence table. If it is not, it must insert new entries in the values and occurrence tables. When a record is deleted, its address must be removed from the occurrence table. If no addresses remain in the occurrence table entry, the corresponding values table entry must also be deleted.

When the secondary-key field of a record is modified, the record address must be removed from one occurrence table entry and inserted into another. If the modification is a new value for the key, an entry must be added to the values table.

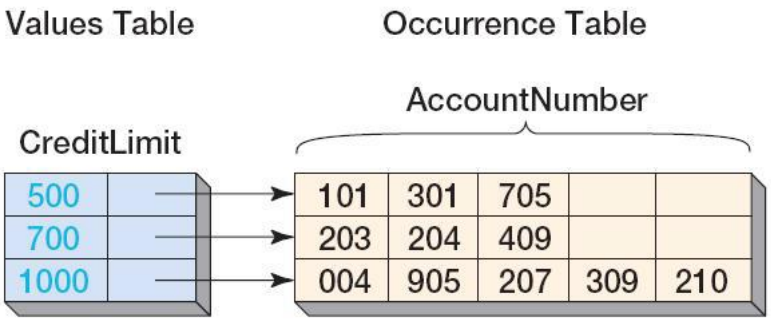


Figure G-37: Values and Occurrence Tables for the Credit Limit key in Figure G-33

The index approach to representing secondary keys overcomes the objections to the linked-list approach. Direct processing of sets is possible. For example, the third record in a set can be retrieved without processing the first or second one. Also, it is possible to dynamically create and delete secondary keys. No changes are made in the records themselves; the DBMS merely creates additional values and occurrence tables. Finally, unique keys can be processed efficiently.

The disadvantages of the index approach are that it requires more file space (the tables use more overhead than the pointers do) and that the DBMS programming task is more complex.

Note that the *application programming* task is not necessarily any more or less difficult—but it is more complex to write DBMS software that processes indexes than it is to write software that processes linked lists. Finally, modifications are usually processed more slowly because of the reading and writing actions required to access and maintain the values in the occurrence tables.

Key Terms

blocks	**B-tree**
circular list	**complex network**
direct access file organization	**flat file**
Head	**Index**
index set	**indexed sequential file organization**
intersection record	**inverted list**
key	**link field**
linked list	**nonunique secondary key**
physical records	**pointer**
pointer field	**primary key**
relative record number (RRN)	**ring**
secondary key	**sequence set**
sequential file organization	**sequential list**

set	**simple network**
tree	**two-way linked list**
unique secondary key	

Review Questions

G.1 Define a flat file. Give an example (other than one in this text) of a flat file and an example of a file that is not flat.

G.2 Show how sequential lists can be used to maintain the file in Review Question G.1 in two different orders simultaneously.

G.3 Show how linked lists can be used to maintain the file in Review Question G.1 in two different orders simultaneously.

G.4 Show how inverted lists (indexes) can be used to maintain the file in Review Question G.1 in two different orders simultaneously.

G.5 Define the term *tree*. Offer an example tree structure (other than one in this text).

G.6 Give an occurrence of the tree in Review Question G.5.

G.7 Represent the occurrence in Review Question G.6 using linked lists.

G.8 Represent the occurrence in Review Question G.6 using indexes.

G.9 Define a simple network and give an example structure(other than one in this text).

G.10 Give an occurrence of the simple network in Review Question G.9.

G.11 Represent the occurrence in Review Question G.10 using linked lists.

G.12 Represent the occurrence in Review Question G.10 using indexes.

G.13 Define complex network. Offer an example of a complex network structure (other than one in this text).

G.14 Give an occurrence of the complex network in Review Question G.13.

G.15 Decompose the complex network in Review Question G.14 into a simple network and represent an occurrence of it using indexes.

G.16 Explain the difference between primary and secondary keys.

G.17 Explain the difference between unique and nonunique keys.

G.18 Give an example of a file containing a unique secondary key (other than one in this text). Represent an occurrence of that file
using an index on the secondary key.

G.19 Define a nonunique secondary key for the file in Review Question G.18. Represent an occurrence of that file using a linked list on the secondary key.

G.20 Perform the same task as in Review Question G.19, but use an index to represent the secondary key.

G.21 Develop an algorithm to produce a report listing the IDs of students enrolled in each class using the linked-list structure in Figure G-4.

G.22 Develop an algorithm to insert records into the structure in Figure G-4. The resulting structure should resemble the one shown in Figure G-5.

G.23 Develop an algorithm to produce a report listing the IDs of students enrolled in each class using the index structure shown in Figures G-8(a), G-8(b), and G-8(c).

G.24 Develop an algorithm to insert a record into the structure shown in Figure G-8(a). Be sure to modify both of the associated indexes shown in Figure G-8(b) and G-8(c).

G.25 Develop an algorithm to delete a record from the structure shown in Figure G-34, which shows a secondary key represented with a linked list. If all records for one of the credit-limit categories (say, $1000) are deleted, should the associated head pointer also be deleted? Why or why not?

G.26 Develop an algorithm to insert a record into the structure shown in Figure G-34. Suppose that the new record has a credit-limit value different from those already established. Should the record be inserted and a new linked list established? Or should the record be rejected? Who should make that decision?