CHAPTER 12

THE PHYSICAL DESIGN PROCESS

CHAPTER OBJECTIVES

- Study the transition to physical design from logical design
- Understand how physical design conforms to the target DBMS
- Learn the physical design process for the relational database
- Discuss storing of data on physical storage
- Examine various indexing techniques to improve data access
- Identify special performance considerations

With Chapter 11, our discussions on logical design have to come to a state of completion. You now know the purpose of the logical design process; you have reviewed the steps, and you have learned the steps and tasks of the process. First, you model the information requirements by creating a semantic data model. Then you complete the logical design process by transforming the semantic data model into a conventional data model such as the relational, hierarchical, or network data model. Because the relational data model is superior to the others and because it is widely used, we emphasized the relational data model in our discussions. You have also studied two data modeling techniques for creating a semantic data model—the object-based data modeling and entity-relationship data modeling techniques.

Remember that a semantic data model is a generic data model. It has no relational, hierarchical, or network flavor. A semantic data model does not consist of tables with rows and columns; only a relational data model does. Only when you move from a semantic data model to a conventional data model such as the

Database Design and Development: An Essential Guide for IT Professionals by Paulraj Ponniah
ISBN 0-471-21877-4 Copyright © 2003 by John Wiley and Sons, Inc.

relational data model, do you represent and perceive data as being contained in two-dimensional tables or relations.

From the relational data model we proceed to the physical design to represent the information requirements in physical hardware storage. Semantic and conventional data models are logical data models; these are conceptual data models. They are conceptual representations, not representations of how data are actually stored. Actual data are going to reside on physical storage, and physical design stipulates how, where, and which data are stored in physical storage. This chapter covers the physical design of relational database systems. The emphasis is exclusively on relational databases. When we discuss any aspect of physical design, assume that the discussion refers to relational databases.

INTRODUCTION TO PHYSICAL DESIGN

For a moment, think of the logical design process in terms of its output. Take, for instance, the logical design of a relational database system. At the end of the process, you arrive at a set of two-dimensional tables or relations. In the standard notation, you represent the set of relations, indicating the columns, primary keys, and foreign keys. This is the output from the logical design. The output is still at a conceptual level, not at the practical and physical level for storing actual data.

To store and manage data on physical storage, the representation of the logical design must be transformed into a representation for actual data storage. Why is this necessary? Computer systems do not store data in the form of tables with columns and rows; they typically store data in a different way. Computer storage systems store data as files and records. Physical design is the process of implementing the database on physical storage. Therefore, in the physical design process, you are concerned with the features of the storage system on which your database will reside. Furthermore, you are also concerned with the functions, features, and facilities of the DBMS selected to manage your database.

Figure 12-1 illustrates the transition to physical design in the DDLC. Note the place of physical design in the overall design process.

Logical to Physical Design

Logical design produces a data model in one of the standard conventional models. For the purpose of our discussions here, we will focus on the relational data model as the chosen conventional model. Although logical design is independent of any particular DBMS and details of physical hardware and systems software, physical design must necessarily take these into account. Physical design has to be tailored to a target DBMS. The features of the host computer system must be used to shape the output of the physical design process. The physical designer must know how the target DBMS works and must be very familiar with the capabilities and working of the host computer system.

So in physical design, the last stage of the database design process, your objective is to proceed from the logical design to implement the database system as a set of records, files, and other data structures on physical storage. When doing this, you

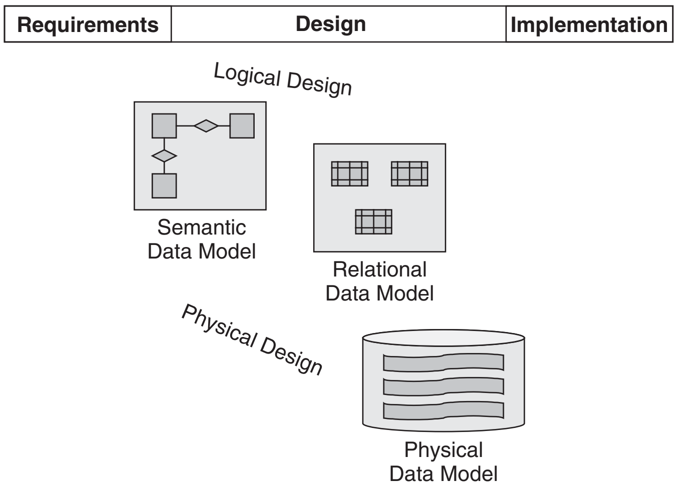


Figure 12-1 Physical design in the overall design process.

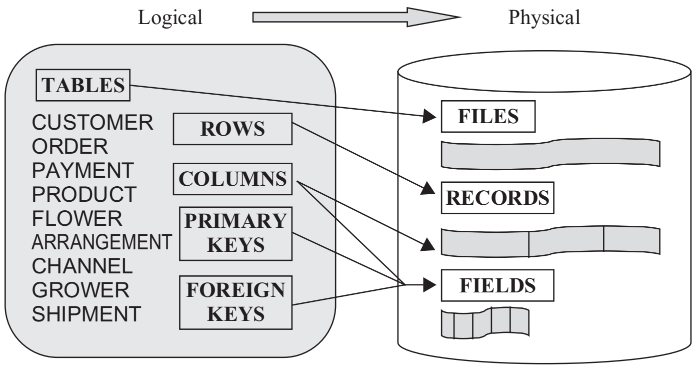


Figure 12-2 Logical design to physical design.

want to ensure adequate performance levels and provide for security and data integrity. You want to design your data structures on storage in such a way as to safeguard against any data loss.

Let us get back to the logical design for the florist business discussed in Chapter 11. How do you make the transition from this logical design to the physical design for the database to support the business? What are the overall components of the physical design? How do the components of the logical design map to those of the physical design? Figure 12-2 portrays the transition to physical design from logical design.

Examine the mapping of the components between logical design and physical design. As a result of the logical design, you have tables or relations with columns and rows for each table. These are the data as represented in the logical model for a relational database system. Now, in physical storage, you have files and records within files. How do you map relations, columns, and rows to files and records? Do you store one relation per file? Or is it more efficient to store several relations in one computer file? Are there advantages if you intend to store related rows from two relations in one file? What happens when the database system grows in size and more transactions begin to access the system? Decisions such as these make up the physical design process. We will consider all the relevant issues and describe the goals and tasks of physical design.

Goals and Design Decisions

Of course, the main objective of physical design is the implementation of the logical design on physical storage. Only then can you begin to store data in the target database and provide for its usage by users. Although this objective appears to be straightforward, when designing the data structures to be stored you need to be concerned with certain crucial issues.

If the way you store data in storage makes data retrieval cumbersome and inefficient, your physical design lacks quality and usefulness. If your data storage method is not compatible with the features of the host system and is difficult for the target DBMS to work with, then again your physical design becomes ineffective. Therefore, let us consider the main goals and decision issues.

Physical Design Goals Two major goals stand out for the physical design process. All tasks must be conditioned to achieve these goals.

Database performance. Your physical design must ensure optimal performance. This means, as data are retrieved from storage, retrieval must be as fast as possible. Again, when a piece of data is written on storage, it must be done as quickly as possible. Fast data operations constitute a major goal. In terms of physical storage, this translates into minimizing the time for each input/output (I/O) operation. This concern for database performance must continue as usage intensifies after the initial deployment. The physical design must be open and flexible to accommodate all future adjustments for continued better performance.

Data management. As data are stored in your database systems and begin to be used, you now have a vital resource to be managed well and protected. Your physical design must enable effective management of the database. This may involve grouping of tables to be managed as a collection for the purpose of backup and recovery. This may require setting up of proper data views for individual user groups and protecting the database from unauthorized usage.

In addition to the two major goals of database performance and data management, let us list a few other underlying objectives in the physical design process. Consider the following:

Scalability. As the usage of the database system continues to increase, you must ensure that the physical design enables upward scalability of the system. When you need to consider upgrades to hardware, operating system, and the DBMS itself, the physical design must be able to adapt itself to the enhancements.

Openness and flexibility. Business conditions do not stay static but keep changing all the time. What happens when business conditions change? These changes must be reflected in your database system so that it can support business operations adequately. On the basis of the changes, the logical data model will change and evolve. How should this affect the actual implementation of the database system? The physical design must be open and flexible enough to accommodate revisions to the logical design.

Security. Safeguarding and protecting the database system from unauthorized access is accomplished through the physical design. You must ensure that the security features in the physical design provide thorough and comprehensive protection.

Ease of administration. Administrative details include database backup, recovery from failures, and ongoing space allocation. The physical design must ensure easy administration with simple and straightforward features.

Physical Design Decisions Initially and on an ongoing basis, the following design issues require serious consideration and appropriate decisions:

- Features, functions, and options supported by the target DBMS
- Features and capabilities of host computer system
- Disk storage configuration
- Usage of data—access patterns and access frequencies
- Data volumes—initial volumes and growth rates
- Hardware and software—initial setup and ongoing upgrades

Physical Design Components

When you finish with the physical design phase, what collective set of components do you expect to produce? At the end of physical design, what is the set of components that are expected to emerge? What major components make up the physical design of your database system? Here is a list of the essential components:

File organization For each file in the physical model, the method or technique for arranging the records of the file on physical storage

Indexes Types and list of index files to improve data access

Integrity constraints Constraints or rules in the form of data edit rules and business rules; rules for enforcement of referential integrity on inserts and deletes

Data volumes Sizing for each data and index file

Data usage Analysis of expected data access paths and access frequencies

Data distribution Strategy for providing data to users by means of centralized, partitioned, or replicated database systems

Physical Design Tasks

To understand the significance of the physical design components, we need to discuss the tasks that produce these components. As we go through the physical design tasks, you will note how these tasks take into account the target DBMS and physical storage. The features of the DBMS and the configuration of physical storage structures affect and influence the physical design to a large extent.

Consider the following list of major tasks in the physical design process:

Translate global logical data model into physical data model for the target DBMS Design files conforming to DBMS and physical storage configuration for the base relations in the global logical data model. Include integrity constraints—data edit rules, business rules, and constraints for referential integrity.

Design representation on physical storage Analyze potential transactions. Determine organization for each file. Choose secondary indexes. Establish other performance improvement techniques. Estimate storage space requirements.

Design security procedures Design user views to regulate usage of data elements by select user groups. Design data access rules.

Use of Data Dictionary

The data dictionary records the components of the design phase. In the logical design phase for a relational database system, you determine all the relations that comprise the logical data model. Now in the physical design phase, you conform your representation to the features of the selected DBMS, the physical hardware, and the systems software configuration to come up with the set of physical files. You also determine index files and other performance improvement strategies. Furthermore, you include the integrity constraints.

Types of Information The data dictionary or catalog is used to record all of the data about the contents of your database system. The following is a list of the types of information contained in the data dictionary or catalog:

Relations Relation names, number of attributes in each relation

Attributes Attribute names within each relation, data type and length for each attribute, value domain for each attribute

Indexes Index names, names of corresponding relations, index types, index attributes for each index

Constraints Data edit rules, business rules, referential integrity constraints

Users Names of authorized users, user groups, accounting information about users, passwords

Data views Names of the views, definition of each data view

Security Authorization for individual users or user groups through data views

Statistical Number of tuples in each relation, data usage statistics

Space Management Storage space allocation for database and files in terms of database records, disk extents, disk block management parameters, storage method for each relation—clustered or nonclustered

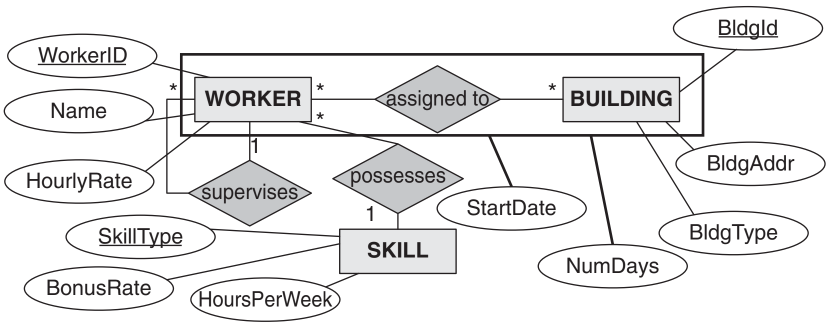
Defining Design Components As noted above, the data dictionary contains definitions for all the design components. As you know, the language used for defining these components is known as Data Definition Language (DDL). For the relational data model, SQL (Structured Query Language), which has evolved as the standard, contains a DDL component. Let us take a specific example of a logical data model and express the definitions for the data dictionary with SQL.

First, this is a partial data model consisting of just four relations. The data model represents the information requirements for the assignment of workers to building construction. Figure 12-3 shows the semantic data model and the relational data model denoted with standard notation.

Figure 12-4 lists the SQL statements used for defining the data model in the data dictionary.

Note the following statements in the figure:

- Definition for each relation with CREATE TABLE statement
- Definition of attributes with their data types, data lengths, and value domains
- Default values for selected attributes
- Primary key statements
- Foreign keys to represent the relationships



Relational Notation

WORKER (<u>WorkerId</u>, Name, HourlyRate, SkillType, SupvId)
Foreign Keys: **SkillType** REFERENCES **SKILL**
SupvId REFERENCES **WORKER**
BUILDING (<u>BldgId</u>, BldgAddr, BldgType)
SKILL (<u>SkillType</u>, BonusRate, HoursPerWeek)
ASSIGNMENT (<u>WorkerId</u>, <u>BldgId</u>, StartDate, NumDays)
Foreign Keys: **WorkerId** REFERENCES **WORKER**
BldgId REFERENCES **BUILDING**

Figure 12-3 Logical model for building construction.

```
CREATE SCHEMA RARITAN-CONSTRUCTION
  AUTHORIZATION TOM NETHERTON
CREATE TABLE WORKER (
  WorkerId          CHARACTER (6)
          PRIMARY KEY,
  Name              CHARACTER(25),
  HourlyRate        NUMERIC (5,2),
  SkillType         CHARACTER (8),
  SupvId            CHARACTER (6),
  FOREIGN KEY SupvId REFERENCES WORKER
                       ON DELETE SET NULL)
CREATE TABLE BUILDING (
  BldgId            CHARACTER (6)
          PRIMARY KEY,
  BldgAddr          CHARACTER (35),
  BldgType          CHARACTER (9)
          DEFAULT 'Office'
          CHECK (BldgType IN
          ('Office', 'Warehouse', 'Retail', 'Home') )
CREATE TABLE SKILL (
  SkillType         CHARACTER (8)
          PRIMARY KEY ,
  BonusRate         NUMERIC (5,2) ,
  HoursPerWeek      NUMERIC (2)
          DEFAULT 40
          CHECK (HoursPerWeek > 0 and
                    HoursPerWeek < = 60 ))
CREATE TABLE ASSIGNMENT (
  WorkerId          CHARACTER (6) ,
  BldgId            CHARACTER (6) ,
  StartDate         DATE ,
  NumDays           Numeric (3) ,
  PRIMARY KEY (WorkerId, BldgId) ,
  FOREIGN KEY WorkerId REFERENCES WORKER
                    ON DELETE CASCADE
  FOREIGN KEY BldgId REFERENCES BUILDING
                    ON DELETE CASCADE )
TABLE SPACE Main_Extent
STORAGE ( INTIAL 1M, NEXT 100K,
  MINEXTENTS 1, MAXEXTENTS 10, PCTINCREASE 10);
```

Figure 12-4 SQL statements for building construction data model

- Data edits with CHECK statements
- Referential integrity constraints
  - ON DELETE SET NULL (set foreign key value to be null when corresponding parent entity is deleted)
  - ON DELETE CASCADE (if a parent entity is deleted, cascade the delete and delete child entity as well)
- Storage statement indicating disk space extents

DATA STORAGE AND ACCESS

Physical design is the process of implementing the database on physical storage. Therefore, a close look at data storage mechanisms and the methods of data access will help you get a better understanding of physical design. What are the various levels of data storage? How are data organized as physical files that reside on physical storage? When a user requests for a particular set of data from the database system, how do different software and hardware layers execute the request? What are the implications of data storage management in the physical design process?

We will cover the various aspects of storage management and trace the handling of data requests. Data are stored as records within file blocks. We will explore the nature of file blocks and understand how block management is part of the physical design process. Records may be arranged in files in a variety of ways. Considering the major access patterns, records must be arranged with the most optimal method. This is called file organization, and we will review the major file organizations.

Storage Management

Data in a relational table may be taken as a collection of records. For example, consider a relational table containing customer data. Each row in the CUSTOMER relational table refers to one specific customer. All the columns in each row refer to the attributes of the particular customer. A table consists of rows, and each row may be considered as a record. Thus customer data contained in the CUSTOMER relational table are a collection of customer records.

Data in physical storage are kept as blocks. A block of data spans a given number of physical disk tracks. A physical file has a certain number of disk blocks allocated to it. For example, the customer file could be allocated 100 blocks. How much space is then allocated to the customer file? That depends on how big the block size is. File blocks may be specified in multiples of the block size determined in the operating system. If the operating system determines the block size to be 2K, then you may have file blocks with sizes of 2K, 4K, and so on.

From the perspective of the higher levels of data storage software, data reside in physical storage as files containing data blocks. Getting back to our customer table example, customer records will reside in the blocks of the customer file. Depending on the block size and the size of the customer records, a certain number of records may be stored in one block. All of the customer records stored in the various blocks make up the customer file. Figure 12-5 illustrates how records are stored in file blocks. The figure also shows how file blocks are related to the storage sectors of the storage medium.

DBMS and the operating system work together to manage the file blocks. They continuously keep track of which blocks are filled up and which ones have room for more records. They monitor the blocks that are being requested to be fetched and those that are already fetched and in use. Storage management encompasses all tasks to manage the movement of data in file blocks.

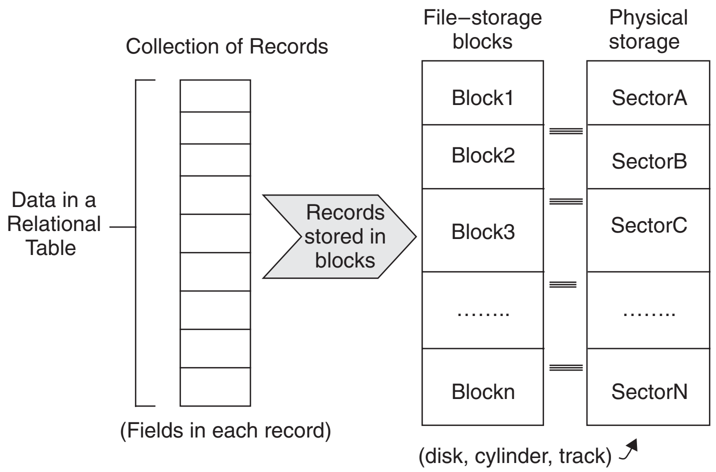


Figure 12-5 Data records in file storage blocks.

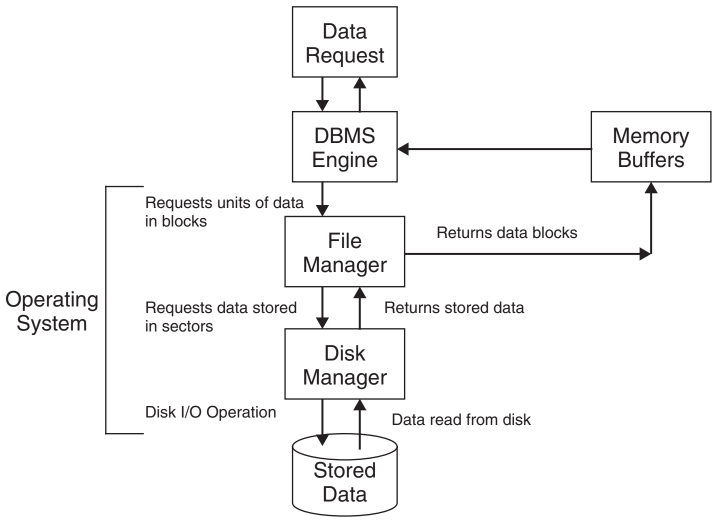


Figure 12-6 How physical data are accessed.

Access of Physical Data

Before proceeding further, you need to understand how data are accessed and fetched from physical storage. You need to know the significance of arranging data in file blocks. A block is the unit of data transfer from physical storage into main memory. Typically, a block of data is fetched from the physical storage device in one input/output (I/O) operation.

Assume that a request is made to obtain the data of customer number 12345678 from storage. You know that the data record for this customer resides in one of the blocks of the customer file. How is the requested customer data retrieved? First, the DBMS determines the particular block on which this customer record resides. Then the DBMS requests the operating system to fetch the specific block. Figure 12-6 illustrates the operation for fetching the specific block. The figure also shows how the request is processed through different software and hardware layers. Note that data are fetched into main memory as blocks and not as individual records.

Files, Blocks, and Records

A quick recapitulation of the essential points covered we have covered so far:

- Data are stored as files in physical storage.
- A file is made up of data blocks.
- A block contains a certain number of records.
- A block is the unit of transfer for reading data from physical storage or for writing to physical storage.
- Data are retrieved as blocks from physical storage into main memory.

Block Addressing Block addressing is the general technique for database records. Suppose that a file of data is allocated *n* blocks and that each block can hold up to *m* records. That means the file may contain up to a maximum of $(m \times n)$ records. Each record gets placed in a specific block. To read or write to a specific record, the DBMS addresses the block in which the record must reside. This is determined according to the way the file is organized. When a record is deleted from a block, the space occupied by the record in the block is freed up. As more and more records are deleted, the block may contain a number of fragmented empty spaces.

Figure 12-7 indicates the contents of a storage block.

Let us review the figure. Note the block header and the important information the header holds. The holes shown in the figure are empty spaces resulting from deletion of records from the block. The holes are chained together by physical addresses so that the system software can look for appropriate contiguous empty space for inserting a new record in a block.

Note the size of each hole. Let us say that a new record of size 1.3K has to be inserted in the block. The system software searches for an empty space large enough in the block to store this record. Now the first hole has enough space, and so the record gets stored there. Let us now say that another record of size 1.8K has to be inserted in the block. Neither of the two remaining holes has contiguous space large enough for this second record. Where will the system software insert the block?

The system software chooses one of the two predefined options:

1. Because there is no single hole large enough, place the entire new record in the overflow block. Therefore, when the need to retrieve the record arises, the system software will first look for it in the original block and then proceed to the overflow block.

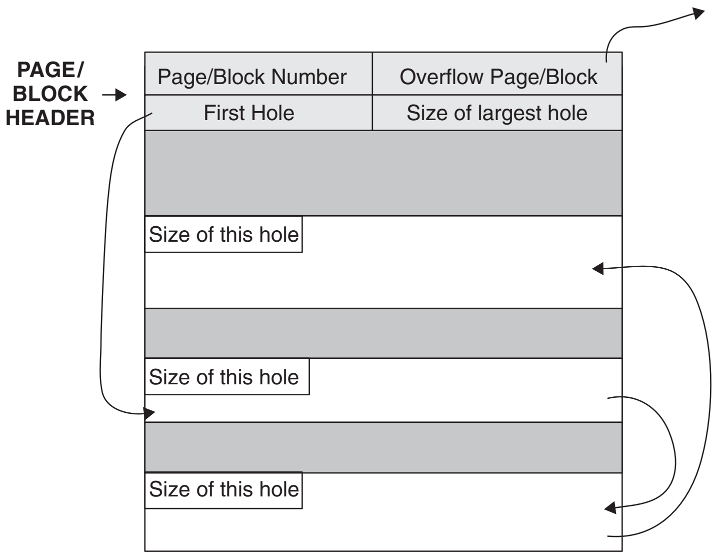


Figure 12-7 Contents of a storage block.

2. Place part of the new record in the original block and the rest of the record in the overflow block. The two parts of the records are chained together by physical addresses.

You must already have realized the imposition of system overhead while writing or reading when either the whole record or part of the record is placed in the overflow block. Block chaining and record chaining come with a price. Therefore, block management becomes significant.

Block Size As you know, the file block is the unit of data transfer between disk storage and main memory. The goal is to be able to transfer large volumes of data from disk storage in a few I/O operations. How can you accomplish this? It would seem that if you make the block size large, you fit more records in a block and therefore move a greater number of records in fewer I/O operations. Let us consider the option of making the block size large.

First, make sure your operating system does not fix the block size. If the operating system supports a certain fixed block size and does not allow larger file blocks, then this option of using large block sizes is not open to you. Usually, file block sizes are allowed to be defined in multiples of the operating system block size. If this is the case in your environment, take advantage of this option.

Each block contains a header area and a data area. The data area represents the effective utilization of the block. The header has control information, and the size of the header does not change even if the data area becomes larger. For a small block, the header utilizes a larger percentage of the total block size. As a percentage of the total block size, the header utilizes a small percentage of a large block.

What are the effects of having large blocks?

- Decrease in the number of I/O operations
- Less space utilized by block headers as percentage of total block size
- Too many unnecessary records retrieved in the full block even when only a few records are requested for

You cannot arbitrarily keep on increasing the block size assuming that every increase results in greater efficiency. If the data requests in your environment typically require just a few records for each request, then very large block sizes result in wasteful data retrievals into memory buffers. What will work is not the largest block size, only the optimal block size. Determine the optimal block size for each file on the basis of the record sizes and the data access patterns.

Block Usage Parameters Consider the storage and removal of records in a block. As transactions happen in the database environment, new records are added to a block, using up empty space in the block; records get deleted from a block, freeing up space in the block; records change their lengths when they are updated, resulting in either using up more space in the block or freeing up some space in the block. DBMSs provide two methods for optimizing block usage. In the various commercial DBMSs, these block usage parameters may go by different names. Nevertheless, the purpose and function of each block usage parameter are the same. Let us examine these parameters and understand how they work.

Block Percent Free This parameter sets the limits on how much space is reserved in each block for expansion of records during updates. When a record has variable length fields, it is likely to expand when transactions update such fields. Also, a record is likely to expand when null values of attributes are replaced with full data values.

Let us take a specific example of this block usage parameter: PCTFREE 20

This means that 20% of each block is left empty for the records in the block to expand into when updates to the records take place. Assume a file with records having many variable length fields subject to many frequent updates. Then you should allocate enough empty space in each block for updated records to occupy. The Block Percent Free parameter, in such a case, must be set at a high level to allow a larger percentage of each block to be left empty. However, if you set the parameter at too high a level, large chunks of blocks may be reserved unnecessarily and wasted.

Block Percent Used Block management by the DBMS involves keeping track of blocks ready and available for insertion of new records. DBMS can do this efficiently only if it marks a block as available for new records only when the block is fairly empty. This reduces the overhead for managing too many "free" blocks. The Block Percent Used parameter sets an indicator when a block is ready and available for new records.

Take a particular example of this block usage parameter: PCTUSED 40

What is the meaning of this parameter? It is like a watermark below which the usage of a block must dip before new records are allowed to be inserted in the block. Only when space used in a block falls below 40%, can new records be accepted in the block. Obviously, deletion of records keeps changing the actual percentage of the block being used at a given time. If, at a given time, the percentage of the block used is 41%, leaving 59% free, new records are still not allowed to be inserted into the block. If a record deletion causes the percent block used to go down to 39%, new records are allowed to be inserted into the block. If a file is fairly volatile, with many deletions and insertions of records, then set this parameter as low as possible.

File Organization

Go back to Figure 12-5 showing how a physical file contains data records. A file is an arrangement of records, and these records are spread over the storage blocks allocated to that file. Data records reside in the database as records arranged in files. Before proceeding further about how records may be arranged in a file, let us review the major operations on records stored in a database.

Here are the main types of operations:

Insert. Add a record to the database. Identify the file and the block within the file, and insert a record in the block.

Delete. Identify the file and the block where the record resides. Delete the record from the block and free up the space.

Equality selection. Fetch only the records where the value of a specific attribute equals a given value.

Range selection. Fetch only those records where the value of a specific attribute is within a given range of values.

Scan. Full file scan. Fetch all the records in a file.

How can you arrange the records in a file? Whether you keep the records in a certain sequence of values of a specific attribute or just throw the records into the file in a random fashion, the arrangement of records in a file affects the types of data operations listed above. The arrangement of records in a file determines how each record gets stored in storage blocks. Therefore, how you arrange the records in a file becomes important and influences efficiency of data access. The database administrator must choose the right arrangement of records in each file.

DBMSs support several methods for arranging records in a file. File organization refers to the way records are arranged in a file. There are three basic file organizations: heap, sequential, and hash. Each file organization affects the database operations in a certain way. Let us examine each of the basic organizations and note the circumstances for which each organization is best suited.

Heap or Pile Organization

- Records placed in the order of creation
- No ordering of records
- Usually single file for each relational table
- New record inserted on last block of file
- Inserting records very efficient
- Searching very inefficient—linear search from beginning to end of file
- Deleting leaves wasted space in the block
- Useful for collecting and storing data for later use
- Good for database backup and transportation of data
- Best if most common operation is full file scan

Sequential Organization

- Records sorted on some search key field—only one search key field per file
- Records placed in sequence or chained together in the order of data values in search key field
- Allows records to be read in search key order
- Fast retrieval in search key order
- Inserting consecutive records takes place in the same block or overflow blocks
- Deletion leaves wasted space in the block

- Searching for random records inefficient
- Best if most common operation is range selection

Hash or Direct Organization

- A hash function computes addresses based on values in specified field
- Hash function specifies the block where a record must be placed
- Search for record in a block done in memory buffer
- Very fast retrieval for random data
- Inserting specific record fast and direct
- Deletion leaves wasted space in the block
- Not good for sequential retrieval
- Best if most common operation is equality selection
- Hashing can cause collisions when many records hash to the same block
- Methods for resolution of collisions (linear probing or multiple hashing)

Figure 12-8 illustrates the problem of collisions in direct file organization.

In this figure, you will note that a very simple hashing function is used—the last two digits of the employee social security number are used to indicate the storage address for the employee record. Employee 102-49-2121 comes on board, and his record gets stored in Block 21. Then employee 352-11-3421 joins the company and the hash algorithm calculates the storage address as 21, but Block 21 is already filled

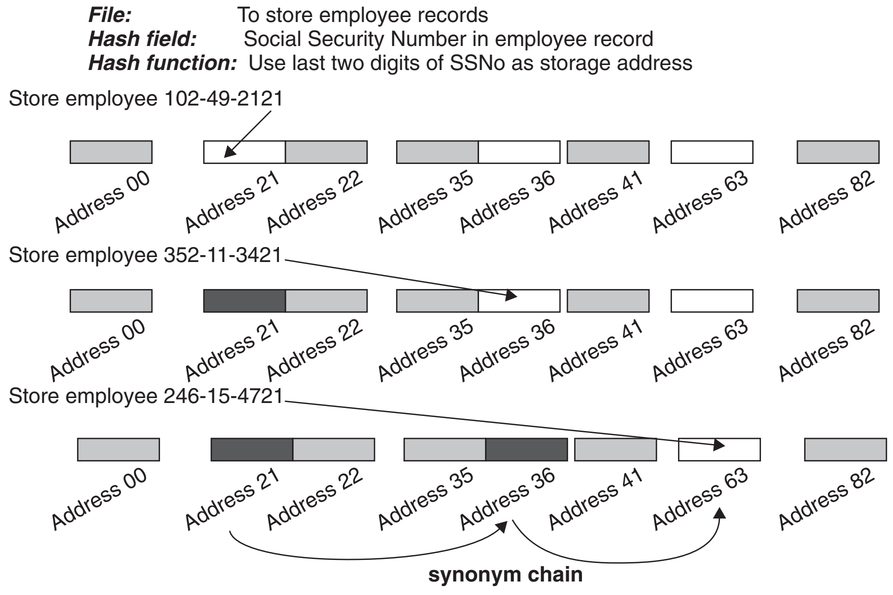

File: To store employee records
Hash field: Social Security Number in employee record
Hash function: Use last two digits of SSNo as storage address
Store employee 102-49-2121
Address 00
Address 21
Address 22
Address 35
Address 36
Address 41
Address 63
Address 82
Store employee 352-11-3421
Store employee 246-15-4721
synonym chain

Figure 12-8 Direct file organization: resolution of collisions.

up. Therefore, DBMS does linear probing and sequentially searches for the next available empty block. That happens to be Block 36, and the employee record for 352-11-3421 is stored there. In the same way, when employee 246-15-4721 joins the company, her record gets stored in the next available empty block, namely, Block 63. Instead of linear probing, another method for resolving collisions is to find the appropriate address by means of applying multiple hash functions. The data records that hash to the same storage address are known as synonyms in direct file organization.

Now imagine that you want to retrieve the record for employee 246-15-4721. You apply the hash function and determine that the record must be found at address 21. However, the record is not in that block; the block contains record for employee 352-11-3421. So how does the DBMS find the record for 246-15-4721? Does it have to search through all blocks sequentially? That would be very inefficient. The DBMS maintains a synonym chain to link all the records that hash to the same block address. This is a list of pointers linking the blocks in which the synonyms are stored. Therefore, when record for 246-15-4721 is not found at address 21, the DBMS follows the synonym chain and locates the record at address 63.

Linking of Related Data Elements

In the previous section, we mentioned pointers and linking of storage units. The storage blocks where synonyms are stored need to be linked together by physical addresses. Frequently in physical implementation of databases, you will come across the need to link records by means of physical addresses or pointers. You will observe the need for physical pointers when we discuss indexes a little later in this chapter. Therefore, let us briefly discuss how related data elements are linked together in physical storage.

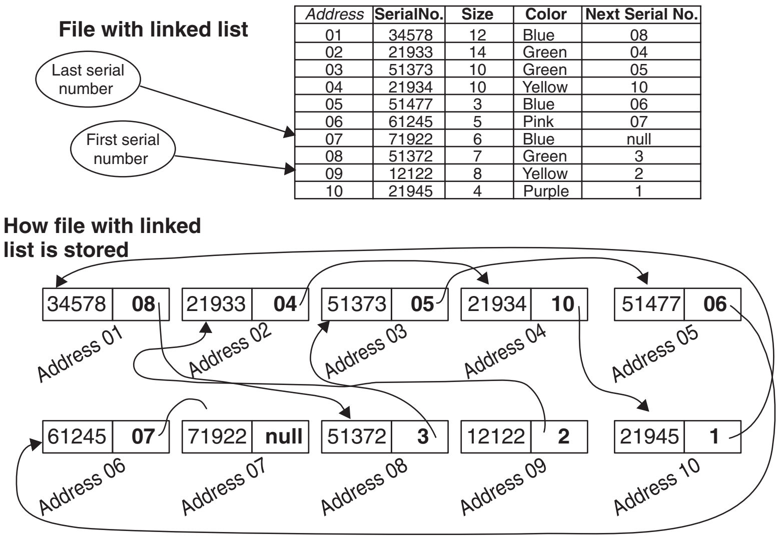
You link two records by placing the physical address of one record as part of the other record itself. These addresses placed in records to connect other records are known as pointers. The list of records linked together forms a linked list. You may link records within the same file through intrafile pointers; similarly, you may link records in more than one file through interfile pointers.

Figure 12-9 shows a file in which the records are connected to form a linked list.

In the example illustrated in the figure, the file consists of 10 records apparently stored in a random sequence. How can you arrange the records in the sequence of serial numbers without moving the records? You can accomplish this by forming a linked list of the records connecting the records in serial number sequence. Note that the address in each record points to the next record in serial number sequence. Also, observe the arrows pointing to the consecutive records in the linked list. The last record in the sequence does not have to point anywhere and therefore contains a *null* address in the pointer field.

What types of addresses are the addresses shown in the above figure? Are these the actual addresses of the physical storage locations? The addresses may be of one of two types.

Absolute addresses. Each record in the linked list contains the actual physical address to point to the next record.



Address	SerialNo.	Size	Color	Next Serial No.
01	34578	12	Blue	08
02	21933	14	Green	04
03	51373	10	Green	05
04	21934	10	Yellow	10
05	51477	3	Blue	06
06	61245	5	Pink	07
07	71922	6	Blue	null
08	51372	7	Green	3
09	12122	8	Yellow	2
10	21945	4	Purple	1

Figure 12-9 File with a linked list.

- Absolute addresses are device-dependent. If you need to reorganize the file or move the file to a different storage medium, you have to change all the addresses in the linked list. This could be a major disadvantage.
- On the other hand, absolute addresses provide a fast method for retrieving records in the sequence of the linked list. You fetch each record, get the address of the next record, and use that address directly to get the next record, and so on.

Relative addresses. Each record in the linked list contains a relative address, not the actual physical address, to point to the next record. The relative address of a storage location is a number indicating the relative offset of the location from the beginning of the storage device.

- Relative addresses are device-independent. If you need to reorganize the file or move the file to a different storage medium, you do not have to change the addresses in the linked list.
- On the other hand, data storage and retrieval are slower with relative addresses. Every time you want to store or retrieve a record, you have to calculate the actual physical address from the relative address. You have to contend with address-resolution overhead.

RAID Technology Basics

Currently, in most organizations, disk storage devices form the primary media for databases. Because storage management is part of the physical design considerations, we need to discuss the nature of disk storage devices. Although disk storage devices have improved tremendously over the past decade, still they have two inherent problems. First, despite major advances, disk storage is still slow for data movement compared to the other hardware components such as main memory and the processor. Second, disk storage devices have more mechanical parts compared to the other components. Therefore, disk storage devices are more prone to failures.

RAID (redundant array of inexpensive disks) technology addresses these two problems as follows:

- Improve performance through data striping. Stripe or spread data across several disks so that storage or retrieval of required data may be accomplished by parallel reads across many disks. When data is stored on a single disk unit, you may need sequential reads, each sequential read waiting for the previous one to finish.
- Improve reliability by storing redundant data to be used only for reconstructing data on failed units.

We will now examine how RAID technology provides these improvements. In today's database environment, the use of RAID technology has become essential.

Performance Improvement Data striping enables an array of disks to be abstracted and viewed as one large logical disk. While striping data across disks, you take a specific unit of data and store it on the first disk in the array, then take the next unit of data and store it on the next disk in the array, and so on. These units of data are called striping units. A striping unit may be one full disk block, or it may even be one single bit. The striping units are spread across the array of disks in a round-robin fashion. Let us say that the data you want to stripe across 6 disks in an array consists of 24 striping units. These 24 units of data are striped across the disks beginning from the first disk, going to the last disk, and then wrapping around. Figure 12-10 illustrates the storage of the 24 striping units on the 6 disks.

Now assume that you want to retrieve the first five striping units in a certain query to the database. These units will be read in parallel from all the first five disks in just the time taken to read one unit of data. Striping units of data across the disk array results in tremendous reduction in data access times.

Reliability Improvement The greater number of disks in an array, the better the access performance is likely to be. This is because you can retrieve from more disks in parallel. However, having more disks also increases the risk of failure. In RAID technology, redundant parity data written on check disks enable recovery from disk failures. Therefore, in the RAID system, over and above the data disks, you need additional check disks to store parity data. Let us see how the parity data stored in the check disks are used for recovery from disk failures.

How Parity Works Assume that a disk array consists of eight disks. Now, consider the first data bit on each of these eight disks. Review the 1s and 0s written as the first bit on each disk. Let us say that five of these first bits are 1s as follows:

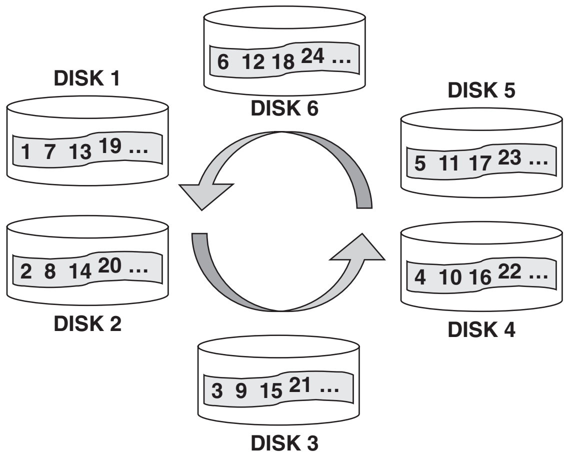


Figure 12-10 Striping units of data across a disk array.

1 0 1 1 0 1 0 1

The first parity bit written on the check disk depends on whether the total number of 1s written as the first bit on the data disks is odd or even. In our case, this number is 5. Because the number 5 is odd, the first parity bit is written as 1. If the number were even, the first parity bit would be written as 0.

When a disk failure occurs, you must be able to recover from the failure. You must be able to recreate the contents of the failed disk. Let us assume that the fourth disk failed. After the failure, the recovery mechanism counts the number of 1s in the first bit positions of all the disks other than the failed disk. This number of 1s is 4—an even number. Now examine the first parity bit on the check disk. It is 1. Therefore, because of the way parity bits are written and because the number of 1s in the first bit position of the remaining disk is even, the first bit of the failed disk must have been 1. In this manner, by looking at the data in the remaining disks and reviewing the parity data, the data in the failed disk may be completely recovered. RAID systems may be implemented at different levels. The number of check disks needed to write parity data depends on the RAID level.

RAID Levels Let us discuss the implementation at different RAID levels for data on four data disks. The disk array will then consist of these four data disks and a certain number of check disks for parity data. As far as storage of data is concerned, the check disks do not contribute to space utilization for data storage; the check disks contain redundant data.

Level 0—Nonredundant

- Just data striping
- Best write performance
- Reliability still a problem
- RAID system consists of 4 data disks and no check disks
- Effective space utilization 100%

Level 1—Mirrored

- Keeps 2 identical copies of data
- Write block on one disk followed by write block on second disk
- Does not stripe data across disks
- Most expensive
- No improvement in reading data
- RAID system consists of 4 data disks and 4 check disks
- Effective space utilization 50%

Level 0 + 1—Striped and Mirrored

- Combines features of Levels 0 and 1
- RAID system consists of 4 data disks and 4 check disks
- Effective space utilization 50%

Level 2—Error-Correcting Codes

- Single-bit striping
- Even small unit of read requires reading all 4 disks
- Good for large workloads, not for small
- RAID system consists of 4 data disks and 3 check disks
- Effective space utilization 57%

Level 3—Bit-Interleaved Parity

- Only 1 check disk needed
- Need to identify failed disk through disk controller
- Performance similar to Level 2
- Most expensive
- No improvement in reading data
- RAID system consists of 4 data disks and 1 check disk
- Effective space utilization 80%

Level 4—Block-Interleaved Parity

- Block as striping unit
- Check disk written for each data write
- RAID system consists of 4 data disks and 1 check disk
- Effective space utilization 80%

Level 5—Block-Interleaved Distributed Parity

- Improves on Level 4
- Parity blocks distributed uniformly over all disks

- Best overall performance
- RAID system consists of 5 disks of data and parity
- Effective space utilization 80%

INDEXING TECHNIQUES

Take the example of an employee file consisting of data records where each data record refers to one employee. If this is a file for a large organization, the employee file may contain thousands of records stored on a huge number of storage blocks. Depending on the file organization, to search for the record of a particular employee you may have to search through several blocks. You have to search through a large file with perhaps huge records. How can you make your search more efficient?

Imagine another file that contains index records with just two fields—one having the key values of the employee records and the other storing addresses of the data records. Each index record points to the corresponding data record. So when you want to search for an employee record with a particular key value, you do not go directly to the employee file to search for the record. First, you go to the other smaller file, search for the record with the desired key value, find the address of the record in the employee file, and then fetch the record from the employee file. This is the principle of indexing. The smaller file containing the key values and data addresses is the index file.

Figure 12-11 illustrates the principle of indexing. Note the data file and the index file.

The figure shows two types of indexes.

Dense Index. There is an index record for each data record.

Sparse Index. There is an index record for each block, not for each data record.

In Figure 12-11, the index file contains values of the primary key in the employee file. An index file may be created based on any attribute in the data file. Indexes created on the primary key are known as primary indexes. Indexes created on any other field are called secondary indexes. Consider a few of the fields in the employee record: EmployeeNo, JobCode, DeptNo, SkillType.

To speed up data access based on EmployeeNo, a primary index created on the primary key field of EmployeeNo may be used. Now suppose you have data access requests based on SkillType. If you want to retrieve all employees whose SkillType is "Programming," you have to retrieve each employee record and gather all the records with this SkillType. Instead of this method, if you have an index file created based on SkillType, then you can speed up your data access.

Figure 12-12 indicates the usage of primary and secondary indexes for data access to the employee file. Note how, in the case of the primary index, one index value points to one target data record. In secondary index files, one index value may point to several target data records. For example, there could be several employees with the SkillType "Programming."

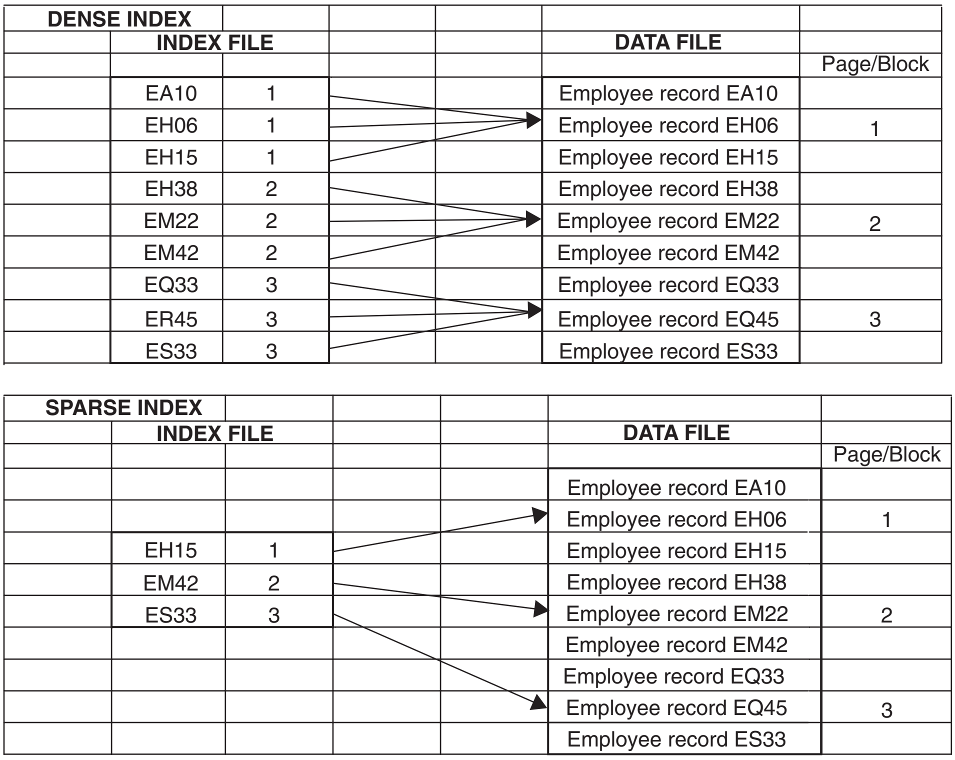


Figure 12-11 Data and index files.

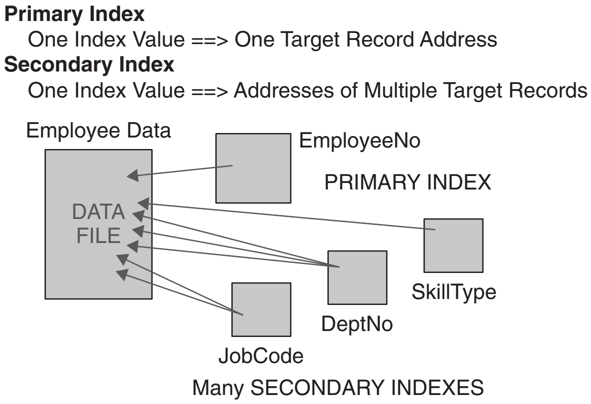


Figure 12-12 Primary and secondary indexes for employee file.

Primary Indexes

For primary indexes, one index record points to one data record. That means there will be as many index records as there are data records. How can you best arrange the index records in the index file to optimize data access? First, consider searching for a specific data record under three different options—directly from the data

Search for data with key value 60

Data File

Address	**Key**	**Data**
01	48	....
06	34	....
11	21	....
12	8	....
15	3	....
23	81	....
44	60	....

Random Index

Key	**Address**
48	01
34	06
21	11
8	12
3	15
81	23
60	44

Sorted Index

Key	**Address**
3	15
8	12
21	11
34	6
48	1
60	44
81	23

Figure 12-13 Primary index search.

file, with an index file with index records arranged in a random manner, or with an index file with index records arranged in the order of the primary key values. Figure 12-13 shows these three options with key values and addresses.

Suppose your intention is to search for a data record with key value 60. It takes seven I/O operations to search the data file directly. When you arrange the index records randomly in the index file, it takes the same number of I/O operations. However, when you arrange the index records sequentially in the index file in the order of the key values, the I/O performance is slightly better, at six I/O operations.

How can we improve the performance even more? Suppose you start your search on the index file at the middle of the index file, that is, at the record with index value 34. Compare the key value of 60 you are searching for with the value 34 at the middle of the index file. Because the value 60 is greater than 34, then key value 60 must be in the second half of the index file. Then go to the middle of the second half and get the index record. In our example, this index record happens to be the one with value 60 and, therefore, our search is over with just two I/O operations.

Step back and observe how we are able to improve the performance in this type of search. We divide the index entries in a binary fashion and compare the search value with the index value at the middle of the file. Immediately we are able to determine in which half of the file the key value lies. We then proceed to that half and compare the key value with the index at the middle of the selected half. This process of binary divisions continues and speeds up the search process. This technique, known as binary search, cuts the number of searches to find the desired index record.

Binary Search

The binary search method utilizes a binary search tree. The index entries are logically arranged in the form of an inverted tree, where the root is at the top. The root

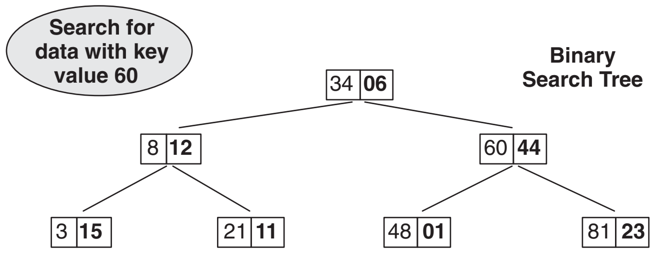


Figure 12-14 Binary search tree.

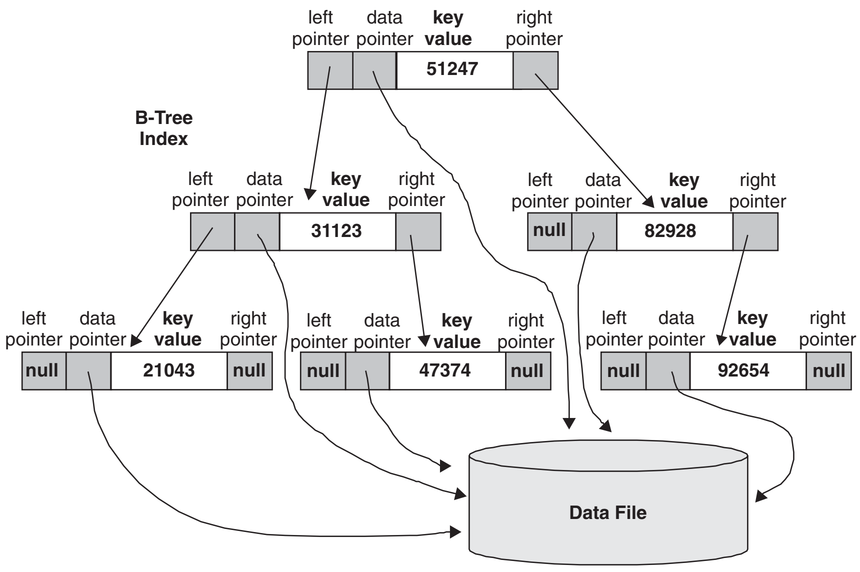


Figure 12-15 B-tree index.

represents the index entry at the middle of the index file. Figure 12-14 presents a binary search tree for the example considered here.

Primary indexes adopt binary search techniques by arranging the index records as nodes in a binary search tree. The lines pointing outward from each node represent pointers to the other nodes on either side. The search for an index record always begins at the root node and then proceeds to the right or the left by following the appropriate pointers.

B-Tree Index

Figure 12-15 shows the contents of each node in a B-tree index and illustrates how the pointers trace the search for a particular record.

Each node contains two pointers for pointing to the index records on the left and right sides. A node also contains a pointer to where the data for the particular key value resides on data storage. These pointers are actually storage addresses. For

Option1:

JobCode	Address
101	**19**
101	**67**
101	**89**
111	**33**
111	**58**
211	**26**
211	**45**
211	**98**
321	**37**
321	**76**
445	**53**
445	**54**

Option 2:

JobCode	Address1	Address2	Address3
101	**19**	**67**	**89**
111	**58**	**33**	
211	**26**	**98**	**45**
321	**76**	**37**	
445	**53**	**54**	

Option3:

	Address
JobCode	of first
101	**19**
111	**58**
211	**26**
321	**76**
445	**53**

Figure 12-16 Secondary indexes.

example, trace the search for the data record with key value 92654. Start the search at the root node containing index value 51247. The search key value 92654 is greater than 51247. Therefore, follow the right pointer from the root and go to the index record with key value 82928. The search value 92654 is greater than 82938, so follow the right pointer from this index record and go to the index record with key value 92654. This is the index record you are looking for. Pick up the data pointer from this index record and go to the data record.

B-tree indexes come with a few important variations. Nevertheless, the indexing principle is the same. Most DBMSs automatically create B-tree indexes for primary keys. The DBA need not explicitly create primary indexes.

Secondary Indexes

Secondary indexes are built on any fields other than the primary key. A secondary index may contain more than one field. Figure 12-16 indicates three options for creating index file records. The figure presents indexes built on the field JobCode in the employee file.

Option 1: One index entry in secondary index file for each value-address pair. The index file contains as many records as the data file. Retrieval of index value 101 results in retrieval of 3 index records and then retrieval of corresponding 3 data records.

Option 2: One index entry in secondary index file for each value with a list of address entries. The index file contains a lower number of records than the data file. Retrieval of index value 101 results in retrieval of just 1 index record and then retrieval of corresponding 3 data records. If there are too many distinct data records for the same index value, this option will not work.

Option 3: One index entry in secondary index file for the first value-address pair, with a pointer in the data record to the next target data record, and so on.

The index file contains a lower number of records than the data file—only 1 index record for each index value. Retrieval of index value 101 results in retrieval of just 1 index record, retrieval of the first corresponding data record, and then retrieval of the other 2 data records by following the pointers in the data records.

Bitmapped Indexing

Bitmapped indexes are ideally suitable for low-selectivity fields. A field with low selectivity is one that consists of a small number of distinct values. If there are only six different job codes for employee data, then JobCode is a low-selectivity field. On the other hand, Zip is not a low-selectivity field; it can have many distinct values.

A bitmap is an ordered series of bits, one for each distinct value of the indexed column. Let us review a sales file containing data about sales of appliances. Assume that the field for Color has three distinct values, namely, white, almond, and black. Construct a bitmap with these three distinct values. Each entry in the bitmap contains three bits. Let us say, the first bit refers to the value white, the second to almond, and the third bit to black. If a product is white in color, the bitmap entry for that product consists of three bits where the first bit is set to 1, the second bit is set to 0, and the third bit is set to 0. If a product is almond in color, the bitmap entry for that product consists of three bits where the first bit is set to 0, the second bit is set to 1, and the third bit is set to 0. You get the picture. Now, please study the bitmapped index example shown in Figure 12-17.

The figure presents an extract of the sales file and bitmapped indexes for the three different fields. Note how each entry in an index contains the ordered bits to represent the distinct values in the field. An entry is created for each record in the sale file. Each entry carries the address of the record.

How do the bitmapped indexes work to retrieve the requested rows? Consider the following data retrieval from the sales file:

```
Select the records from sales file
    Where Product is "Washer" and
    Color is "Almond" and
    Division is "East" or "South"
```

Figure 12-18 illustrates how Boolean logic is applied to find the result set based on the bitmapped indexes shown in Figure 12-17.

As you observe, bitmapped indexes support data retrievals using low-selectivity fields. The strength of this technique rests on its effectiveness when you retrieve data based on values in low-selectivity fields. Bitmapped indexes take significantly less space than B-tree indexes for low-selectivity fields.

On the other hand, if new values are introduced for the indexed fields, bitmapped indexes have to be reconstructed. Another disadvantage relates to the necessity to access the data files all the time after the bitmapped indexes are accessed. B-tree indexes do not require data file access if the requested information is already contained in the index file.

Extract of Sales Data

Address or Rowid	Date	Product	Color	Region	Sale ($)
00001BFE.0012.0111	15-Nov-00	Dishwasher	White	East	300
00001BFE.0013.0114	15-Nov-00	Dryer	Almond	West	450
00001BFF.0012.0115	16-Nov-00	Dishwasher	Almond	West	350
00001BFF.0012.0138	16-Nov-00	Washer	Black	North	550
00001BFF.0012.0145	17-Nov-00	Washer	White	South	500
00001BFF.0012.0157	17-Nov-00	Dryer	White	East	400
00001BFF.0014.0165	17-Nov-00	Washer	Almond	South	575

Bitmapped Index for Product Column

Ordered bits: Washer, Dryer, Dishwasher

Address or Rowid	Bitmap
00001BFE.0012.0111	001
00001BFE.0013.0114	010
00001BFF.0012.0115	001
00001BFF.0012.0138	100
00001BFF.0012.0145	100
00001BFF.0012.0157	010
00001BFF.0014.0165	100

Bitmapped Index for Color Column

Ordered bits: White, Almond, Black

Address or Rowid	Bitmap
00001BFE.0012.0111	100
00001BFE.0013.0114	010
00001BFF.0012.0115	010
00001BFF.0012.0138	001
00001BFF.0012.0145	100
00001BFF.0012.0157	100
00001BFF.0014.0165	010

Bitmapped Index for Region Column

Ordered bits: East, West, North, South

Address or Rowid	Bitmap
00001BFE.0012.0111	1000
00001BFE.0013.0114	0100
00001BFF.0012.0115	0100
00001BFF.0012.0138	0010
00001BFF.0012.0145	0001
00001BFF.0012.0157	1000
00001BFF.0014.0165	0001

Figure 12-17 Bitmapped indexes.

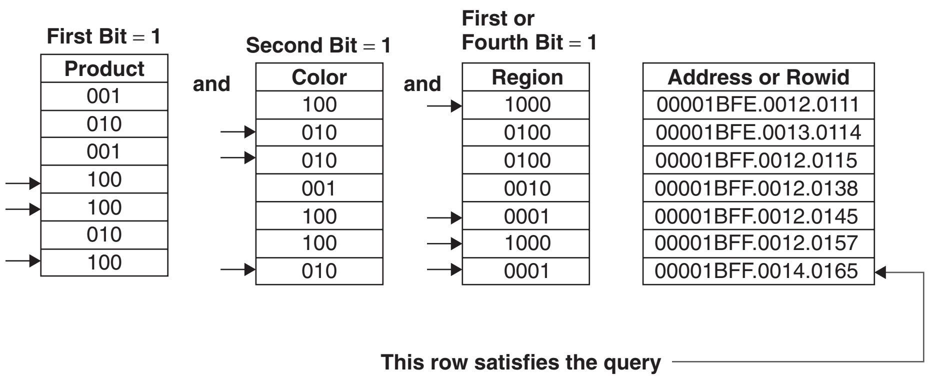


Figure 12-18 Data retrieval with bitmapped indexes.

OTHER PERFORMANCE CONSIDERATIONS

Indexing is by far the widely used and most effective method for performance improvement in a database system. In addition to the primary indexes automatically created by the DBMS, the database administrator can create secondary indexes based on other fields in the various files. Both primary indexes and secondary indexes serve well in improving performance.

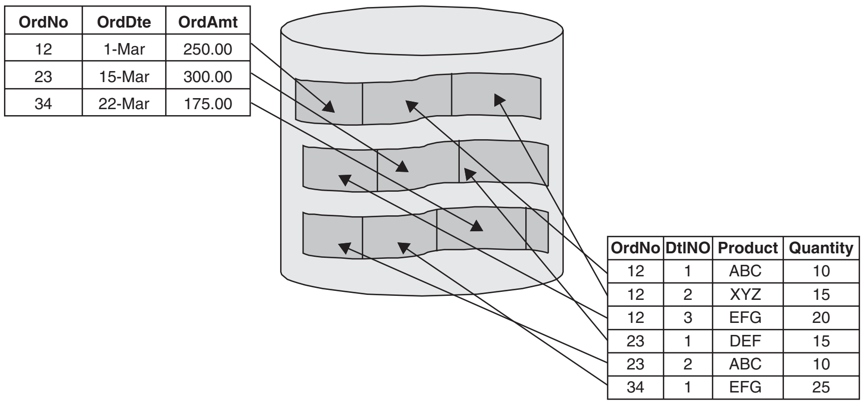
In addition to indexing, a few other options are available to enhance data access performance and to improve storage management. We will now explore some of these other improvement techniques. Some of these techniques may readily apply to your particular database system. A few others may have universal applicability. Study the techniques as we discuss them and try to determine their applicability to your environment.

Clustering

Consider two relational tables common in most database environments—the ORDER table and the ORDER DETAIL table. Typically, the details of each order are retrieved along with order data. Most of the typical data access to order information requires retrieval of an order row from the ORDER table together with rows from the ORDER DETAIL table for that order. Let us see if we can improve data access performance in such situations when rows from two tables are usually retrieved together in one transaction. How can we do this?

Instead of placing the rows from the ORDER table and the rows from the ORDER DETAIL table in two separate files, what if we place the rows from both tables as records in one file? Furthermore, what if we interleave the order detail records with the corresponding order records in this single file? Then, as the records in the file are allocated to storage, it is most likely that an order and its details will reside on one block. What is the big advantage of this arrangement? In one I/O operation, order and corresponding order detail records can be retrieved. This principle of interleaving related records from two tables in one file is called clustering. Figure 12-19 illustrates clustering of ORDER and ORDER DETAIL records.

If your DBMS supports data clustering, you can utilize this significant feature to improve performance. To create a cluster, you need to identify the tables whose rows must be clustered and the key columns that link those tables. The DBMS will then store and retrieve related records from the same cluster.



OrdNo	OrdDte	OrdAmt
12	1-Mar	250.00
23	15-Mar	300.00
34	22-Mar	175.00

OrdNo	DtlNO	Product	Quantity
12	1	ABC	10
12	2	XYZ	15
12	3	EFG	20
23	1	DEF	15
23	2	ABC	10
34	1	EFG	25

Figure 12-19 Data clustering.

Denormalization

You know clearly the significance of normalizing the tables in a relational data model. Normalization eliminates data anomalies, eliminates data redundancies, supports simpler logic, makes the design application-independent, and encourages data sharing. But, in practice, we come across cases in which strict normalization sometimes has an adverse effect on data access performance.

Consider the following cases and note how denormalization can improve data access performance.

PRODUCT-LINE (<u>LineNo</u>, ProductLineDesc, ………….)
PRODUCT (<u>ProductId</u>, ProductDesc, UnitCost, ………, LineNo)
Foreign Key: LineNo REFERENCES PRODUCT-LINE

Very common data access: product data along with ProductLineDesc

Normal method of retrieval: data from both tables by joining them

Improved method: Denormalize PRODUCT table by adding Product LineDesc so that all the required data may be obtained from only the PRODUCT table

STUDENT (<u>StudentNo</u>, StudentName, ………………)
CLASS (<u>ClassId</u>, ClassDesc, …………………………)
ENROLLMENT (<u>StudentNo</u>, <u>ClassId</u>, ………………..)
Foreign Key: StudentNo REFERENCES STUDENT
ClassId REFERENCES CLASS

Very common data access: enrollment with student name and class description

Normal method of retrieval: data from both the two primary tables through joins

Improved method: Denormalize ENROLLMENT table by adding Student-Name and ClassDesc so that all the required data may be obtained from only the ENROLLMENT table

Although there are no specific rules on when to denormalize, you must exercise sufficient caution before attempting to denormalize. Here are some general guidelines.

Many-to-many relationships. Add nonkey attributes whenever joining of the three tables to get data from the two primary tables is too frequent in your environment.

One-to-one relationships. Even when one table has a lower number of rows, if matching rows in these tables occur most of the time, consider combining the two tables into one.

One-to-many relationships. Add only the most needed attribute from the table on the “one” side to the table on the “many” side.

Fragmentation

When some tables in your database system are huge, with many attributes and a large number of rows, it is not easy to manage and administer their storage requirements. Huge tables need special considerations for improving data access. You can manage a huge table if you partition it in the best possible manner.

In some cases, part of the data in a huge table may be used less frequently than the rest of the data. Then you may want to partition the more frequently used data and store it on fast storage. In other cases, some parts of a huge table may be used primarily by one user group and the rest used by other user groups. In such cases, you may partition the table by user groups.

In the relational data model, a table may be partitioned and stored in multiple files in the following ways.

Horizontal Partitioning Split the table into groups of rows. This type of partitioning is appropriate for separating the data for user groups in geographically dispersed areas. Horizontal partitioning is also applicable for separating active data from old data.

Vertical Partitioning Split the table into groups of columns. This type of partitioning is appropriate if some columns are used more frequently than others. Vertical partitioning is also applicable if certain sets of columns are used by different user groups.

Figure 12-20 indicates how a CUSTOMER table may be partitioned horizontally or vertically.

Memory Buffer Management

The buffer manager is the software component responsible for bringing blocks of data from disk to main memory as and when they are needed. Main memory is partitioned into collections of blocks called buffer pools. Buffer pool slots hold slots for blocks to be transferred between disk and main memory. The buffer pool slots must be managed efficiently to influence data access performance. Buffer pool slots must be made available by clearing the data that are no longer required for processing. The buffer manager must be able to keep track of buffer usage properly so as to be able move the necessary volumes of data between disk and main memory efficiently.

Proper buffer management improves data access performance. You may set the DBMS parameters to accomplish this objective. You may improve performance in the following ways:

- Allocating the right number of memory buffers so that only the needed data are kept in the buffers and the needed data are retained in the buffers for the right length of time.
- Determining the right buffer sizes so that the buffers are of optimum sizes—large enough to accommodate the needed data and small enough not to waste buffer space.

CUSTOMER data file

CustomerNo	Name	Address	Phone	CreditCde	Balance
2235	Rolland				
5598	Williams				
6699	Jones				
7755	Hathaway				
9655	Cervino				
0089	Cuccia				
0109	Harreld				
0412	McKeown				

Horizontal Partitioning

CustomerNo	Name	Address	Phone	CreditCde	Balance
2235	Rolland				
5598	Williams				
6699	Jones				
7755	Hathaway				
9655	Cervino				

CustomerNo	Name	Address	Phone	CreditCde	Balance
0089	Cuccia				
0109	Harreld				
0412	McKeown				

Vertical Partitioning

CustomerNo	Name	Address	Phone
2235	Rolland		
5598	Williams		
6699	Jones		
7755	Hathaway		
9655	Cervino		
0089	Cuccia		
0109	Harreld		
0412	McKeown		

CustomerNo	CreditCde	Balance
2235		
5598		
6699		
7755		
9655		
0089		
0109		
0412		

Figure 12-20 Horizontal and vertical partitioning.

- Taking advantage of special features of the DBMS such as prefetching of blocks that are likely to be needed for the next processing.

Preconstructed Joins

Consider three related tables for a product distributing business: CUSTOMER, PRODUCT, and SALE. The CUSTOMER table contains data about each customer. The PRODUCT table contains data about each product. The SALE table contains data such as date of sale, product sold, manufacturer of product, customer number, sale representative identification, and so on.

Assume that a large number of data retrieval requests require data from all three of these tables. How will these data retrieval requests be satisfied? For every such request, the three tables must be joined and then the requested data have to be retrieved. In such situations, joining these tables for every request causes big overhead and slows the data retrievals. Is there an alternative to these numerous joins?

Suppose you do the joins before hand and preconstruct another table as a directory containing join-attribute values with pointers to the CUSTOMER and PRODUCT table rows. If you do that, then all the corresponding data requests can

CUSTOMER relation

Storage Address	**CustNo**	**CustName**
10	123	Customer1
11	234	Customer2
12	345	Customer3
13	456	Customer4
14	567	Customer5

PRODUCT relation

Storage Address	**ProdNo**	**ProdDesc**
20	1	Prod1
21	2	Prod2
22	3	Prod3
23	4	Prod4
24	5	Prod5

SALE relation

Storage Address	**SaleDte**	**CustNo**	**ProdNo**	**Units**
30	2/1/2003	345	1	25
31	2/15/2003	234	3	35
32	2/15/2003	567	3	40
33	2/22/2002	123	5	45
34	2/24/2002	123	4	60

Directory with join-attribute values

CustNo Value	Addr to CUSTOMER	Addr to PRODIUCT
123	10	24
123	10	23
234	11	22
345	12	20
567	14	22

Figure 12-21 Preconstructed join directory table.

be made against this new table first and then followed by retrievals from the base tables. Figure 12-21 illustrates this method of performance improvement.

You should use this method for large primary tables where joins on the fly are expensive in terms of performance overhead. However, if you create too many preconstructed join tables, you may be propagating too much data redundancy. Every time you add new rows to the participating tables, you have to insert new rows in the directory table. This additional processing may offset any advantage gained with the preconstructed join table.

CHAPTER SUMMARY

- Physical design is the final stage in the design process.
- Physical design implements the database on physical storage.
- When you make the transition from logical design to physical design, you map the components of one to those of the other.
- Database performance and data management form the primary focus of physical design.
- Physical design components: file organization, indexes, integrity constraints, data volumes, data usage, and data distribution.
- Physical design major tasks: translate global logical model into physical model, design representation on physical storage, and design security procedures.
- Data dictionary or catalog records the components of the design phase.
- Data is stored as files in physical storage; a file consists of data blocks; a block contains records; typically, a record is a row of a relational table.

- A block is the unit of transfer of data between physical storage and the main memory of the computer system; block addressing is the general technique for finding records.
- Three major file organizations: heap, sequential, and hash.
- RAID technology improves data retrieval performance and reliability of disk storage.
- Indexing with primary and secondary indexes forms the primary method for performance improvement of database systems.
- Other performance considerations include data clustering, denormalization, data fragmentation, memory buffer management, and preconstructed joins.

REVIEW QUESTIONS

1. List the primary goals of the physical design process. Which ones are the most important?
2. Describe very briefly the physical design components.
3. What are the major physical design tasks?
4. How is the data dictionary used to record the contents of the database system? What are the types of information stored in the data dictionary?
5. Describe how rows of a relational table are stored in physical storage.
6. What is your understanding of file organization? What are the major types of file organizations?
7. How does RAID technology improve reliability of disk storage?
8. What is a B-tree index? Explain with a simple example.
9. What is data clustering? How does it improve data access performance?
10. Distinguish between horizontal and vertical partitioning. State the conditions under which each option is useful.

EXERCISES

1. Indicate whether true or false:
   A. Physical design may ignore the features of the target DBMS.
   B. Proper physical design ensures improved database performance.
   C. File organization refers to how blocks are allocated to files.
   D. A row in a relational table is the unit of data transfer from physical storage.
   E. Sequential file organization is best if the most common operation is range selection.
   F. Data striping in RAID technology improves disk reliability.
   G. You can build a secondary index only on certain fields in the file.
   H. Denormalization increases data redundancy but improves data access performance.
   I. Most DBMSs automatically create primary indexes.

   J. Absolute addresses in linked lists provide faster data retrieval than relative addresses.

2. You are the database administrator in the database project for a department store. Name any six relational tables that will be part of the logical design. Describe how you will allocate storage for these tables and how you will assign the data to physical files.
3. Explain how you can improve data access performance by adjusting block sizes and block usage parameters.
4. What is hash or direct file organization? How are the records organized in the file? Explain the problem of collisions in this method of file organization and describe how collisions are resolved.
5. What is bitmapped indexing? When is it effective? Show, with a simple example, how bitmapped indexes speed up data access.