Chapter 12

Physical Database Design

Chapter Outline

What You Will Learn in This Chapter

- How does a DBMS store data for efficient retrieval?
- How does a DBMS interact with the file system?
- What are the common database operations?
- What options does a DBMS have for storing tables?
- How is one data row stored?
- How can you improve performance by specifying where data is stored?
- How does a DBA control file storage?
- What performance issues might arise at Sally's Pet Store?

A Developer's View

Ariel: How is the new job going, Miranda?

Miranda: Great! The other developers are really fun to work with.

Ariel: So you're not bored with the job yet?

Miranda: No. I don't think that will ever happen—everything keeps changing. Now they want me to set up a Web site for the sales application. They want a site where customers can check on their order status and maybe even enter new orders.

Ariel: That sounds hard. I know a little about HTML, but I don't have any idea of how you access a database over the Web.

Miranda: Well, there are some nice tools out there now. With SQL and a little programming, it should not be too hard.

Ariel: That sounds like a great opportunity. If you learn how to build Web sites that access databases, you can write your ticket to a job anywhere.

Getting Started

A DBMS sometimes provides options on how to physically store data. Most enable you to add indexes to improve query performance. Some systems enable you to select hashed or direct storage for data that needs immediate access. You can also use data clustering and partitioning to handle large data tables more efficiently.

Introduction

How does a DBMS store data for efficient retrieval? Any database application is created through the basic steps described in Chapters 2 through 9. You get the user requirements, design the database through normalization, create the queries using SQL, build forms and reports and then add the details to create a complete application. However, with large applications, one more step is critical to the success of your application. You must analyze its performance. Performance is largely controlled by telling the DBMS how to physically store and retrieve the data.

If computers were fast enough, how the DBMS physically stored the data for each table might not matter. Today, for small applications, this situation is probably true. The default storage method provides acceptable levels of performance, and you could skip this chapter. However, as databases and applications become larger or contain specialized types of data, physical storage becomes an important issue in the performance of your application. Large business applications routinely hold millions or even trillions of rows of data in tables. Proper configuration is essential—otherwise, even simple queries could take minutes or hours to run.

Two basic questions must be answered to store data tables: (1) How should each row of data be stored and accessed? and (2) How should individual columns be stored? The first question is more difficult to answer and is determined largely by how the data is used. Hence we must first examine the possible uses of the database. The answer to the second question depends largely on the type of data being stored. For traditional business data (numbers and small text), the answers are straightforward. If your application stores more complex data objects, the second question becomes more critical.

Two-Minute Chapter

Up to this point in the book, the features of relational databases have been discussed without the need to understand how data is actually stored and retrieved by the DBMS. In fact, that is one of the key features of a DBMS—it is free to optimize the storage of data without affecting the overall application design. However, sometimes it becomes useful to understand some of the underlying data storage techniques. When databases get huge and you need to find ways to improve performance, some DBMSs provide storage options that can make a big difference in usability.

Indexes are the most common method of improving performance for data retrieval. Most systems use linked lists and a B+-tree approach to storing indexed data. These topics are routinely covered in a second programming course in computer science disciplines (data structures). Some examples are given here but programming details are left for CS courses.

As pointed out in Chapter 9, adding too many indexes to a table can degrade performance when inserting or updating data in the table. So two other primary methods of storing data are sometimes available: simple sequential and hashed-key tables. It might seem strange, but sometimes sequential storage can be the fastest approach to handling big tables—as long as the data rarely changes and is generally retrieved as a large batch. Removing all other overhead items can substantially improve the raw transfer of the data. Hashed-key tables are trickier and not always available. They are useful when the data always has its own key value and you need rapid access to an individual item. For example, a bar-code number can be used as a key, or a transponder value from an RFID toll device (FasTrak in California or E-ZPass on the East Coast). The number provided is hashed (simplified) and directly converted into a physical location in the database file. So individual items can be retrieved or updated almost instantaneously.

Another approach that is used for huge databases is to partition the data or cluster items together. Remember the common Sales form that leads to separate tables for Sales and SaleItems. The related data from these tables (linked by SaleID) is almost always retrieved together. So some systems provide methods to store the related data together—making it faster to retrieve from typical disk drives.

Most applications work well with the standard B+-tree indexes, and as disk drive performance improves (such as using solid-state drives), this approach can be fast enough for most common business data. But for some specialized situations, performance can be dramatically improved with different storage approaches. Your job is to recognize when those tools are needed.

Physical Data Storage

How does a DBMS interact with the file system? Developers see database storage in terms of tables, but these tables ultimately need to be stored in files on the operating system. The main job of the database engine is to translate the concept of tables and rows into physical storage on the computer's disk drives. In computer science classes (particularly the data structures class), you will spend a lot of time coding different ways to store this data. This chapter simply introduces the basic concepts. Figure 12.1 shows how the operating system is responsible

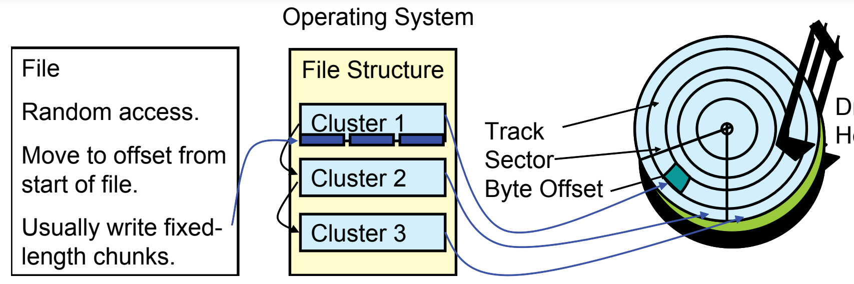


Figure 12.1

Physical data storage. The operating system breaks files into clusters and writes the clusters onto the physical disk drive. It uses internal pointers to retrieve the data sequentially, or randomly based on an offset from the start of the file. The DBMS uses file read/write commands to store chunks of data.

for translating files into physical storage on the disk drive. The file system (such as NTFS for Windows), breaks a file into clusters and uses internal pointers to record the physical location of each cluster. In most cases, the DBMS ignores the direct disk drive issues and lets the operating system handle the details. Instead, the DBMS uses the operating system's file read and write commands. For the main data storage, the DBMS creates a file and reserves a specified amount of disk space. The DBMS extends the allocated file space as the data grows. The DBMS can write a chunk of data anywhere within the allocated space by using the standard write command. The DBMS keeps track of which portions of the space are used by recording the **offset** (count of the number of bytes) from the start of the file. If you are familiar with programming, you should recognize the role of the standard fopen, fseek, fread, and fwrite commands available in stdio in C (and similar languages), or the fstream objects with open, seekg, read, and write methods in C++.

The challenge for programmers who create the DBMS is to translate the concepts of tables and rows into this file structure—so that the data can be stored efficiently and retrieved quickly. Several common storage methods have been developed over the past few years. One of them (B+tree) is commonly used for general data access and is useful in most situations. However, for huge databases, you might need more control over how the table rows are stored. Some DBMSs give you more choices and even if you do not intend to become a DBMS programmer, you need to understand their strengths and weaknesses.

Table Operations

What are the common database operations? To understand the differences between storage methods, you must first understand how the DBMS will use the data. Then by evaluating how each storage method affects the various table operations, you can choose the best method for your particular application. As shown in Figure 12.2, three major categories of operations affect tables: (1) retrieving data, (2) storing data, and (3) reorganizing the database. Each category contains more detailed tasks that are described in the following sections. Every application will perform all of the operations within the categories. As a developer you need to examine the application and identify the operations that are affecting performance.

- Retrieve data
  - Read entire table.
  - Read next row.
  - Read arbitrary row.
- Store data
  - Insert a row.
  - Delete a row.
  - Modify a row.
- Reorganize database
  - Remove deleted rows.
  - Recover unused space.

Figure 12.2

Table operations. Every application must perform these operations. The key is to determine which operation is causing delays.

Retrieve Data

Retrieving data constitutes some of the most common activities in a database application. These operations also present the best opportunity to improve performance. Applications commonly perform three types of data retrieval. They read the entire table, read the next row in a sequence, and find and retrieve an arbitrary row.

Reading the entire table, or large portions of it, might not seem like a common operation, but it does occur relatively often when printing reports. For the example in Figure 12.3, to print weekly paychecks, the application will have to read every row in the employee table. But what if hourly workers are paid weekly, but managers are paid monthly? In most companies the managers represent only

Figure 12.3

Read a table sequentially. Sequential retrieval requires the data to be sorted; for example, this customer data is sorted alphabetically by LastName and FirstName. Fortunately, sort methods are so fast that they do not generally affect the application performance.

LastName	FirstName	Phone
Adams	Kimberly	(406) 987-9338
Adkins	Inga	(706) 977-4337
Albright	Searoba	(619) 281-2485
Anderson	Charlotte	(701) 384-5623
Baez	Bessie	(606) 661-2765
Baez	Lou Ann	(502) 029-3909
Bailey	Gayle	(360) 649-9754
Bell	Luther	(717) 244-3484
Carter	Phillip	(219) 263-2040
Cartwright	Glen	(502) 595-1052
Carver	Bernice	(804) 020-5842
Craig	Melinda	(502) 691-7565

a small percentage of the total workers, and retrieving 90 percent of a table is no different in performance than retrieving 100 percent.

Reading the next row in a sequence is related to retrieving all the data in a table. When an application needs to read an entire table, it is generally retrieved in some order or sequence. For example, paychecks might be printed in alphabetical order by employee name, department name, or postal code.

The more challenging retrieval operation is the ability to retrieve any arbitrary row. It is sometimes called random access because the database does not know which record might be requested. For example, any customer could place an order at random, and the database would have to retrieve the matching data for that customer.

This lookup process is one of the most critical elements to affect the performance of your application. It is easy to spot in situations like the customer example. The clerk enters a customer name or number, and the database has to retrieve the matching data. Clearly, you want to keep the lookup time as short as possible to avoid delays for the customer and the clerks.

Yet there is a more critical problem involving lookups. Any time you build a query, two types of random lookups come into play. First, joining two tables requires the database to match the values in one table with those in a second table. Second, any time you impose a condition with the WHERE statement, you are asking the DBMS to find rows that match that condition. So query performance is directly related to how fast the database can perform lookups and match the data requested. These lookups are critical because they are so numerous. Joining two tables could require thousands or millions of lookups—depending on the number of rows in the two tables. Remember that many tasks throughout the application use queries. Sequential lookups that retrieve large portions of the table require minimal optimization, because you have to read the entire table. The random retrievals and random lookups require more thought about optimization. However, storing data sequentially causes other problems when you need to insert, delete, or modify rows.

Store Data

A DBMS has to perform three basic operations involved with storing data: inserting a new row, deleting a row, or modifying the data in a row. Most systems implement a fast delete operation—they do not actually remove the deleted data. As shown in Figure 12.4, it is much faster to just mark the row as deleted. Then when the database wants to retrieve an item, the DBMS first checks to see whether the item has been deleted. If so, the DBMS ignores that row. Similarly, a good DBMS attempts to store data in fixed block lengths, so that if a row is modified, the DBMS can simply overwrite the data. With highly variable-length data, this operation is not always possible, so the DBMS must perform a delete and an insert operation.

In terms of performance, the biggest issue with delete operations involves storage space instead of speed. Although a row has been deleted, it still takes up physical space. Sometimes the DBMS can overwrite the old data, but after a while, there can be millions of bytes of unused fragments.

Inserting a new row of data is one of the more challenging aspects in a database management system. Next to random lookups, it is the source of the most performance problems. In fact, there is generally a trade-off between the two issues. If a system is good at random lookups, it is not as efficient at storing new data rows.

LastName	FirstName	Phone
Adams	Kimberly	(406) 987-9338
Adkins	Inga	(706) 977-4337
Albright	Searoba	(619) 281-2485
Anderson	Charlotte	(701) 384-5623
Baez	Bessie	(606) 661-2765
xBaez	Lou Ann	(502) 029-3909
Bailey	Gayle	(360) 649-9754
Bell	Luther	(717) 244-3484
Carter	Phillip	(219) 263-2040
Cartwright	Glen	(502) 595-1052
Carver	Bernice	(804) 020-5842
Craig	Melinda	(502) 691-7565

Figure 12.4

Delete a row. Deletion is fast because the DBMS just marks the row as deleted. It does not actually remove the data.

That is, the techniques used to improve random lookups often require significantly more time to add data rows.

The performance issues of adding new data are somewhat technical and will be explained in more detail in the section on data storage methods. For now, examine your application to identify which tables will add new data on a regular basis and which tables might add data only occasionally. For example, a firm might add only a few new items a year to the Products table. However, thousands of new rows could be added to the Order table every day.

Reorganize the Database

Largely because of the deletion method, a database can become disorganized over time. Data that is flagged as deleted is still hiding in the table space. Empty holes of storage space are too small to hold new data and data rows that are used together are no longer stored near each other.

These problems are particularly challenging with relational databases. In a relational database the system data is also stored in tables. For example, the form layout that you redesigned 20 times is stored as rows in a table. Each time you redesigned it, the database flagged the old version as deleted and saved the new version. Complex forms could take up several thousand bytes of storage.

Most systems have an administrative command to reorganize or **pack** the database. This command causes the DBMS to go through the data and rewrite each table—clearing up the storage space. A major challenge to database administration is to determine how often to run this command. Two complications exist. First, it can take several hours for this command to process large databases. Second, a few systems require that all users be logged off the DBMS before the administrator can run this command. You want to avoid database systems with the second requirement. It prevents you from providing 24-hour access to the database. However, even if other people can still use the system, database reorganization can affect the overall performance of the application, so the process generally needs to be performed during slow periods (e.g., at night).

On the flip side, if you forget to periodically reorganize the database, it can rapidly fill with wasted space. It is not uncommon for even a small Access database to grow from under 1 megabyte to 5 or 6 megabytes of storage space during development. Be sure to use the database utilities to compact the database. Doing so will make it much easier and faster to back up and copy the data files.

Identifying Problems

During the database design stage, you should be able to identify potential problems. You need to analyze the database usage and volume statistics collected in Chapter 3. In particular, look for large tables; heavily used tables; transaction tables requiring fast database responses; and queries with multiple joins, complex criteria, or detailed subqueries. You should also perform tests during the development of the applications. Generate large sample tables and test the performance of the queries, forms, and reports. Once the database application is operational, you can use the performance monitoring tools described in Chapter 11 to locate bottlenecks.

Once you identify the form, report, or query that is causing delays; you need to determine the cause of the problem: data retrieval, data storage, or data reorganization. You can use the programming debug feature to step through code that utilizes many different operations. By timing procedures and loops, you can determine which section is causing the longest delays. You can also use the Timer function to record the times of various operations.

Once you have identified the location of the delays, you can test various strategies for improving performance. If the delays involve your program, explore different ways to reorganize your code to improve performance. If delays are due to data retrieval or storage, think about ways to perform data operations in larger blocks. For example, your program might run faster if it writes individual changes to a temporary table and then uses SQL statements to transfer the changes to the primary tables in one large operation.

A second method to improve performance is to alter the way the data is stored. Each DBMS provides different controls over data storage. The following sections summarize the most common techniques.

Data Storage Methods

What options does a DBMS have for storing tables? Three primary methods are used to store data tables—each with several variations. The simplest method is sequential storage—putting the data into tables in the order in which it is most commonly accessed. To provide faster access, particularly for random lookups, a second approach is to create indexes of the data. A third approach known as direct or hashed-key storage is radically different and is designed to optimize random lookup at all costs.

Sequential storage is relatively easy to understand, but probably the least useful. Hashed storage methods are also straightforward, but have their own limitations. Indexed tables are by far the most common means of storing and accessing data today. They are complex and have many variations. To choose the best storage method, you sometimes have to understand the differences between the variations.

Pointers and linked lists are key topics in understanding how indexes work. You might have heard computer science students discussing these topics. Do not panic. You do not need to know how to program routines using pointers and linked lists.

ID	LastName	FirstName	DateHired
1	Reeves	Keith	1/29/....
2	Gibson	Bill	3/31/....
3	Reasoner	Katy	2/17/....
4	Hopkins	Alan	2/8/....
5	James	Leisha	1/6/....
6	Eaton	Anissa	8/23/....
7	Farris	Dustin	3/28/....
8	Carpenter	Carlos	12/29/....
9	O'Connor	Jessica	7/23/....
10	Shields	Howard	7/13/....

Figure 12.5

Sequential file. Each row is stored in some predefined order. Sequential storage is used primarily for backup or for transferring data to a different database.

To understand their strengths and weaknesses, you just need to be able to draw some basic diagrams.

Sequential Storage

Sequential files are the simplest method of storing data. Each row is stored in a predefined order as shown in Figure 12.5. As long as the data is retrieved in the order specified, access is fast and storage space is used efficiently. The real problems arise when data is added or when users need to retrieve data in several different sequences.

Uses

Sequential storage is useful when data is always retrieved in a fixed order. It is also useful when the file contains a lot of common data. For example, if most customers have the same ZIP code, you might as well leave the ZIP code data in simple sequential storage.

Another use of sequential files is for backup or transporting data to a different system. Each database system stores data in a proprietary internal format. To transfer data from one system to another generally requires exporting the data to a common format, moving the data, and importing it into the new database. A sequential ASCII file is a popular export/import format that most database systems support.

Drawbacks

To understand the drawbacks to sequential storage, consider the steps involved in performing the basic database operations listed in Figure 12.2. Reading the entire table and retrieving the next sequential row are easy. Finding an arbitrary row is much slower. If the rows can hold different lengths of data, the only way to find an item is to search from the start of the table until the desired row is found. With N rows of data, the expected number of retrievals required to find a random row is (N + 1)/2, or a table with 1,000,000 rows would require 500,000 lookups on average to find a matching row. Obviously a bad idea.

Another major drawback can be seen by examining the data storage operations. As with every method, flagged deletion is fast and relatively efficient. The real

ID	LastName	FirstName	DateHired
8	Carpenter	Carlos	12/29/....
6	Eaton	Anissa	8/23/....
7	Farris	Dustin	3/28/....
2	Gibson	Bill	3/31/....
11	**Inez**	**Maria**	**1/15/....**
4	Hopkins	Alan	2/8/....
5	James	Leisha	1/6/....
9	O'Connor	Jessica	7/23/....
3	Reasoner	Katy	2/17/....
1	Reeves	Keith	1/29/....
10	Shields	Howard	7/13/....

Figure 12.6

Insert into a sequential table. Copy the top of the table to a new table. Store the new data row (Inez). Copy the rest of the data. The system must read every row in the table.

problems arise when you want to insert a new row. Examine Figure 12.5 and decide how you would insert data for a new employee with the last name of Inez. The basic steps are shown in Figure 12.6. If you had to write a program to insert a row, the most efficient method is to follow four steps: (1) Read each row. (2) Decide if this row comes before the new row. If so, store it in a new table. (3) When you reach the insertion point, save the new row of data. (4) Append the rest of the data to the end of the new table. The main drawback to this approach is that any time you want to add a row of data, the database has to retrieve (and probably rewrite) every row in the table.

Pointers and Indexes

The most common solution to the problems of sequential tables is to store each row separately and use pointers to find a row. This approach also uses indexes to establish the sequential retrieval of data and to improve searches. Separating rows of data means that each row is stored as an independent group. (Actually, you can break rows into smaller chunks, but for now, think of each row stored independently.) When a row of data is stored, it is stored at some location. This location is called an **address**, and a variable that holds this address is called a **pointer**. With most file systems, the address (and pointer value) is a number that represents the offset in bytes from the start of the file.

Figure 12.7 illustrates how the data is separated. It also shows how an index is used to retrieve the data. The data is linked to the index via the address pointers. To retrieve the data sequentially, the DBMS simply loops through the index and follows the pointers to retrieve the data. The data rows can be stored in any order in the file structure.

An **index** is the most common method used to provide faster access to data. An index sorts and stores the key values from the original table along with a pointer to the rest of the data in each row. Figure 12.8 illustrates the concept. Notice that a table can have many indexes. Indexes can also be based on several columns of data. The ability to create multiple indexes in a table indicates their first strength.

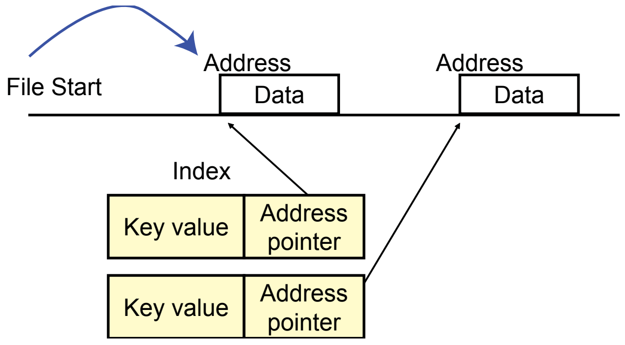
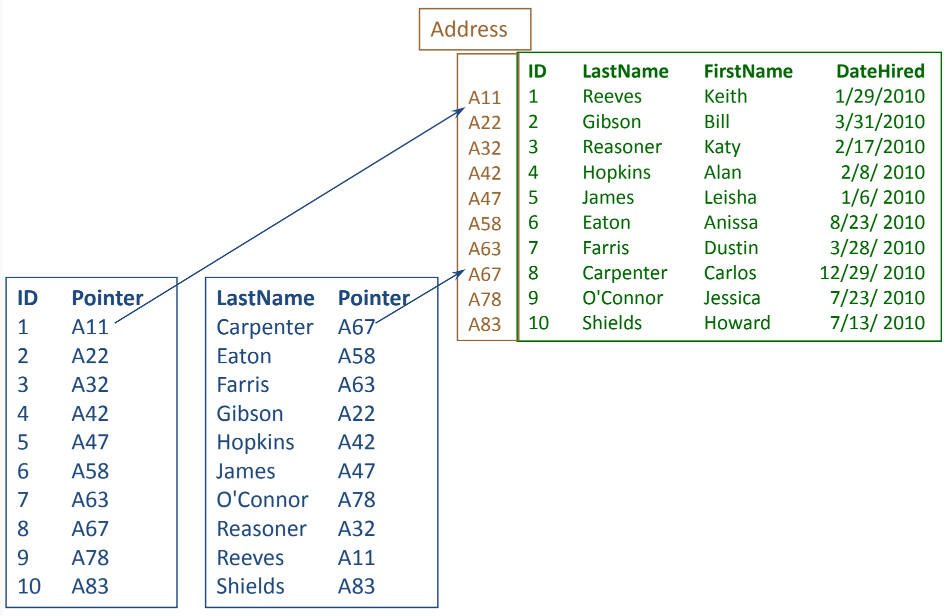


Figure 12.7

Use of pointers. The database searches the key values. When it finds the appropriate key, it follows the pointer to retrieve the associated data stored on the disk.

Figure 12.8

Indexes. An index sorts and stores a key value along with a pointer to the rest of the data. Indexes can be built for any column or combination of columns in the table. The two separate indexes provide different sorts and searches for one table.



Address	ID	LastName	FirstName	DateHired
A11	1	Reeves	Keith	1/29/2010
A22	2	Gibson	Bill	3/31/2010
A32	3	Reasoner	Katy	2/17/2010
A42	4	Hopkins	Alan	2/8/ 2010
A47	5	James	Leisha	1/6/ 2010
A58	6	Eaton	Anissa	8/23/ 2010
A63	7	Farris	Dustin	3/28/ 2010
A67	8	Carpenter	Carlos	12/29/ 2010
A78	9	O'Connor	Jessica	7/23/ 2010
A83	10	Shields	Howard	7/13/ 2010

ID	Pointer
1	A11
2	A22
3	A32
4	A42
5	A47
6	A58
7	A63
8	A67
9	A78
10	A83

LastName	Pointer
Carpenter	A67
Eaton	A58
Farris	A63
Gibson	A22
Hopkins	A42
James	A47
O'Connor	A78
Reasoner	A32
Reeves	A11
Shields	A83

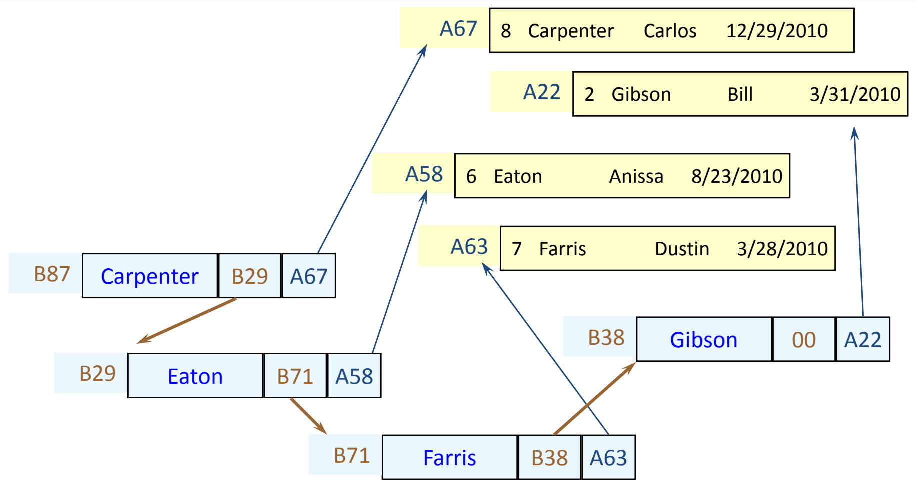


Figure 12.9

Linked list. The index is split into separate index elements. Each element contains a key value (LastName), a pointer to the next index element, and a pointer to the rest of the data for that row. To retrieve data sequentially, start at the first element (Carpenter) and follow the link (pointer) to the next element (Eaton).

They enable relatively fast, sorted access to a table based on any criteria. Indexes generally provide a clear advantage over straight sequential files because they support high-speed access to any data columns.

The astute reader will recognize that the index has not really solved all of the problems—it has simply transferred them to the index file. That is, to store and retrieve data, you face the same problems in building the index. On the plus side, the index is smaller and easier to manipulate. It is also possible to create multiple indexes for any table, so it can be searched or retrieved using different key columns. But, it would be nice to find a better way to handle the index itself.

Linked Lists

To solve the insert problem, indexes are generally based on linked lists instead of sequential lists. A linked list is a technique that splits data even further than a sequential index. With a linked list, any index element can be stored separately. A pointer is then used to link to the next index item. Figure 12.9 illustrates the basic concepts. In this example each row of data is stored separately. Then an index is created that is keyed on LastName. However, each element of the index is stored separately. An index element consists of three parts: the key value, a pointer to the associated data element, and a pointer to the next index element.

To retrieve data sequentially, start at the first element for Carpenter. Follow the pointer to the next element (B29 points to Eaton). Each element of the index is found by following the link (pointer) to the next element. The data pointer in each index element provides the link to the entire data row for that key value (A67 points to the Carpenter row).

The strength of a linked list lies in its ability to easily and rapidly insert and delete data. Remember the difficulty in inserting data with a sequential table. Even with a sequential index, inserting a new row generally results in copying half the index (or more). For large tables this approach is clearly inefficient.

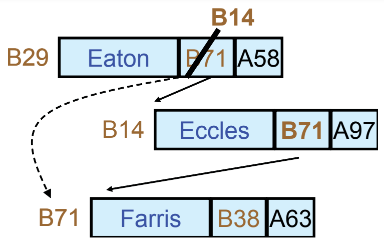


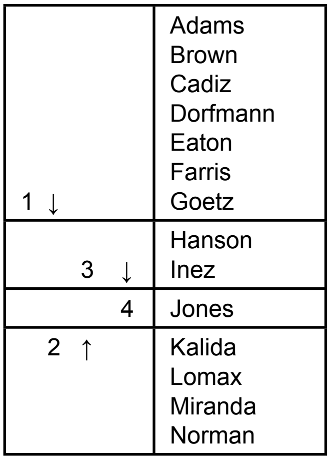
Figure 12.10

Insert into a linked list. To add the index element for Eccles: store the new data element, keep the address (B14); find the sort location—between Eaton and Farris; move the link pointer from Eaton into Eccles (B71); store the pointer for Eccles (B14) in Eaton.

On the other hand, as shown in Figure 12.10, inserting a new key row into a linked list requires three basic steps. (1) Store the data and store the index element—keeping the address of each. (2) Find the point in the index to insert the new row using a binary search. In the example, Eccles comes between Eaton and Farris. (3) Change the link pointers. The link in Eaton should point to Eccles (change B71 to B14) and the link in Eccles should point to Farris (insert the B71). Those are the only steps needed. No copying of data keys and no complicated code.

Figure 12.11

Binary search. A sorted index can be searched rapidly using a binary search. To find the entry for Jones, find the middle of the list (Goetz). Jones is past Goetz, so split the second half in half (Kalida). Keep splitting the remainder in half until you find the entry.



	Adams
	Brown
	Cadiz
	Dorfmann
	Eaton
	Farris
1 ↓	Goetz
	Hanson
3 ↓	Inez
4	Jones
2 ↑	Kalida
	Lomax
	Miranda
	Norman

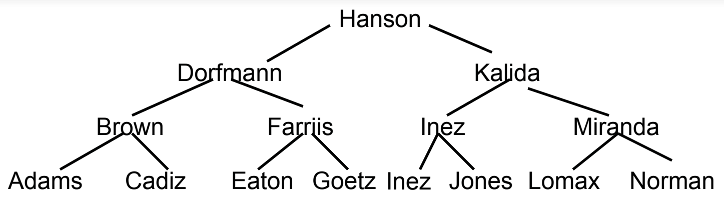


Figure 12.12

Simple tree. Each node element has a key value, a pointer to data for that key, and two link pointers. One pointer is for values less than the key. One is for values greater than or equal to the key.

Linked lists have substantial advantages for most of the standard table operations. In particular, they are the most efficient way to insert and change data because the code simply edits the link pointers to add or delete something from the list. But, how can linked lists improve searching for and retrieving random items in the list?

B+Trees

As noted in Chapter 9, sorted lists like indexes provide a relatively efficient method to search for data. A binary search can take advantage of the sorted data by cutting the search in half at each step. Figure 12.11 shows the search process. Recall that a binary search can find any specific entry with no more than log2(N) retrievals. In this example with 14 entries, log2(14) is 3.8, or a maximum of 4 lookups. The example specifically uses Jones because it requires all 4 retrievals.

It is clear that binary searches are efficient, but how does that help with linked lists on indexes. First, recognize that indexes are sorted, so it should be possible to use a similar approach. Second, think about the list for a few minutes, and you can see that it can be reorganized. Instead of trying to store it sequentially, grab the middle entry (the starting point for any search), and build a tree structure. In many ways, a tree is just a more complex way of storing a linked list. Instead of linear, it contains multiple links.

One version of a tree is shown in Figure 12.12. Only the key values are shown in this figure. In practice, each **node** or element on the tree would contain an index element much like those in Figure 12.10. That is, each element would contain the key value, a pointer to the rest of the data, and two link pointers. For the particular tree in Figure 12.12, each element has at most two links. One link (the line to the left) points to elements that have lower values. The other link (line to the right) points to elements that have a value greater than or equal to the value in the node. The **root** is the highest node on the tree. The bottom nodes are called **leaves** because they are at the end of the tree branches.

The power of the tree lies in its ability to find a data element. To find the data for Jones, start at the top of the tree (Hanson). Jones is alphabetically greater than Hanson, so go to the right side. Track down the tree depending on the key value until you reach the bottom element for Jones. Notice that every element requires at most four searches because there are only four levels in the tree. Notice that the search was exactly the same as the binary search. The number of searches is given by the **depth** of the tree, which is the number of nodes between the root and the leaves. Notice that if you compress a B+tree down to one level, each element

- Set the degree (m)
  - m >= 3
  - Usually an odd number.
- Every node (except the root) must have between m/2 and m children.
- All leaves are at the same level/depth.
- All key values are displayed on the bottom leaves.
- A nonleaf node with n children will contain n-1 key values.
- Leaves are connected by pointers (sequential access).

Figure 12.13

B+tree rules. These rules will generate a tree structure that provides good database performance under a variety of conditions.

would be in one long key row. In other words, you would end up with indexed sequential access.

The power of a B^+tree for searching is clear, but what if you want to retrieve the data sequentially? The answer is that the leaves or bottom nodes contain a link to the next item. When the DBMS reads to the leftmost leaf (Adams), it can follow points to the right to retrieve each final item in sequence.

B^+Tree Definition

On examining Figure 12.12, it quickly becomes clear that there are many ways to organize a tree. For example, why is Brown listed beneath Cadiz instead of beneath Adams? There is no good answer to this first question. Minor positional choices like this one are arbitrary and do not affect the tree. But the question shows that there is some flexibility in the final tree. Bigger questions do affect the tree significantly, such as why is the tree approximately symmetrical—that is, why not let one side reach lower than the other side? Why does each node split into two branches—why not three or more?

Answers to each of these questions will affect the layout of the tree. As the layout changes, so does the performance. Computer scientists have studied these structures in detail. For database purposes, they have determined that the best overall performance is provided by a B^+tree, which follows the six basic rules shown in Figure 12.13. The rules are not as complicated as they may first appear.

First, you have to choose the degree of the tree. The **degree** represents the maximum number of children that can fall below any node. Choosing the degree determines how fast the database can find any particular item. In Figure 12.12 the degree was 2, which produced a binary search. Higher degrees result in trees that are broader, requiring even fewer searches to find any item. Two rules that give the B^+tree its power are that each node must have at least m/2 children (and no more than m children) and that all leaves must be at the same depth. In other words, the tree cannot be lopsided, but must be balanced so that data is distributed relatively evenly across the tree.

Figure 12.14 shows a small B^+tree of degree 3. With a degree of 3, a node can point to three different children. If it does, the node must have two key values, such as (458, 792). To understand why, search the tree to find key value 692. Start at the top and note that 692 is greater than 315, so go to the right branch. Now 692 falls between 458 and 792, so branch to the middle child and then drop down to find the entry on the bottom leaf, which contains a pointer to the rest of the data. A node with three children must have two keys. Any value lower than the left-most

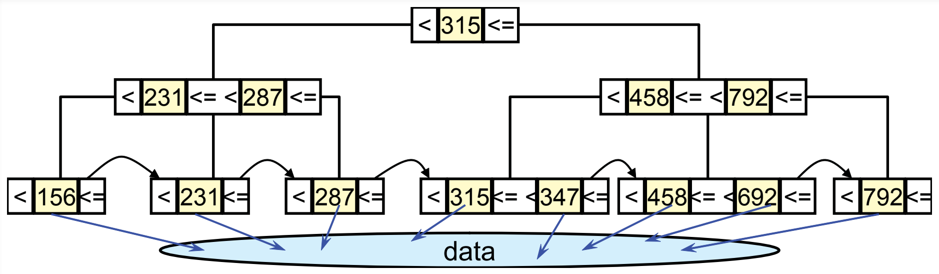


Figure 12.14

Sample B+tree of degree 3. Start at the top to find the value 692. It is larger than 315, so go to the right branch. It is between 458 and 792 so go down the middle. The bottom leaf points to the rest of the data. The bottom leaves also contain links to provide sequential access.

key goes to the left. A value greater than the right-most key goes to the right. Anything between the keys follows the middle path.

Uses

The main strength of the B$^+$tree is that it provides a guaranteed level of performance for access to the data. Every element can be found in the same number of searches—which is determined by the depth of the tree. The tree also provides fast sequential retrieval. The other power comes from the ability to add or delete elements from the tree. As in a linked list, adding new items to a tree is relatively easy. The process is a little more complicated in a tree, because the rules require the tree to be rearranged periodically as data is added. You can study the details of programming a B$^+$tree in a computer science class. The basic operations are straightforward, but somewhat tedious. You can also buy software to create and manipulate B$^+$trees. However, adding items to a tree is still relatively fast and efficient.

Overall, the B$^+$tree approach provides the best general access to data. If you do not know anything useful about the data or how it will be used, you should always choose the B$^+$tree method to store a table. It provides the best overall performance for typical data—for sequential retrieval, random lookup, and for changes to the data.

Drawbacks

The drawbacks to B$^+$tree storage are relatively minor. It has been shown to be the best general purpose storage method, and most DBMSs use it as the main storage method. One criticism has been that the coding is relatively complex, but standard algorithms have been developed for several years, so it is not really an issue. The bigger problem is that for large tables that involve constant changes, it takes time to reorganize the index for every change. The problem is worse when you create multiple B$^+$tree indexes on a table. Inserting a row could trigger changes in several indexes and result in restructuring millions of items in each index. Many systems recommend that if you are going to bulk insert thousands of rows of data, you should turn off all indexing, insert the data, then index the table one time. The only other solution is to use indexes sparingly on tables that have heavy transaction changes.

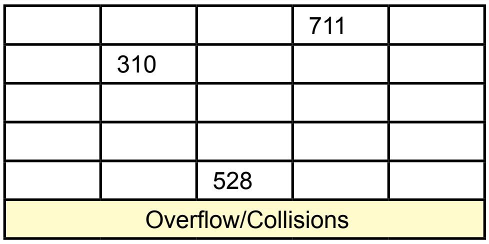


Figure 12.15

Hashed-key access. The key value (528) is converted directly into a storage location by dividing by a prime number (101). The remainder (23) is used to identify the storage position. If two keys have the same remainder, one is stored in an overflow location.

Direct or Hashed Access

Some situations require super fast random access to data. For example, in transaction situations you might need virtually instantaneous retrieval of some data items. When a grocery store clerk scans an item, the DBMS must retrieve the price immediately. A delay of even 5 seconds would be incredibly annoying and costly given the huge number of items that are scanned every day. In this example, the computer is given a unique bar-code number and needs to retrieve the matching data. It makes sense to optimize the search for this situation.

A **direct access** or **hashed-key** storage method solves this problem better than any other approach. The method works by first setting aside enough space to store all the key values you might need in numbered storage locations. Then the key value (bar code) is converted to a storage location number. Computer researchers have determined that a prime modulus function usually provides the best conversion. For example, you might have 100 elements with key values ranging from 100 to 9911. You choose a prime number approximately equal to the number of elements. For this case 101 is a good prime number. Then you divide each key value by the prime number and look at the remainder. As shown in Figure 12.15, a key value of 528 has a remainder (or modulus) of 23. Hence data for that key will be stored in location number 23. There is one catch—some keys might have the same modulus. The system sets aside an overflow area for these collisions, which it searches sequentially.

Uses

The hashed-key approach is extremely fast for finding and storing random data. The key's value is immediately converted into a storage location, and data can be retrieved in one pass to the disk. This method works best for transaction operations that require instantaneous retrieval of small amounts of data.

The hashed-key storage method requires you to know approximately how many items will be stored in the table. It also works best if the data does not change very often. It is acceptable to set aside enough space to add a few items. The method begins to deteriorate if key values are constantly being added to the table.

Drawbacks

One drawback to the hashed-key storage method is that it has little or no provision for sequential retrieval of data. It is possible to retrieve the data and sort it. Some

Operation	Sequential	B+Tree	Hashed
Read one	••	••••	•••••
Read next	•••••	••••	•••
Read all	•••••	••••	•••
Insert	•	••••	••••
Delete	•	••••	••••
Modify	•	••••	••••
Reorganize	••	••••	•••

Figure 12.16

Comparison of access methods. The B+tree is the best overall method to store and retrieve data. Sequential is useful for large tables that do not change often and need only sequential access. Hashed is useful for rapid access to individual items.

order-preserving hash functions exist to keep the keys in a predefined order. However, sequential retrieval will be slower than with a B$^+$tree index.

A second drawback is that the method sets aside storage space for the data, so you have to know how much space will be needed before you collect the data. If you add items to the table, they tend to end up in overflow storage, which is substantially slower. Performance can be improved by reorganizing the table—which creates more space and uses a new prime number. However, it takes time to reorganize the table, which should be done when the data is not being heavily used.

Bitmap Index

Some vendors (e.g., Oracle) provide highly compressed bitmap indexes for large tables. With a **bitmap index** each data key is encoded down to a small set of bits. The bitmap (binary) image of the entire index is usually small enough to fit in RAM. High-speed bit operations are used to make comparisons and search for key values. Hence the bitmap indexes are extremely fast. Bitmap indexes are particularly useful for columns like secondary keys that contain large amounts of repeating data. They should not be used for a column that contains all unique values. For example, in a typical SaleItem(SaleID, ItemID, Quantity) table, you could consider using a bitmap index for the SaleID and ItemID columns. But you would not want to use a bitmap index for the SaleID column in the Sale table. In Oracle, you use the CREATE BITMAP INDEX to generate a new bitmap index.

Comparison of Access Methods

All of these access methods are critical to computer scientists who create the DBMS. As an application developer, you do not need to know the gory technical details of the various methods. However, you do need to understand the strengths, weaknesses, and best uses of the methods. A good DBMS will let you choose how you want to store each table. At a minimum the DBMS will provide the ability to specify indexes for various columns. To determine which method should be used to store and retrieve data, you need to know two things: the primary operations that will be performed on the table and which method best supports those operations. Figure 12.16 answers the second question by summarizing the comments from the previous sections.

In practice, you have only three choices. First, the B$^+$tree is the best overall method to store and retrieve data. In almost any table the primary-key columns should be stored in a B$^+$tree index to speed the join operations in queries. Second, hashed access should be used for tables that do not change often and the application requires fast retrieval or storage of data based on a key value. Third, sequential storage can be used if a table almost never changes and the application always retrieves data sequentially and in large chunks. Generally, your choice comes down to B$^+$tree or hashed access. If you have tables that change often, you should consider removing indexes—which creates a sequential table. Most modern databases use some version of B$^+$tree storage. Primary keys are almost always indexed this way by default.

Storing Data Columns

How is one data row stored? The previous section explored the various methods of storing and retrieving individual rows of data. The second issue in storing data is how to store individual columns of data within a single row. For basic business data consisting of numbers and short text, it rarely matters how individual columns are stored. However, applications are being developed that need to store more complex data such as large amounts of text, graphics, sound, and even video clips. This data is relatively complex and requires significantly more storage space. Despite the declining cost of storage space, some of these objects are so large that you must be careful in how the database allocates storage for each item.

Text and Numbers

Fixed-width or positional storage is the simplest means of storing a row of data as shown in Figure 12.17. Each column is allocated a fixed number of bytes, and the data is stored in a set position. When the DBMS retrieves a row, it can find each column because the table definition lists the starting position of each column. The biggest drawback to this method is that at the start you must decide on the width of each column. Any data that does not fit into the assigned width will be truncated. This decision causes problems. For example, how much space should you set aside for a customer name? If you pick a small number, you risk throwing away part of a customer's name. If you pick a large number, the database sets aside that much space for every row of data—wasting space for most situations. This type of storage is used when you specify the domain as numeric or a CHAR column with a fixed width.

Figure 12.17

Fixed-width or positional-column storage. If data widths do not vary much, this method is a fast, efficient means to store columns. If descriptions can be short or very long, then you will have to allocate space for the longest possible description, which wastes space for the short descriptions.

ID	Price	QOH	Description
4	110.00		Dog Kennel-Extra Large
18	1.00	1874	Cat Food-Can-Premium
29	6.00	240	Flea Collar-Cat

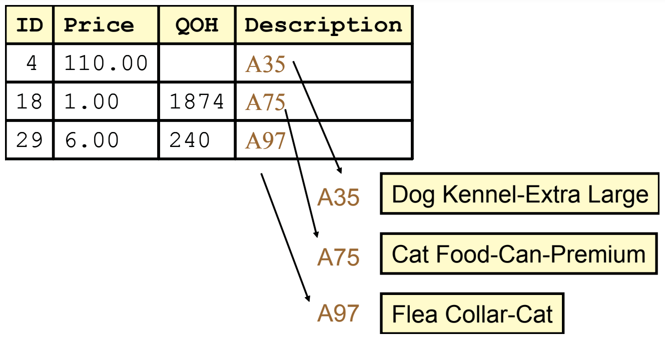


Figure 12.18

Variable length columns. Text columns that can be variable should be stored as variable width (varchar). The DBMS stores a pointer to the data that is stored in a pool.

The problem of deciding how much text space to allocate is common. Hence, a solution was developed to accommodate text data that is highly variable in length. For example, descriptions, comments, and memos can be long or short. In these situations the best storage method to use is the **variable length** method shown in Figure 12.18. In this case only a pointer is held in the actual row of the table. The data is stored in a separate pool. In SQL databases you specify this type of storage by selecting the **VARCHAR** column type. Some databases also provide a memo or comment data type to implement this type of storage. For example, Access provides a Memo type, which can hold large chunks of text. The Memo type can hold up to 64,000 characters, whereas text columns are limited to 255. For most systems you should always use the VARCHAR instead of fixed-width CHAR to store a text column. The exception is that small text columns, such as a two-letter state code, will be slightly more efficient if you use fixed width.
Note that numeric data is almost never stored as characters. Instead, it is stored in binary format to save space. The numbers used in these figures are just for illustration. You rarely have to worry about the width of numeric columns; they typically use either 4 or 8 bytes of storage.

One of the more challenging problems is storing variable-length string data, particularly when the lengths can vary widely, such as comments. If the system allocates a fixed amount of space, every row would be at the maximum value, and most of the space would be wasted. On the other hand, if the system allocates space for each row dynamically, then some rows will be shorter than others. This approach saves space, but makes it more difficult to handle modifications of the data. If the new data is longer than the old row, the system cannot just overwrite the old row. Some systems (e.g., Oracle) solve this dilemma by allocating data blocks to hold a group of rows. Each block contains a certain amount of free space. The DBMS uses this free space to store modified data that is longer than the existing row. In Oracle, you can control the amount of free space through two parameters: PCTFREE and PCTUSED. If the current data block is fuller than the PCTFREE value, no new rows are added to the block. The remaining space is kept for expansion of existing rows. See the *Oracle Server Administrator's Guide* for details and suggestions on values for these parameters.

Image and Binary Data

Most DBMSs provide the ability to store binary data within the database itself. For example, you can create a column to hold a picture for each row. Unfortunately, no standards exist for defining these columns or using this data. Hence, if you use these features, it is difficult to convert your database to another vendor's format. The data type varies depending on the DBMS: Access uses an OLE Object column, Oracle uses LONG RAW or BLOB (binary large object), SQL Server uses image. More importantly, the internal data format, and the storage and retrieval methods are different for each vendor. It is relatively easy to create and store data in these formats. The only problem is if you need to transfer data from one DBMS to another. Usually, the only answer is to retrieve each object one at a time, return it to its native format, and then store it in the new DBMS. It is a relatively painful process that you want to avoid.

From a performance standpoint, you will have to experiment with each application to decide if it is worthwhile to use these binary data types. The advantage of storing binary data within the DBMS is that you gain the use of the concurrency protection and database backup facilities. The main drawback is the difficulty in accessing the binary data using other software. Most software applications (e.g., drawing packages) do not know how to store and retrieve data from your DBMS, so you need to create an intermediate program to handle the exchange.

The alternative to storing binary files within the DBMS is to store them in a separate subdirectory, and then store only the file name within a text column in the database. This method is commonly used for Web-based applications. The Pet Store example uses this method to provide support with different databases.

Transferring Data with Delimited Files

If you need to transfer data to a different database or a different application, you often have to use a delimited file. It is often called a delimited file because the table is converted to standard text characters (no binary numbers). As shown in Figure 12.19, each column is separated by a specific character or delimiter. A comma is a common delimiter. Because text data might contain commas or other special characters, text columns are enclosed in quotation marks. Spaces are eliminated unless they are in quoted text columns. Missing data is simply not displayed, so if a column is missing, the data row would have two adjacent commas (e.g., 110,,"Dog …"). This technique is not very useful for permanent use within a database. Every time it retrieves a row, the DBMS has to search for the commas and interpret the quotes to find a particular column. However, it is a good way to transfer data between different systems. It is also good at saving space—particularly when many columns are missing.

Figure 12.19

Delimited files. Each column is separated by a special delimiter character (,). Text columns are quoted to protect spaces and hide special characters like commas. This method is often used to transfer files to different databases or other applications.

```
4, 110, , "Dog Kennel-Extra Large"
18, 1, 1874, "Cat Food-Can-Premium"
29, 6, 240, "Flea Collar-Cat"
```

Data Clustering and Partitioning

How can you improve performance by specifying where data is stored? Another way to improve database performance is to control the location of individual components of the table. For example, some parts of your application may always be retrieved together, so performance might improve if the two sets of data are retrieved together. On the other hand, sometimes you collect data that might not be accessed very often. It is still worthwhile to keep the data, but it might be better to store it on cheaper, slower drives. A third technique exists to speed up access to data by spreading it across several disk drives. All three situations are related in that they involve partitioning data and controlling where it is stored to improve performance. The key to understanding these methods is to remember that mechanical disk drives are slow. Every access to the disk that can be avoided will improve the application's speed.

Data Clustering

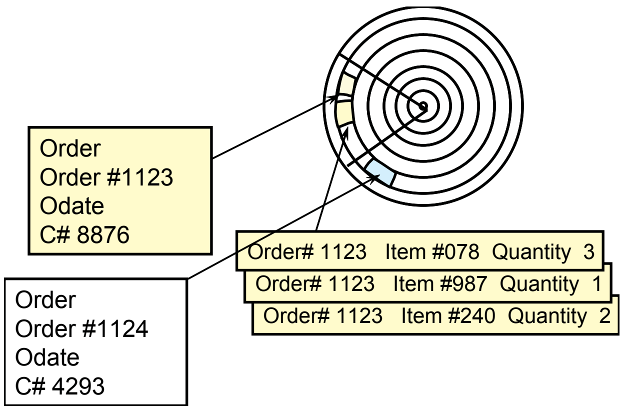
To improve general system performance, most computers retrieve data in chunks. They try to anticipate the next demand and read ahead of the current request. If the system guesses correctly, the next data request can be filled from RAM, which is substantially faster than waiting for the drive to spin around again.

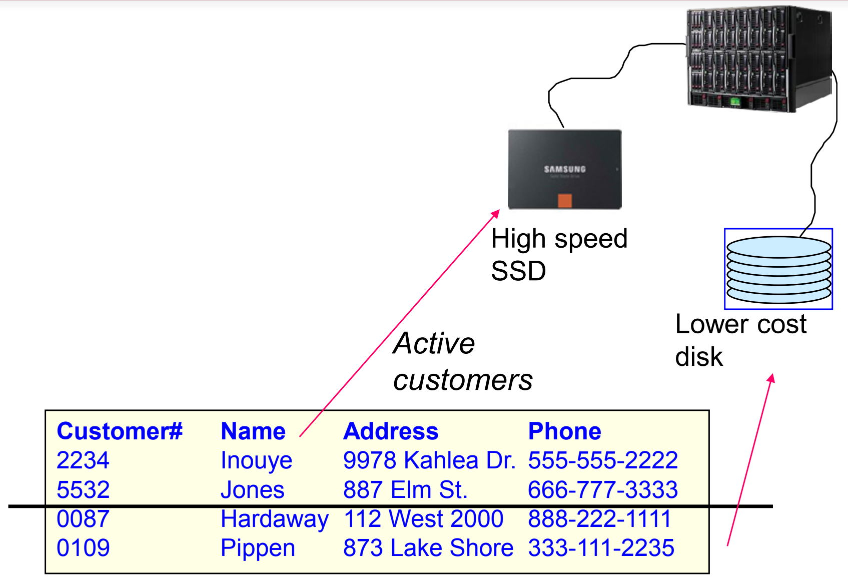
Database systems designers have used this concept to improve performance of database applications. Some parts of an application are generally used at the same time. Consider the example presented in Figure 12.20. Generally, when users look at order items, they also want to see the related data stored in the order table. By storing all the data for Order 1123 in the same data block, the data can be retrieved in one pass. The application will run faster because it avoids a second trip to the disk drive.

If you are using a DBMS that supports data clustering, you can improve performance by identifying data that is commonly accessed together. To create a cluster, you need to specify the tables involved and the key columns that link those tables.

Figure 12.20

Data clustering. Order and OrderItem data are usually needed at the same time. By storing them close to each other, the computer can retrieve them in one pass. Clustering the data improves application speed by reducing the number of disk accesses.


Order
Order #1123
Odate
C# 8876
Order
Order #1124
Odate
C# 4293
Order# 1123 Item #078 Quantity 3
Order# 1123 Item #987 Quantity 1
Order# 1123 Item #240 Quantity 2



Customer#	Name	Address	Phone
2234	Inouye	9978 Kahlea Dr.	555-555-2222
5532	Jones	887 Elm St.	666-777-3333
0087	Hardaway	112 West 2000	888-222-1111
0109	Pippen	873 Lake Shore	333-111-2235

Figure 12.21

Horizontal partition. Data for currently active customers is stored on high-speed drives. Older data is moved to cheaper, slower drives. The user does not need to know about the split because the DBMS automatically retrieves the data.

The DBMS then automatically stores and retrieves the related data in the same cluster. Only some of the large transaction-oriented database systems support clustering. For example, Oracle has a CREATE CLUSTER command to define the tables and key columns.

Data Partitioning

Another situation that commonly arises in business applications is that some data is used more frequently than other data. Even in the same table, you might collect data that is used only occasionally. For example, a basic customer table could contain information on customers who have not placed orders for several years that the marketing department wants to keep. Because the data is rarely used, it would be nice to move it to a cheaper storage location.

As shown in Figure 12.21, this situation would involve a **horizontal partition**. Some of the rows (currently active customers) will be stored in one location, and other rows (inactive customers) will be stored in a different location. The active data will be stored on high-speed disk drives. In extreme situations, some of this data could be stored on solid-state RAM drives, which hold all data in semiconductor RAM. On the other hand, the less-used data can be placed on slower-speed optical drives. The optical drives can hold huge amounts of data at a low cost; however, their access speeds are somewhat slower.

The key to making this approach work is that after you set it up, a good DBMS automatically retrieves the data from the appropriate drive. The user does not have to know that the data is stored on different drives. A single SQL query will retrieve the data—wherever it is stored. The high-end DBMSs provide several methods for determining the partition. Common methods include range partitioning (e.g.,

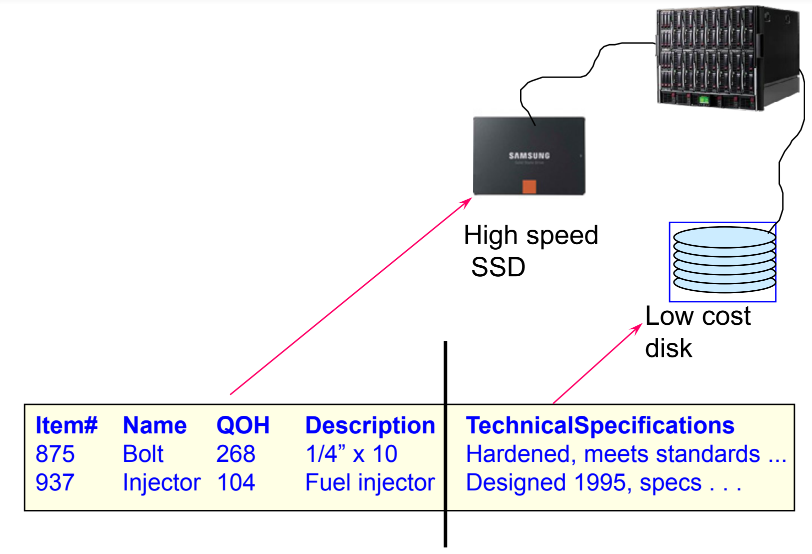

SAMSUNG
High speed SSD
Low cost disk
Item# Name QOH Description TechnicalSpecifications
875 Bolt 268 1/4" x 10 Hardened, meets standards ...
937 Injector 104 Fuel injector Designed 1995, specs . . .

Figure 12.22

Vertical partition. Technical data and images that are not accessed very often can be stored on a high-capacity, low-cost, but slower hard drive or even an optical drive.

specify a range of ID values) and list partitioning (e.g., list the key values that fall into each partition).

Vertical partitioning uses the same logic. The only difference is that with vertical partitioning, some columns of data are stored on a faster drive, whereas others are moved to cheaper and slower drives. Figure 12.22 shows how a product table might be split into two pieces. Basic business data used in transactions is stored on a high-speed disk. Detailed technical specifications and images are stored on high-capacity optical disks. Most day-to-day operations will use the basic data stored on the high-speed drive. However, the detailed data is readily available to anyone who needs it. The only difference is users will wait a little longer to retrieve the data on the slower drive.

In theory, data can be partitioned using any DBMS. Simply define two tables that can be joined by a common key. Then store each table on the appropriate drive. The difficulty with this approach is that anyone who wants to use the data will have to know that it is stored in different tables. You can circumvent this issue by building a query that automatically combines the tables. Then users can pull data from the query without having to know where each piece is stored.

In practice, horizontal partitioning is often used to split data so that it can be stored in locations where it will be used the most. For instance, you might split a customer table so that each regional office has the set of customers that it deals with the most.

On the other hand, vertical partitioning is useful for limiting the amount of data that you need to read into memory. If some columns are rarely used, they can be stored in a separate table. Overall performance will improve because the DBMS will be able to retrieve more of the smaller rows.

Managing Tablespaces

How does a DBA control file storage? Each vendor provides different methods to monitor and control database performance. These tools are a major selling point for each vendor. Smaller systems like Microsoft Access provide only limited control over the physical storage of data. System developers generally use the storage methods that are appropriate for the most general situations (B^+tree). You control column storage by the data type you assign.

Larger systems like Oracle provide a variety of tools to help evaluate and manage the performance of the database. For example, Oracle sets clustering and provides hashed access with the CREATE CLUSTER command. Indexed files can also be partitioned and clustered. Oracle database performance can also be tuned with various parameters. For example, the PCTFREE and PCTUSED options specify how tightly the data should be packed into the defined space. Various STORAGE parameters specify how the database should be expanded as it grows. Tables and indexes are stored in tablespaces, which are areas that the database administrator allocates on a drive. By specifying the location of the tablespaces, you can allocate data on specific drives. You can improve performance by storing each element in a tablespace on a different drive. For example, large databases should store transaction and recovery logs and main data on different drives.

Sally's Pet Store

What performance issues might arise at Sally's Pet Store? At the start the Pet Store database should have few performance problems. Beginning in one store, an ambitious system might store the database on a central computer, which is connected to three or four other computers in the store. Reasonably up-to-date personal computers should be able to handle the initial database. As accounting functions are added, or if the system needs to expand beyond a single store, then the system would have to be reevaluated.

At the current time, there should be few concerns about performance tuning. However, to improve performance, all primary keys should be indexed. Microsoft Access generally defines these indexes by default, but you should examine each table to be sure. Be careful when assigning indexes to columns that are part of a concatenated key. The index on a partial key must allow duplicates.

One potential area for problems is the City table. This table currently holds basic data on cities throughout the United States. Performance could be improved by reducing the number of cities—on the assumption that most customers would come from the surrounding communities. However, if you choose to keep the data, you can improve performance by thinking about how the table will be accessed. In particular, it is often searched by ZIP code. Similarly, because users often want a sorted list of the cities, it would be useful to index the City column. Are there too many indexes for one table? You could test the performance of retrievals before and after adding the indexes. However, note that the City table is predominantly used for retrieval and rarely used to add data. Hence building additional indexes makes sense.

The same situation probably exists for the Merchandise table. Most applications and users will retrieve data from the Merchandise table, with few updates, deletions, or insertions. Hence you might build additional indexes on that table.

For now, partitioning and clustering are not warranted. Over time, as the business expands, you might want to move some of the older data to less expensive storage devices. For instance, data on animals sold more than 5 or 10 years ago will probably not be used often and could be placed on slower CD-ROM drives. Similarly, inactive customer data, and older order data can be moved from the primary tables. The exact dates will depend on the cost of storage, discussions with Sally, observation of retrieval patterns, and legal needs.

Summary

Large application databases sometimes need to be fine-tuned to improve their performance. Some systems provide control over how the data is stored and retrieved. Three basic types of controls can be used to determine (1) how table rows are stored and retrieved, (2) how individual columns are stored, and (3) how data is clustered or partitioned.

The primary choices for storing rows of data are B^+tree indexes, hashed-key access, and sequential files. The method depends on how the data is used in terms of the standard database operations. The most challenging operations are searching for random entries and adding new data to the table. The B^+tree approach is the most common because it provides the best overall access for a variety of situations. In particular, it provides reasonably fast random access, good sequential retrieval, and good performance for inserting and deleting rows of data. In contrast, the hashed-key approach provides high-speed random access to any data element, but it is poor at retrieving data sequentially. Sequential files are rarely used, because although they use a minimum of space, they provide weak access to random rows of data.

Most DBMSs provide some control over how individual columns can be stored. The most common feature enables developers to control the storage of text data. Large text columns should be stored in varying-character columns instead of fixed-width columns. You should also be familiar with using delimited files for transferring data to different systems.

Some systems can cluster data in common locations on the disk drive. This approach improves performance by enabling the disk drive to retrieve related data in one pass. Another useful technique is to partition data so that data that is used less often can be moved to less expensive, slower disk drives. RAID systems provide another performance gain by splitting data and storing it on independent disk drives within the same system. The RAID drives can store and retrieve data substantially faster than a single disk drive can. RAID drives can also provide automatic backup by storing each component on two different drives.

Be careful when attempting to improve the performance of an application. Changes that help one area can adversely affect other operations. This trade-off is important when creating indexes for columns in a table. Indexes tend to improve data retrieval but slow down the processing when data is added to the table.

A Developer's View

As Miranda's problems indicate, database performance can become an important issue. Performance problems should be anticipated and solved as early as possible in design and development. You do not have to be intimately familiar with how the DBMS stores data. However, you do need to know which options are available to you. With many systems, the most important control you have is in choosing which columns to index. Sometimes you can choose the exact storage method. You need to understand the strengths and weaknesses of the various methods so that you can choose the method that best fits your application's needs. For your class project, you should identify the columns that should be indexed. You might have to generate sample data and compare processing time for various operations.

Key Terms

address
bitmap index
degree
depth
direct access
fixed-width
hashed-key
horizontal partition
index
leaves
node
offset
pack
pointer
root
VARCHAR
variable length
vertical partition

Review Questions

1. What basic data operations are performed on tables?
2. What are the primary data storage methods for tables?
3. What are the strengths and weaknesses of sequential storage?
4. How do linked lists solve insert and delete problems?
5. What are the strengths and weaknesses of indexed (B^+tree) storage?
6. What are the strengths and weaknesses of hashed (direct access) data storage?
7. How does data clustering improve database performance?
8. How does data partitioning improve database performance?
9. How is storage different for CHAR versus VARCHAR data types?

Exercises

1. Using the documentation for one DBMS, write the commands to create a table using a hashed-key index on an integer primary key column.
2. Based on the sample data in Figure 12.10, write the logic for the code to insert a new element in a linked list.
3. Research the documentation, DBAs, magazine, or Internet sources and find two methods or tricks that can be used to improve performance of your DBMS. Identify the specific problem the hint is designed to solve.
4. Create a B$^+$tree (degree 3). Show each final tree.
   a) The base tree holds the following key values: 1038, 1164, 2314, 3678, 4164, 5931, 6104, 7368, 7547, 8442, 8556, 8777, and 9114.
   b) Add the key value 8655.
   c) Add the key value 2715.
   d) Add the key value 10911.
   e) Add the key value 2941.
   f) Delete the key value 9114.
5. Draw a linked list.
   a) Start with the following key values: 341, 492, 561, 678, 781, and 856.
   b) Show how to insert the key value 603.
   c) Show how to delete the key 781.
6. Create a hashed storage example. Use a prime number of 53. Show the storage of the following numbers: 781, 467, 198, 435, 351, 782, and 149.
7. Write the commands to partition a Customer table based on the CustomerID. Older data has lower values for CustomerID, so split the table into three partitions based on values of 10,000 and 20,000.

Sally's Pet Store

8. The basic version of the database is relatively small and there should not be any current performance problems. However, if the company expands into several cities with multiple stores, performance could become more important. Outline a plan for how you could expand the database to handle this situation. Identify the DBMS software you would choose.
9. Go through the list of tables and classify them into two groups: (1) Transaction tables that receive many updates, and (2) Lookup or analytical tables that are used in transactions but are seldom updated, so they can include more indexes.
10. Copy the City table and remove all of the indexes from the copy. Create a query that counts the number of customers from each state using the original City table. Create a second copy of the query that uses the copy of the City table. Run both queries and comment on the performance of the two queries.

Rolling Thunder Bicycles

11. Make a copy of the Rolling Thunder database. Write SQL statements to perform the following operations on the Bicycle table: (a) add a row, (b) delete a row, (c) select all rows, and (d) write a program to change one value in every row. Write four short programs to perform these operations in a loop that repeats at least 100 times. Run the programs and record the time it takes to perform the operations. Next, index every column in the Bicycle table and rerun your tests. Record and analyze your results.
12. Examine the tables and the usage of each table in the Rolling Thunder application. Identify the primary uses of each table in terms of the table operations described in this chapter. Use this list to identify desired indexes and appropriate storage methods for each table if the database becomes large.
13. Examine the tables in Rolling Thunder and identify which tables should be clustered. Which tables could gain from partitioning? If the application is expanded, what new data could be added that might gain from partitioning?

Corner Med

14. Examine the tables and the usage of each table in the Corner Med database. Assume the database is going to become relatively large when it is used at multiple locations. Identify the tables that are primarily transaction based versus the lookup and analysis tables. Use these lists to specify additional indexes that might be added (or removed) to improve performance. What other options could be used to improve performance?
15. Assuming the company has operated for five years, how would you partition the data to reduce storage needs and improve performance?
16. If the company decides to digitize other medical records (x-rays, photos, lab results, prescriptions, and so on), what performance problems can be expected? How will you minimize these issues?

Web Site References

http://www.sql-server-performance.com	Hints on improving performance for SQL Server.
http://docs.oracle.com/cd/B19306_01/server.102/b14211/toc.htm	Oracle performance tuning.
http://www.bluerwhite.org/btree/	General B-tree information and coding.

Additional Reading

Baeza-Yates, R. and Ribeiro-Neto, B. *Modern Information Retrieval*, Reading, MA: Addison-Wesley, 1991. [Computer science approach to storing and retrieving data, includes Web access and multimedia.]

Goetz, G., Modern B-Tree Techniques, Boston: Now Publishers, 2011. [Basic textbook on B-tree processing.]

Korfhage, R., *Information Storage and Retrieval*, New York: Wiley & Sons, 1997. [Summary of data storage methods.]

Loomis, M. *Data Management and File Processing*, Upper Saddle River, NJ: Prentice-Hall, 1983. [In-depth treatment of data storage issues such as B-trees.]

Dunham, J. *Database Performance Tuning Handbook*, Berkeley: McGraw-Hill, 1997. [In-depth treatment of improving your application's performance.]