Part 5

Selected Database Issues

Chapter 19 Security

Chapter Objectives

In this chapter you will learn:

- The scope of database security.
- Why database security is a serious concern for an organization.
- The types of threat that can affect a database system.
- How to protect a computer system using computer-based controls.
- The security measures provided by Microsoft Office Access and Oracle DBMSs.
- Approaches for securing a DBMS on the Web.

Data is a valuable resource that must be strictly controlled and managed, as with any corporate resource. Part or all of the corporate data may have strategic importance to an organization and should therefore be kept secure and confidential.

In Chapter 2 we discussed the database environment and, in particular, the typical functions and services of a Database Management System (DBMS). These functions and services include authorization services, such that a DBMS must furnish a mechanism to ensure that only authorized users can access the database. In other words, the DBMS must ensure that the database is secure. The term **security** refers to the protection of the database against unauthorized access, either intentional or accidental. Besides the services provided by the DBMS, discussions on database security could also include broader issues associated with securing the database and its environment. However, these issues are outwith the scope of this book and the interested reader is referred to Pfleeger (1997).

Structure of this Chapter

In Section 19.1 we discuss the scope of database security and examine the types of threat that may affect computer systems in general. In Section 19.2 we consider the range of computer-based controls that are available as countermeasures to these threats. In Sections 19.3 and 19.4 we describe the security measures provided by Microsoft Office Access 2003 DBMS and Oracle9*i* DBMS. In Section 19.5 we identify the security measures associated with DBMSs and the Web. The examples used throughout this chapter are taken from the *DreamHome* case study described in Section 10.4 and Appendix A.

19.1 Database Security

In this section we describe the scope of database security and discuss why organizations must take potential threats to their computer systems seriously. We also identify the range of threats and their consequences on computer systems.

Database security The mechanisms that protect the database against intentional or accidental threats.

Security considerations apply not only to the data held in a database: breaches of security may affect other parts of the system, which may in turn affect the database. Consequently, database security encompasses hardware, software, people, and data. To effectively implement security requires appropriate controls, which are defined in specific mission objectives for the system. This need for security, while often having been neglected or overlooked in the past, is now increasingly recognized by organizations. The reason for this turnaround is the increasing amounts of crucial corporate data being stored on computer and the acceptance that any loss or unavailability of this data could prove to be disastrous.

A database represents an essential corporate resource that should be properly secured using appropriate controls. We consider database security in relation to the following situations:

- theft and fraud;
- loss of confidentiality (secrecy);
- loss of privacy;
- loss of integrity;
- loss of availability.

These situations broadly represent areas in which the organization should seek to reduce risk, that is the possibility of incurring loss or damage. In some situations, these areas are closely related such that an activity that leads to loss in one area may also lead to loss in another. In addition, events such as fraud or loss of privacy may arise because of either

intentional or unintentional acts, and do not necessarily result in any detectable changes to the database or the computer system.

Theft and fraud affect not only the database environment but also the entire organization. As it is people who perpetrate such activities, attention should focus on reducing the opportunities for this occurring. Theft and fraud do not necessarily alter data, as is the case for activities that result in either loss of confidentiality or loss of privacy.

Confidentiality refers to the need to maintain secrecy over data, usually only that which is critical to the organization, whereas privacy refers to the need to protect data about individuals. Breaches of security resulting in loss of confidentiality could, for instance, lead to loss of competitiveness, and loss of privacy could lead to legal action being taken against the organization.

Loss of data integrity results in invalid or corrupted data, which may seriously affect the operation of an organization. Many organizations are now seeking virtually continuous operation, the so-called 24/7 availability (that is, 24 hours a day, 7 days a week). Loss of availability means that the data, or the system, or both cannot be accessed, which can seriously affect an organization's financial performance. In some cases, events that cause a system to be unavailable may also cause data corruption.

Database security aims to minimize losses caused by anticipated events in a cost-effective manner without unduly constraining the users. In recent times, computer-based criminal activities have significantly increased and are forecast to continue to rise over the next few years.

Threats 19.1.1

Threat Any situation or event, whether intentional or accidental, that may adversely affect a system and consequently the organization.

A threat may be caused by a situation or event involving a person, action, or circumstance that is likely to bring harm to an organization. The harm may be tangible, such as loss of hardware, software, or data, or intangible, such as loss of credibility or client confidence. The problem facing any organization is to identify all possible threats. Therefore, as a minimum an organization should invest time and effort in identifying the most serious threats.

In the previous section we identified areas of loss that may result from intentional or unintentional activities. While some types of threat can be either intentional or unintentional, the impact remains the same. Intentional threats involve people and may be perpetrated by both authorized users and unauthorized users, some of whom may be external to the organization.

Any threat must be viewed as a potential breach of security which, if successful, will have a certain impact. Table 19.1 presents examples of various types of threat, listed under the area on which they may have an impact. For example, 'viewing and disclosing unauthorized data' as a threat may result in theft and fraud, loss of confidentiality, and loss of privacy for the organization.

Table 19.1 Examples of threats.

Threat	Theft and fraud	Loss of confidentiality	Loss of privacy	Loss of integrity	Loss of availability
Using another person's means of access	✓	✓	✓		
Unauthorized amendment or copying of data	✓			✓	
Program alteration	✓			✓	✓
Inadequate policies and procedures that allow a mix of confidential and normal output	✓	✓	✓		
Wire tapping	✓	✓	✓		
Illegal entry by hacker	✓	✓	✓		
Blackmail	✓	✓	✓		
Creating 'trapdoor' into system	✓	✓	✓		
Theft of data, programs, and equipment	✓	✓	✓		✓
Failure of security mechanisms, giving greater access than normal		✓	✓	✓	
Staff shortages or strikes				✓	✓
Inadequate staff training		✓	✓	✓	✓
Viewing and disclosing unauthorized data	✓	✓	✓		
Electronic interference and radiation				✓	✓
Data corruption owing to power loss or surge				✓	✓
Fire (electrical fault, lightning strike, arson), flood, bomb				✓	✓
Physical damage to equipment				✓	✓
Breaking cables or disconnection of cables				✓	✓
Introduction of viruses				✓	✓

The extent that an organization suffers as a result of a threat's succeeding depends upon a number of factors, such as the existence of countermeasures and contingency plans. For example, if a hardware failure occurs corrupting secondary storage, all processing activity must cease until the problem is resolved. The recovery will depend upon a number of factors, which include when the last backups were taken and the time needed to restore the system.

An organization needs to identify the types of threat it may be subjected to and initiate appropriate plans and countermeasures, bearing in mind the costs of implementing them. Obviously, it may not be cost-effective to spend considerable time, effort, and money on potential threats that may result only in minor inconvenience. The organization's business may also influence the types of threat that should be considered, some of which may be rare. However, rare events should be taken into account, particularly if their impact would be significant. A summary of the potential threats to computer systems is represented in Figure 19.1.

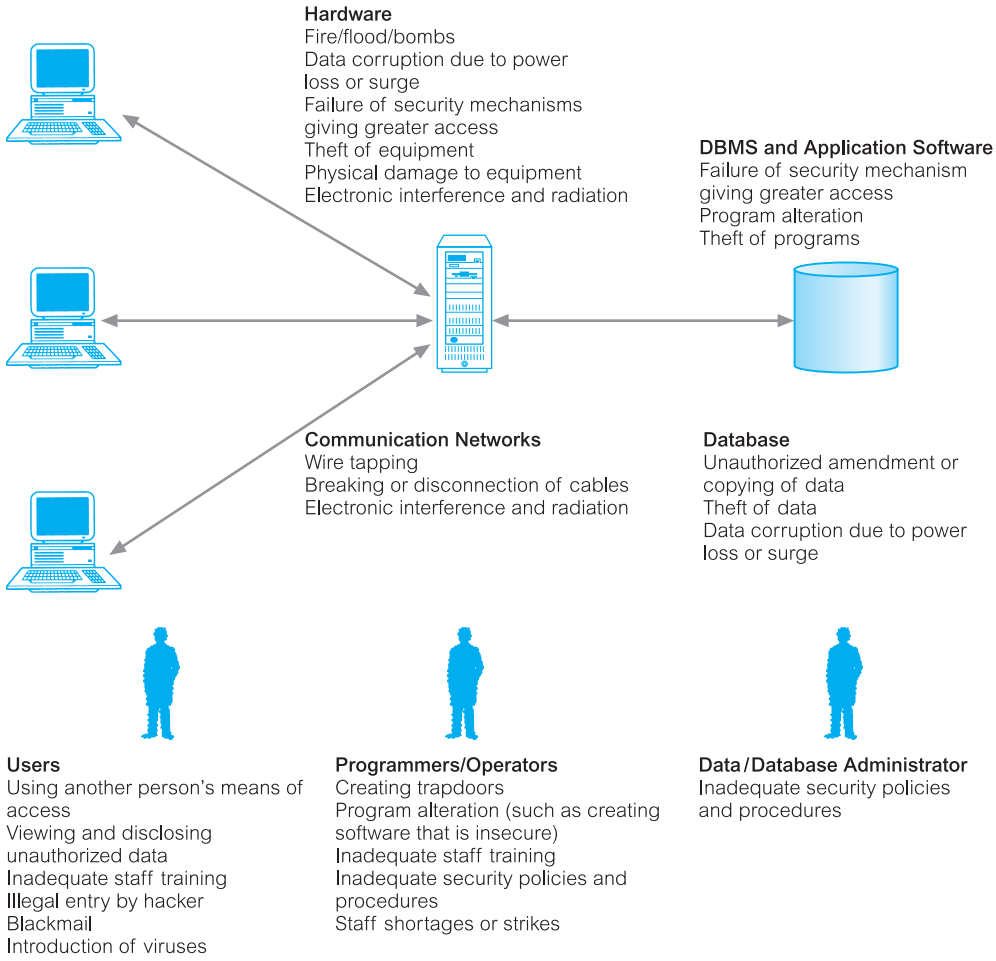


Figure 19.1 Summary of potential threats to computer systems.

19.2 Countermeasures – Computer-Based Controls

The types of countermeasure to threats on computer systems range from physical controls to administrative procedures. Despite the range of computer-based controls that are available, it is worth noting that, generally, the security of a DBMS is only as good as that of the operating system, owing to their close association. Representation of a typical multi-user computer environment is shown in Figure 19.2. In this section we focus on the following computer-based security controls for a multi-user environment (some of which may not be available in the PC environment):

- authorization
- access controls
- views
- backup and recovery
- integrity
- encryption
- RAID technology.

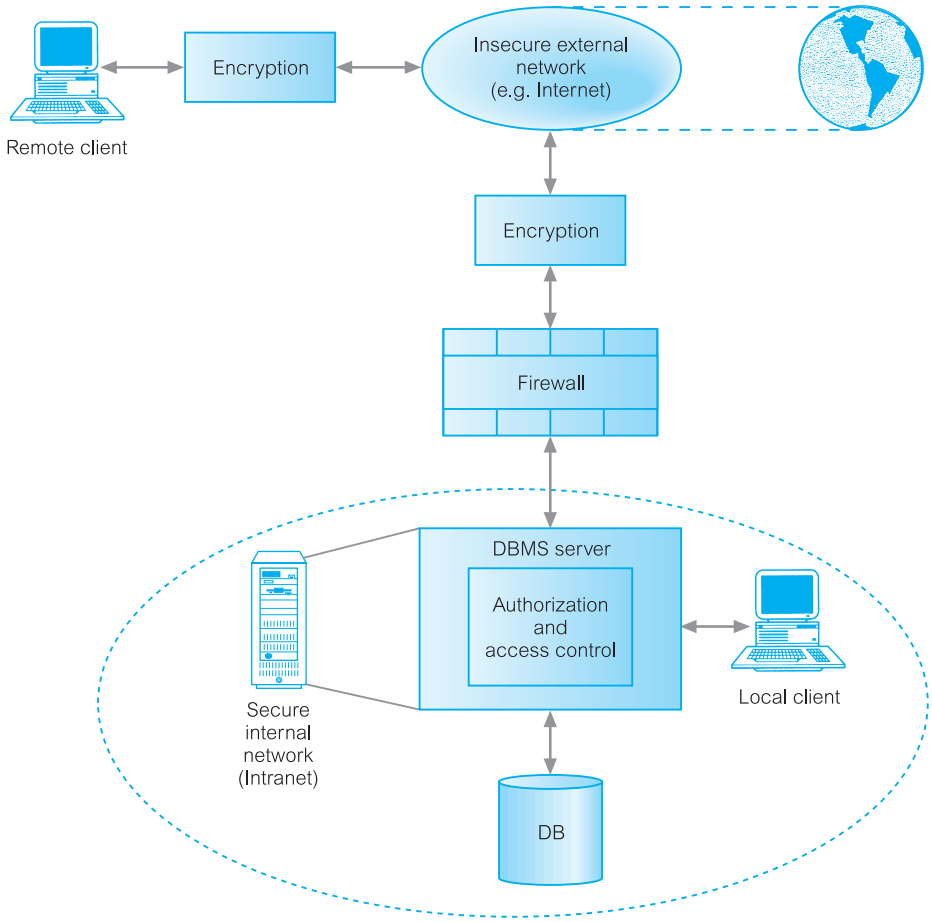


Figure 19.2
Representation of a typical multi-user computer environment.

19.2.1 Authorization

Authorization The granting of a right or privilege that enables a subject to have legitimate access to a system or a system's object.

Authorization controls can be built into the software, and govern not only what system or object a specified user can access, but also what the user may do with it. The process of authorization involves authentication of subjects requesting access to objects, where 'subject' represents a user or program and 'object' represents a database table, view, procedure, trigger, or any other object that can be created within the system.

Authentication A mechanism that determines whether a user is who he or she claims to be.

A system administrator is usually responsible for allowing users to have access to a computer system by creating individual user accounts. Each user is given a unique identifier, which is used by the operating system to determine who they are. Associated with each identifier is a password, chosen by the user and known to the operating system, which must be supplied to enable the operating system to verify (or authenticate) who the user claims to be.

This procedure allows authorized use of a computer system but does not necessarily authorize access to the DBMS or any associated application programs. A separate, similar procedure may have to be undertaken to give a user the right to use the DBMS. The responsibility to authorize use of the DBMS usually rests with the Database Administrator (DBA), who must also set up individual user accounts and passwords using the DBMS itself.

Some DBMSs maintain a list of valid user identifiers and associated passwords, which can be distinct from the operating system's list. However, other DBMSs maintain a list whose entries are validated against the operating system's list based on the current user's login identifier. This prevents a user from logging on to the DBMS with one name, having already logged on to the operating system using a different name.

Access Controls 19.2.2

The typical way to provide access controls for a database system is based on the granting and revoking of privileges. A **privilege** allows a user to create or access (that is read, write, or modify) some database object (such as a relation, view, or index) or to run certain DBMS utilities. Privileges are granted to users to accomplish the tasks required for their jobs. As excessive granting of unnecessary privileges can compromise security: a privilege should only be granted to a user if that user cannot accomplish his or her work without that privilege. A user who creates a database object such as a relation or a view automatically gets all privileges on that object. The DBMS subsequently keeps track of how these privileges are granted to other users, and possibly revoked, and ensures that at all times only users with necessary privileges can access an object.

Discretionary Access Control (DAC)

Most commercial DBMSs provide an approach to managing privileges that uses SQL called **Discretionary Access Control** (**DAC**). The SQL standard supports DAC through the GRANT and REVOKE commands. The GRANT command gives privileges to users, and the REVOKE command takes away privileges. We discussed how the SQL standard supports discretionary access control in Section 6.6.

Discretionary access control, while effective, has certain weaknesses. In particular, an unauthorized user can trick an authorized user into disclosing sensitive data. For example, an unauthorized user such as an Assistant in the *DreamHome* case study can create a relation to capture new client details and give access privileges to an authorized user such as a Manager without their knowledge. The Assistant can then alter some application programs that the Manager uses to include some hidden instruction to copy sensitive data from the Client relation that only the Manager has access to, into the new relation created by the Assistant. The unauthorized user, namely the Assistant, now has a copy of the sensitive data, namely new clients of *DreamHome*, and to cover up his or her actions now modifies the altered application programs back to the original form.

Clearly, an additional security approach is required to remove such loopholes, and this requirement is met in an approach called Mandatory Access Control (MAC), which we discuss in detail below. Although discretionary access control is typically provided by most commercial DBMSs, only some also provide support for mandatory access control.

Mandatory Access Control (MAC)

Mandatory Access Control (**MAC**) is based on system-wide policies that cannot be changed by individual users. In this approach each database object is assigned a *security class* and each user is assigned a *clearance* for a security class, and *rules* are imposed on reading and writing of database objects by users. The DBMS determines whether a given user can read or write a given object based on certain rules that involve the security level of the object and the clearance of the user. These rules seek to ensure that sensitive data can never be passed on to another user without the necessary clearance. The SQL standard does not include support for MAC.

A popular model for MAC is called Bell–LaPadula model (Bell and LaPadula, 1974), which is described in terms of **objects** (such as relations, views, tuples, and attributes), **subjects** (such as users and programs), **security classes**, and **clearances**. Each database object is assigned a *security class*, and each subject is assigned a *clearance* for a security class. The security classes in a system are ordered, with a most secure class and a least secure class. For our discussion of the model, we assume that there are four classes: *top secret (TS)*, *secret (S)*, *confidential (C)*, and *unclassified (U)*, and we denote the class of an object or subject A as *class (A)*. Therefore for this system, $TS > S > C > U$, where A > B means that class A data has a higher security level than class B data.

The Bell–LaPadula model imposes two restrictions on all reads and writes of database objects:

1. **Simple Security Property:** Subject S is allowed to read object O only if class (*S*) >= class (*O*). For example, a user with *TS* clearance can read a relation with *C* clearance, but a user with *C* clearance cannot read a relation with *TS* classification.
2. ***_Property:** Subject S is allowed to write object O only if class (*S*) <=class (*O*). For example, a user with *S* clearance can only write objects with *S* or *TS* classification.

If discretionary access controls are also specified, these rules represent additional restrictions. Thus to read or write a database object, a user must have the necessary privileges provided through the SQL GRANT command (see Section 6.6) and the security classes of the user and the object must satisfy the restrictions given above.

Multilevel Relations and Polyinstantiation

In order to apply mandatory access control policies in a relational DBMS, a security class must be assigned to each database object. The objects can be at the granularity of relations, tuples, or even individual attribute values. Assume that each tuple is assigned a security class. This situation leads to the concept of a **multilevel relation**, which is a relation that reveals different tuples to users with different security clearances.

For example, the Client relation with an additional attribute displaying the security class for each tuple is shown in Figure 19.3(a).

Users with *S* and *TS* clearance will see all tuples in the Client relation. However, a user with *C* clearance will only see the first two tuples and a user with *U* clearance will see no tuples at all. Assume that a user with clearance *C* wishes to enter a tuple (CR74, David, Sinclaire) into the Client relation, where the primary key of the relation is clientNo. This insertion is disallowed because it violates the primary key constraint (see Section 3.2.5) for this relation. However, the inability to insert this new tuple informs the user with clearance *C* that a tuple exists with a primary key value of CR74 at a higher security class than *C*. This compromises the security requirement that users should not be able to infer any information about objects that have a higher security classification.

This problem of inference can be solved by including the security classification attribute as part of the primary key for a relation. In the above example, the insertion of the new tuple into the Client relation is allowed, and the relation instance is modified as shown in Figure 19.3(b). Users with clearance *C* see the first two tuples and the newly added tuple, but users with clearance *S* or *TS* see all five tuples. The result is a relation with two tuples with a clientNo of CR74, which can be confusing. This situation may be dealt with by assuming that the tuple with the higher classification takes priority over the other, or by only revealing a single tuple according to the user's clearance. The presence of data objects that appear to have different values to users with different clearances is called **polyinstantiation**.

clientNo	fName	lName	telNo	prefType	maxRent	securityClass
CR76	John	Kay	0207-774-5632	Flat	425	*C*
CR56	Aline	Stewart	0141-848-1825	Flat	350	*C*
CR74	Mike	Ritchie	01475-392178	House	750	*S*
CR62	Mary	Tregar	01224-196720	Flat	600	*S*

Figure 19.3(a) The Client relation with an additional attribute displaying the security class for each tuple.

clientNo	fName	lName	telNo	prefType	maxRent	securityClass
CR76	John	Kay	0207-774-5632	Flat	425	*C*
CR56	Aline	Stewart	0141-848-1825	Flat	350	*C*
CR74	Mike	Ritchie	01475-392178	House	750	*S*
CR62	Mary	Tregar	01224-196720	Flat	600	*S*
CR74	David	Sinclaire				*C*

Figure 19.3(b) The Client relation with two tuples displaying clientNo as CR74. The primary key for this relation is (clientNo, securityClass).

Although mandatory access control does address a major weakness of discretionary access control, a major disadvantage of MAC is the rigidity of the MAC environment. For example, MAC policies are often established by database or systems administrators, and the classification mechanisms are sometimes considered to be inflexible.

19.2.3 Views

View A view is the dynamic result of one or more relational operations operating on the base relations to produce another relation. A view is a *virtual relation* that does not actually exist in the database, but is produced upon request by a particular user, at the time of request.

The view mechanism provides a powerful and flexible security mechanism by hiding parts of the database from certain users. The user is not aware of the existence of any attributes or rows that are missing from the view. A view can be defined over several relations with a user being granted the appropriate privilege to use it, but not to use the base relations. In this way, using a view is more restrictive than simply having certain privileges granted to a user on the base relation(s). We discussed views in detail in Sections 3.4 and 6.4.

19.2.4 Backup and Recovery

Backup The process of periodically taking a copy of the database and log file (and possibly programs) on to offline storage media.

A DBMS should provide backup facilities to assist with the recovery of a database following failure. It is always advisable to make backup copies of the database and log file at regular intervals and to ensure that the copies are in a secure location. In the event of a failure that renders the database unusable, the backup copy and the details captured in the log file are used to restore the database to the latest possible consistent state. A description of how a log file is used to restore a database is described in more detail in Section 20.3.3.

Journaling The process of keeping and maintaining a log file (or journal) of all changes made to the database to enable recovery to be undertaken effectively in the event of a failure.

A DBMS should provide logging facilities, sometimes referred to as journaling, which keep track of the current state of transactions and database changes, to provide support for recovery procedures. The advantage of journaling is that, in the event of a failure, the database can be recovered to its last known consistent state using a backup copy of the database and the information contained in the log file. If no journaling is enabled on a

failed system, the only means of recovery is to restore the database using the latest backup version of the database. However, without a log file, any changes made after the last backup to the database will be lost. The process of journaling is discussed in more detail in Section 20.3.3.

Integrity 19.2.5

Integrity constraints also contribute to maintaining a secure database system by preventing data from becoming invalid, and hence giving misleading or incorrect results. Integrity constraints were discussed in detail in Section 3.3.

Encryption 19.2.6

Encryption The encoding of the data by a special algorithm that renders the data unreadable by any program without the decryption key.

If a database system holds particularly sensitive data, it may be deemed necessary to encode it as a precaution against possible external threats or attempts to access it. Some DBMSs provide an encryption facility for this purpose. The DBMS can access the data (after decoding it), although there is a degradation in performance because of the time taken to decode it. Encryption also protects data transmitted over communication lines. There are a number of techniques for encoding data to conceal the information; some are termed 'irreversible' and others 'reversible'. Irreversible techniques, as the name implies, do not permit the original data to be known. However, the data can be used to obtain valid statistical information. Reversible techniques are more commonly used. To transmit data securely over insecure networks requires the use of a **cryptosystem**, which includes:

- an *encryption key* to encrypt the data (plaintext);
- an *encryption algorithm* that, with the encryption key, transforms the plaintext into *ciphertext*;
- a *decryption key* to decrypt the ciphertext;
- a *decryption algorithm* that, with the decryption key, transforms the ciphertext back into plaintext.

One technique, called **symmetric encryption**, uses the same key for both encryption and decryption and relies on safe communication lines for exchanging the key. However, most users do not have access to a secure communication line and, to be really secure, the keys need to be as long as the message (Leiss, 1982). However, most working systems are based on user keys shorter than the message. One scheme used for encryption is the **Data Encryption Standard (DES)**, which is a standard encryption algorithm developed by IBM. This scheme uses one key for both encryption and decryption, which must be kept secret, although the algorithm need not be. The algorithm transforms each 64-bit block of

plaintext using a 56-bit key. The DES is not universally regarded as being very secure, and some authors maintain that a larger key is required. For example, a scheme called PGP (Pretty Good Privacy) uses a 128-bit symmetric algorithm for bulk encryption of the data it sends.

Keys with 64 bits are now probably breakable by major governments with special hardware, albeit at substantial cost. However, this technology will be within the reach of organized criminals, major organizations, and smaller governments in a few years. While it is envisaged that keys with 80 bits will also become breakable in the future, it is probable that keys with 128 bits will remain unbeakable for the foreseeable future. The terms 'strong authentication' and 'weak authentication' are sometimes used to distinguish between algorithms that, to all intents and purposes, cannot be broken with existing technologies and knowledge (strong) from those that can be (weak).

Another type of cryptosystem uses different keys for encryption and decryption, and is referred to as **asymmetric encryption**. One example is **public key** cryptosystems, which use two keys, one of which is public and the other private. The encryption algorithm may also be public, so that anyone wishing to send a user a message can use the user's publicly known key in conjunction with the algorithm to encrypt it. Only the owner of the private key can then decipher the message. Public key cryptosystems can also be used to send a 'digital signature' with a message and prove that the message came from the person who claimed to have sent it. The most well known asymmetric encryption is **RSA** (the name is derived from the initials of the three designers of the algorithm).

Generally, symmetric algorithms are much faster to execute on a computer than those that are asymmetric. However, in practice, they are often used together, so that a public key algorithm is used to encrypt a randomly generated encryption key, and the random key is used to encrypt the actual message using a symmetric algorithm. We discuss encryption in the context of the Web in Section 19.5.

19.2.7 RAID (Redundant Array of Independent Disks)

The hardware that the DBMS is running on must be *fault-tolerant*, meaning that the DBMS should continue to operate even if one of the hardware components fails. This suggests having redundant components that can be seamlessly integrated into the working system whenever there is one or more component failures. The main hardware components that should be fault-tolerant include disk drives, disk controllers, CPU, power supplies, and cooling fans. Disk drives are the most vulnerable components with the shortest times between failure of any of the hardware components.

One solution is the use of **Redundant Array of Independent Disks** (**RAID**) technology. RAID originally stood for *Redundant Array of Inexpensive Disks*, but more recently the 'I' in RAID has come to stand for *Independent*. RAID works on having a large disk array comprising an arrangement of several independent disks that are organized to improve reliability and at the same time increase performance.

Performance is increased through *data striping*: the data is segmented into equal-size partitions (the *striping unit*) which are transparently distributed across multiple disks. This gives the appearance of a single large, fast disk where in actual fact the data is distributed across several smaller disks. Striping improves overall I/O performance by allowing

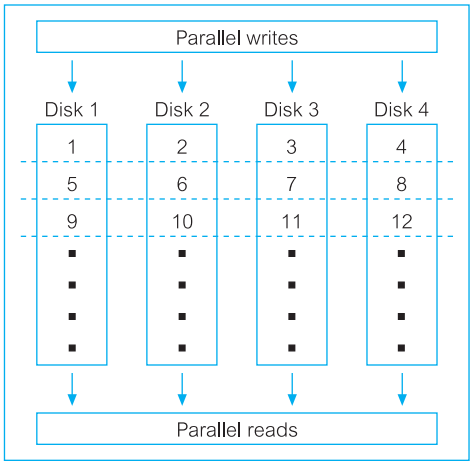
multiple I/Os to be serviced in parallel. At the same time, data striping also balances the load among disks.

Reliability is improved through storing redundant information across the disks using a *parity* scheme or an *error-correcting* scheme, such as Reed-Solomon codes (see, for example, Pless, 1989). In a parity scheme, each byte may have a parity bit associated with it that records whether the number of bits in the byte that are set to 1 is even or odd. If the number of bits in the byte becomes corrupted, the new parity of the byte will not match the stored parity. Similarly, if the stored parity bit becomes corrupted, it will not match the data in the byte. Error-correcting schemes store two or more additional bits, and can reconstruct the original data if a single bit becomes corrupt. These schemes can be used through striping bytes across disks.

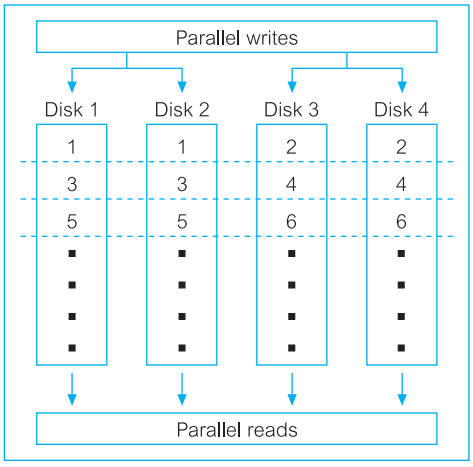
There are a number of different disk configurations with RAID, termed RAID *levels*. A brief description of each RAID level is given below together with a diagrammatic representation for each of the *main* levels in Figure 19.4. In this figure the numbers represent sequential data blocks and the letters indicate segments of a data block.

- RAID 0 – Nonredundant This level maintains no redundant data and so has the best write performance since updates do not have to be replicated. Data striping is performed at the level of blocks. A diagrammatic representation of RAID 0 is shown in Figure 19.4(a).
- RAID 1 – Mirrored This level maintains (*mirrors*) two identical copies of the data across different disks. To maintain consistency in the presence of disk failure, writes may not be performed simultaneously. This is the most expensive storage solution. A diagrammatic representation of RAID 1 is shown in Figure 19.4(b).
- RAID 0+1 – Nonredundant and Mirrored This level combines striping and mirroring.
- RAID 2 – Memory-Style Error-Correcting Codes With this level, the striping unit is a single bit and Hamming codes are used as the redundancy scheme. A diagrammatic representation of RAID 2 is shown in Figure 19.4(c).
- RAID 3 – Bit-Interleaved Parity This level provides redundancy by storing parity information on a single disk in the array. This parity information can be used to recover the data on other disks should they fail. This level uses less storage space than RAID 1 but the parity disk can become a bottleneck. A diagrammatic representation of RAID 3 is shown in Figure 19.4(d).
- RAID 4 – Block-Interleaved Parity With this level, the striping unit is a disk block – a parity block is maintained on a separate disk for corresponding blocks from a number of other disks. If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk. A diagrammatic representation of RAID 4 is shown in Figure 19.4(e).
- RAID 5 – Block-Interleaved Distributed Parity This level uses parity data for redundancy in a similar way to RAID 3 but stripes the parity data across all the disks, similar to the way in which the source data is striped. This alleviates the bottleneck on the parity disk. A diagrammatic representation of RAID 5 is shown in Figure 19.4(f).
- RAID 6 – P+Q Redundancy This level is similar to RAID 5 but additional redundant data is maintained to protect against multiple disk failures. Error-correcting codes are used instead of using parity.

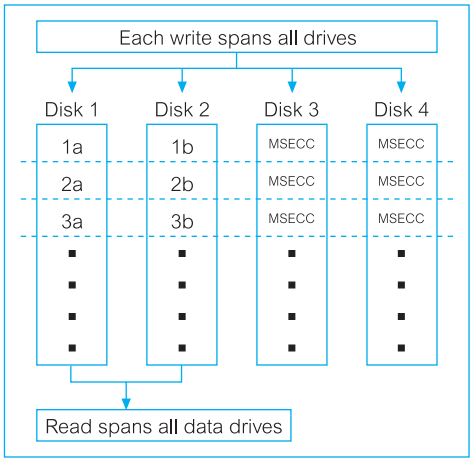
Figure 19.4
RAID levels. The numbers represent sequential data blocks and the letters indicate segments of a data block.



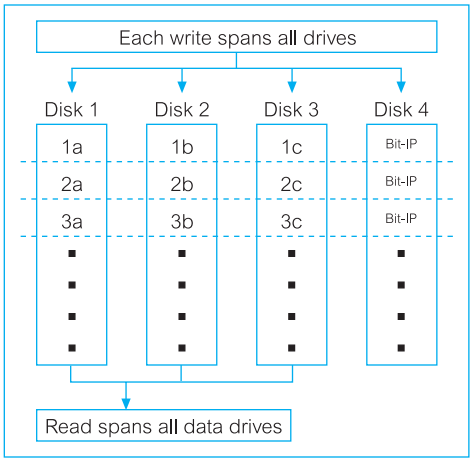
(a) RAID 0 – Nonredundant



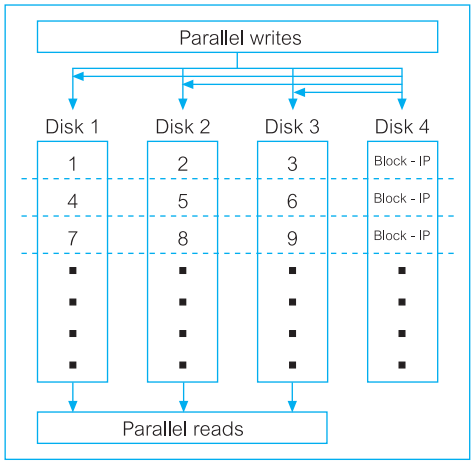
(b) RAID 1 – Mirrored



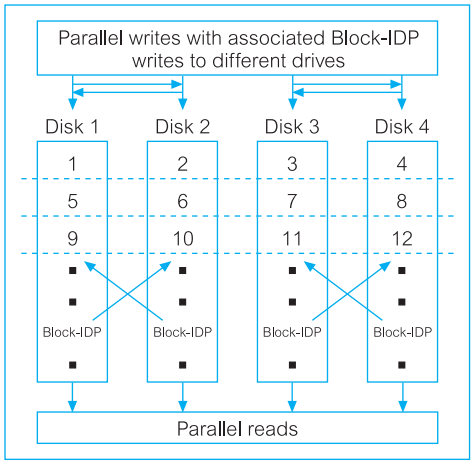
(c) RAID 2 – Memory-Style Error-Correcting Codes (MSECC)



(d) RAID 3 – Bit Interleaved Parity (Bit-IP)



(e) RAID 4 – Block-Interleaved Parity (Block-IP)



(f) RAID 5 – Block-Interleaved Distributed Parity (Block-IDP)

Oracle, for example, recommends use of RAID 1 for the redo log files. For the database files, Oracle recommends either RAID 5, provided the write overhead is acceptable, otherwise Oracle recommends either RAID 1 or RAID 0+1. A fuller discussion of RAID is outwith the scope of this book and the interested reader is referred to the papers by Chen and Patterson (1990) and Chen *et al*. (1994).

19.3 Security in Microsoft Office Access DBMS

In Section 8.1 we provided an overview of Microsoft Office Access 2003 DBMS. In this section we focus on the security measures provided by Office Access. In Section 6.6 we described the SQL GRANT and REVOKE statements; Microsoft Office Access 2003 does not support these statements but instead provides the following two methods for securing a database:

- setting a password for opening a database (referred to as *system security* by Microsoft Office Access);
- user-level security, which can be used to limit the parts of the database that a user can read or update (referred to as *data security* by Microsoft Office Access).

In this section we briefly discuss how Microsoft Office Access provides these two types of security mechanism.

Setting a Password

The simpler security method is to set a password for opening the database. Once a password has been set (from the **Tools**, **Security** menu), a dialog box requesting the password will be displayed whenever the database is opened. Only users who type the correct password will be allowed to open the database. This method is secure as Microsoft Office Access encrypts the password so that it cannot be accessed by reading the database file directly. However, once a database is open, all the objects contained within the database are available to the user. Figure 19.5(a) shows the dialog box to set the password and Figure 19.5(b) shows the dialog box requesting the password whenever the database is opened.

User-Level Security

User-level security in Microsoft Office Access is similar to methods used in most network systems. Users are required to identify themselves and type a password when they start Microsoft Office Access. Within the Microsoft Office Access *workgroup information file*, users are identified as members of a **group**. Access provides two default groups: administrators (*Admins* group) and users (*Users* group), but additional groups can be defined. Figure 19.6 displays the dialog box used to define the security level for user and group accounts. It shows a non-default group called Assistants, and a user called Assistant who is a member of the Users and Assistants groups.

Permissions are granted to groups and users to regulate how they are allowed to work with each object in the database using the User and Group Permissions dialog box. Table 19.2 shows the permissions that can be set in Microsoft Office Access. For example, Figure 19.7 shows the dialog box for a user called Assistant who has only read access to

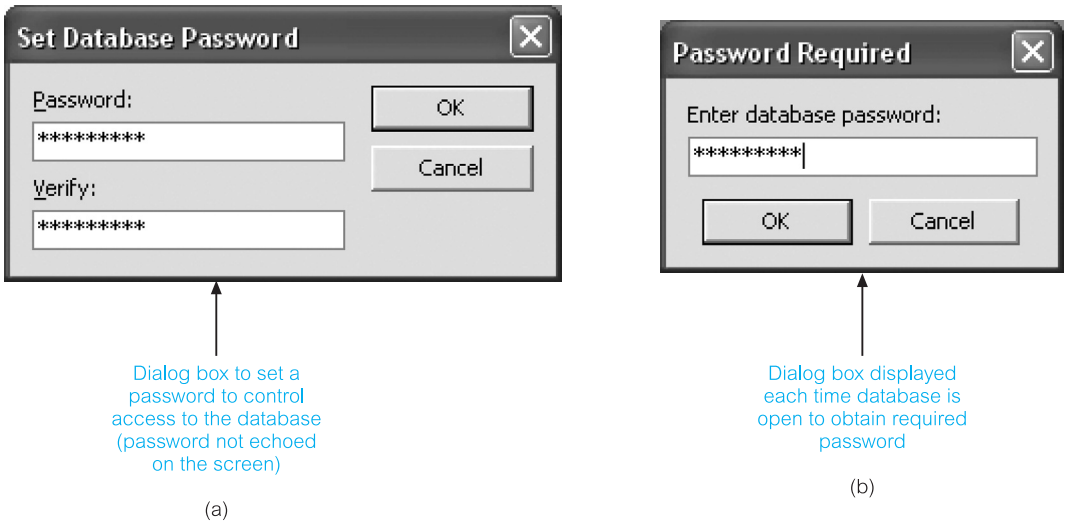


Figure 19.5 Securing the *DreamHome* database using a password: (a) the Set Database Password dialog box; (b) the Password Required dialog box shown at startup.

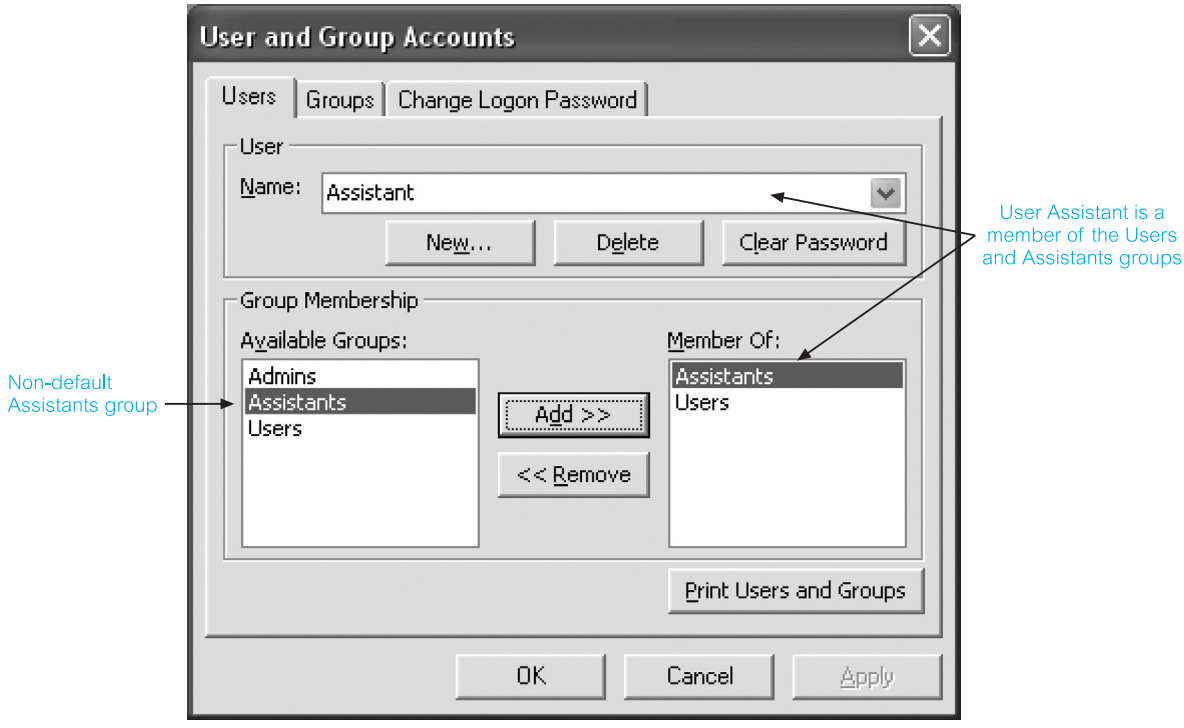


Figure 19.6 The User and Group Accounts dialog box for the *DreamHome* database.

Table 19.2 Microsoft Office Access permissions.

Permission	Description
Open/Run	Open a database, form, report, or run a macro
Open Exclusive	Open a database with exclusive access
Read Design	View objects in Design view
Modify Design	View and change database objects, and delete them
Administer	For databases, set database password, replicate database, and change startup properties
	Full access to database objects including ability to assign permissions
Read Data	View data
Update Data	View and modify data (but not insert or delete data)
Insert Data	View and insert data (but not update or delete data)
Delete Data	View and delete data (but not insert or update data)

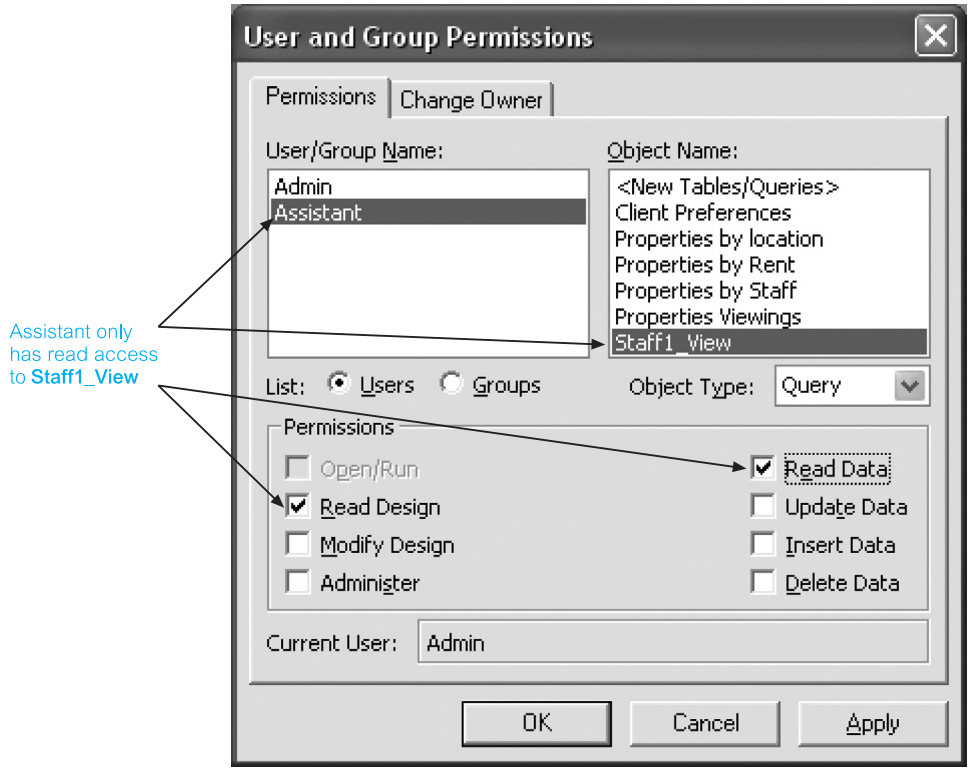


Figure 19.7 User and Group Permissions dialog box showing the Assistant user has only read access to the Staff1_View query.

a stored query called Staff1_View. In a similar way, all access to the base table Staff would be removed so that the Assistant user could only view the data in the Staff table using this view.

19.4 Security in Oracle DBMS

In Section 8.2 we provided an overview of Oracle9*i* DBMS. In this section, we focus on the security measures provided by Oracle. In the previous section we examined two types of security in Microsoft Office Access: system security and data security. In this section we examine how Oracle provides these two types of security. As with Office Access, one form of system security used by Oracle is the standard user name and password mechanism, whereby a user has to provide a valid user name and password before access can be gained to the database, although the responsibility to authenticate users can be devolved to the operating system. Figure 19.8 illustrates the creation of a new user called Beech with password authentication set. Whenever user Beech tries to connect to the database, this user will be presented with a Connect or Log On dialog box similar to the

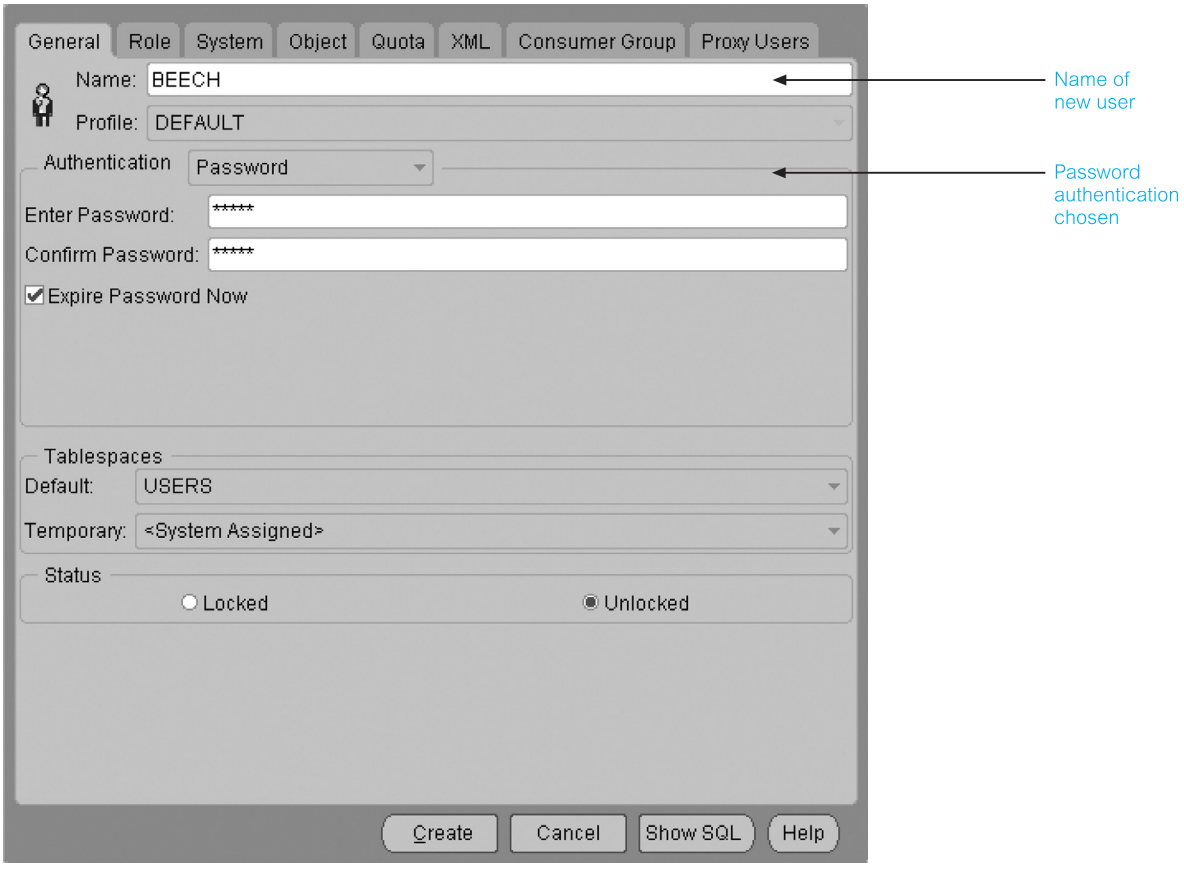


Figure 19.8 Creation of a new user called Beech with password authentication set.

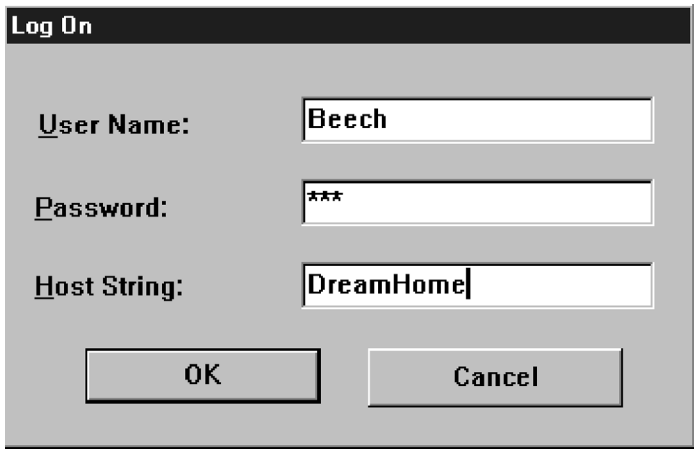


Figure 19.9
Log On dialog box requesting user name, password, and the name of the database the user wishes to connect to.

one illustrated in Figure 19.9, prompting for a user name and password to access the specified database.

Privileges

As we discussed in Section 19.2.2, a **privilege** is a right to execute a particular type of SQL statement or to access another user's objects. Some examples of Oracle privileges include the right to:

- connect to the database (create a session);
- create a table;
- select rows from another user's table.

In Oracle, there are two distinct categories of privileges:

- system privileges;
- object privileges.

System privileges

A **system privilege** is the right to perform a particular action or to perform an action on any schema objects of a particular type. For example, the privileges to create tablespaces and to create users in a database are system privileges. There are over eighty distinct system privileges in Oracle. System privileges are granted to, or revoked from, users and **roles** (discussed below) using either of the following:

- Grant System Privileges/Roles dialog box and Revoke System Privileges/Roles dialog box of the Oracle Security Manager;
- SQL GRANT and REVOKE statements (see Section 6.6).

However, only users who are granted a specific system privilege with the ADMIN OPTION or users with the GRANT ANY PRIVILEGE system privilege can grant or revoke system privileges.

Object privileges

An **object privilege** is a privilege or right to perform a particular action on a specific table, view, sequence, procedure, function, or package. Different object privileges are available for different types of object. For example, the privilege to delete rows from the Staff table is an object privilege.

Some schema objects (such as clusters, indexes, and triggers) do not have associated object privileges; their use is controlled with system privileges. For example, to alter a cluster, a user must own the cluster or have the ALTER ANY CLUSTER system privilege.

A user automatically has all object privileges for schema objects contained in his or her schema. A user can grant any object privilege on any schema object he or she owns to any other user or role. If the grant includes the WITH GRANT OPTION (of the GRANT statement), the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users. The object privileges for tables and views are shown in Table 19.3.

Table 19.3 What each object privilege allows a grantee to do with tables and views.

Object privilege	Table	View
ALTER	Change the table definition with the ALTER TABLE statement.	N/A
DELETE	Remove rows from the table with the DELETE statement. Note: SELECT privilege on the table must be granted along with the DELETE privilege.	Remove rows from the view with the DELETE statement.
INDEX	Create an index on the table with the CREATE INDEX statement.	N/A
INSERT	Add new rows to the table with the INSERT statement.	Add new rows to the view with the INSERT statement.
REFERENCES	Create a constraint that refers to the table. Cannot grant this privilege to a role.	N/A
SELECT	Query the table with the SELECT statement.	Query the view with the SELECT statement.
UPDATE	Change data in the table with the UPDATE statement. Note: SELECT privilege on the table must be granted along with the UPDATE privilege.	Change data in the view with the UPDATE statement.

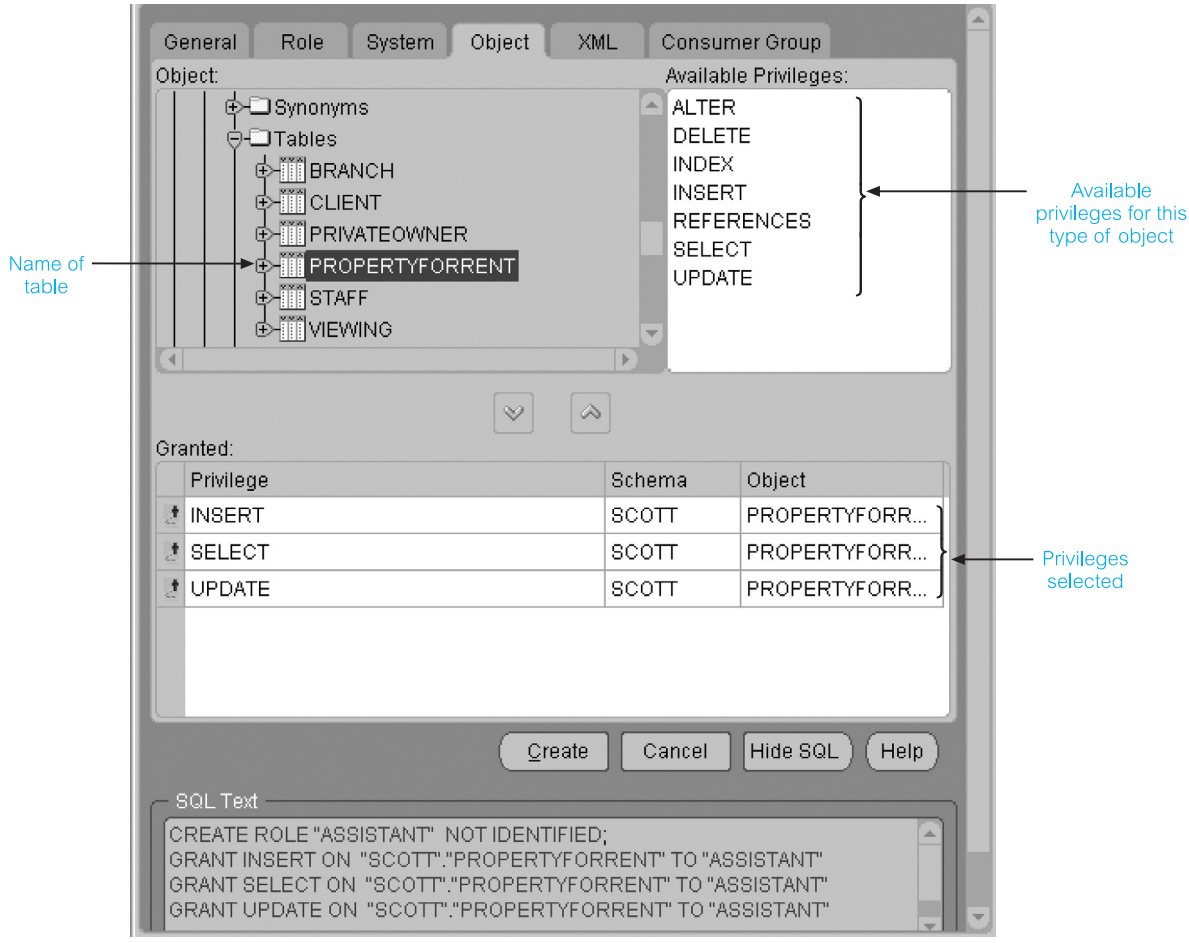


Figure 19.10
Setting the Insert, Select, and Update privileges on the PropertyForRent table to the role Assistant.

Roles

A user can receive a privilege in two different ways:

(1) Privileges can be granted to users explicitly. For example, a user can explicitly grant the privilege to insert rows into the PropertyForRent table to the user Beech:

GRANT INSERT ON PropertyForRent **TO** Beech;

(2) Privileges can also be granted to a **role** (a named group of privileges), and then the role granted to one or more users. For example, a user can grant the privileges to select, insert, and update rows from the PropertyForRent table to the role named Assistant, which in turn can be granted to the user Beech. A user can have access to several roles, and several users can be assigned the same roles. Figure 19.10 illustrates the granting of these privileges to the role Assistant using the Oracle Security Manager.

Because roles allow for easier and better management of privileges, privileges should normally be granted to roles and not to specific users.

19.5 DBMSs and Web Security

In Chapter 29 we provide a general overview of DBMSs on the Web. In this section we focus on how to make a DBMS secure on the Web. Those readers unfamilair with the terms and technologies associated with DBMSs on the Web are advised to read Chapter 29 before reading this section.

Internet communication relies on TCP/IP as the underlying protocol. However, TCP/IP and HTTP were not designed with security in mind. Without special software, all Internet traffic travels 'in the clear' and anyone who monitors traffic can read it. This form of attack is relatively easy to perpetrate using freely available 'packet sniffing' software, since the Internet has traditionally been an open network. Consider, for example, the implications of credit card numbers being intercepted by unethical parties during transmission when customers use their cards to purchase products over the Internet. The challenge is to transmit and receive information over the Internet while ensuring that:

- it is inaccessible to anyone but the sender and receiver (privacy);
- it has not been changed during transmission (integrity);
- the receiver can be sure it came from the sender (authenticity);
- the sender can be sure the receiver is genuine (non-fabrication);
- the sender cannot deny he or she sent it (non-repudiation).

However, protecting the transaction only solves part of the problem. Once the information has reached the Web server, it must also be protected there. With the three-tier architecture that is popular in a Web environment, we also have the complexity of ensuring secure access to, and of, the database. Today, most parts of such architecture can be secured, but it generally requires different products and mechanisms.

One other aspect of security that has to be addressed in the Web environment is that information transmitted to the client's machine may have executable content. For example, HTML pages may contain ActiveX controls, JavaScript/VBScript, and/or one or more Java applets. Executable content can perform the following malicious actions, and measures need to be taken to prevent them:

- corrupt data or the execution state of programs;
- reformat complete disks;
- perform a total system shutdown;
- collect and download confidential data, such as files or passwords, to another site;
- usurp identity and impersonate the user or user's computer to attack other targets on the network;
- lock up resources making them unavailable for legitimate users and programs;
- cause non-fatal but unwelcome effects, especially on output devices.

In earlier sections we identified general security mechanisms for database systems. However, the increasing accessibility of databases on the public Internet and private intranets requires a re-analysis and extension of these approaches. In this section we address some of the issues associated with database security in these environments.

Proxy Servers 19.5.1

In a Web environment, a proxy server is a computer that sits between a Web browser and a Web server. It intercepts all requests to the Web server to determine if it can fulfill the requests itself. If not, it forwards the requests to the Web server. Proxy servers have two main purposes: to improve performance and filter requests.

Improve performance

Since a proxy server saves the results of all requests for a certain amount of time, it can significantly improve performance for groups of users. For example, assume that user A and user B access the Web through a proxy server. First, user A requests a certain Web page and, slightly later, user B requests the same page. Instead of forwarding the request to the Web server where that page resides, the proxy server simply returns the cached page that it had already fetched for user A. Since the proxy server is often on the same network as the user, this is a much faster operation. Real proxy servers, such as those employed by Compuserve and America Online, can support thousands of users.

Filter requests

Proxy servers can also be used to filter requests. For example, an organization might use a proxy server to prevent its employees from accessing a specific set of Web sites.

Firewalls 19.5.2

The standard security advice is to ensure that Web servers are unconnected to any in-house networks and regularly backed up to recover from inevitable attacks. When the Web server has to be connected to an internal network, for example to access the company database, firewall technology can help to prevent unauthorized access, provided it has been installed and maintained correctly.

A **firewall** is a system designed to prevent unauthorized access to or from a private network. Firewalls can be implemented in both hardware and software, or a combination of both. They are frequently used to prevent unauthorized Internet users from accessing private networks connected to the Internet, especially intranets. All messages entering or leaving the intranet pass through the firewall, which examines each message and blocks those that do not meet the specified security criteria. There are several types of firewall technique:

- **Packet filter**, which looks at each packet entering or leaving the network and accepts or rejects it based on user-defined rules. Packet filtering is a fairly effective mechanism and transparent to users, but can be difficult to configure. In addition, it is susceptible to IP spoofing. (IP spoofing is a technique used to gain unauthorized access to computers, whereby the intruder sends messages to a computer with an IP address indicating that the message is coming from a trusted port.)

- **Application gateway**, which applies security mechanisms to specific applications, such as FTP and Telnet servers. This is a very effective mechanism, but can degrade performance.
- **Circuit-level gateway**, which applies security mechanisms when a TCP or UDP (User Datagram Protocol) connection is established. Once the connection has been made, packets can flow between the hosts without further checking.
- **Proxy server**, which intercepts all messages entering and leaving the network. The proxy server in effect hides the true network addresses.

In practice, many firewalls provide more than one of these techniques. A firewall is considered a first line of defense in protecting private information. For greater security, data can be encrypted, as discussed below and earlier in Section 19.2.6.

19.5.3 Message Digest Algorithms and Digital Signatures

A message digest algorithm, or one-way hash function, takes an arbitrarily sized string (the *message*) and generates a fixed-length string (the *digest* or *hash*). A digest has the following characteristics:

- it should be computationally infeasible to find another message that will generate the same digest;
- the digest does not reveal anything about the message.

A digital signature consists of two pieces of information: a string of bits that is computed from the data that is being 'signed', along with the private key of the individual or organization wishing the signature. The signature can be used to verify that the data comes from this individual or organization. Like a handwritten signature, a digital signature has many useful properties:

- its authenticity can be verified, using a computation based on the corresponding public key;
- it cannot be forged (assuming the private key is kept secret);
- it is a function of the data signed and cannot be claimed to be the signature for any other data;
- the signed data cannot be changed, otherwise the signature will no longer verify the data as being authentic.

Some digital signature algorithms use message digest algorithms for parts of their computations; others, for efficiency, compute the digest of a message and digitally sign the digest rather than signing the message itself.

19.5.4 Digital Certificates

A digital certificate is an attachment to an electronic message used for security purposes, most commonly to verify that a user sending a message is who he or she claims to be, and to provide the receiver with the means to encode a reply.

An individual wishing to send an encrypted message applies for a digital certificate from a Certificate Authority (CA). The CA issues an encrypted digital certificate containing the applicant's public key and a variety of other identification information. The CA makes its own public key readily available through printed material or perhaps on the Internet.

The recipient of an encrypted message uses the CA's public key to decode the digital certificate attached to the message, verifies it as issued by the CA, and then obtains the sender's public key and identification information held within the certificate. With this information, the recipient can send an encrypted reply.

Clearly, the CA's role in this process is critical, acting as a go-between for the two parties. In a large, distributed complex network like the Internet, this third-party trust model is necessary as clients and servers may not have an established mutual trust yet both parties want to have a secure session. However, because each party trusts the CA, and because the CA is vouching for each party's identification and trustworthiness by signing their certificates, each party recognizes and implicitly trusts each other. The most widely used standard for digital certificates is X.509.

Kerberos 19.5.5

Kerberos is a server of secured user names and passwords (named after the three-headed monster in Greek mythology that guarded the gate of hell). The importance of Kerberos is that it provides one centralized security server for all data and resources on the network. Database access, login, authorization control, and other security features are centralized on trusted Kerberos servers. Kerberos has a similar function to that of a Certificate server: to identify and validate a user. Security companies are currently investigating a merger of Kerberos and Certificate servers to provide a network-wide secure system.

Secure Sockets Layer and Secure HTTP 19.5.6

Many large Internet product developers agreed to use an encryption protocol known as Secure Sockets Layer (SSL) developed by Netscape for transmitting private documents over the Internet. SSL works by using a private key to encrypt data that is transferred over the SSL connection. Both Netscape Navigator and Internet Explorer support SSL, and many Web sites use this protocol to obtain confidential user information, such as credit card numbers. The protocol, layered between application-level protocols such as HTTP and the TCP/IP transport-level protocol, is designed to prevent eavesdropping, tampering, and message forgery. Since SSL is layered under application-level protocols, it may be used for other application-level protocols such as FTP and NNTP.

Another protocol for transmitting data securely over the Web is Secure HTTP (S-HTTP), a modified version of the standard HTTP protocol. S-HTTP was developed by Enterprise Integration Technologies (EIT), which was acquired by Verifone, Inc. in 1995. Whereas SSL creates a secure connection between a client and a server, over which any amount of data can be sent securely, S-HTTP is designed to transmit individual messages securely. SSL and S-HTTP, therefore, can be seen as complementary rather than competing technologies.

Both protocols have been submitted to the Internet Engineering Task Force (IETF) for approval as standards. By convention, Web pages that require an SSL connection start with **https:** instead of **http:**. Not all Web browsers and servers support SSL/S-HTTP.

Basically, these protocols allow the browser and server to authenticate one another and secure information that subsequently flows between them. Through the use of cryptographic techniques such as encryption, and digital signatures, these protocols:

- allow Web browsers and servers to authenticate each other;
- permit Web site owners to control access to particular servers, directories, files, or services;
- allow sensitive information (for example, credit card numbers) to be shared between browser and server, yet remain inaccessible to third parties;
- ensure that data exchanged between browser and server is reliable, that is, cannot be corrupted either accidentally or deliberately, without detection.

A key component in the establishment of secure Web sessions using the SSL or S-HTTP protocols is the digital certificate, discussed above. Without authentic and trustworthy certificates, protocols like SSL and S-HTTP offer no security at all.

19.5.7 Secure Electronic Transactions and Secure Transaction Technology

The Secure Electronic Transactions (SET) protocol is an open, interoperable standard for processing credit card transactions over the Internet, created jointly by Netscape, Microsoft, Visa, Mastercard, GTE, SAIC, Terisa Systems, and VeriSign. SET's goal is to allow credit card transactions to be as simple and secure on the Internet as they are in retail stores. To address privacy concerns, the transaction is split in such a way that the merchant has access to information about what is being purchased, how much it costs, and whether the payment is approved, but no information on what payment method the customer is using. Similarly, the card issuer (for example, Visa) has access to the purchase price but no information on the type of merchandise involved.

Certificates are heavily used by SET, both for certifying a cardholder and for certifying that the merchant has a relationship with the financial institution. The mechanism is illustrated in Figure 19.11. While both Microsoft and Visa International are major participants in the SET specifications, they currently provide the Secure Transaction Technology (STT) protocol, which has been designed to handle secure bank payments over the Internet. STT uses DES encryption of information, RSA encryption of bankcard information, and strong authentication of all parties involved in the transaction.

19.5.8 Java Security

In Section 29.8 we introduce the Java language as an increasingly important language for Web development. Those readers unfamiliar with Java are advised to read Section 29.8 before reading this section.

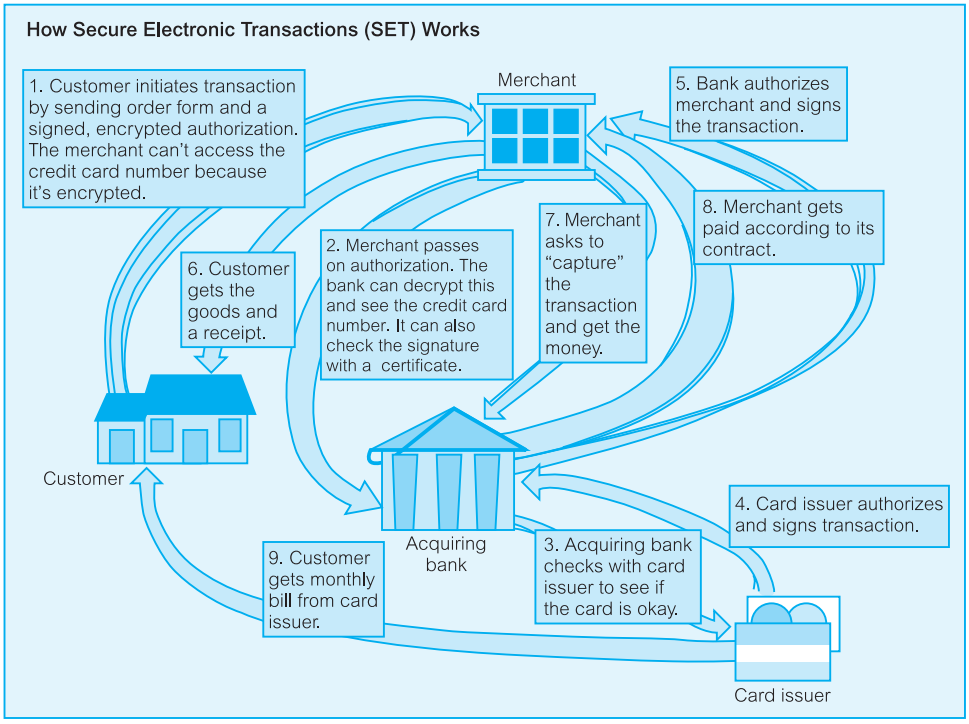


Figure 19.11
A SET transaction.

Safety and security are integral parts of Java's design, with the 'sandbox' ensuring that an untrusted, possibly malicious, application cannot gain access to system resources. To implement this sandbox, three components are used: a class loader, a bytecode verifier, and a security manager. The safety features are provided by the Java language and the Java Virtual Machine (JVM), and enforced by the compiler and the runtime system; security is a policy that is built on top of this safety layer.

Two safety features of the Java language relate to strong typing and automatic garbage collection. In this section we look at two other features: the class loader and the bytecode verifier. To complete this section on Java security, we examine the JVM Security Manager.

The class loader

The class loader, as well as loading each required class and checking it is in the correct format, additionally checks that the application/applet does not violate system security by allocating a *namespace*. Namespaces are hierarchical and allow the JVM to group classes based on where they originate (local or remote). A class loader never allows a class from a 'less protected' namespace to replace a class from a more protected namespace. In this way, the file system's I/O primitives, which are defined in a local Java class, cannot be invoked or indeed overridden by classes from outside the local machine. An executing JVM allows multiple class loaders, each with its own namespace, to be active simultaneously. As browsers and Java applications can typically provide their own class loader,

albeit based on a recommended template from Sun Microsystems, this may be viewed as a weakness in the security model. However, some argue that this is a strength of the language, allowing system administrators to implement their own (presumably tighter) security measures.

The bytecode verifier

Before the JVM will allow an application/applet to run, its code must be verified. The verifier assumes that all code is meant to crash or violate system security and performs a series of checks, including the execution of a theorem prover, to ensure that this is not the case. Typical checks include verifying that:

- compiled code is correctly formatted;
- internal stacks will not overflow/underflow;
- no 'illegal' data conversions will occur (for example, integer to pointer) – this ensures that variables will not be granted access to restricted memory areas;
- bytecode instructions are appropriately typed;
- all class member accesses are valid.

The Security Manager

The Java security policy is application specific. A Java application, such as a Java-enabled Web browser or a Web server, defines and implements its own security policy. Each of these applications implements its own Security Manager. A Java-enabled Web browser contains its own applet Security Manager, and any applets downloaded by this browser are subject to its policies. Generally, the Security Manager performs runtime verification of potentially 'dangerous' methods, that is, methods that request I/O, network access, or wish to define a new class loader. In general, downloaded applets are prevented from:

- reading and writing files on the client's file system. This also prevents applets storing persistent data (for example, a database) on the client side, although the data could be sent back to the host for storage;
- making network connections to machines other than the host that provided the compiled '.class' files. This is either the host where the HTML page came from, or the host specified in the CODEBASE parameter in the applet tag, with CODEBASE taking precedence;
- starting other programs on the client;
- loading libraries;
- defining method calls. Allowing an applet to define native method calls would give the applet direct access to the underlying operating system.

These restrictions apply to applets that are downloaded over the public Internet or company intranet. They do not apply to applets on the client's local disk and in a directory that is on the client's CLASSPATH. Local applets are loaded by the file system loader and, as well as being able to read and write files, are allowed to exit the virtual machine and are

not passed through the bytecode verifier. The JDK (Java Development Kit) Appletviewer also slightly relaxes these restrictions, by letting the user define an explicit list of files that can be accessed by downloaded applets. In a similar way, Microsoft's Internet Explorer 4.0 introduced the concept of 'zones', and some zones may be trusted and others untrusted. Java applets loaded from certain zones are able to read and write to files on the client's hard drive. The zones with which this is possible are customizable by the Network Administrators.

Enhanced applet security

The sandbox model was introduced with the first release of the Java applet API in January 1996. Although this model does generally protect systems from untrusted code obtained from the network, it does not address several other security and privacy issues. Authentication is needed to ensure that an applet comes from where it claims to have come from. Further, digitally signed and authenticated applets can then be raised to the status of trusted applets, and subsequently allowed to run with fewer security restrictions.

The Java Security API, available in JDK 1.1, contains APIs for digital signatures, message digests, key management, and encryption/decryption (subject to United States export control regulations). Work is in progress to define an infrastructure that allows flexible security policies for signed applets.

ActiveX Security 19.5.9

The ActiveX security model is considerably different from Java applets. Java achieves security by restricting the behavior of applets to a safe set of instructions. ActiveX, on the other hand, places no restrictions on what a control can do. Instead, each ActiveX control can be digitally signed by its author using a system called Authenticode™. The digital signatures are then certified by a Certificate Authority (CA). This security model places the responsibility for the computer's security on the user. Before the browser downloads an ActiveX control that has not been signed or has been certified by an unknown CA, it presents a dialog box warning the user that this action may not be safe. The user can then abort the transfer or continue and accept the consequences.

Chapter Summary

- **Database security** is the mechanisms that protect the database against intentional or accidental threats.
- Database security is concerned with avoiding the following situations: theft and fraud, loss of confidentiality (secrecy), loss of privacy, loss of integrity, and loss of availability.
- A **threat** is any situation or event, whether intentional or accidental, that will adversely affect a system and consequently an organization.
- **Computer-based security controls** for the multi-user environment include: authorization, access controls, views, backup and recovery, integrity, encryption, and RAID technology.
- **Authorization** is the granting of a right or privilege that enables a subject to have legitimate access to a system or a system's object. **Authentication** is a mechanism that determines whether a user is who he or she claims to be.
- Most commercial DBMSs provide an approach called **Discretionary Access Control** (**DAC**), which manages privileges using SQL. The SQL standard supports DAC through the GRANT and REVOKE commands. Some commercial DBMSs also provide an approach to access control called **Mandatory Access Control** (**MAC**), which is based on system-wide policies that cannot be changed by individual users. In this approach each database object is assigned a *security class* and each user is assigned a *clearance* for a security class, and *rules* are imposed on reading and writing of database objects by users. The SQL standard does not include support for MAC.
- A **view** is the dynamic result of one or more relational operations operating on the base relations to produce another relation. A view is a **virtual relation** that does not actually exist in the database but is produced upon request by a particular user at the time of request. The view mechanism provides a powerful and flexible security mechanism by hiding parts of the database from certain users.
- **Backup** is the process of periodically taking a copy of the database and log file (and possibly programs) on to offline storage media. **Journaling** is the process of keeping and maintaining a log file (or journal) of all changes made to the database to enable recovery to be undertaken effectively in the event of a failure.
- **Integrity constraints** also contribute to maintaining a secure database system by preventing data from becoming invalid, and hence giving misleading or incorrect results.
- **Encryption** is the encoding of the data by a special algorithm that renders the data unreadable by any program without the decryption key.
- **Microsoft Office Access DBMS** and **Oracle DBMS** provide two types of security measure: system security and data security. **System security** enables the setting of a password for opening a database, and **data security** provides user-level security which can be used to limit the parts of a database that a user can read and update.
- The security measures associated with **DBMSs on the Web** include: proxy servers, firewalls, message digest algorithms and digital signatures, digital certificates, kerberos, Secure Sockets Layer (SSL) and Secure HTTP (S-HTTP), Secure Electronic Transactions (SET) and Secure Transaction Technology (SST), Java security, and ActiveX security.

Review Questions

19.1 Explain the purpose and scope of database security.

19.2 List the main types of threat that could affect a database system, and for each describe the controls that you would use to counteract each of them.

19.3 Explain the following in terms of providing security for a database:
(a) authorization;
(b) access controls;
(c) views;
(d) backup and recovery;
(e) integrity;
(f) encryption;
(g) RAID technology.

19.4 Describe the security measures provided by Microsoft Office Access or Oracle DBMSs.

19.5 Describe the approaches for securing DBMSs on the Web.

Exercises

19.6 Examine any DBMS used by your organization and identify the security measures provided.

19.7 Identify the types of security approach that are used by your organization to secure any DBMSs that are accessible over the Web.

19.8 Consider the *DreamHome* case study described in Chapter 10. List the potential threats that could occur and propose countermeasures to overcome them.

19.9 Consider the *Wellmeadows Hospital* case study described in Appendix B.3. List the potential threats that could occur and propose countermeasures to overcome them.

Chapter 20 Transaction Management

Chapter Objectives

In this chapter you will learn:

- The purpose of concurrency control.
- The purpose of database recovery.
- The function and importance of transactions.
- The properties of a transaction.
- The meaning of serializability and how it applies to concurrency control.
- How locks can be used to ensure serializability.
- How the two-phase locking (2PL) protocol works.
- The meaning of deadlock and how it can be resolved.
- How timestamps can be used to ensure serializability.
- How optimistic concurrency control techniques work.
- How different levels of locking may affect concurrency.
- Some causes of database failure.
- The purpose of the transaction log file.
- The purpose of checkpoints during transaction logging.
- How to recover following database failure.
- Alternative models for long duration transactions.
- How Oracle handles concurrency control and recovery.

In Chapter 2 we discussed the functions that a Database Management System (DBMS) should provide. Among these are three closely related functions that are intended to ensure that the database is reliable and remains in a consistent state, namely transaction support, concurrency control services, and recovery services. This reliability and consistency must be maintained in the presence of failures of both hardware and software components, and when multiple users are accessing the database. In this chapter we concentrate on these three functions.

Although each function can be discussed separately, they are mutually dependent. Both concurrency control and recovery are required to protect the database from data

inconsistencies and data loss. Many DBMSs allow users to undertake simultaneous operations on the database. If these operations are not controlled, the accesses may interfere with one another and the database can become inconsistent. To overcome this, the DBMS implements a **concurrency control** protocol that prevents database accesses from interfering with one another.

Database recovery is the process of restoring the database to a correct state following a failure. The failure may be the result of a system crash due to hardware or software errors, a media failure, such as a head crash, or a software error in the application, such as a logical error in the program that is accessing the database. It may also be the result of unintentional or intentional corruption or destruction of data or facilities by system administrators or users. Whatever the underlying cause of the failure, the DBMS must be able to recover from the failure and restore the database to a consistent state.

Structure of this Chapter

Central to an understanding of both concurrency control and recovery is the notion of a **transaction**, which we describe in Section 20.1. In Section 20.2 we discuss concurrency control and examine the protocols that can be used to prevent conflict. In Section 20.3 we discuss database recovery and examine the techniques that can be used to ensure the database remains in a consistent state in the presence of failures. In Section 20.4 we examine more advanced transaction models that have been proposed for transactions that are of a long duration (from hours to possibly even months) and have uncertain developments, so that some actions cannot be foreseen at the beginning. In Section 20.5 we examine how Oracle handles concurrency control and recovery.

In this chapter we consider transaction support, concurrency control, and recovery for a centralized DBMS, that is a DBMS that consists of a single database. Later, in Chapter 23, we consider these services for a distributed DBMS, that is a DBMS that consists of multiple logically related databases distributed across a network.

20.1 Transaction Support

Transaction An action, or series of actions, carried out by a single user or application program, which reads or updates the contents of the database.

A transaction is a **logical unit of work** on the database. It may be an entire program, a part of a program, or a single command (for example, the SQL command INSERT or UPDATE), and it may involve any number of operations on the database. In the database context, the execution of an application program can be thought of as one or more transactions with non-database processing taking place in between. To illustrate the concepts of a transaction, we examine two relations from the instance of the *DreamHome* rental database shown in Figure 3.3:

Figure 20.1 Example transactions.

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, salary)
```

(a)

```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)

Staff (staffNo, fName, lName, position, sex, DOB, salary, branchNo)
PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)

A simple transaction against this database is to update the salary of a particular member of staff given the staff number, *x*. At a high level, we could write this transaction as shown in Figure 20.1(a). In this chapter we denote a database read or write operation on a data item *x* as read(x) or write(x). Additional qualifiers may be added as necessary; for example, in Figure 20.1(a), we have used the notation read(staffNo = *x*, salary) to indicate that we want to read the data item salary for the tuple with primary key value *x*. In this example, we have a **transaction** consisting of two database operations (read and write) and a non-database operation (salary = salary*1.1).

A more complicated transaction is to delete the member of staff with a given staff number *x*, as shown in Figure 20.1(b). In this case, as well as having to delete the tuple in the Staff relation, we also need to find all the PropertyForRent tuples that this member of staff managed and reassign them to a different member of staff, newStaffNo say. If all these updates are not made, referential integrity will be lost and the database will be in an **inconsistent state**: a property will be managed by a member of staff who no longer exists in the database.

A transaction should always transform the database from one consistent state to another, although we accept that consistency may be violated while the transaction is in progress. For example, during the transaction in Figure 20.1(b), there may be some moment when one tuple of PropertyForRent contains the new newStaffNo value and another still contains the old one, *x*. However, at the end of the transaction, all necessary tuples should have the new newStaffNo value.

A transaction can have one of two outcomes. If it completes successfully, the transaction is said to have **committed** and the database reaches a new consistent state. On the other hand, if the transaction does not execute successfully, the transaction is **aborted**. If a transaction is aborted, the database must be restored to the consistent state it was in before the transaction started. Such a transaction is **rolled back** or **undone**. A committed transaction cannot be aborted. If we decide that the committed transaction was a mistake, we must perform another **compensating transaction** to reverse its effects (as we discuss in Section 20.4.2). However, an aborted transaction that is rolled back can be restarted later and, depending on the cause of the failure, may successfully execute and commit at that time.

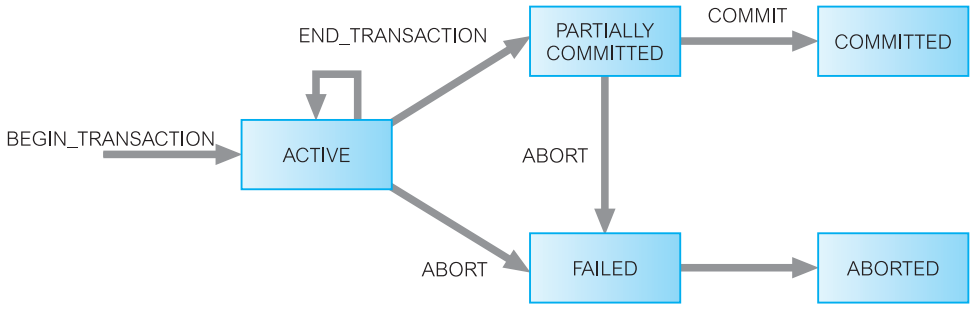


Figure 20.2 State transition diagram for a transaction.

The DBMS has no inherent way of knowing which updates are grouped together to form a single logical transaction. It must therefore provide a method to allow the user to indicate the boundaries of a transaction. The keywords BEGIN TRANSACTION, COMMIT, and ROLLBACK (or their equivalent[†]) are available in many data manipulation languages to delimit transactions. If these delimiters are not used, the entire program is usually regarded as a single transaction, with the DBMS automatically performing a COMMIT when the program terminates correctly and a ROLLBACK if it does not.

Figure 20.2 shows the state transition diagram for a transaction. Note that in addition to the obvious states of ACTIVE, COMMITTED, and ABORTED, there are two other states:

- PARTIALLY COMMITTED, which occurs after the final statement has been executed. At this point, it may be found that the transaction has violated serializability (see Section 20.2.2) or has violated an integrity constraint and the transaction has to be aborted. Alternatively, the system may fail and any data updated by the transaction may not have been safely recorded on secondary storage. In such cases, the transaction would go into the FAILED state and would have to be aborted. If the transaction has been successful, any updates can be safely recorded and the transaction can go to the COMMITTED state.
- FAILED, which occurs if the transaction cannot be committed or the transaction is aborted while in the ACTIVE state, perhaps due to the user aborting the transaction or as a result of the concurrency control protocol aborting the transaction to ensure serializability.

Properties of Transactions 20.1.1

There are properties that all transactions should possess. The four basic, or so-called **ACID**, properties of a transaction are (Haerder and Reuter, 1983):

- *Atomicity* The 'all or nothing' property. A transaction is an indivisible unit that is either performed in its entirety or is not performed at all. It is the responsibility of the recovery subsystem of the DBMS to ensure atomicity.
- *Consistency* A transaction must transform the database from one consistent state to another consistent state. It is the responsibility of both the DBMS and the application developers to ensure consistency. The DBMS can ensure consistency by enforcing all

[†] With the ISO SQL standard, BEGIN TRANSACTION is implied by the first *transaction-initiating* SQL statement (see Section 6.5).

the constraints that have been specified on the database schema, such as integrity and enterprise constraints. However, in itself this is insufficient to ensure consistency. For example, suppose we have a transaction that is intended to transfer money from one bank account to another and the programmer makes an error in the transaction logic and debits one account but credits the wrong account, then the database is in an inconsistent state. However, the DBMS would not have been responsible for introducing this inconsistency and would have had no ability to detect the error.

- *Isolation* Transactions execute independently of one another. In other words, the partial effects of incomplete transactions should not be visible to other transactions. It is the responsibility of the concurrency control subsystem to ensure isolation.
- *Durability* The effects of a successfully completed (committed) transaction are permanently recorded in the database and must not be lost because of a subsequent failure. It is the responsibility of the recovery subsystem to ensure durability.

20.1.2 Database Architecture

In Chapter 2 we presented an architecture for a DBMS. Figure 20.3 represents an extract from Figure 2.8 identifying four high-level database modules that handle transactions, concurrency control, and recovery. The **transaction manager** coordinates transactions on behalf of application programs. It communicates with the **scheduler**, the module responsible for implementing a particular strategy for concurrency control. The scheduler is sometimes referred to as the **lock manager** if the concurrency control protocol is locking-based. The objective of the scheduler is to maximize concurrency without allowing

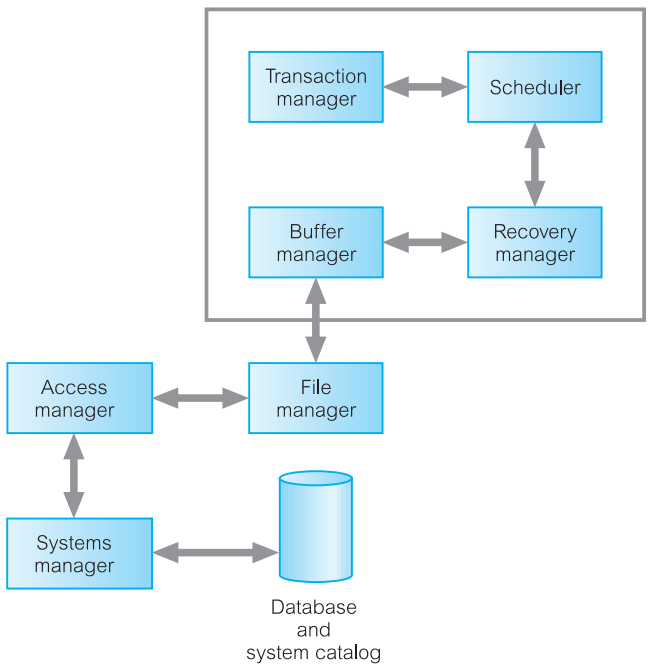


Figure 20.3
DBMS transaction subsystem.

concurrently executing transactions to interfere with one another, and so compromise the integrity or consistency of the database.

If a failure occurs during the transaction, then the database could be inconsistent. It is the task of the **recovery manager** to ensure that the database is restored to the state it was in before the start of the transaction, and therefore a consistent state. Finally, the **buffer manager** is responsible for the efficient transfer of data between disk storage and main memory.

20.2 Concurrency Control

In this section we examine the problems that can arise with concurrent access and the techniques that can be employed to avoid these problems. We start with the following working definition of concurrency control.

Concurrency control	The process of managing simultaneous operations on the database without having them interfere with one another.

20.2.1 The Need for Concurrency Control

A major objective in developing a database is to enable many users to access shared data concurrently. Concurrent access is relatively easy if all users are only reading data, as there is no way that they can interfere with one another. However, when two or more users are accessing the database simultaneously and at least one is updating data, there may be interference that can result in inconsistencies.

This objective is similar to the objective of multi-user computer systems, which allow two or more programs (or transactions) to execute at the same time. For example, many systems have input/output (I/O) subsystems that can handle I/O operations independently, while the main central processing unit (CPU) performs other operations. Such systems can allow two or more transactions to execute simultaneously. The system begins executing the first transaction until it reaches an I/O operation. While the I/O is being performed, the CPU suspends the first transaction and executes commands from the second transaction. When the second transaction reaches an I/O operation, control then returns to the first transaction and its operations are resumed from the point at which it was suspended. The first transaction continues until it again reaches another I/O operation. In this way, the operations of the two transactions are **interleaved** to achieve concurrent execution. In addition, **throughput** – the amount of work that is accomplished in a given time interval – is improved as the CPU is executing other transactions instead of being in an idle state waiting for I/O operations to complete.

However, although two transactions may be perfectly correct in themselves, the interleaving of operations in this way may produce an incorrect result, thus compromising the integrity and consistency of the database. We examine three examples of potential problems caused by concurrency: the **lost update problem**, the **uncommitted dependency problem**, and the **inconsistent analysis problem**. To illustrate these problems, we use a simple bank account relation that contains the *DreamHome* staff account balances. In this context, we are using the transaction as the *unit of concurrency control*.

Example 20.1 The lost update problem

An apparently successfully completed update operation by one user can be overridden by another user. This is known as the **lost update problem** and is illustrated in Figure 20.4, in which transaction T_1 is executing concurrently with transaction T_2. T_1 is withdrawing £10 from an account with balance bal_x, initially £100, and T_2 is depositing £100 into the same account. If these transactions are executed **serially**, one after the other with no interleaving of operations, the final balance would be £190 no matter which transaction is performed first.

Transactions T_1 and T_2 start at nearly the same time, and both read the balance as £100. T_2 increases bal_x by £100 to £200 and stores the update in the database. Meanwhile, transaction T_1 decrements its copy of bal_x by £10 to £90 and stores this value in the database, overwriting the previous update, and thereby 'losing' the £100 previously added to the balance.

The loss of T_2's update is avoided by preventing T_1 from reading the value of bal_x until after T_2's update has been completed.

Figure 20.4 The lost update problem.

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Example 20.2 The uncommitted dependency (or dirty read) problem

The uncommitted dependency problem occurs when one transaction is allowed to see the intermediate results of another transaction before it has committed. Figure 20.5 shows an example of an uncommitted dependency that causes an error, using the same initial value for balance bal_x as in the previous example. Here, transaction T_4 updates bal_x to £200,

Figure 20.5 The uncommitted dependency problem.

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

but it aborts the transaction so that bal_x should be restored to its original value of £100. However, by this time transaction T_3 has read the new value of bal_x (£200) and is using this value as the basis of the £10 reduction, giving a new incorrect balance of £190, instead of £90. The value of bal_x read by T_3 is called *dirty data*, giving rise to the alternative name, *the dirty read problem*.

The reason for the rollback is unimportant; it may be that the transaction was in error, perhaps crediting the wrong account. The effect is the assumption by T_3 that T_4's update completed successfully, although the update was subsequently rolled back. This problem is avoided by preventing T_3 from reading bal_x until after the decision has been made to either commit or abort T_4's effects.

The two problems in these examples concentrate on transactions that are updating the database and their interference may corrupt the database. However, transactions that only read the database can also produce inaccurate results if they are allowed to read partial results of incomplete transactions that are simultaneously updating the database. We illustrate this with the next example.

Example 20.3 The inconsistent analysis problem

The problem of inconsistent analysis occurs when a transaction reads several values from the database but a second transaction updates some of them during the execution of the first. For example, a transaction that is summarizing data in a database (for example, totaling balances) will obtain inaccurate results if, while it is executing, other transactions are updating the database. One example is illustrated in Figure 20.6, in which a summary transaction T_6 is executing concurrently with transaction T_5. Transaction T_6 is totaling the balances of account x (£100), account y (£50), and account z (£25). However, in the meantime, transaction T_5 has transferred £10 from bal_x to bal_z, so that T_6 now has the wrong result (£10 too high). This problem is avoided by preventing transaction T_6 from reading bal_x and bal_z until after T_5 has completed its updates.

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

Figure 20.6
The inconsistent analysis problem.

Another problem can occur when a transaction T rereads a data item it has previously read but, in between, another transaction has modified it. Thus, T receives two different values for the same data item. This is sometimes referred to as a **nonrepeatable** (or **fuzzy**) **read**. A similar problem can occur if transaction T executes a query that retrieves a set of tuples from a relation satisfying a certain predicate, re-executes the query at a later time but finds that the retrieved set contains an additional (**phantom**) tuple that has been inserted by another transaction in the meantime. This is sometimes referred to as a **phantom read**.

20.2.2 Serializability and Recoverability

The objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference between them, and hence prevent the types of problem described in the previous section. One obvious solution is to allow only one transaction to execute at a time: one transaction is *committed* before the next transaction is allowed to *begin*. However, the aim of a multi-user DBMS is also to maximize the degree of concurrency or parallelism in the system, so that transactions that can execute without interfering with one another can run in parallel. For example, transactions that access different parts of the database can be scheduled together without interference. In this section, we examine serializability as a means of helping to identify those executions of transactions that are *guaranteed* to ensure consistency (Papadimitriou, 1979). First, we give some definitions.

Schedule A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.

A transaction comprises a sequence of operations consisting of read and/or write actions to the database, followed by a commit or abort action. A schedule S consists of a sequence of the operations from a set of n transactions $T_1, T_2, \ldots, T_n$, subject to the constraint that the order of operations for each transaction is preserved in the schedule. Thus, for each transaction T_i in schedule S, the order of the operations in T_i must be the same in schedule S.

Serial schedule A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.

In a serial schedule, the transactions are performed in serial order. For example, if we have two transactions T_1 and T_2, serial order would be T_1 followed by T_2, or T_2 followed by T_1. Thus, in serial execution there is no interference between transactions, since only one is executing at any given time. However, there is no guarantee that the results of all serial executions of a given set of transactions will be identical. In banking, for example, it matters whether interest is calculated on an account before a large deposit is made or after.

Nonserial schedule A schedule where the operations from a set of concurrent transactions are interleaved.

The problems described in Examples 20.1–20.3 resulted from the mismanagement of concurrency, which left the database in an inconsistent state in the first two examples and presented the user with the wrong result in the third. Serial execution prevents such problems occurring. No matter which serial schedule is chosen, serial execution never leaves the database in an inconsistent state, so every serial execution is considered correct, although different results may be produced. The objective of **serializability** is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.

If a set of transactions executes concurrently, we say that the (nonserial) schedule is correct if it *produces the same results as some serial execution*. Such a schedule is called **serializable**. To prevent inconsistency from transactions interfering with one another, it is essential to guarantee serializability of concurrent transactions. In serializability, the ordering of read and write operations is important:

- If two transactions only read a data item, they do not conflict and order is not important.
- If two transactions either read or write completely separate data items, they do not conflict and order is not important.
- If one transaction writes a data item and another either reads or writes the same data item, the order of execution is important.

Consider the schedule S_1 shown in Figure 20.7(a) containing operations from two concurrently executing transactions T_7 and T_8. Since the write operation on bal_x in T_8 does not conflict with the subsequent read operation on bal_y in T_7, we can change the order of these operations to produce the equivalent schedule S_2 shown in Figure 20.7(b). If we also now change the order of the following non-conflicting operations, we produce the equivalent serial schedule S_3 shown in Figure 20.7(c):

- Change the order of the write(bal_x) of T_8 with the write(bal_y) of T_7.
- Change the order of the read(bal_x) of T_8 with the read(bal_y) of T_7.
- Change the order of the read(bal_x) of T_8 with the write(bal_y) of T_7.

Figure 20.7 Equivalent schedules: (a) nonserial schedule S_1; (b) nonserial schedule S_2 equivalent to S_1; (c) serial schedule S_3, equivalent to S_1 and S_2.

Time	T_7	T_8
t_1	begin_transaction	
t_2	read($\mathbf{bal_x}$)	
t_3	write($\mathbf{bal_x}$)	
t_4		begin_transaction
t_5		read($\mathbf{bal_x}$)
t_6		write($\mathbf{bal_x}$)
t_7	read($\mathbf{bal_y}$)	
t_8	write($\mathbf{bal_y}$)	
t_9	commit	
t_{10}		read($\mathbf{bal_y}$)
t_{11}		write($\mathbf{bal_y}$)
t_{12}		commit

(a)

T_7	T_8
begin_transaction	
read($\mathbf{bal_x}$)	
write($\mathbf{bal_x}$)	
	begin_transaction
	read($\mathbf{bal_x}$)
read($\mathbf{bal_y}$)	
	write($\mathbf{bal_x}$)
write($\mathbf{bal_y}$)	
commit	
	read($\mathbf{bal_y}$)
	write($\mathbf{bal_y}$)
	commit

(b)

T_7	T_8
begin_transaction	
read($\mathbf{bal_x}$)	
write($\mathbf{bal_x}$)	
read($\mathbf{bal_y}$)	
write($\mathbf{bal_y}$)	
commit	
	begin_transaction
	read($\mathbf{bal_x}$)
	write($\mathbf{bal_x}$)
	read($\mathbf{bal_y}$)
	write($\mathbf{bal_y}$)
	commit

(c)

Schedule S_3 is a serial schedule and, since S_1 and S_2 are equivalent to S_3, S_1 and S_2 are serializable schedules.

This type of serializability is known as **conflict serializability**. A conflict serializable schedule orders any conflicting operations in the same way as some serial execution.

Testing for conflict serializability

Under the **constrained write rule** (that is, a transaction updates a data item based on its old value, which is first read by the transaction), a **precedence** (or **serialization**) **graph** can be produced to test for conflict serializability. For a schedule S, a precedence graph is a directed graph G = (N, E) that consists of a set of nodes N and a set of directed edges E, which is constructed as follows:

- Create a node for each transaction.
- Create a directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_i.
- Create a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.
- Create a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been written by T_i.

If an edge $T_i \rightarrow T_j$ exists in the precedence graph for S, then in any serial schedule S′ equivalent to S, T_i must appear before T_j. If the precedence graph contains a cycle the schedule is not conflict serializable.

Example 20.4 Non-conflict serializable schedule

Consider the two transactions shown in Figure 20.8. Transaction T_9 is transferring £100 from one account with balance bal_x to another account with balance bal_y, while T_{10} is increasing the balance of these two accounts by 10%. The precedence graph for this schedule, shown in Figure 20.9, has a cycle and so is not conflict serializable.

Figure 20.8
Two concurrent update transactions that are not conflict serializable.

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read($\mathbf{bal_x}$)	
t_3	$\mathbf{bal_x} = \mathbf{bal_x} + 100$	
t_4	write($\mathbf{bal_x}$)	begin_transaction
t_5		read($\mathbf{bal_x}$)
t_6		$\mathbf{bal_x} = \mathbf{bal_x} * 1.1$
t_7		write($\mathbf{bal_x}$)
t_8		read($\mathbf{bal_y}$)
t_9		$\mathbf{bal_y} = \mathbf{bal_y} * 1.1$
t_{10}		write($\mathbf{bal_y}$)
t_{11}	read($\mathbf{bal_y}$)	commit
t_{12}	$\mathbf{bal_y} = \mathbf{bal_y} - 100$	
t_{13}	write($\mathbf{bal_y}$)	
t_{14}	commit	

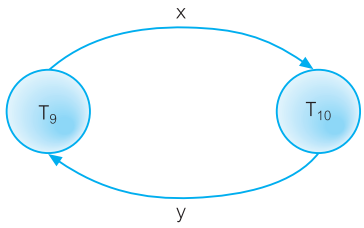


Figure 20.9
Precedence graph for Figure 20.8 showing a cycle, so schedule is not conflict serializable.

View serializability

There are several other types of serializability that offer less stringent definitions of schedule equivalence than that offered by conflict serializability. One less restrictive definition is called **view serializability**. Two schedules S_1 and S_2 consisting of the same operations from n transactions $T_1, T_2, \ldots, T_n$ are view equivalent if the following three conditions hold:

- For each data item x, if transaction T_i reads the initial value of x in schedule S_1, then transaction T_i must also read the initial value of x in schedule S_2.
- For each read operation on data item x by transaction T_i in schedule S_1, if the value read by x has been written by transaction T_j, then transaction T_i must also read the value of x produced by transaction T_j in schedule S_2.
- For each data item x, if the last write operation on x was performed by transaction T_i in schedule S_1, the same transaction must perform the final write on data item x in schedule S_2.

A schedule is view serializable if it is view equivalent to a serial schedule. Every conflict serializable schedule is view serializable, although the converse is not true. For example, the schedule shown in Figure 20.10 is view serializable, although it is not conflict serializable. In this example, transactions T_{12} and T_{13} do not conform to the constrained write rule; in other words, they perform *blind writes*. It can be shown that any view serializable schedule that is not conflict serializable contains one or more blind writes.

Time	T_{11}	T_{12}	T_{13}
t_1	begin_transaction		
t_2	read($\mathbf{bal_x}$)		
t_3		begin_transaction	
t_4		write($\mathbf{bal_x}$)	
t_5		commit	
t_6	write($\mathbf{bal_x}$)		
t_7	commit		
t_8			begin_transaction
t_9			write($\mathbf{bal_x}$)
t_{10}			commit

Figure 20.10
View serializable schedule that is not conflict serializable.

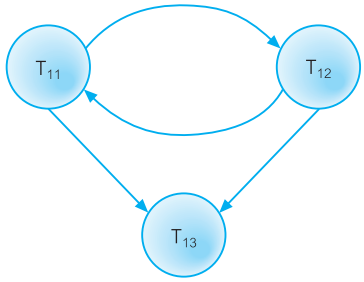


Figure 20.11
Precedence graph for the view serializable schedule of Figure 20.10.

Testing for view serializability

Testing for view serializability is much more complex than testing for conflict serializability. In fact, it has been shown that testing for view serializability is NP-complete, thus it is highly improbable that an efficient algorithm can be found (Papadimitriou, 1979). If we produce a (conflict serializable) precedence graph corresponding to the schedule given in Figure 20.10 we would get the graph shown in Figure 20.11. This graph contains a cycle indicating that the schedule is not conflict serializable, however, we know that it is view serializable since it is equivalent to the serial schedule T_{11} followed by T_{12} followed by T_{13}. When we examine the rules for the precedence graph given above, we can see that the edge $T_{12} \rightarrow T_{11}$ should not have been inserted into the graph as the values of bal_x written by T_{11} and T_{12} were never used by any other transaction because of the blind writes.

As a result, to test for view serializability we need a method to decide whether an edge should be inserted into the precedence graph. The approach we take is to construct a *labeled precedence graph* for the schedule as follows:

(1) Create a node for each transaction.

(2) Create a node labeled T_{bw}. T_{bw} is a dummy transaction inserted at the beginning of the schedule containing a write operation for each data item accessed in the schedule.

(3) Create a node labeled T_{fr}. T_{fr} is a dummy transaction added at the end of the schedule containing a read operation for each data item accessed in the schedule.

(4) Create a directed edge $T_i \xrightarrow{0} T_j$, if T_j reads the value of an item written by T_i.

(5) Remove all directed edges incident on transaction T_i for which there is no path from T_i to T_{fr}.

(6) For each data item that T_j reads, which has been written by T_i, and T_k writes ($T_k \neq T_{bw}$) then:

(a) If $T_i = T_{bw}$ and $T_j \neq T_{fr}$, then create a directed edge $T_j \xrightarrow{0} T_k$.

(b) If $T_i \neq T_{bw}$ and $T_j = T_{fr}$, then create a directed edge $T_k \xrightarrow{0} T_i$.

(c) If $T_i \neq T_{bw}$ and $T_j \neq T_{fr}$, then create a pair of directed edges $T_k \xrightarrow{x} T_i$ and $T_j \xrightarrow{x} T_k$, where x is a unique positive integer that has not been used for labeling an earlier directed edge. This rule is a more general case of the preceding two rules indicating that if transaction T_i writes an item that T_j subsequently reads then any transaction, T_k, that writes the same item must *either* precede T_i or succeed T_j.

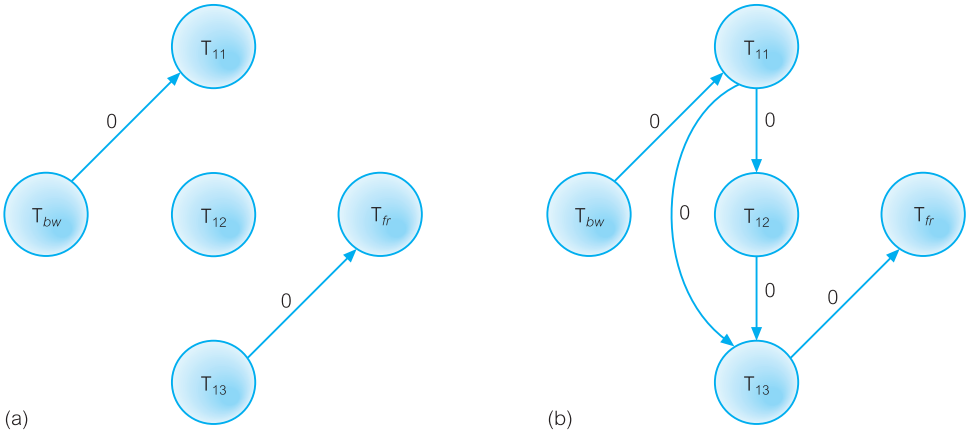


Figure 20.12
Labeled precedence graph for the view serializable schedule of Figure 20.10.

Applying the first five rules to the schedule in Figure 20.10 produces the precedence graph shown in Figure 20.12(a). Applying rule 6(a), we add the edges $T_{11} \rightarrow T_{12}$ and $T_{11} \rightarrow T_{13}$, both labeled 0; applying rule 6(b), we add the edges $T_{11} \rightarrow T_{13}$ (which is already present) and $T_{12} \rightarrow T_{13}$, again both labeled 0. The final graph is shown in Figure 20.12(b).

Based on this labeled precedence graph, the test for view serializability is as follows:

(1) If the graph contains no cycles, the schedule is view serializable.

(2) The presence of a cycle, however, is not a sufficient condition to conclude that the schedule is not view serializable. The actual test is based on the observation that rule 6(c) generates m distinct directed edge pairs, resulting in 2^m different graphs containing just one edge from each pair. If *any* one of these graphs is acyclic, then the corresponding schedule is view serializable and the serializability order is determined by the topological sorting of the graph with the dummy transactions T_{bw} and T_{fr} removed.

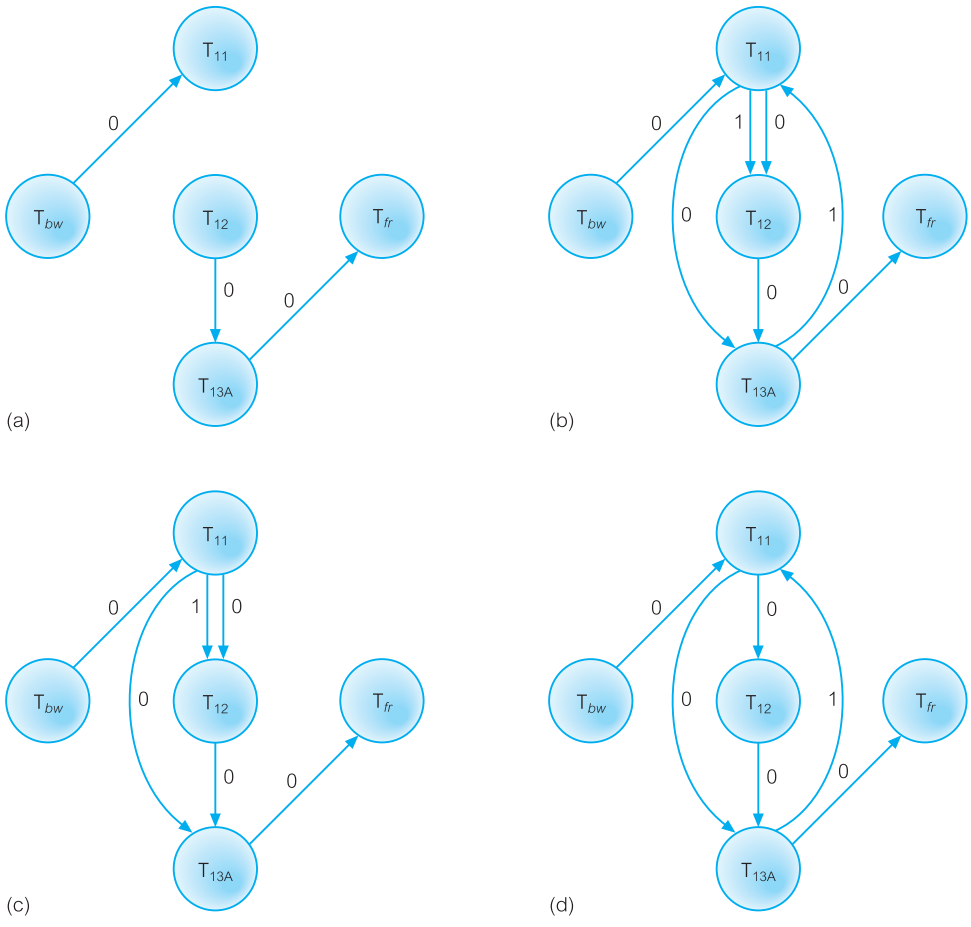
Applying these tests to the graph in Figure 20.12(b), which is acyclic, we conclude that the schedule is view serializable. As another example, consider the slightly modified variant of the schedule of Figure 20.10 shown in Figure 20.13 containing an additional read operation in transaction T_{13A}. Applying the first five rules to this schedule produces the precedence graph shown in Figure 20.14(a). Applying rule 6(a), we add the edges $T_{11} \rightarrow T_{12}$ and $T_{11} \rightarrow T_{13A}$, both labeled 0; applying rule 6(b), we add the edges $T_{11} \rightarrow T_{13A}$ (which is already present) and $T_{12} \rightarrow T_{13A}$ (again already present), both labeled 0. Applying rule 6(c), we add the pair of edges $T_{11} \rightarrow T_{12}$ and $T_{13A} \rightarrow T_{11}$, this time both labeled 1. The final graph is shown in Figure 20.14(b). From this we can produce two different graphs containing only one edge, as shown in Figure 20.14(c) and 20.14(d). As Figure 20.14(c) is acyclic, we can conclude that this schedule is also view serializable (corresponding to the serial schedule $T_{11} \rightarrow T_{12} \rightarrow T_{13A}$).

In practice, a DBMS does not test for the serializability of a schedule. This would be impractical, as the interleaving of operations from concurrent transactions is determined by the operating system. Instead, the approach taken is to use protocols that are known to produce serializable schedules. We discuss such protocols in the next section.

Figure 20.13
A modified version of the schedule in Figure 20.10 containing an additional read operation.

Time	T_{11}	T_{12}	T_{13A}
t_1	begin_transaction		
t_2	read(bal_x)		
t_3		begin_transaction	
t_4		write(bal_x)	
t_5		commit	
t_6			begin_transaction
t_7			read(bal_x)
t_8	write(bal_x)		
t_9	commit		
t_{10}			write(bal_x)
t_{11}			commit

Figure 20.14
Labeled precedence graph for another view serializable schedule in Figure 20.13.



Recoverability

Serializability identifies schedules that maintain the consistency of the database, assuming that none of the transactions in the schedule fails. An alternative perspective examines the *recoverability* of transactions within a schedule. If a transaction fails, the atomicity property requires that we undo the effects of the transaction. In addition, the durability property states that once a transaction commits, its changes cannot be undone (without running another, compensating, transaction). Consider again the two transactions shown in Figure 20.8 but instead of the commit operation at the end of transaction T_9, assume that T_9 decides to roll back the effects of the transaction. T_{10} has read the update to bal_x performed by T_9, and has itself updated bal_x and committed the change. Strictly speaking, we should undo transaction T_{10} because it has used a value for bal_x that has been undone. However, the durability property does not allow this. In other words, this schedule is a *nonrecoverable schedule*, which should not be allowed. This leads to the definition of a recoverable schedule.

Recoverable schedule A schedule where, for each pair of transactions T_i and T_j, if T_j reads a data item previously written by T_i, then the commit operation of T_i precedes the commit operation of T_j.

Concurrency control techniques

Serializability can be achieved in several ways. There are two main concurrency control techniques that allow transactions to execute safely in parallel subject to certain constraints: locking and timestamp methods.

Locking and timestamping are essentially **conservative** (or **pessimistic**) approaches in that they cause transactions to be delayed in case they conflict with other transactions at some time in the future. **Optimistic** methods, as we see later, are based on the premise that conflict is rare so they allow transactions to proceed unsynchronized and only check for conflicts at the end, when a transaction commits. We discuss locking, timestamping, and optimistic concurrency control techniques in the following sections.

Locking Methods 20.2.3

Locking A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results.

Locking methods are the most widely used approach to ensure serializability of concurrent transactions. There are several variations, but all share the same fundamental characteristic, namely that a transaction must claim a **shared** (*read*) or **exclusive** (*write*) lock on a data item before the corresponding database read or write operation. The **lock** prevents another transaction from modifying the item or even reading it, in the case of an exclusive lock. Data items of various sizes, ranging from the entire database down to a field, may

be locked. The size of the item determines the fineness, or **granularity**, of the lock. The actual lock might be implemented by setting a bit in the data item to indicate that portion of the database is locked, or by keeping a list of locked parts of the database, or by other means. We examine lock granularity further in Section 20.2.8. In the meantime, we continue to use the term 'data item' to refer to the lock granularity. The basic rules for locking are set out in the following box.

Shared lock	If a transaction has a shared lock on a data item, it can read the item but not update it.
Exclusive lock	If a transaction has an exclusive lock on a data item, it can both read and update the item.

Since read operations cannot conflict, it is permissible for more than one transaction to hold shared locks simultaneously on the same item. On the other hand, an exclusive lock gives a transaction exclusive access to that item. Thus, as long as a transaction holds the exclusive lock on the item, no other transactions can read or update that data item. Locks are used in the following way:

- Any transaction that needs to access a data item must first lock the item, requesting a shared lock for read only access or an exclusive lock for both read and write access.
- If the item is not already locked by another transaction, the lock will be granted.
- If the item is currently locked, the DBMS determines whether the request is compatible with the existing lock. If a shared lock is requested on an item that already has a shared lock on it, the request will be granted; otherwise, the transaction must **wait** until the existing lock is released.
- A transaction continues to hold a lock until it explicitly releases it either during execution or when it terminates (aborts or commits). It is only when the exclusive lock has been released that the effects of the write operation will be made visible to other transactions.

In addition to these rules, some systems permit a transaction to issue a shared lock on an item and then later to **upgrade** the lock to an exclusive lock. This in effect allows a transaction to examine the data first and then decide whether it wishes to update it. If upgrading is not supported, a transaction must hold exclusive locks on all data items that it may update at some time during the execution of the transaction, thereby potentially reducing the level of concurrency in the system. For the same reason, some systems also permit a transaction to issue an exclusive lock and then later to **downgrade** the lock to a shared lock.

Using locks in transactions, as described above, does not guarantee serializability of schedules by themselves, as Example 20.5 shows.

Example 20.5 Incorrect locking schedule

Consider again the two transactions shown in Figure 20.8. A valid schedule that may be employed using the above locking rules is:

$S = \{write\_lock(T_9, bal_x), read(T_9, bal_x), write(T_9, bal_x), unlock(T_9, bal_x),$
$write\_lock(T_{10}, bal_x), read(T_{10}, bal_x), write(T_{10}, bal_x), unlock(T_{10}, bal_x),$
$write\_lock(T_{10}, bal_y), read(T_{10}, bal_y), write(T_{10}, bal_y), unlock(T_{10}, bal_y),$
$commit(T_{10}), write\_lock(T_9, bal_y), read(T_9, bal_y), write(T_9, bal_y),$
$unlock(T_9, bal_y), commit(T_9)\}$

If, prior to execution, $bal_x = 100$, $bal_y = 400$, the result should be $bal_x = 220$, $bal_y = 330$, if T_9 executes before T_{10}, or $bal_x = 210$ and $bal_y = 340$, if T_{10} executes before T_9. However, the result of executing schedule S would give $bal_x = 220$ and $bal_y = 340$. (S is **not** a serializable schedule.)

The problem in this example is that the schedule releases the locks that are held by a transaction as soon as the associated read/write is executed and that lock item (say bal_x) no longer needs to be accessed. However, the transaction itself is locking other items (bal_y), after it releases its lock on bal_x. Although this may seem to allow greater concurrency, it permits transactions to interfere with one another, resulting in the loss of total isolation and atomicity.

To guarantee serializability, we must follow an additional protocol concerning the positioning of the lock and unlock operations in every transaction. The best-known protocol is **two-phase locking** (**2PL**).

Two-phase locking (2PL)

2PL A transaction follows the two-phase locking protocol if all locking operations precede the first unlock operation in the transaction.

According to the rules of this protocol, every transaction can be divided into two phases: first a **growing phase**, in which it acquires all the locks needed but cannot release any locks, and then a **shrinking phase**, in which it releases its locks but cannot acquire any new locks. There is no requirement that all locks be obtained simultaneously. Normally, the transaction acquires some locks, does some processing, and goes on to acquire additional locks as needed. However, it never releases any lock until it has reached a stage where no new locks are needed. The rules are:

- A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.
- Once the transaction releases a lock, it can never acquire any new locks.

If upgrading of locks is allowed, upgrading can take place only during the growing phase and may require that the transaction wait until another transaction releases a shared lock on the item. Downgrading can take place only during the shrinking phase. We now look at how two-phase locking is used to resolve the three problems identified in Section 20.2.1.

Example 20.6 Preventing the lost update problem using 2PL

A solution to the lost update problem is shown in Figure 20.15. To prevent the lost update problem occurring, T_2 first requests an exclusive lock on bal_x. It can then proceed to read the value of bal_x from the database, increment it by £100, and write the new value back to the database. When T_1 starts, it also requests an exclusive lock on bal_x. However, because the data item bal_x is currently exclusively locked by T_2, the request is not immediately granted and T_1 has to **wait** until the lock is released by T_2. This occurs only once the commit of T_2 has been completed.

Figure 20.15 Preventing the lost update problem.

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Example 20.7 Preventing the uncommitted dependency problem using 2PL

A solution to the uncommitted dependency problem is shown in Figure 20.16. To prevent this problem occurring, T_4 first requests an exclusive lock on bal_x. It can then proceed to read the value of bal_x from the database, increment it by £100, and write the new value

Figure 20.16 Preventing the uncommitted dependency problem.

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	**rollback**/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

back to the database. When the rollback is executed, the updates of transaction T_4 are undone and the value of bal_x in the database is returned to its original value of £100. When T_3 starts, it also requests an exclusive lock on bal_x. However, because the data item bal_x is currently exclusively locked by T_4, the request is not immediately granted and T_3 has to wait until the lock is released by T_4. This occurs only once the rollback of T_4 has been completed.

Example 20.8 Preventing the inconsistent analysis problem using 2PL

A solution to the inconsistent analysis problem is shown in Figure 20.17. To prevent this problem occurring, T_5 must precede its reads by exclusive locks, and T_6 must precede its reads with shared locks. Therefore, when T_5 starts it requests and obtains an exclusive lock on bal_x. Now, when T_6 tries to share lock bal_x the request is not immediately granted and T_6 has to wait until the lock is released, which is when T_5 commits.

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

Figure 20.17 Preventing the inconsistent analysis problem.

It can be proved that if *every* transaction in a schedule follows the two-phase locking protocol, then the schedule is guaranteed to be conflict serializable (Eswaran *et al.*, 1976). However, while the two-phase locking protocol guarantees serializability, problems can occur with the interpretation of when locks can be released, as the next example shows.

Example 20.9 Cascading rollback

Consider a schedule consisting of the three transactions shown in Figure 20.18, which conforms to the two-phase locking protocol. Transaction T_{14} obtains an exclusive lock on bal_x then updates it using bal_y, which has been obtained with a shared lock, and writes the value of bal_x back to the database before releasing the lock on bal_x. Transaction T_{15} then obtains an exclusive lock on bal_x, reads the value of bal_x from the database, updates it, and writes the new value back to the database before releasing the lock. Finally, T_{16} share locks bal_x and reads it from the database. By now, T_{14} has failed and has been rolled back. However, since T_{15} is dependent on T_{14} (it has read an item that has been updated by T_{14}), T_{15} must also be rolled back. Similarly, T_{16} is dependent on T_{15}, so it too must be rolled back. This situation, in which a single transaction leads to a series of rollbacks, is called **cascading rollback**.

Figure 20.18 Cascading rollback with 2PL.

Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock($\mathbf{bal_x}$)		
t_3	read($\mathbf{bal_x}$)		
t_4	read_lock($\mathbf{bal_y}$)		
t_5	read($\mathbf{bal_y}$)		
t_6	$\mathbf{bal_x = bal_y + bal_x}$		
t_7	write($\mathbf{bal_x}$)		
t_8	unlock($\mathbf{bal_x}$)	begin_transaction	
t_9	⋮	write_lock($\mathbf{bal_x}$)	
t_{10}	⋮	read($\mathbf{bal_x}$)	
t_{11}	⋮	$\mathbf{bal_x = bal_x + 100}$	
t_{12}	⋮	write($\mathbf{bal_x}$)	
t_{13}	⋮	unlock($\mathbf{bal_x}$)	
t_{14}	⋮	⋮	
t_{15}	**rollback**	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock($\mathbf{bal_x}$)
t_{18}		**rollback**	⋮
t_{19}			**rollback**

Cascading rollbacks are undesirable since they potentially lead to the undoing of a significant amount of work. Clearly, it would be useful if we could design protocols that prevent cascading rollbacks. One way to achieve this with two-phase locking is to leave the release of *all* locks until the end of the transaction, as in the previous examples. In this way, the problem illustrated here would not occur, as T_{15} would not obtain its exclusive lock until after T_{14} had completed the rollback. This is called **rigorous 2PL**. It can be shown that with rigorous 2PL, transactions can be serialized in the order in which they commit. Another variant of 2PL, called **strict 2PL**, only holds *exclusive lock*s until the end of the transaction. Most database systems implement one of these two variants of 2PL.

Another problem with two-phase locking, which applies to all locking-based schemes, is that it can cause **deadlock**, since transactions can wait for locks on data items. If two transactions wait for locks on items held by the other, deadlock will occur and the deadlock detection and recovery scheme described in the Section 20.2.4 is needed. It is also possible for transactions to be in **livelock**, that is left in a wait state indefinitely, unable to acquire any new locks, although the DBMS is not in deadlock. This can happen if the waiting algorithm for transactions is unfair and does not take account of the time that transactions have been waiting. To avoid livelock, a priority system can be used, whereby the longer a transaction has to wait, the higher its priority, for example, a *first-come-first-served* queue can be used for waiting transactions.

Concurrency control with index structures

Concurrency control for an index structure (see Appendix C) can be managed by treating each page of the index as a data item and applying the two-phase locking protocol described above. However, since indexes are likely to be frequently accessed, particularly the higher levels of trees (as searching occurs from the root downwards), this simple concurrency control strategy may lead to high lock contention. Therefore, a more efficient locking protocol is required for indexes. If we examine how tree-based indexes are traversed, we can make the following two observations:

- The search path starts from the root and moves down to the leaf nodes of the tree but the search never moves back up the tree. Thus, once a lower-level node has been accessed, the higher-level nodes in that path will not be used again.
- When a new index value (a key and a pointer) is being inserted into a leaf node, then if the node is not full, the insertion will not cause changes to the higher-level nodes. This suggests that we only have to exclusively lock the leaf node in such a case, and only exclusively lock higher-level nodes if a node is full and has to be split.

Based on these observations, we can derive the following locking strategy:

- For searches, obtain shared locks on nodes starting at the root and proceeding downwards along the required path. Release the lock on a (parent) node once a lock has been obtained on the child node.
- For insertions, a conservative approach would be to obtain exclusive locks on all nodes as we descend the tree to the leaf node to be modified. This ensures that a split in the leaf node can propagate all the way up the tree to the root. However, if a child node is not full, the lock on the parent node can be released. A more optimistic approach would be to obtain shared locks on all nodes as we descend to the leaf node to be modified, where we obtain an exclusive lock on the leaf node itself. If the leaf node has to split, we upgrade the shared lock on the parent node to an exclusive lock. If this node also has to split, we continue to upgrade the locks at the next higher level. In the majority of cases, a split is not required making this a better approach.

The technique of locking a child node and releasing the lock on the parent node if possible is known as **lock-coupling** or **crabbing**. For further details on the performance of concurrency control algorithms for trees, the interested reader is referred to Srinivasan and Carey (1991).

Latches

DBMSs also support another type of lock called a **latch**, which is held for a much shorter duration than a normal lock. A latch can be used before a page is read from, or written to, disk to ensure that the operation is atomic. For example, a latch would be obtained to write a page from the database buffers to disk, the page would then be written to disk, and the latch immediately unset. As the latch is simply to prevent conflict for this type of access, latches do not need to conform to the normal concurrency control protocol such as two-phase locking.

20.2.4 Deadlock

Deadlock An impasse that may result when two (or more) transactions are each waiting for locks to be released that are held by the other.

Figure 20.19 shows two transactions, T_{17} and T_{18}, that are deadlocked because each is waiting for the other to release a lock on an item it holds. At time t_2, transaction T_{17} requests and obtains an exclusive lock on item bal_x, and at time t_3 transaction T_{18} obtains an exclusive lock on item bal_y. Then at t_6, T_{17} requests an exclusive lock on item bal_y. Since T_{18} holds a lock on bal_y, transaction T_{17} waits. Meanwhile, at time t_7, T_{18} requests a lock on item bal_x, which is held by transaction T_{17}. Neither transaction can continue because each is waiting for a lock it cannot obtain until the other completes. Once deadlock occurs, the applications involved cannot resolve the problem. Instead, the DBMS has to recognize that deadlock exists and break the deadlock in some way.

Unfortunately, there is only one way to break deadlock: abort one or more of the transactions. This usually involves undoing all the changes made by the aborted transaction(s). In Figure 20.19, we may decide to abort transaction T_{18}. Once this is complete, the locks held by transaction T_{18} are released and T_{17} is able to continue again. Deadlock should be transparent to the user, so the DBMS should automatically restart the aborted transaction(s).

Figure 20.19
Deadlock between two transactions.

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock($\mathbf{bal_x}$)	begin_transaction
t_3	read($\mathbf{bal_x}$)	write_lock($\mathbf{bal_y}$)
t_4	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	read($\mathbf{bal_y}$)
t_5	write($\mathbf{bal_x}$)	$\mathbf{bal_y} = \mathbf{bal_y} + 100$
t_6	write_lock($\mathbf{bal_y}$)	write($\mathbf{bal_y}$)
t_7	WAIT	write_lock($\mathbf{bal_x}$)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

There are three general techniques for handling deadlock: timeouts, deadlock prevention, and deadlock detection and recovery. With timeouts, the transaction that has requested a lock waits for at most a specified period of time. Using **deadlock prevention**, the DBMS looks ahead to determine if a transaction would cause deadlock, and never allows deadlock to occur. Using **deadlock detection and recovery**, the DBMS allows deadlock to occur but recognizes occurrences of deadlock and breaks them. Since it is more difficult to prevent deadlock than to use timeouts or testing for deadlock and breaking it when it occurs, systems generally avoid the deadlock prevention method.

Timeouts

A simple approach to deadlock prevention is based on *lock timeouts*. With this approach, a transaction that requests a lock will wait for only a system-defined period of time. If the lock has not been granted within this period, the lock request times out. In this case, the DBMS assumes the transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction. This is a very simple and practical solution to deadlock prevention and is used by several commercial DBMSs.

Deadlock prevention

Another possible approach to deadlock prevention is to order transactions using transaction timestamps, which we discuss in Section 20.2.5. Two algorithms have been proposed by Rosenkrantz *et al*. (1978). One algorithm, *Wait-Die*, allows only an older transaction to wait for a younger one, otherwise the transaction is aborted (*dies*) and restarted with the same timestamp, so that eventually it will become the oldest active transaction and will not die. The second algorithm, *Wound-Wait*, uses a symmetrical approach: only a younger transaction can wait for an older one. If an older transaction requests a lock held by a younger one, the younger one is aborted (*wounded*).

A variant of 2PL, called **conservative 2PL**, can also be used to prevent deadlock. Using conservative 2PL, a transaction obtains all its locks when it begins or it waits until all the locks are available. This protocol has the advantage that if lock contention is heavy, the time that locks are held is reduced because transactions are never blocked and therefore never have to wait for locks. On the other hand, if lock contention is low then locks are held longer under this protocol. Further, the overhead for setting locks is high because all the locks must be obtained and released all at once. Thus, if a transaction fails to obtain one lock it must release all the current locks it has obtained and start the lock process again. From a practical perspective, a transaction may not know at the start which locks it may actually need and, therefore, may have to set more locks than is required. This protocol is not used in practice.

Deadlock detection

Deadlock detection is usually handled by the construction of a **wait-for graph** (WFG) that shows the transaction dependencies; that is, transaction T_i is dependent on T_j if transaction T_j holds the lock on a data item that T_i is waiting for. The WFG is a directed graph G = (N, E) that consists of a set of nodes N and a set of directed edges E, which is constructed as follows:

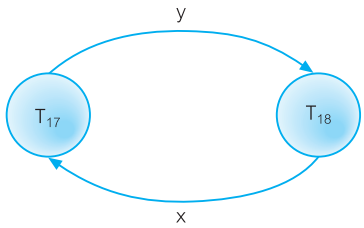


Figure 20.20
WFG with a cycle showing deadlock between two transactions.

- Create a node for each transaction.
- Create a directed edge $T_i \rightarrow T_j$, if transaction T_i is waiting to lock an item that is currently locked by T_j.

Deadlock exists if and only if the WFG contains a cycle (Holt, 1972). Figure 20.20 shows the WFG for the transactions in Figure 20.19. Clearly, the graph has a cycle in it ($T_{17} \rightarrow T_{18} \rightarrow T_{17}$), so we can conclude that the system is in deadlock.

Frequency of deadlock detection

Since a cycle in the wait-for graph is a necessary and sufficient condition for deadlock to exist, the deadlock detection algorithm generates the WFG at regular intervals and examines it for a cycle. The choice of time interval between executions of the algorithm is important. If the interval chosen is too small, deadlock detection will add considerable overhead; if the interval is too large, deadlock may not be detected for a long period. Alternatively, a dynamic deadlock detection algorithm could start with an initial interval size. Each time no deadlock is detected, the detection interval could be increased, for example, to twice the previous interval, and each time deadlock is detected, the interval could be reduced, for example, to half the previous interval, subject to some upper and lower limits.

Recovery from deadlock detection

As we mentioned above, once deadlock has been detected the DBMS needs to abort one or more of the transactions. There are several issues that need to be considered:

(1) *Choice of deadlock victim* In some circumstances, the choice of transactions to abort may be obvious. However, in other situations, the choice may not be so clear. In such cases, we would want to abort the transactions that incur the minimum costs. This may take into consideration:
   (a) how long the transaction has been running (it may be better to abort a transaction that has just started rather than one that has been running for some time);
   (b) how many data items have been updated by the transaction (it would be better to abort a transaction that has made little change to the database rather than one that has made significant changes to the database);
   (c) how many data items the transaction is still to update (it would be better to abort a transaction that has many changes still to make to the database rather than one that has few changes to make). Unfortunately, this may not be something that the DBMS would necessarily know.

(2) *How far to roll a transaction back* Having decided to abort a particular transaction, we have to decide how far to roll the transaction back. Clearly, undoing all the changes

made by a transaction is the simplest solution, although not necessarily the most efficient. It may be possible to resolve the deadlock by rolling back only part of the transaction.

(3) *Avoiding starvation* Starvation occurs when the same transaction is always chosen as the victim, and the transaction can never complete. Starvation is very similar to livelock mentioned in Section 20.2.3, which occurs when the concurrency control protocol never selects a particular transaction that is waiting for a lock. The DBMS can avoid starvation by storing a count of the number of times a transaction has been selected as the victim and using a different selection criterion once this count reaches some upper limit.

Timestamping Methods 20.2.5

The use of locks, combined with the two-phase locking protocol, guarantees serializability of schedules. The order of transactions in the equivalent serial schedule is based on the order in which the transactions lock the items they require. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. A different approach that also guarantees serializability uses transaction timestamps to order transaction execution for an equivalent serial schedule.

Timestamp methods for concurrency control are quite different from locking methods. No locks are involved, and therefore there can be no deadlock. Locking methods generally prevent conflicts by making transactions wait. With timestamp methods, there is no waiting: transactions involved in conflict are simply rolled back and restarted.

Timestamp A unique identifier created by the DBMS that indicates the relative starting time of a transaction.

Timestamps can be generated by simply using the system clock at the time the transaction started, or, more normally, by incrementing a logical counter every time a new transaction starts.

Timestamping A concurrency control protocol that orders transactions in such a way that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.

With timestamping, if a transaction attempts to read or write a data item, then the read or write is only allowed to proceed if the *last update on that data item* was carried out by an older transaction. Otherwise, the transaction requesting the read/write is restarted and given a new timestamp. New timestamps must be assigned to restarted transactions to prevent their being continually aborted and restarted. Without new timestamps, a transaction with an old timestamp might not be able to commit owing to younger transactions having already committed.

Besides timestamps for transactions, there are timestamps for data items. Each data item contains a **read_timestamp**, giving the timestamp of the last transaction to read the item, and

a **write_timestamp**, giving the timestamp of the last transaction to write (update) the item. For a transaction T with timestamp ts(T), the timestamp ordering protocol works as follows.

(1) *Transaction T issues a read(x)*

(a) Transaction T asks to read an item (x) that has already been updated by a younger (later) transaction, that is ts(T) < write_timestamp(x). This means that an earlier transaction is trying to read a value of an item that has been updated by a later transaction. The earlier transaction is too late to read the previous outdated value, and any other values it has acquired are likely to be inconsistent with the updated value of the data item. In this situation, transaction T must be aborted and restarted with a new (later) timestamp.

(b) Otherwise, ts(T) ≥ write_timestamp(x), and the read operation can proceed. We set read_timestamp(x) = max(ts(T), read_timestamp(x)).

(2) *Transaction T issues a write(x)*

(a) Transaction T asks to write an item (x) whose value has already been read by a younger transaction, that is ts(T) < read_timestamp(x). This means that a later transaction is already using the current value of the item and it would be an error to update it now. This occurs when a transaction is late in doing a write and a younger transaction has already read the old value or written a new one. In this case, the only solution is to roll back transaction T and restart it using a later timestamp.

(b) Transaction T asks to write an item (x) whose value has already been written by a younger transaction, that is ts(T) < write_timestamp(x). This means that transaction T is attempting to write an obsolete value of data item x. Transaction T should be rolled back and restarted using a later timestamp.

(c) Otherwise, the write operation can proceed. We set write_timestamp(x) = ts(T).

This scheme, called **basic timestamp ordering**, guarantees that transactions are conflict serializable, and the results are equivalent to a serial schedule in which the transactions are executed in chronological order of the timestamps. In other words, the results will be as if all of transaction 1 were executed, then all of transaction 2, and so on, with no interleaving. However, basic timestamp ordering does not guarantee recoverable schedules. Before we show how these rules can be used to generate a schedule using timestamping, we first examine a slight variation to this protocol that provides greater concurrency.

Thomas's write rule

A modification to the basic timestamp ordering protocol that relaxes conflict serializability can be used to provide greater concurrency by rejecting obsolete write operations (Thomas, 1979). The extension, known as **Thomas's write rule**, modifies the checks for a write operation by transaction T as follows:

(a) Transaction T asks to write an item (x) whose value has already been read by a younger transaction, that is ts(T) < read_timestamp(x). As before, roll back transaction T and restart it using a later timestamp.

(b) Transaction T asks to write an item (x) whose value has already been written by a younger transaction, that is ts(T) < write_timestamp(x). This means that a later transaction has already updated the value of the item, and the value that the older transaction is writing must be based on an obsolete value of the item. In this case, the write

operation can safely be ignored. This is sometimes known as the **ignore obsolete write rule**, and allows greater concurrency.

(c) Otherwise, as before, the write operation can proceed. We set write_timestamp(x) = ts(T).

The use of Thomas's write rule allows schedules to be generated that would not have been possible under the other concurrency protocols discussed in this section. For example, the schedule shown in Figure 20.10 is not conflict serializable: the write operation on bal_x by transaction T_{11} following the write by T_{12} would be rejected, and T_{11} would need to be rolled back and restarted with a new timestamp. In contrast, using Thomas's write rule, this view serializable schedule would be valid without requiring any transactions to be rolled back.

We examine another timestamping protocol that is based on the existence of multiple versions of each data item in the next section.

Example 20.10 Basic timestamp ordering

Three transactions are executing concurrently, as illustrated in Figure 20.21. Transaction T_{19} has a timestamp of $ts(T_{19})$, T_{20} has a timestamp of $ts(T_{20})$, and T_{21} has a timestamp of $ts(T_{21})$, such that $ts(T_{19}) < ts(T_{20}) < ts(T_{21})$. At time t_8, the write by transaction T_{20} violates the first write rule and so T_{20} is aborted and restarted at time t_{14}. Also at time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
t_8	write(bal_y)		write(bal_y)†	
t_9	$bal_y = bal_y + 30$			$bal_y = bal_y + 30$
t_{10}	write(bal_y)			write(bal_y)
t_{11}	$bal_z = 100$			$bal_z = 100$
t_{12}	write(bal_z)			write(bal_z)
t_{13}	$bal_z = 50$	$bal_z = 50$		commit
t_{14}	write(bal_z)	write(bal_z)‡	begin_transaction	
t_{15}	read(bal_y)	commit	read(bal_y)	
t_{16}	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	
t_{17}	write(bal_y)		write(bal_y)	
t_{18}			commit	

† At time t_8, the write by transaction T_{20} violates the first timestamping write rule described above and therefore is aborted and restarted at time t_{14}.
‡ At time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Figure 20.21 Timestamping example.

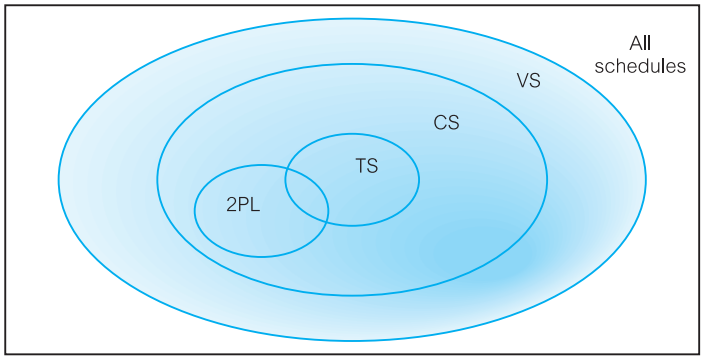


Figure 20.22
Comparison of conflict serializability (CS), view serializability (VS), two-phase locking (2PL), and timestamping (TS).

Comparison of methods

Figure 20.22 illustrates the relationship between conflict serializability (CS), view serializability (VS), two-phase locking (2PL), and timestamping (TS). As can be seen, view serializability encompasses the other three methods, conflict serializability encompasses 2PL and timestamping, while 2PL and timestamping overlap. Note, in the last case, that there are schedules common to both 2PL and timestamping but, equally well, there are also schedules that can be produced by 2PL but not timestamping and vice versa.

20.2.6 Multiversion Timestamp Ordering

Versioning of data can also be used to increase concurrency, since different users may work concurrently on different versions of the same object instead of having to wait for each others' transactions to complete. In the event that the work appears faulty at any stage, it should be possible to roll back the work to some valid state. Versions have been used as an alternative to the nested and multilevel concurrency control protocols we discuss in Section 20.4 (for example, see Beech and Mahbod, 1988; Chou and Kim, 1986, 1988). In this section we briefly examine one concurrency control scheme that uses versions to increase concurrency based on timestamps (Reed, 1978; 1983). In Section 20.5 we briefly discuss how Oracle uses this scheme for concurrency control.

The basic timestamp ordering protocol discussed in the previous section assumes that only one version of a data item exists, and so only one transaction can access a data item at a time. This restriction can be relaxed if we allow multiple transactions to read and write different versions of the same data item, and ensure that each transaction sees a consistent set of versions for all the data items it accesses. In multiversion concurrency control, each write operation creates a new version of a data item while retaining the old version. When a transaction attempts to read a data item, the system selects one of the versions that ensures serializability.

For each data item x, we assume that the database holds n versions $x_1, x_2, \ldots, x_n$. For each version i, the system stores three values:

- the value of version x_i;
- read_timestamp(x_i), which is the largest timestamp of all transactions that have successfully read version x_i;
- write_timestamp(x_i), which is the timestamp of the transaction that created version x_i.

Let ts(T) be the timestamp of the current transaction. The multiversion timestamp ordering protocol uses the following two rules to ensure serializability:

(1) *Transaction T issues a write(x)* If transaction T wishes to write data item x, we must ensure that the data item has not been read already by some other transaction T_j such that ts(T) < ts(T_j). If we allow transaction T to perform this write operation, then for serializability its change should be seen by T_j but clearly T_j, which has already read the value, will not see T's change.

Thus, if version x_j has the largest write timestamp of data item x that is *less than or equal to* ts(T) (that is, write_timestamp(x_j) ≤ ts(T)) and read_timestamp(x_j) > ts(T), transaction T must be aborted and restarted with a new timestamp. Otherwise, we create a new version x_i of x and set read_timestamp(x_i) = write_timestamp(x_i) = ts(T).

(2) *Transaction T issues a read(x)* If transaction T wishes to read data item x, we must return the version x_j that has the largest write timestamp of data item x that is *less than or equal to* ts(T). In other words, return write_timestamp(x_j) such that write_timestamp(x_j) ≤ ts(T). Set the value of read_timestamp(x_j) = max(ts(T), read_timestamp(x_j)). Note that with this protocol a read operation never fails.

Versions can be deleted once they are no longer required. To determine whether a version is required, we find the timestamp of the oldest transaction in the system. Then, for any two versions x_i and x_j of data item x with write timestamps less than this oldest timestamp, we can delete the older version.

Optimistic Techniques 20.2.7

In some environments, conflicts between transactions are rare, and the additional processing required by locking or timestamping protocols is unnecessary for many of the transactions. **Optimistic techniques** are based on the assumption that conflict is rare, and that it is more efficient to allow transactions to proceed without imposing delays to ensure serializability (Kung and Robinson, 1981). When a transaction wishes to commit, a check is performed to determine whether conflict has occurred. If there has been a conflict, the transaction must be rolled back and restarted. Since the premise is that conflict occurs very infrequently, rollback will be rare. The overhead involved in restarting a transaction may be considerable, since it effectively means redoing the entire transaction. This could be tolerated only if it happened very infrequently, in which case the majority of transactions will be processed without being subjected to any delays. These techniques potentially allow greater concurrency than traditional protocols since no locking is required.

There are two or three phases to an optimistic concurrency control protocol, depending on whether it is a read-only or an update transaction:

- *Read phase* This extends from the start of the transaction until immediately before the commit. The transaction reads the values of all data items it needs from the database and stores them in local variables. Updates are applied to a local copy of the data, not to the database itself.
- *Validation phase* This follows the read phase. Checks are performed to ensure serializability is not violated if the transaction updates are applied to the database. For a read-only transaction, this consists of checking that the data values read are still the current values for the corresponding data items. If no interference occurred, the transaction is committed. If interference occurred, the transaction is aborted and restarted. For a transaction that has updates, validation consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained. If not, the transaction is aborted and restarted.
- *Write phase* This follows the successful validation phase for update transactions. During this phase, the updates made to the local copy are applied to the database.

The validation phase examines the reads and writes of transactions that may cause interference. Each transaction T is assigned a timestamp at the start of its execution, *start(T)*, one at the start of its validation phase, *validation(T)*, and one at its finish time, *finish(T)*, including its write phase, if any. To pass the validation test, one of the following must be true:

(1) All transactions S with earlier timestamps must have finished before transaction T started; that is, *finish(S)* < *start(T)*.

(2) If transaction T starts before an earlier one S finishes, then:
   (a) the set of data items written by the earlier transaction are not the ones read by the current transaction; *and*
   (b) the earlier transaction completes its write phase before the current transaction enters its validation phase, that is *start(T)* < *finish(S)* < *validation(T)*.

Rule 2(a) guarantees that the writes of an earlier transaction are not read by the current transaction; rule 2(b) guarantees that the writes are done serially, ensuring no conflict.

Although optimistic techniques are very efficient when there are few conflicts, they can result in the rollback of individual transactions. Note that the rollback involves only a local copy of the data so there are no cascading rollbacks, since the writes have not actually reached the database. However, if the aborted transaction is of a long duration, valuable processing time will be lost since the transaction must be restarted. If rollback occurs often, it is an indication that the optimistic method is a poor choice for concurrency control in that particular environment.

20.2.8 Granularity of Data Items

Granularity The size of data items chosen as the *unit of protection* by a concurrency control protocol.

All the concurrency control protocols that we have discussed assume that the database consists of a number of 'data items', without explicitly defining the term. Typically, a data item is chosen to be one of the following, ranging from coarse to fine, where fine granularity refers to small item sizes and coarse granularity refers to large item sizes:

- the entire database;
- a file;
- a page (sometimes called an area or database space – a section of physical disk in which relations are stored);
- a record;
- a field value of a record.

The size or granularity of the data item that can be locked in a single operation has a significant effect on the overall performance of the concurrency control algorithm. However, there are several tradeoffs that have to be considered in choosing the data item size. We discuss these tradeoffs in the context of locking, although similar arguments can be made for other concurrency control techniques.

Consider a transaction that updates a single tuple of a relation. The concurrency control algorithm might allow the transaction to lock only that single tuple, in which case the granule size for locking is a single record. On the other hand, it might lock the entire database, in which case the granule size is the entire database. In the second case, the granularity would prevent any other transactions from executing until the lock is released. This would clearly be undesirable. On the other hand, if a transaction updates 95% of the records in a file, then it would be more efficient to allow it to lock the entire file rather than to force it to lock each record separately. However, escalating the granularity from field or record to file may increase the likelihood of deadlock occurring.

Thus, the coarser the data item size, the lower the degree of concurrency permitted. On the other hand, the finer the item size, the more locking information that needs to be stored. The best item size depends upon the nature of the transactions. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity at the record level. On the other hand, if a transaction typically accesses many records of the same file, it may be better to have page or file granularity so that the transaction considers all those records as one (or a few) data items.

Some techniques have been proposed that have dynamic data item sizes. With these techniques, depending on the types of transaction that are currently executing, the data item size may be changed to the granularity that best suits these transactions. Ideally, the DBMS should support mixed granularity with record, page, and file level locking. Some systems automatically upgrade locks from record or page to file if a particular transaction is locking more than a certain percentage of the records or pages in the file.

Hierarchy of granularity

We could represent the granularity of locks in a hierarchical structure where each node represents data items of different sizes, as shown in Figure 20.23. Here, the root node represents the entire database, the level 1 nodes represent files, the level 2 nodes represent

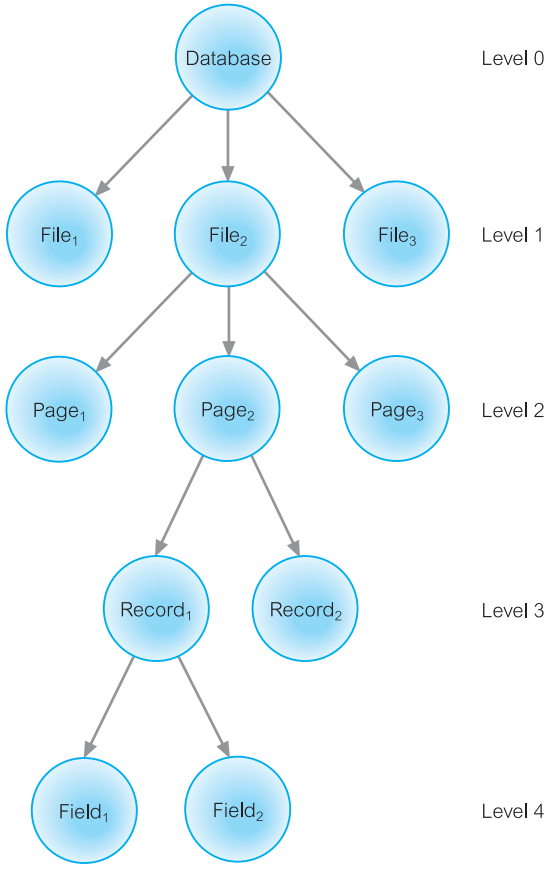


Figure 20.23
Levels of locking.

pages, the level 3 nodes represent records, and the level 4 leaves represent individual fields. Whenever a node is locked, all its descendants are also locked. For example, if a transaction locks a page, $Page_2$, all its records ($Record_1$ and $Record_2$) as well as all their fields ($Field_1$ and $Field_2$) are also locked. If another transaction requests an incompatible lock on the *same* node, the DBMS clearly knows that the lock cannot be granted.

If another transaction requests a lock on any of the *descendants* of the locked node, the DBMS checks the hierarchical path from the root to the requested node to determine if any of its ancestors are locked before deciding whether to grant the lock. Thus, if the request is for an exclusive lock on record $Record_1$, the DBMS checks its parent ($Page_2$), its grandparent ($File_2$), and the database itself to determine if any of them are locked. When it finds that $Page_2$ is already locked, it denies the request.

Additionally, a transaction may request a lock on a node and a descendant of the node is already locked. For example, if a lock is requested on $File_2$, the DBMS checks every page in the file, every record in those pages, and every field in those records to determine if any of them are locked.

Table 20.1 Lock compatibility table for multiple-granularity locking.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

✓ = compatible, ✗ = incompatible

Multiple-granularity locking

To reduce the searching involved in locating locks on descendants, the DBMS can use another specialized locking strategy called **multiple-granularity locking**. This strategy uses a new type of lock called an **intention lock** (Gray *et al.*, 1975). When any node is locked, an intention lock is placed on all the ancestors of the node. Thus, if some descendant of $File_2$ (in our example, $Page_2$) is locked and a request is made for a lock on $File_2$, the presence of an intention lock on $File_2$ indicates that some descendant of that node is already locked.

Intention locks may be either Shared (read) or eXclusive (write). An *intention shared* (IS) lock conflicts only with an exclusive lock; an *intention exclusive* (IX) lock conflicts with both a shared and an exclusive lock. In addition, a transaction can hold a *shared and intention exclusive* (SIX) lock that is logically equivalent to holding both a shared and an IX lock. A SIX lock conflicts with any lock that conflicts with either a shared or IX lock; in other words, a SIX lock is compatible only with an IS lock. The lock compatibility table for multiple-granularity locking is shown in Table 20.1.

To ensure serializability with locking levels, a two-phase locking protocol is used as follows:

- No lock can be granted once any node has been unlocked.
- No node may be locked until its parent is locked by an intention lock.
- No node may be unlocked until all its descendants are unlocked.

In this way, locks are applied from the root down using intention locks until the node requiring an actual read or exclusive lock is reached, and locks are released from the bottom up. However, deadlock is still possible and must be handled as discussed previously.

20.3 Database Recovery

Database recovery The process of restoring the database to a correct state in the event of a failure.

At the start of this chapter we introduced the concept of database recovery as a service that should be provided by the DBMS to ensure that the database is reliable and remains in a consistent state in the presence of failures. In this context, reliability refers to both the resilience of the DBMS to various types of failure and its capability to recover from them. In this section we consider how this service can be provided. To gain a better understanding of the potential problems we may encounter in providing a reliable system, we start by examining the need for recovery and the types of failure that can occur in a database environment.

20.3.1 The Need for Recovery

The storage of data generally includes four different types of media with an increasing degree of reliability: main memory, magnetic disk, magnetic tape, and optical disk. Main memory is **volatile** storage that usually does not survive system crashes. Magnetic disks provide **online non-volatile** storage. Compared with main memory, disks are more reliable and much cheaper, but slower by three to four orders of magnitude. Magnetic tape is an **offline non-volatile** storage medium, which is far more reliable than disk and fairly inexpensive, but slower, providing only sequential access. Optical disk is more reliable than tape, generally cheaper, faster, and providing random access. Main memory is also referred to as **primary storage** and disks and tape as **secondary storage**. **Stable storage** represents information that has been replicated in several non-volatile storage media (usually disk) with independent failure modes. For example, it may be possible to simulate stable storage using RAID (Redundant Array of Independent Disks) technology, which guarantees that the failure of a single disk, even during data transfer, does not result in loss of data (see Section 19.2.6).

There are many different types of failure that can affect database processing, each of which has to be dealt with in a different manner. Some failures affect main memory only, while others involve non-volatile (secondary) storage. Among the causes of failure are:

- **system crashes** due to hardware or software errors, resulting in loss of main memory;
- **media failures**, such as head crashes or unreadable media, resulting in the loss of parts of secondary storage;
- **application software errors**, such as logical errors in the program that is accessing the database, which cause one or more transactions to fail;
- **natural physical disasters**, such as fires, floods, earthquakes, or power failures;
- **carelessness** or unintentional destruction of data or facilities by operators or users;
- **sabotage**, or intentional corruption or destruction of data, hardware, or software facilities.

Whatever the cause of the failure, there are two principal effects that we need to consider: the loss of main memory, including the database buffers, and the loss of the disk copy of the database. In the remainder of this chapter we discuss the concepts and techniques that can minimize these effects and allow recovery from failure.

Transactions and Recovery 20.3.2

Transactions represent the basic *unit of recovery* in a database system. It is the role of the recovery manager to guarantee two of the four *ACID* properties of transactions, namely *atomicity* and *durability*, in the presence of failures. The recovery manager has to ensure that, on recovery from failure, either all the effects of a given transaction are permanently recorded in the database or none of them are. The situation is complicated by the fact that database writing is not an atomic (single-step) action, and it is therefore possible for a transaction to have committed but for its effects not to have been permanently recorded in the database, simply because they have not yet reached the database.

Consider again the first example of this chapter, in which the salary of a member of staff is being increased, as shown at a high level in Figure 20.1(a). To implement the read operation, the DBMS carries out the following steps:

- find the address of the disk block that contains the record with primary key value *x*;
- transfer the disk block into a database buffer in main memory;
- copy the salary data from the database buffer into the variable *salary*.

For the write operation, the DBMS carries out the following steps:

- find the address of the disk block that contains the record with primary key value *x*;
- transfer the disk block into a database buffer in main memory;
- copy the salary data from the variable *salary* into the database buffer;
- write the database buffer back to disk.

The database buffers occupy an area in main memory from which data is transferred to and from secondary storage. It is only once the buffers have been **flushed** to secondary storage that any update operations can be regarded as permanent. This flushing of the buffers to the database can be triggered by a specific command (for example, transaction commit) or automatically when the buffers become full. The explicit writing of the buffers to secondary storage is known as **force-writing**.

If a failure occurs between writing to the buffers and flushing the buffers to secondary storage, the recovery manager must determine the status of the transaction that performed the write at the time of failure. If the transaction had issued its commit, then to ensure durability the recovery manager would have to **redo** that transaction's updates to the database (also known as **rollforward**).

On the other hand, if the transaction had not committed at the time of failure, then the recovery manager would have to **undo** (**rollback**) any effects of that transaction on the database to guarantee transaction atomicity. If only one transaction has to be undone, this is referred to as **partial undo**. A partial undo can be triggered by the scheduler when a transaction is rolled back and restarted as a result of the concurrency control protocol, as described in the previous section. A transaction can also be aborted unilaterally, for example, by the user or by an exception condition in the application program. When all active transactions have to be undone, this is referred to as **global undo**.

Example 20.11 Use of UNDO/REDO

Figure 20.24 illustrates a number of concurrently executing transactions $T_1, \ldots, T_6$. The DBMS starts at time t_0 but fails at time t_f. We assume that the data for transactions T_2 and T_3 has been written to secondary storage before the failure.

Clearly T_1 and T_6 had not committed at the point of the crash, therefore at restart the recovery manager must *undo* transactions T_1 and T_6. However, it is not clear to what extent the changes made by the other (committed) transactions T_4 and T_5 have been propagated to the database on non-volatile storage. The reason for this uncertainty is the fact that the volatile database buffers may or may not have been written to disk. In the absence of any other information, the recovery manager would be forced to *redo* transactions T_2, T_3, T_4, and T_5.

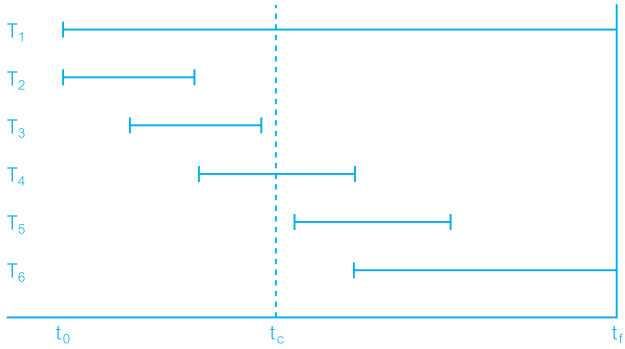


Figure 20.24 Example of UNDO/REDO.

Buffer management

The management of the database buffers plays an important role in the recovery process and we briefly discuss their management before proceeding. As we mentioned at the start of this chapter, the buffer manager is responsible for the efficient management of the database buffers that are used to transfer pages to and from secondary storage. This involves reading pages from disk into the buffers until the buffers become full and then using a *replacement strategy* to decide which buffer(s) to force-write to disk to make space for new pages that need to be read from disk. Example replacement strategies are *first-in-first-out* (FIFO) and *least recently used* (LRU). In addition, the buffer manager should not read a page from disk if it is already in a database buffer.

One approach is to associate two variables with the management information for each database buffer: pinCount and dirty, which are initially set to zero for each database buffer. When a page is requested from disk, the buffer manager will check to see whether the page is already in one of the database buffers. If it is not, the buffer manager will:

(1) use the replacement strategy to choose a buffer for replacement (which we will call the *replacement buffer*) and increment its pinCount. The requested page is now **pinned**

in the database buffer and cannot be written back to disk yet. The replacement strategy will not choose a buffer that has been pinned;

(2) if the dirty variable for the replacement buffer is set, it will write the buffer to disk;

(3) read the page from disk into the replacement buffer and reset the buffer's dirty variable to zero.

If the same page is requested again, the appropriate pinCount is incremented by 1. When the system informs the buffer manager that it has finished with the page, the appropriate pinCount is decremented by 1. At this point, the system will also inform the buffer manager if it has modified the page and the dirty variable is set accordingly. When a pinCount reaches zero, the page is **unpinned** and the page can be written back to disk if it has been modified (that is, if the dirty variable has been set).

The following terminology is used in database recovery when pages are written back to disk:

- A **steal policy** allows the buffer manager to write a buffer to disk before a transaction commits (the buffer is unpinned). In other words, the buffer manages '*steals*' a page from the transaction. The alternative policy is **no-steal**.
- A **force** policy ensures that all pages updated by a transaction are immediately written to disk when the transaction commits. The alternative policy is **no-force**.

The simplest approach from an implementation perspective is to use a no-steal, force policy: with *no-steal* we do not have to undo changes of an aborted transaction because the changes will not have been written to disk, and with *force* we do not have to redo the changes of a committed transaction if there is a subsequent crash because all the changes will have been written to disk at commit. The deferred update recovery protocol we discuss shortly uses a no-steal policy.

On the other hand, the *steal* policy avoids the need for a very large buffer space to store all updated pages by a set of concurrent transactions, which in practice may be unrealistic anyway. In addition, the *no-force* policy has the distinct advantage of not having to rewrite a page to disk for a later transaction that has been updated by an earlier committed transaction and may still be in a database buffer. For these reasons, most DBMSs employ a steal, no-force policy.

Recovery Facilities 20.3.3

A DBMS should provide the following facilities to assist with recovery:

- a backup mechanism, which makes periodic backup copies of the database;
- logging facilities, which keep track of the current state of transactions and database changes;
- a checkpoint facility, which enables updates to the database that are in progress to be made permanent;
- a recovery manager, which allows the system to restore the database to a consistent state following a failure.

Backup mechanism

The DBMS should provide a mechanism to allow backup copies of the database and the *log file* (discussed next) to be made at regular intervals without necessarily having to stop the system first. The backup copy of the database can be used in the event that the database has been damaged or destroyed. A backup can be a complete copy of the entire database or an incremental backup, consisting only of modifications made since the last complete or incremental backup. Typically, the backup is stored on offline storage, such as magnetic tape.

Log file

To keep track of database transactions, the DBMS maintains a special file called a **log** (or **journal**) that contains information about all updates to the database. The log may contain the following data:

- **Transaction records**, containing:
  - transaction identifier;
  - type of log record (transaction start, insert, update, delete, abort, commit);
  - identifier of data item affected by the database action (insert, delete, and update operations);
  - **before-image** of the data item, that is, its value before change (update and delete operations only);
  - **after-image** of the data item, that is, its value after change (insert and update operations only);
  - log management information, such as a pointer to previous and next log records for that transaction (all operations).
- **Checkpoint records**, which we describe shortly.

The log is often used for purposes other than recovery (for example, for performance monitoring and auditing). In this case, additional information may be recorded in the log file (for example, database reads, user logons, logoffs, and so on), but these are not relevant to recovery and therefore are omitted from this discussion. Figure 20.25 illustrates a

Figure 20.25
A segment of a log file.

Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

segment of a log file that shows three concurrently executing transactions T1, T2, and T3. The columns pPtr and nPtr represent pointers to the previous and next log records for each transaction.

Owing to the importance of the transaction log file in the recovery process, the log may be duplexed or triplexed (that is, two or three separate copies are maintained) so that if one copy is damaged, another can be used. In the past, log files were stored on magnetic tape because tape was more reliable and cheaper than magnetic disk. However, nowadays DBMSs are expected to be able to recover quickly from minor failures. This requires that the log file be stored online on a fast direct-access storage device.

In some environments where a vast amount of logging information is generated every day (a daily logging rate of 10^4 megabytes is not uncommon), it is not possible to hold all this data online all the time. The log file is needed online for quick recovery following minor failures (for example, rollback of a transaction following deadlock). Major failures, such as disk head crashes, obviously take longer to recover from and may require access to a large part of the log. In these cases, it would be acceptable to wait for parts of the log file to be brought back online from offline storage.

One approach to handling the offlining of the log is to divide the online log into two separate random access files. Log records are written to the first file until it reaches a high-water mark, for example 70% full. A second log file is then opened and all log records for *new* transactions are written to the second file. *Old* transactions continue to use the first file until they have finished, at which time the first file is closed and transferred to offline storage. This simplifies the recovery of a single transaction as all the log records for that transaction are either on offline or online storage. It should be noted that the log file is a potential bottleneck and the speed of the writes to the log file can be critical in determining the overall performance of the database system.

Checkpointing

The information in the log file is used to recover from a database failure. One difficulty with this scheme is that when a failure occurs we may not know how far back in the log to search and we may end up redoing transactions that have been safely written to the database. To limit the amount of searching and subsequent processing that we need to carry out on the log file, we can use a technique called **checkpointing**.

Checkpoint The point of synchronization between the database and the transaction log file. All buffers are force-written to secondary storage.

Checkpoints are scheduled at predetermined intervals and involve the following operations:

- writing all log records in main memory to secondary storage;
- writing the modified blocks in the database buffers to secondary storage;
- writing a checkpoint record to the log file. This record contains the identifiers of all transactions that are active at the time of the checkpoint.

If transactions are performed serially, then, when a failure occurs, we check the log file to find the last transaction that started before the last checkpoint. Any earlier transactions would have committed previously and would have been written to the database at the checkpoint. Therefore, we need only redo the one that was active at the checkpoint and any subsequent transactions for which both start and commit records appear in the log. If a transaction is active at the time of failure, the transaction must be undone. If transactions are performed concurrently, we redo all transactions that have committed since the checkpoint and undo all transactions that were active at the time of the crash.

Example 20.12 Use of UNDO/REDO with checkpointing

Returning to Example 20.11, if we now assume that a checkpoint occurred at point t_c, then we would know that the changes made by transactions T_2 and T_3 had been written to secondary storage. In this case, the recovery manager would be able to omit the redo for these two transactions. However, the recovery manager would have to redo transactions T_4 and T_5, which have committed since the checkpoint, and undo transactions T_1 and T_6, which were active at the time of the crash.

Generally, checkpointing is a relatively inexpensive operation, and it is often possible to take three or four checkpoints an hour. In this way, no more than 15–20 minutes of work will need to be recovered.

20.3.4 Recovery Techniques

The particular recovery procedure to be used is dependent on the extent of the damage that has occurred to the database. We consider two cases:

- If the database has been extensively damaged, for example a disk head crash has occurred and destroyed the database, then it is necessary to restore the last backup copy of the database and reapply the update operations of committed transactions using the log file. This assumes, of course, that the log file has not been damaged as well. In Step 8 of the physical database design methodology presented in Chapter 18, it was recommended that, where possible, the log file be stored on a disk separate from the main database files. This reduces the risk of both the database files and the log file being damaged at the same time.
- If the database has not been physically damaged but has become inconsistent, for example the system crashed while transactions were executing, then it is necessary to undo the changes that caused the inconsistency. It may also be necessary to redo some transactions to ensure that the updates they performed have reached secondary storage. Here, we do not need to use the backup copy of the database but can restore the database to a consistent state using the **before-** and **after-images** held in the log file.

We now look at two techniques for recovery from the latter situation, that is, the case where the database has not been destroyed but is in an inconsistent state. The techniques, known as **deferred update** and **immediate update**, differ in the way that updates are written to secondary storage. We also look briefly at an alternative technique called **shadow paging**.

Recovery techniques using deferred update

Using the *deferred update* recovery protocol, updates are not written to the database until after a transaction has reached its commit point. If a transaction fails before it reaches this point, it will not have modified the database and so no undoing of changes will be necessary. However, it may be necessary to redo the updates of committed transactions as their effect may not have reached the database. In this case, we use the log file to protect against system failures in the following way:

- When a transaction starts, write a *transaction start* record to the log.
- When any write operation is performed, write a log record containing all the log data specified previously (excluding the before-image of the update). Do not actually write the update to the database buffers or the database itself.
- When a transaction is about to commit, write a *transaction commit* log record, write all the log records for the transaction to disk, and then commit the transaction. Use the log records to perform the actual updates to the database.
- If a transaction aborts, ignore the log records for the transaction and do not perform the writes.

Note that we write the log records to disk before the transaction is actually committed, so that if a system failure occurs while the actual database updates are in progress, the log records will survive and the updates can be applied later. In the event of a failure, we examine the log to identify the transactions that were in progress at the time of failure. Starting at the last entry in the log file, we go back to the most recent checkpoint record:

- Any transaction with *transaction start* and *transaction commit* log records should be **redone**. The redo procedure performs all the writes to the database using the after-image log records for the transactions, *in the order in which they were written to the log*. If this writing has been performed already, before the failure, the write has no effect on the data item, so there is no damage done if we write the data again (that is, the operation is **idempotent**). However, this method guarantees that we will update any data item that was not properly updated prior to the failure.
- For any transactions with *transaction start* and *transaction abort* log records, we do nothing since no actual writing was done to the database, so these transactions do not have to be undone.

If a second system crash occurs during recovery, the log records are used again to restore the database. With the form of the write log records, it does not matter how many times we redo the writes.

Recovery techniques using immediate update

Using the *immediate update* recovery protocol, updates are applied to the database as they occur without waiting to reach the commit point. As well as having to redo the updates of committed transactions following a failure, it may now be necessary to undo the effects of transactions that had not committed at the time of failure. In this case, we use the log file to protect against system failures in the following way:

- When a transaction starts, write a *transaction start* record to the log.
- When a write operation is performed, write a record containing the necessary data to the log file.
- Once the log record is written, write the update to the database buffers.
- The updates to the database itself are written when the buffers are next flushed to secondary storage.
- When the transaction commits, write a *transaction commit* record to the log.

It is essential that log records (or at least certain parts of them) are written *before* the corresponding write to the database. This is known as the **write-ahead log protocol**. If updates were made to the database first, and failure occurred before the log record was written, then the recovery manager would have no way of undoing (or redoing) the operation. Under the write-ahead log protocol, the recovery manager can safely assume that, if there is no *transaction commit* record in the log file for a particular transaction then that transaction was still active at the time of failure and must therefore be undone.

If a transaction aborts, the log can be used to undo it since it contains all the old values for the updated fields. As a transaction may have performed several changes to an item, the writes are undone *in reverse order*. Regardless of whether the transaction's writes have been applied to the database itself, writing the before-images guarantees that the database is restored to its state prior to the start of the transaction.

If the system fails, recovery involves using the log to undo or redo transactions:

- For any transaction for which both a *transaction start* and *transaction commit* record appear in the log, we redo using the log records to write the after-image of updated fields, as described above. Note that if the new values have already been written to the database, these writes, although unnecessary, will have no effect. However, any write that did not actually reach the database will now be performed.
- For any transaction for which the log contains a *transaction start* record but not a *transaction commit* record, we need to undo that transaction. This time the log records are used to write the before-image of the affected fields, and thus restore the database to its state prior to the transaction's start. The undo operations are performed *in the reverse order to which they were written to the log*.

Shadow paging

An alternative to the log-based recovery schemes described above is **shadow paging** (Lorie, 1977). This scheme maintains two-page tables during the life of a transaction: a *current* page table and a *shadow* page table. When the transaction starts, the two-page

tables are the same. The shadow page table is never changed thereafter, and is used to restore the database in the event of a system failure. During the transaction, the current page table is used to record all updates to the database. When the transaction completes, the current page table becomes the shadow page table. Shadow paging has several advantages over the log-based schemes: the overhead of maintaining the log file is eliminated, and recovery is significantly faster since there is no need for undo or redo operations. However, it has disadvantages as well, such as data fragmentation and the need for periodic garbage collection to reclaim inaccessible blocks.

Recovery in a Distributed DBMS 20.3.5

In Chapters 22 and 23 we discuss the distributed DBMS (DDBMS), which consists of a logically interrelated collection of databases physically distributed over a computer network, each under the control of a local DBMS. In a DDBMS, **distributed transactions** (transactions that access data at more than one site) are divided into a number of **subtransactions**, one for each site that has to be accessed. In such a system, atomicity has to be maintained for both the subtransactions and the overall (global) transaction. The techniques described above can be used to ensure the atomicity of subtransactions. Ensuring atomicity of the global transaction means ensuring that the subtransactions either all commit or all abort. The two common protocols for distributed recovery are known as two-phase commit (2PC) and three-phase commit (3PC) and will be examined in Section 23.4.

Advanced Transaction Models 20.4

The transaction protocols that we have discussed so far in this chapter are suitable for the types of transaction that arise in traditional business applications, such as banking and airline reservation systems. These applications are characterized by:

- the simple nature of the data, such as integers, decimal numbers, short character strings, and dates;
- the short duration of transactions, which generally finish within minutes, if not seconds.

In Section 25.1 we examine the more advanced types of database application that have emerged. For example, design applications such as Computer-Aided Design, Computer-Aided Manufacturing, and Computer-Aided Software Engineering have some common characteristics that are different from traditional database applications:

- A design may be very large, perhaps consisting of millions of parts, often with many interdependent subsystem designs.
- The design is not static but evolves through time. When a design change occurs, its implications must be propagated through all design representations. The dynamic nature of design may mean that some actions cannot be foreseen.
- Updates are far-reaching because of topological relationships, functional relationships, tolerances, and so on. One change is likely to affect a large number of design objects.

- Often, many design alternatives are being considered for each component, and the correct version for each part must be maintained. This involves some form of version control and configuration management.
- There may be hundreds of people involved with the design, and they may work in parallel on multiple versions of a large design. Even so, the end-product must be consistent and coordinated. This is sometimes referred to as *cooperative engineering*. Cooperation may require interaction and sharing between other concurrent activities.

Some of these characteristics result in transactions that are very complex, access many data items, and are of long duration, possibly running for hours, days, or perhaps even months. These requirements force a re-examination of the traditional transaction management protocols to overcome the following problems:

- As a result of the time element, a **long-duration transaction** is more susceptible to failures. It would be unacceptable to abort this type of transaction and potentially lose a significant amount of work. Therefore, to minimize the amount of work lost, we require that the transaction be recovered to a state that existed shortly before the crash.
- Again, as a result of the time element, a long-duration transaction may access (for example, lock) a large number of data items. To preserve transaction isolation, these data items are then inaccessible to other applications until the transaction commits. It is undesirable to have data inaccessible for extended periods of time as this limits concurrency.
- The longer the transaction runs, the more likely it is that deadlock will occur if a locking-based protocol is used. It has been shown that the frequency of deadlock increases to the fourth power of the transaction size (Gray, 1981).
- One way to achieve cooperation among people is through the use of shared data items. However, the traditional transaction management protocols significantly restrict this type of cooperation by requiring the isolation of incomplete transactions.

In the remainder of this section, we consider the following advanced transaction models:

- nested transaction model;
- sagas;
- multilevel transaction model;
- dynamic restructuring;
- workflow models.

20.4.1 Nested Transaction Model

Nested transaction model	A transaction is viewed as a collection of related subtasks, or *subtransactions*, each of which may also contain any number of subtransactions.

The **nested transaction model** was introduced by Moss (1981). In this model, the complete transaction forms a tree, or hierarchy, of **subtransactions**. There is a top-level transaction that can have a number of child transactions; each child transaction can also have nested transactions. In Moss's original proposal, only the leaf-level subtransactions (the subtransactions at the lowest level of nesting) are allowed to perform the database operations. For example, in Figure 20.26 we have a reservation transaction (T_1) that consists of booking flights (T_2), hotel (T_5), and hire car (T_6). The flight reservation booking itself is split into two subtransactions: one to book a flight from London to Paris (T_3) and a second to book a connecting flight from Paris to New York (T_4). Transactions have to commit from the bottom upwards. Thus, T_3 and T_4 must commit before parent transaction T_2, and T_2 must commit before parent T_1. However, a transaction abort at one level does not have to affect a transaction in progress at a higher level. Instead, a parent is allowed to perform its own recovery in one of the following ways:

- Retry the subtransaction.
- Ignore the failure, in which case the subtransaction is deemed to be *non-vital*. In our example, the car rental may be deemed non-vital and the overall reservation can proceed without it.
- Run an alternative subtransaction, called a *contingency subtransaction*. In our example, if the hotel reservation at the Hilton fails, an alternative booking may be possible at another hotel, for example, the Sheraton.
- Abort.

The updates of committed subtransactions at intermediate levels are visible only within the scope of their immediate parents. Thus, when T_3 commits the changes are visible only to T_2. However, they are not visible to T_1 or any transaction external to T_1. Further, a commit of a subtransaction is conditionally subject to the commit or abort of its superiors. Using this model, top-level transactions conform to the traditional ACID properties of a **flat transaction**.

```
begin_transaction T1                                  Complete Reservation
    begin_transaction T2                              Airline_reservation
        begin_transaction T3                          First_flight
            reserve_airline_seat(London, Paris);
        commit T3;
        begin_transaction T4                          Connecting_flight
            reserve_airline_seat(Paris, New York);
        commit T4;
    commit T2;
    begin_transaction T5                              Hotel_reservation
        book_hotel(Hilton);
    commit T5;
    begin_transaction T6                              Car_reservation
        book_car();
    commit T6;
commit T1;
```

Figure 20.26
Nested transactions.

Moss also proposed a concurrency control protocol for nested transactions, based on strict two-phase locking. The subtransactions of parent transactions are executed as if they were separate transactions. A subtransaction is allowed to hold a lock if any other transaction that holds a conflicting lock is the subtransaction's parent. When a subtransaction commits, its locks are inherited by its parent. In inheriting a lock, the parent holds the lock in a more exclusive mode if both the child and the parent hold a lock on the same data item.

The main advantages of the nested transaction model are its support for:

- *Modularity* A transaction can be decomposed into a number of subtransactions for the purposes of concurrency and recovery.
- *A finer level of granularity for concurrency control and recovery* Occurs at the level of the subtransaction rather than the transaction.
- *Intra-transaction parallelism* Subtransactions can execute concurrently.
- *Intra-transaction recovery* Uncommitted subtransactions can be aborted and rolled back without any side-effects to other subtransactions.

Emulating nested transactions using savepoints

Savepoint An identifiable point in a flat transaction representing some partially consistent state, which can be used as an internal restart point for the transaction if a subsequent problem is detected.

One of the objectives of the nested transaction model is to provide a *unit of recovery* at a finer level of granularity than the transaction. During the execution of a transaction, the user can establish a **savepoint**, for example using a SAVE WORK statement.[†] This generates an identifier that the user can subsequently use to roll the transaction back to, for example using a ROLLBACK WORK <savepoint_identifier> statement.[†] However, unlike nested transactions, savepoints do not support any form of intra-transaction parallelism.

20.4.2 Sagas

Sagas A sequence of (flat) transactions that can be interleaved with other transactions.

The concept of **sagas** was introduced by Garcia-Molina and Salem (1987) and is based on the use of *compensating transactions*. The DBMS guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to recover from partial execution. Unlike a nested transaction, which has an arbitrary level of nesting,

[†] This is not standard SQL, simply an illustrative statement.

a saga has only one level of nesting. Further, for every subtransaction that is defined, there is a corresponding compensating transaction that will semantically undo the subtransaction's effect. Therefore, if we have a saga comprising a sequence of n transactions $T_1, T_2, \ldots, T_n$, with corresponding compensating transactions $C_1, C_2, \ldots, C_n$, then the final outcome of the saga is one of the following execution sequences:

$T_1, T_2, \ldots, T_n$	if the transaction completes successfully
$T_1, T_2, \ldots, T_i, C_{i-1}, \ldots, C_2, C_1$	if subtransaction T_i fails and is aborted

For example, in the reservation system discussed above, to produce a saga we restructure the transaction to remove the nesting of the airline reservations, as follows:

T_3, T_4, T_5, T_6

These subtransactions represent the leaf nodes of the top-level transaction in Figure 20.26. We can easily derive compensating subtransactions to cancel the two flight bookings, the hotel reservation, and the car rental reservation.

Compared with the flat transaction model, sagas relax the property of isolation by allowing a saga to reveal its partial results to other concurrently executing transactions before it completes. Sagas are generally useful when the subtransactions are relatively independent and when compensating transactions can be produced, such as in our example. In some instances though, it may be difficult to define a compensating transaction in advance, and it may be necessary for the DBMS to interact with the user to determine the appropriate compensating effect. In other instances, it may not be possible to define a compensating transaction; for example, it may not be possible to define a compensating transaction for a transaction that dispenses cash from an automatic teller machine.

Multilevel Transaction Model 20.4.3

The nested transaction model presented in Section 20.4.1 requires the commit process to occur in a bottom-up fashion through the top-level transaction. This is called, more precisely, a **closed nested transaction**, as the semantics of these transactions enforce atomicity at the top level. In contrast, we also have **open nested transactions**, which relax this condition and allow the partial results of subtransactions to be observed outside the transaction. The saga model discussed in the previous section is an example of an open nested transaction.

A specialization of the open nested transaction is the **multilevel transaction** model where the tree of subtransactions is balanced (Weikum, 1991; Weikum and Schek, 1991). Nodes at the same depth of the tree correspond to operations of the same level of abstraction in a DBMS. The edges in the tree represent the implementation of an operation by a sequence of operations at the next lower level. The levels of an n-level transaction are denoted $L_0, L_1, \ldots, L_n$, where L_0 represents the lowest level in the tree, and L_n the root of the tree. The traditional flat transaction ensures there are no conflicts at the lowest level (L_0). However, the basic concept in the multilevel transaction model is that two operations at level L_i may not conflict even though their implementations at the next lower level L_{i-1} do conflict. By taking advantage of the level-specific conflict information, multilevel transactions allow a higher degree of concurrency than traditional flat transactions.

Figure 20.27
Non-serializable schedule.

Time	T_7	T_8
t_1	begin_transaction	
t_2	read($\mathbf{bal_x}$)	
t_3	$\mathbf{bal_x} = \mathbf{bal_x} + 5$	
t_4	write($\mathbf{bal_x}$)	
t_5		begin_transaction
t_6		read($\mathbf{bal_y}$)
t_7		$\mathbf{bal_y} = \mathbf{bal_y} + 10$
t_8		write($\mathbf{bal_y}$)
t_9	read($\mathbf{bal_y}$)	
t_{10}	$\mathbf{bal_y} = \mathbf{bal_y} - 5$	
t_{11}	write($\mathbf{bal_y}$)	
t_{12}	commit	
t_{13}		read($\mathbf{bal_x}$)
t_{14}		$\mathbf{bal_x} = \mathbf{bal_x} - 2$
t_{15}		write($\mathbf{bal_x}$)
t_{16}		commit

For example, consider the schedule consisting of two transactions T_7 and T_8 shown in Figure 20.27. We can easily demonstrate that this schedule is not conflict serializable. However, consider dividing T_7 and T_8 into the following subtransactions with higher-level operations:

T_7: T_{71}, which increases bal_x by 5
T_{72}, which subtracts 5 from bal_y

T_8: T_{81}, which increases bal_y by 10
T_{82}, which subtracts 2 from bal_x

With knowledge of the semantics of these operations though, as addition and subtraction are commutative, we can execute these subtransactions in any order, and the correct result will always be generated.

20.4.4 Dynamic Restructuring

At the start of this section we discussed some of the characteristics of design applications, for example, uncertain duration (from hours to months), interaction with other concurrent activities, and uncertain developments, so that some actions cannot be foreseen at the beginning. To address the constraints imposed by the ACID properties of flat transactions, two new operations were proposed: **split-transaction** and **join-transaction** (Pu *et al.*, 1988). The principle behind split-transactions is to split an active transaction into two serializable transactions and divide its actions and resources (for example, locked data items) between the new transactions. The resulting transactions can proceed independently from that point, perhaps controlled by different users, and behave as though they had always been independent. This allows the partial results of a transaction to be shared with other transactions while preserving its semantics; that is, if the original transaction conformed to the ACID properties, then so will the new transactions.

The split-transaction operation can be applied only when it is possible to generate two transactions that are serializable with each other and with all other concurrently executing transactions. The conditions that permit a transaction T to be split into transactions A and B are defined as follows:

(1) AWriteSet $\cap$ BWriteSet $\subseteq$ BWriteLast. This condition states that if both A and B write to the same object, B's write operations must follow A's write operations.

(2) AReadSet $\cap$ BWriteSet = $\emptyset$. This condition states that A cannot see any of the results from B.

(3) BReadSet $\cap$ AWriteSet = ShareSet. This condition states that B may see the results of A.

These three conditions guarantee that A is serialized before B. However, if A aborts, B must also abort because it has read data written by A. If both BWriteLast and ShareSet are empty, then A and B can be serialized in any order and both can be committed independently.

The join-transaction performs the reverse operation of the split-transaction, merging the ongoing work of two or more independent transactions as though these transactions had always been a single transaction. A split-transaction followed by a join-transaction on one of the newly created transactions can be used to transfer resources among particular transactions without having to make the resources available to other transactions.

The main advantages of the dynamic restructuring method are:

- *Adaptive recovery*, which allows part of the work done by a transaction to be committed, so that it will not be affected by subsequent failures.
- *Reducing isolation*, which allows resources to be released by committing part of the transaction.

Workflow Models 20.4.5

The models discussed so far in this section have been developed to overcome the limitations of the flat transaction model for transactions that may be long-lived. However, it has been argued that these models are still not sufficiently powerful to model some business activities. More complex models have been proposed that are combinations of open and nested transactions. However, as these models hardly conform to any of the ACID properties, the more appropriate name *workflow model* has been used instead.

A *workflow* is an activity involving the coordinated execution of multiple tasks performed by different *processing entities*, which may be people or software systems, such as a DBMS, an application program, or an electronic mail system. An example from the *DreamHome* case study is the processing of a rental agreement for a property. The client who wishes to rent a property contacts the appropriate member of staff appointed to manage the desired property. This member of staff contacts the company's credit controller, who verifies that the client is acceptable, using sources such as credit-check bureaux. The credit controller then decides to approve or reject the application and informs the member of staff of the final decision, who passes the final decision on to the client.

There are two general problems involved in workflow systems: the specification of the workflow and the execution of the workflow. Both problems are complicated by the fact that many organizations use multiple, independently managed systems to automate different parts of the process. The following are defined as key issues in specifying a workflow (Rusinkiewicz and Sheth, 1995):

- *Task specification* The execution structure of each task is defined by providing a set of externally observable execution states and a set of transitions between these states.
- *Task coordination requirements* These are usually expressed as intertask-execution dependencies and data-flow dependencies, as well as the termination conditions of the workflow.
- *Execution (correctness) requirements* These restrict the execution of the workflow to meet application-specific correctness criteria. They include failure and execution atomicity requirements and workflow concurrency control and recovery requirements.

In terms of execution, an activity has open nesting semantics that permits partial results to be visible outside its boundary, allowing components of the activity to commit individually. Components may be other activities with the same open nesting semantics, or closed nested transactions that make their results visible to the entire system only when they commit. However, a closed nested transaction can only be composed of other closed nested transactions. Some components in an activity may be defined as vital and, if they abort, their parents must also abort. In addition, compensating and contingency transactions can be defined, as discussed previously.

For a more detailed discussion of advanced transaction models, the interested reader is referred to Korth *et al.* (1988), Skarra and Zdonik (1989), Khoshafian and Abnous (1990), Barghouti and Kaiser (1991), and Gray and Reuter (1993).

20.5 Concurrency Control and Recovery in Oracle

To complete this chapter, we briefly examine the concurrency control and recovery mechanisms in Oracle9*i*. Oracle handles concurrent access slightly differently from the protocols described in Section 20.2. Instead, Oracle uses a *multiversion read consistency* protocol that guarantees a user sees a consistent view of the data requested (Oracle Corporation, 2004a). If another user changes the underlying data during the execution of the query, Oracle maintains a version of the data as it existed at the time the query started. If there are other uncommitted transactions in progress when the query started, Oracle ensures that the query does not see the changes made by these transactions. In addition, Oracle does not place any locks on data for read operations, which means that a read operation never blocks a write operation. We discuss these concepts in the remainder of this chapter. In what follows, we use the terminology of the DBMS – Oracle refers to a relation as a *table* with *columns* and *rows*. We provided an introduction to Oracle in Section 8.2

Oracle's Isolation Levels 20.5.1

In Section 6.5 we discussed the concept of isolation levels, which describe how a transaction is isolated from other transactions. Oracle implements two of the four isolation levels defined in the ISO SQL standard, namely READ COMMITTED and SERIALIZABLE:

- *READ COMMITTED* Serialization is enforced at the **statement level** (this is the default isolation level). Thus, each statement within a transaction sees only data that was committed before the *statement* (not the transaction) started. This does mean that data may be changed by other transactions between executions of the same statement within the same transaction, allowing nonrepeatable and phantom reads.
- *SERIALIZABLE* Serialization is enforced at the **transaction level**, so each statement within a transaction sees only data that was committed before the transaction started, as well as any changes made by the transaction through INSERT, UPDATE, or DELETE statements.

Both isolation levels use row-level locking and both wait if a transaction tries to change a row updated by an uncommitted transaction. If the blocking transaction aborts and rolls back its changes, the waiting transaction can proceed to change the previously locked row. If the blocking transaction commits and releases its locks, then with READ COMMITTED mode the waiting transaction proceeds with its update. However, with SERIALIZABLE mode, an error is returned indicating that the operations cannot be serialized. In this case, the application developer has to add logic to the program to return to the start of the transaction and restart it.

In addition, Oracle supports a third isolation level:

- *READ ONLY* Read-only transactions see only data that was committed before the transaction started.

The isolation level can be set in Oracle using the SQL SET TRANSACTION or ALTER SESSION commands.

Multiversion Read Consistency 20.5.2

In this section we briefly describe the implementation of Oracle's multiversion read consistency protocol. In particular, we describe the use of the rollback segments, system change number (SCN), and locks.

Rollback segments

Rollback segments are structures in the Oracle database used to store undo information. When a transaction is about to change the data in a block, Oracle first writes the before-image of the data to a rollback segment. In addition to supporting multiversion read consistency, rollback segments are also used to undo a transaction. Oracle also maintains one or more *redo logs*, which record all the transactions that occur and are used to recover the database in the event of a system failure.

System change number

To maintain the correct chronological order of operations, Oracle maintains a system change number (SCN). The SCN is a logical timestamp that records the order in which operations occur. Oracle stores the SCN in the redo log to redo transactions in the correct sequence. Oracle uses the SCN to determine which version of a data item should be used within a transaction. It also uses the SCN to determine when to clean out information from the rollback segments.

Locks

Implicit locking occurs for all SQL statements so that a user never needs to lock any resource explicitly, although Oracle does provide a mechanism to allow the user to acquire locks manually or to alter the default locking behavior. The default locking mechanisms lock data at the lowest level of restrictiveness to guarantee integrity while allowing the highest degree of concurrency. Whereas many DBMSs store information on row locks as a list in memory, Oracle stores row-locking information within the actual data block where the row is stored.

As we discussed in Section 20.2, some DBMSs also allow lock escalation. For example, if an SQL statement requires a high percentage of the rows within a table to be locked, some DBMSs will escalate the individual row locks into a table lock. Although this reduces the number of locks the DBMS has to manage, it results in unchanged rows being locked, thereby potentially reducing concurrency and increasing the likelihood of deadlock. As Oracle stores row locks within the data blocks, Oracle never needs to escalate locks.

Oracle supports a number of lock types, including:

- *DDL locks* – used to protect schema objects, such as the definitions of tables and views.
- *DML locks* – used to protect the base data, for example, table locks protect entire tables and row locks protect selected rows. Oracle supports the following types of table lock (least restrictive to most restrictive):
  - row-share table lock (also called a subshare table lock), which indicates that the transaction has locked rows in the table and intends to update them;
  - row-exclusive table lock (also called a subexclusive table lock), which indicates that the transaction has made one or more updates to rows in the table;
  - share table lock, which allows other transactions to query the table;
  - share row exclusive table lock (also called a share-subexclusive table lock);
  - exclusive table lock, which allows the transaction exclusive write access to the table.
- *Internal latches* – used to protect shared data structures in the system global area (SGA).
- *Internal locks* – used to protect data dictionary entries, data files, tablespaces, and rollback segments.
- *Distributed locks* – used to protect data in a distributed and/or parallel server environment.
- *PCM locks* – parallel cache management (PCM) locks are used to protect the buffer cache in a parallel server environment.

Deadlock Detection 20.5.3

Oracle automatically detects deadlock and resolves it by rolling back one of the statements involved in the deadlock. A message is returned to the transaction whose statement is rolled back. Usually the signaled transaction should be rolled back explicitly, but it can retry the rolled-back statement after waiting.

Backup and Recovery 20.5.4

Oracle provides comprehensive backup and recovery services, and additional services to support high availability. A complete review of these services is outwith the scope of this book, and so we touch on only a few of the salient features. The interested reader is referred to the Oracle documentation set for further information (Oracle Corporation, 2004c).

Recovery manager

The Oracle recovery manager (RMAN) provides server-managed backup and recovery. This includes facilities to:

- backup one or more datafiles to disk or tape;
- backup archived redo logs to disk or tape;
- restore datafiles from disk or tape;
- restore and apply archived redo logs to perform recovery.

RMAN maintains a catalog of backup information and has the ability to perform complete backups or incremental backups, in the latter case storing only those database blocks that have changed since the last backup.

Instance recovery

When an Oracle instance is restarted following a failure, Oracle detects that a crash has occurred using information in the control file and the headers of the database files. Oracle will recover the database to a consistent state from the redo log files using rollforward and rollback methods, as we discussed in Section 20.3. Oracle also allows checkpoints to be taken at intervals determined by a parameter in the initialization file (INIT.ORA), although setting this parameter to zero can disable this.

Point-in-time recovery

In an earlier version of Oracle, point-in-time recovery allowed the datafiles to be restored from backups and the redo information to be applied up to a specific time or system change number (SCN). This was useful when an error had occurred and the database had to be recovered to a specific point (for example, a user may have accidentally deleted a table). Oracle has extended this facility to allow point-in-time recovery at the tablespace level, allowing one or more tablespaces to be restored to a particular point.

Standby database

Oracle allows a standby database to be maintained in the event of the primary database failing. The standby database can be kept at an alternative location and Oracle will ship the redo logs to the alternative site as they are filled and apply them to the standby database. This ensures that the standby database is almost up to date. As an extra feature, the standby database can be opened for read-only access, which allows some queries to be offloaded from the primary database.

Chapter Summary

- **Concurrency control** is the process of managing simultaneous operations on the database without having them interfere with one another. **Database recovery** is the process of restoring the database to a correct state after a failure. Both protect the database from inconsistencies and data loss.
- A **transaction** is an action, or series of actions, carried out by a single user or application program, which accesses or changes the contents of the database. A transaction is a logical *unit of work* that takes the database from one consistent state to another. Transactions can terminate successfully (**commit**) or unsuccessfully (**abort**). Aborted transactions must be **undone** or rolled back. The transaction is also the *unit of concurrency* and the *unit of recovery*.
- A transaction should possess the four basic, or so-called **ACID**, properties: atomicity, consistency, isolation, and durability. Atomicity and durability are the responsibility of the recovery subsystem; isolation and, to some extent, consistency are the responsibility of the concurrency control subsystem.
- Concurrency control is needed when multiple users are allowed to access the database simultaneously. Without it, problems of *lost update*, *uncommitted dependency*, and *inconsistent analysis* can arise. Serial execution means executing one transaction at a time, with no interleaving of operations. A **schedule** shows the sequence of the operations of transactions. A schedule is **serializable** if it produces the same results as some serial schedule.
- Two methods that guarantee serializability are **two-phase locking (2PL)** and **timestamping**. Locks may be shared (read) or exclusive (write). In **two-phase locking**, a transaction acquires all its locks before releasing any. With **timestamping**, transactions are ordered in such a way that older transactions get priority in the event of conflict.
- **Deadlock** occurs when two or more transactions are waiting to access data the other transaction has locked. The only way to break deadlock once it has occurred is to abort one or more of the transactions.
- A tree may be used to represent the granularity of locks in a system that allows locking of data items of different sizes. When an item is locked, all its descendants are also locked. When a new transaction requests a lock, it is easy to check all the ancestors of the object to determine whether they are already locked. To show whether any of the node's descendants are locked, an **intention lock** is placed on all the ancestors of any node being locked.
- Some causes of failure are system crashes, media failures, application software errors, carelessness, natural physical disasters, and sabotage. These failures can result in the loss of main memory and/or the disk copy of the database. Recovery techniques minimize these effects.
- To facilitate recovery, one method is for the system to maintain a **log file** containing transaction records that identify the start/end of transactions and the before- and after-images of the write operations. Using **deferred updates**, writes are done initially to the log only and the log records are used to perform actual updates to the database. If the system fails, it examines the log to determine which transactions it needs to **redo**, but there is

no need to **undo** any writes. Using **immediate updates**, an update may be made to the database itself any time after a log record is written. The log can be used to undo and redo transactions in the event of failure.

- **Checkpoints** are used to improve database recovery. At a checkpoint, all modified buffer blocks, all log records, and a checkpoint record identifying all active transactions are written to disk. If a failure occurs, the checkpoint record identifies which transactions need to be redone.
- **Advanced transaction models** include nested transactions, sagas, multilevel transactions, dynamically restructuring transactions, and workflow models.

Review Questions

20.1 Explain what is meant by a transaction. Why are transactions important units of operation in a DBMS?

20.2 The consistency and reliability aspects of transactions are due to the 'ACIDity' properties of transactions. Discuss each of these properties and how they relate to the concurrency control and recovery mechanisms. Give examples to illustrate your answer.

20.3 Describe, with examples, the types of problem that can occur in a multi-user environment when concurrent access to the database is allowed.

20.4 Give full details of a mechanism for concurrency control that can be used to ensure that the types of problem discussed in Question 20.3 cannot occur. Show how the mechanism prevents the problems illustrated from occurring. Discuss how the concurrency control mechanism interacts with the transaction mechanism.

20.5 Explain the concepts of serial, nonserial, and serializable schedules. State the rules for equivalence of schedules.

20.6 Discuss the difference between conflict serializability and view serializability.

20.7 Discuss the types of problem that can occur with locking-based mechanisms for concurrency control and the actions that can be taken by a DBMS to prevent them.

20.8 Why would two-phase locking not be an appropriate concurrency control scheme for indexes? Discuss a more appropriate locking scheme for tree-based indexes.

20.9 What is a timestamp? How do timestamp-based protocols for concurrency control differ from locking based protocols?

20.10 Describe the basic timestamp ordering protocol for concurrency control. What is Thomas's write rule and how does this affect the basic timestamp ordering protocol?

20.11 Describe how versions can be used to increase concurrency.

20.12 Discuss the difference between pessimistic and optimistic concurrency control.

20.13 Discuss the types of failure that may occur in a database environment. Explain why it is important for a multi-user DBMS to provide a recovery mechanism.

20.14 Discuss how the log file (or journal) is a fundamental feature in any recovery mechanism. Explain what is meant by forward and backward recovery and describe how the log file is used in forward and backward recovery. What is the significance of the write-ahead log protocol? How do checkpoints affect the recovery protocol?

20.15 Compare and contrast the deferred update and immediate update recovery protocols.

20.16 Discuss the following advanced transaction models:

(a) nested transactions
(b) sagas
(c) multilevel transactions
(d) dynamically restructuring transactions.

Exercises

20.17 Analyze the DBMSs that you are currently using. What concurrency control protocol does each DBMS use? What type of recovery mechanism is used? What support is provided for the advanced transaction models discussed in Section 20.4?

20.18 For each of the following schedules, state whether the schedule is serializable, conflict serializable, view serializable, recoverable, and whether it avoids cascading aborts:

(a) read(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), write(T_2, bal_x), commit(T_1), commit(T_2)
(b) read(T_1, bal_x), read(T_2, bal_y), write(T_3, bal_x), read(T_2, bal_x), read(T_1, bal_y), commit(T_1), commit(T_2), commit(T_3)
(c) read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), abort(T_2), commit(T_1)
(d) write(T_1, bal_x), read(T_2, bal_x), write(T_1, bal_x), commit(T_2), abort(T_1)
(e) read(T_1, bal_x), write(T_2, bal_x), write(T_1, bal_x), read(T_3, bal_x), commit(T_1), commit(T_2), commit(T_3)

20.19 Draw a precedence graph for each of the schedules (a) to (e) in the previous exercise.

20.20 (a) Explain what is meant by the constrained write rule and explain how to test whether a schedule is conflict serializable under the constrained write rule. Using the above method, determine whether the following schedule is serializable:

$$S = [R_1(Z), R_2(Y), W_2(Y), R_3(Y), R_1(X), W_1(X), W_1(Z), W_3(Y), R_2(X), R_1(Y), W_1(Y), W_2(X), R_3(W), W_3(W)]$$

where $R_i(Z)/W_i(Z)$ indicates a read/write by transaction i on data item Z.

(b) Would it be sensible to produce a concurrency control algorithm based on serializability? Justify your answer. How is serializability used in standard concurrency control algorithms?

20.21 (a) Discuss how you would test for view serializability using a labeled precedence graph.
(b) Using the above method, determine whether the following schedules are conflict serializable:

(i) $S_1 = [R_1(X), W_2(X), W_1(X)]$
(ii) $S_2 = [W_1(X), R_2(X), W_3(X), W_2(X)]$
(iii) $S_3 = [W_1(X), R_2(X), R_3(X), W_3(X), W_4(X), W_2(X)]$

20.22 Produce a wait-for graph for the following transaction scenario, and determine whether deadlock exists:

Transaction	Data items locked by transaction	Data items transaction is waiting for
T_1	x_2	x_1, x_3
T_2	x_3, x_{10}	x_7, x_8
T_3	x_8	x_4, x_5
T_4	x_7	x_1
T_5	x_1, x_5	x_3
T_6	x_4, x_9	x_6
T_7	x_6	x_5

20.23 Write an algorithm for shared and exclusive locking. How does granularity affect this algorithm?

20.24 Write an algorithm that checks whether the concurrently executing transactions are in deadlock.

20.25 Using the sample transactions given in Examples 20.1, 20.2, and 20.3, show how timestamping could be used to produce serializable schedules.

20.26 Figure 20.22 gives a Venn diagram showing the relationships between conflict serializability, view serializability, two-phase locking, and timestamping. Extend the diagram to include optimistic and multiversion concurrency control. Further extend the diagram to differentiate between 2PL and strict 2PL, timestamping without Thomas's write rule, and timestamping with Thomas's write rule.

20.27 Explain why stable storage cannot really be implemented. How would you simulate stable storage?

20.28 Would it be realistic for a DBMS to dynamically maintain a wait-for graph rather than create it each time the deadlock detection algorithm runs? Explain your answer.

Chapter 21 Query Processing

Chapter Objectives

In this chapter you will learn:

- The objectives of query processing and optimization.
- Static versus dynamic query optimization.
- How a query is decomposed and semantically analyzed.
- How to create a relational algebra tree to represent a query.
- The rules of equivalence for the relational algebra operations.
- How to apply heuristic transformation rules to improve the efficiency of a query.
- The types of database statistics required to estimate the cost of operations.
- The different strategies for implementing the relational algebra operations.
- How to evaluate the cost and size of the relational algebra operations.
- How pipelining can be used to improve the efficiency of queries.
- The difference between materialization and pipelining.
- The advantages of left-deep trees.
- Approaches for finding the optimal execution strategy.
- How Oracle handles query optimization.

When the relational model was first launched commercially, one of the major criticisms often cited was inadequate performance of queries. Since then, a significant amount of research has been devoted to developing highly efficient algorithms for processing queries. There are many ways in which a complex query can be performed, and one of the aims of query processing is to determine which one is the most cost effective.

In first generation network and hierarchical database systems, the low-level procedural query language is generally embedded in a high-level programming language such as COBOL, and it is the programmer's responsibility to select the most appropriate execution strategy. In contrast, with declarative languages such as SQL, the user specifies *what* data is required rather than *how* it is to be retrieved. This relieves the user of the responsibility of determining, or even knowing, what constitutes a good execution strategy and makes the language more universally usable. Additionally, giving the DBMS the responsibility

for selecting the best strategy prevents users from choosing strategies that are known to be inefficient and gives the DBMS more control over system performance.

There are two main techniques for query optimization, although the two strategies are usually combined in practice. The first technique uses **heuristic rules** that order the operations in a query. The other technique compares different strategies based on their relative costs and selects the one that minimizes resource usage. Since disk access is slow compared with memory access, disk access tends to be the dominant cost in query processing for a centralized DBMS, and it is the one that we concentrate on exclusively in this chapter when providing cost estimates.

Structure of this Chapter

In Section 21.1 we provide an overview of query processing and examine the main phases of this activity. In Section 21.2 we examine the first phase of query processing, namely query decomposition, which transforms a high-level query into a relational algebra query and checks that it is syntactically and semantically correct. In Section 21.3 we examine the heuristic approach to query optimization, which orders the operations in a query using transformation rules that are known to generate good execution strategies. In Section 21.4 we discuss the cost estimation approach to query optimization, which compares different strategies based on their relative costs and selects the one that minimizes resource usage. In Section 21.5 we discuss pipelining, which is a technique that can be used to further improve the processing of queries. Pipelining allows several operations to be performed in a parallel way, rather than requiring one operation to be complete before another can start. We also discuss how a typical query processor may choose an optimal execution strategy. In the final section, we briefly examine how Oracle performs query optimization.

In this chapter we concentrate on techniques for query processing and optimization in centralized relational DBMSs, being the area that has attracted most effort and the model that we focus on in this book. However, some of the techniques are generally applicable to other types of system that have a high-level interface. Later, in Section 23.7 we briefly examine query processing for distributed DBMSs. In Section 28.5 we see that some of the techniques we examine in this chapter may require further consideration for the Object-Relational DBMS, which supports queries containing user-defined types and user-defined functions.

The reader is expected to be familiar with the concepts covered in Section 4.1 on the relational algebra and Appendix C on file organizations. The examples in this chapter are drawn from the *DreamHome* case study described in Section 10.4 and Appendix A.

21.1 Overview of Query Processing

Query processing	The activities involved in parsing, validating, optimizing, and executing a query.

The aims of query processing are to transform a query written in a high-level language, typically SQL, into a correct and efficient execution strategy expressed in a low-level language (implementing the relational algebra), and to execute the strategy to retrieve the required data.

Query optimization	The activity of choosing an efficient execution strategy for processing a query.

An important aspect of query processing is query optimization. As there are many equivalent transformations of the same high-level query, the aim of query optimization is to choose the one that minimizes resource usage. Generally, we try to reduce the total execution time of the query, which is the sum of the execution times of all individual operations that make up the query (Selinger *et al.*, 1979). However, resource usage may also be viewed as the response time of the query, in which case we concentrate on maximizing the number of parallel operations (Valduriez and Gardarin, 1984). Since the problem is computationally intractable with a large number of relations, the strategy adopted is generally reduced to finding a near optimum solution (Ibaraki and Kameda, 1984).

Both methods of query optimization depend on database statistics to evaluate properly the different options that are available. The accuracy and currency of these statistics have a significant bearing on the efficiency of the execution strategy chosen. The statistics cover information about relations, attributes, and indexes. For example, the system catalog may store statistics giving the cardinality of relations, the number of distinct values for each attribute, and the number of levels in a multilevel index (see Appendix C.5.4). Keeping the statistics current can be problematic. If the DBMS updates the statistics every time a tuple is inserted, updated, or deleted, this would have a significant impact on performance during peak periods. An alternative, and generally preferable, approach is to update the statistics on a periodic basis, for example nightly, or whenever the system is idle. Another approach taken by some systems is to make it the users' responsibility to indicate when the statistics are to be updated. We discuss database statistics in more detail in Section 21.4.1.

As an illustration of the effects of different processing strategies on resource usage, we start with an example.

Example 21.1 Comparison of different processing strategies

Find all Managers who work at a London branch.

We can write this query in SQL as:

```
SELECT *
FROM Staff s, Branch b
WHERE s.branchNo = b.branchNo AND
      (s.position = 'Manager' AND b.city = 'London');
```

Three equivalent relational algebra queries corresponding to this SQL statement are:

(1) $\sigma_{(position='Manager') \wedge (city='London') \wedge (Staff.branchNo=Branch.branchNo)}(Staff \times Branch)$

(2) $\sigma_{(position='Manager') \wedge (city='London')}(Staff \bowtie_{Staff.branchNo=Branch.branchNo} Branch)$

(3) $(\sigma_{position='Manager'}(Staff)) \bowtie_{Staff.branchNo=Branch.branchNo} (\sigma_{city='London'}(Branch))$

For the purposes of this example, we assume that there are 1000 tuples in Staff, 50 tuples in Branch, 50 Managers (one for each branch), and 5 London branches. We compare these three queries based on the number of disk accesses required. For simplicity, we assume that there are no indexes or sort keys on either relation, and that the results of any intermediate operations are stored on disk. The cost of the final write is ignored, as it is the same in each case. We further assume that tuples are accessed one at a time (although in practice disk accesses would be based on blocks, which would typically contain several tuples), and main memory is large enough to process entire relations for each relational algebra operation.

The first query calculates the Cartesian product of Staff and Branch, which requires (1000 + 50) disk accesses to read the relations, and creates a relation with (1000 * 50) tuples. We then have to read each of these tuples again to test them against the selection predicate at a cost of another (1000 * 50) disk accesses, giving a total cost of:

(1000 + 50) + 2*(1000 * 50) = 101 050 disk accesses

The second query joins Staff and Branch on the branch number branchNo, which again requires (1000 + 50) disk accesses to read each of the relations. We know that the join of the two relations has 1000 tuples, one for each member of staff (a member of staff can only work at one branch). Consequently, the Selection operation requires 1000 disk accesses to read the result of the join, giving a total cost of:

2*1000 + (1000 + 50) = 3050 disk accesses

The final query first reads each Staff tuple to determine the Manager tuples, which requires 1000 disk accesses and produces a relation with 50 tuples. The second Selection operation reads each Branch tuple to determine the London branches, which requires 50 disk accesses and produces a relation with 5 tuples. The final operation is the join of the reduced Staff and Branch relations, which requires (50 + 5) disk accesses, giving a total cost of:

1000 + 2*50 + 5 + (50 + 5) = 1160 disk accesses

Clearly the third option is the best in this case, by a factor of 87:1. If we increased the number of tuples in Staff to 10 000 and the number of branches to 500, the improvement would be by a factor of approximately 870:1. Intuitively, we may have expected this as the Cartesian product and Join operations are much more expensive than the Selection operation, and the third option significantly reduces the size of the relations that are being joined together. We will see shortly that one of the fundamental strategies in query processing is to perform the unary operations, Selection and Projection, as early as possible, thereby reducing the operands of any subsequent binary operations.

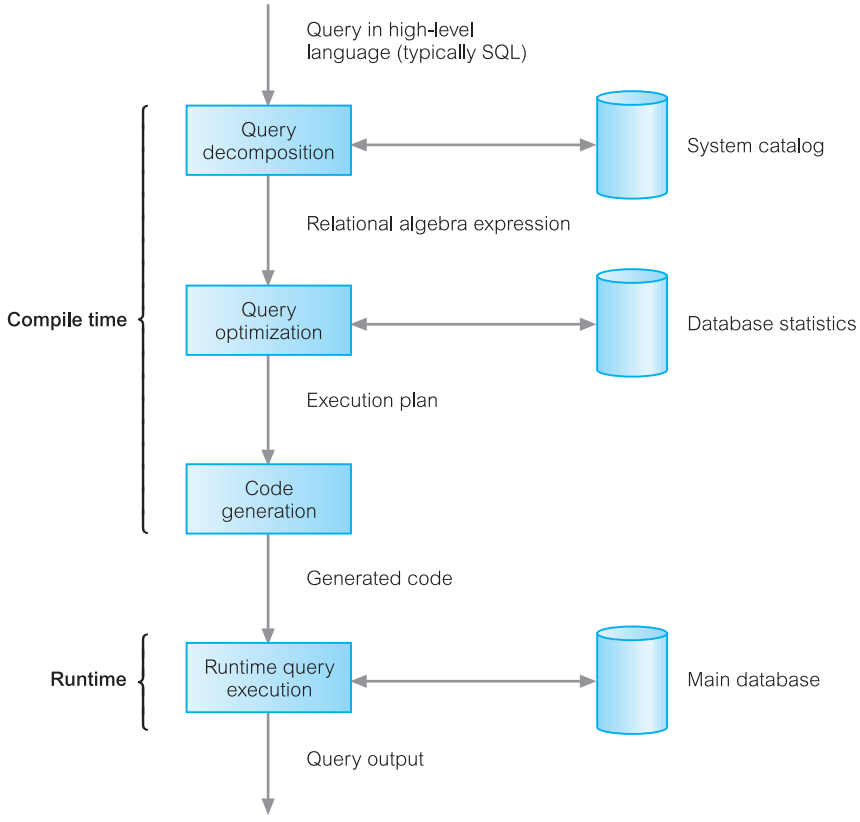


Figure 21.1
Phases of query processing.

Query processing can be divided into four main phases: decomposition (consisting of parsing and validation), optimization, code generation, and execution, as illustrated in Figure 21.1. In Section 21.2 we briefly examine the first phase, decomposition, before turning our attention to the second phase, query optimization. To complete this overview, we briefly discuss when optimization may be performed.

Dynamic versus static optimization

There are two choices for when the first three phases of query processing can be carried out. One option is to dynamically carry out decomposition and optimization every time the query is run. The advantage of *dynamic query optimization* arises from the fact that all information required to select an optimum strategy is up to date. The disadvantages are that the performance of the query is affected because the query has to be parsed, validated, and optimized before it can be executed. Further, it may be necessary to reduce the number of execution strategies to be analyzed to achieve an acceptable overhead, which may have the effect of selecting a less than optimum strategy.

The alternative option is *static query optimization*, where the query is parsed, validated, and optimized once. This approach is similar to the approach taken by a compiler for a programming language. The advantages of static optimization are that the runtime

overhead is removed, and there may be more time available to evaluate a larger number of execution strategies, thereby increasing the chances of finding a more optimum strategy. For queries that are executed many times, taking some additional time to find a more optimum plan may prove to be highly beneficial. The disadvantages arise from the fact that the execution strategy that is chosen as being optimal when the query is compiled may no longer be optimal when the query is run. However, a hybrid approach could be used to overcome this disadvantage, where the query is re-optimized if the system detects that the database statistics have changed significantly since the query was last compiled. Alternatively, the system could compile the query for the first execution in each session, and then cache the optimum plan for the remainder of the session, so the cost is spread across the entire DBMS session.

21.2 Query Decomposition

Query decomposition is the first phase of query processing. The aims of query decomposition are to transform a high-level query into a relational algebra query, and to check that the query is syntactically and semantically correct. The typical stages of query decomposition are analysis, normalization, semantic analysis, simplification, and query restructuring.

(1) Analysis

In this stage, the query is lexically and syntactically analyzed using the techniques of programming language compilers (see, for example, Aho and Ullman, 1977). In addition, this stage verifies that the relations and attributes specified in the query are defined in the system catalog. It also verifies that any operations applied to database objects are appropriate for the object type. For example, consider the following query:

```
SELECT staffNumber
FROM Staff
WHERE position > 10;
```

This query would be rejected on two grounds:

(1) In the select list, the attribute staffNumber is not defined for the Staff relation (should be staffNo).

(2) In the WHERE clause, the comparison '>10' is incompatible with the data type position, which is a variable character string.

On completion of this stage, the high-level query has been transformed into some internal representation that is more suitable for processing. The internal form that is typically chosen is some kind of query tree, which is constructed as follows:

- A leaf node is created for each base relation in the query.
- A non-leaf node is created for each intermediate relation produced by a relational algebra operation.
- The root of the tree represents the result of the query.
- The sequence of operations is directed from the leaves to the root.

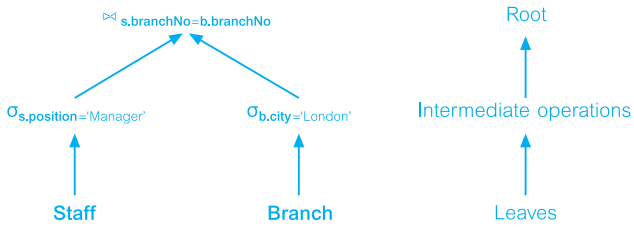


Figure 21.2
Example relational algebra tree.

Figure 21.2 shows an example of a query tree for the SQL statement of Example 21.1 that uses the relational algebra in its internal representation. We refer to this type of query tree as a **relational algebra tree**.

(2) Normalization

The normalization stage of query processing converts the query into a normalized form that can be more easily manipulated. The predicate (in SQL, the WHERE condition), which may be arbitrarily complex, can be converted into one of two forms by applying a few transformation rules (Jarke and Koch, 1984):

- *Conjunctive normal form* A sequence of conjuncts that are connected with the ∧ (AND) operator. Each conjunct contains one or more terms connected by the ∨ (OR) operator. For example:

  (position = 'Manager' ∨ salary > 20000) ∧ branchNo = 'B003'

  A conjunctive selection contains only those tuples that satisfy all conjuncts.
- *Disjunctive normal form* A sequence of disjuncts that are connected with the ∨ (OR) operator. Each disjunct contains one or more terms connected by the ∧ (AND) operator. For example, we could rewrite the above conjunctive normal form as:

  (position = 'Manager' ∧ branchNo = 'B003') ∨ (salary > 20000 ∧ branchNo = 'B003')

  A disjunctive selection contains those tuples formed by the union of all tuples that satisfy the disjuncts.

(3) Semantic analysis

The objective of semantic analysis is to reject normalized queries that are incorrectly formulated or contradictory. A query is incorrectly formulated if components do not contribute to the generation of the result, which may happen if some join specifications are missing. A query is contradictory if its predicate cannot be satisfied by any tuple. For example, the predicate (position = 'Manager' ∧ position = 'Assistant') on the Staff relation is contradictory, as a member of staff cannot be both a Manager and an Assistant simultaneously. However, the predicate ((position = 'Manager' ∧ position = 'Assistant') ∨ salary > 20000) could be simplified to (salary > 20000) by interpreting the contradictory clause

as the boolean value FALSE. Unfortunately, the handling of contradictory clauses is not consistent between DBMSs.

Algorithms to determine correctness exist only for the subset of queries that do not contain disjunction and negation. For these queries, we could apply the following checks:

(1) Construct a *relation connection graph* (Wong and Youssefi, 1976). If the graph is not connected, the query is incorrectly formulated. To construct a relation connection graph, we create a node for each relation and a node for the result. We then create edges between two nodes that represent a join, and edges between nodes that represent the source of Projection operations.

(2) Construct a *normalized attribute connection graph* (Rosenkrantz and Hunt, 1980). If the graph has a cycle for which the valuation sum is negative, the query is contradictory. To construct a normalized attribute connection graph, we create a node for each reference to an attribute, or constant 0. We then create a directed edge between nodes that represent a join, and a directed edge between an attribute node and a constant 0 node that represents a Selection operation. Next, we weight the edges a $\rightarrow$ b with the value c, if it represents the inequality condition (a $\leq$ b + c), and weight the edges 0 $\rightarrow$ a with the value $-c$, if it represents the inequality condition (a $\geq$ c).

Example 21.2 Checking semantic correctness

Consider the following SQL query:

```
SELECT p.propertyNo, p.street
FROM Client c, Viewing v, PropertyForRent p
WHERE c.clientNo = v.clientNo AND
      c.maxRent >= 500 AND c.prefType = 'Flat' AND p.ownerNo = 'CO93';
```

The relation connection graph shown in Figure 21.3(a) is not fully connected, implying that the query is not correctly formulated. In this case, we have omitted the join condition (v.propertyNo = p.propertyNo) from the predicate.

Now consider the query:

```
SELECT p.propertyNo, p.street
FROM Client c, Viewing v, PropertyForRent p
WHERE c.maxRent > 500 AND c.clientNo = v.clientNo AND
      v.propertyNo = p.propertyNo AND c.prefType = 'Flat' AND c.maxRent < 200;
```

The normalized attribute connection graph for this query shown in Figure 21.3(b) has a cycle between the nodes c.maxRent and 0 with a negative valuation sum, which indicates that the query is contradictory. Clearly, we cannot have a client with a maximum rent that is both greater than £500 and less than £200.

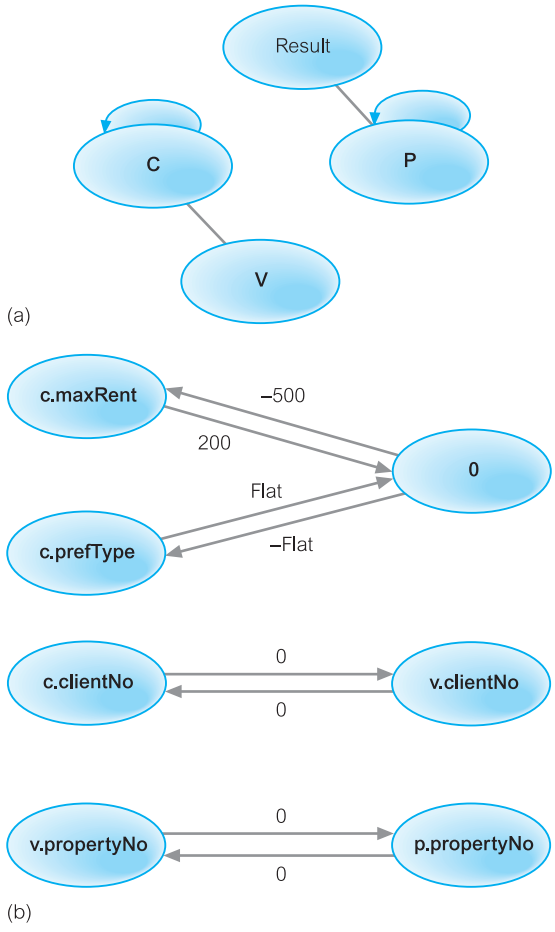


Figure 21.3
(a) Relation connection graph showing query is incorrectly formulated; (b) normalized attribute connection graph showing query is contradictory.

(4) Simplification

The objectives of the simplification stage are to detect redundant qualifications, eliminate common subexpressions, and transform the query to a semantically equivalent but more easily and efficiently computed form. Typically, access restrictions, view definitions, and integrity constraints are considered at this stage, some of which may also introduce redundancy. If the user does not have the appropriate access to all the components of the query, the query must be rejected. Assuming that the user has the appropriate access privileges, an initial optimization is to apply the well-known idempotency rules of boolean algebra, such as:

$p \wedge (p) \equiv p$	$p \vee (p) \equiv p$
$p \wedge false \equiv false$	$p \vee false \equiv p$
$p \wedge true \equiv p$	$p \vee true \equiv true$
$p \wedge (\sim p) \equiv false$	$p \vee (\sim p) \equiv true$
$p \wedge (p \vee q) \equiv p$	$p \vee (p \wedge q) \equiv p$

For example, consider the following view definition and query on the view:

```
CREATE VIEW Staff3 AS
SELECT staffNo, fName, lName, salary, branchNo
FROM Staff
WHERE branchNo = 'B003';

SELECT *
FROM Staff3
WHERE (branchNo = 'B003' AND
       salary > 20000);
```

As discussed in Section 6.4.3, during view resolution this query will become:

```
SELECT staffNo, fName, lName, salary, branchNo
FROM Staff
WHERE (branchNo = 'B003' AND salary > 20000) AND branchNo = 'B003';
```

and the WHERE condition reduces to (branchNo = 'B003' **AND** salary > 20000).

Integrity constraints may also be applied to help simplify queries. For example, consider the following integrity constraint, which ensures that only Managers have a salary greater than £20,000:

```
CREATE ASSERTION OnlyManagerSalaryHigh
CHECK ((position <> 'Manager' AND salary < 20000)
OR (position = 'Manager' AND salary > 20000));
```

and consider the effect on the query:

```
SELECT *
FROM Staff
WHERE (position = 'Manager' AND salary < 15000);
```

The predicate in the WHERE clause, which searches for a manager with a salary below £15,000, is now a contradiction of the integrity constraint so there can be no tuples that satisfy this predicate.

(5) Query restructuring

In the final stage of query decomposition, the query is restructured to provide a more efficient implementation. We consider restructuring further in the next section.

21.3 Heuristical Approach to Query Optimization

In this section we look at the heuristical approach to query optimization, which uses transformation rules to convert one relational algebra expression into an equivalent form that is known to be more efficient. For example, in Example 21.1 we observed that it was more efficient to perform the Selection operation on a relation before using that relation in a Join, rather than perform the Join and then the Selection operation. We will see in Section 21.3.1 that there is a transformation rule allowing the order of Join and Selection operations to be changed so that Selection can be performed first. Having discussed what transformations are valid, in Section 21.3.2 we present a set of heuristics that are known to produce 'good' (although not necessarily optimum) execution strategies.

21.3.1 Transformation Rules for the Relational Algebra Operations

By applying transformation rules, the optimizer can transform one relational algebra expression into an equivalent expression that is known to be more efficient. We will use these rules to restructure the (canonical) relational algebra tree generated during query decomposition. Proofs of the rules can be found in Aho *et al.* (1979). In listing these rules, we use three relations R, S, and T, with R defined over the attributes $A = \{A_1, A_2, \ldots, A_n\}$, and S defined over $B = \{B_1, B_2, \ldots, B_n\}$; p, q, and r denote predicates, and L, L_1, L_2, M, M_1, M_2, and N denote sets of attributes.

(1) Conjunctive Selection operations can cascade into individual Selection operations (and vice versa).

$$\sigma_{p \wedge q \wedge r}(R) = \sigma_p(\sigma_q(\sigma_r(R)))$$

This transformation is sometimes referred to as *cascade of selection*. For example:

$$\sigma_{branchNo='B003' \wedge salary>15000}(Staff) = \sigma_{branchNo='B003'}(\sigma_{salary>15000}(Staff))$$

(2) Commutativity of Selection operations.

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

For example:

$$\sigma_{branchNo='B003'}(\sigma_{salary>15000}(Staff)) = \sigma_{salary>15000}(\sigma_{branchNo='B003'}(Staff))$$

(3) In a sequence of Projection operations, only the last in the sequence is required.

$$\Pi_L \Pi_M \ldots \Pi_N(R) = \Pi_L(R)$$

For example:

$$\Pi_{lName} \Pi_{branchNo, lName}(Staff) = \Pi_{lName}(Staff)$$

(4) Commutativity of Selection and Projection.
If the predicate p involves only the attributes in the projection list, then the Selection and Projection operations commute:

$$\Pi_{A_1, \ldots, A_m}(\sigma_p(R)) = \sigma_p(\Pi_{A_1, \ldots, A_m}(R)) \qquad \text{where } p \in \{A_1, A_2, \ldots, A_m\}$$

For example:

$$\Pi_{fName, lName}(\sigma_{lName='Beech'}(Staff)) = \sigma_{lName='Beech'}(\Pi_{fName, lName}(Staff))$$

(5) Commutativity of Theta join (and Cartesian product).

$$R \bowtie_p S = S \bowtie_p R$$

$$R \times S = S \times R$$

As the Equijoin and Natural join are special cases of the Theta join, then this rule also applies to these Join operations. For example, using the Equijoin of Staff and Branch:

$$\text{Staff} \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} \text{Branch} = \text{Branch} \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} \text{Staff}$$

(6) Commutativity of Selection and Theta join (or Cartesian product).
If the selection predicate involves only attributes of one of the relations being joined, then the Selection and Join (or Cartesian product) operations commute:

$$\sigma_p(R \bowtie_r S) = (\sigma_p(R)) \bowtie_r S$$

$$\sigma_p(R \times S) = (\sigma_p(R)) \times S \qquad \text{where } p \in \{A_1, A_2, \ldots, A_n\}$$

Alternatively, if the selection predicate is a conjunctive predicate of the form $(p \wedge q)$, where p involves only attributes of R, and q involves only attributes of S, then the Selection and Theta join operations commute as:

$$\sigma_{p \wedge q}(R \bowtie_r S) = (\sigma_p(R)) \bowtie_r (\sigma_q(S))$$

$$\sigma_{p \wedge q}(R \times S) = (\sigma_p(R)) \times (\sigma_q(S))$$

For example:

$$\sigma_{\text{position}=\text{'Manager'} \wedge \text{city}=\text{'London'}}(\text{Staff} \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} \text{Branch}) =$$

$$(\sigma_{\text{position}=\text{'Manager'}}(\text{Staff})) \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} (\sigma_{\text{city}=\text{'London'}}(\text{Branch}))$$

(7) Commutativity of Projection and Theta join (or Cartesian product).
If the projection list is of the form $L = L_1 \cup L_2$, where L_1 involves only attributes of R, and L_2 involves only attributes of S, then provided the join condition only contains attributes of L, the Projection and Theta join operations commute as:

$$\Pi_{L_1 \cup L_2}(R \bowtie_r S) = (\Pi_{L_1}(R)) \bowtie_r (\Pi_{L_2}(S))$$

For example:

$$\Pi_{\text{position, city, branchNo}}(\text{Staff} \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} \text{Branch}) =$$

$$(\Pi_{\text{position, branchNo}}(\text{Staff})) \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} (\Pi_{\text{city, branchNo}}(\text{Branch}))$$

If the join condition contains additional attributes not in L, say attributes $M = M_1 \cup M_2$ where M_1 involves only attributes of R, and M_2 involves only attributes of S, then a final Projection operation is required:

$$\Pi_{L_1 \cup L_2}(R \bowtie_r S) = \Pi_{L_1 \cup L_2}(\Pi_{L_1 \cup M_1}(R)) \bowtie_r (\Pi_{L_2 \cup M_2}(S))$$

For example:

$$\Pi_{\text{position, city}}(\text{Staff} \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} \text{Branch}) =$$

$$\Pi_{\text{position, city}}((\Pi_{\text{position, branchNo}}(\text{Staff})) \bowtie_{\text{Staff.branchNo}=\text{Branch.branchNo}} (\Pi_{\text{city, branchNo}}(\text{Branch})))$$

(8) Commutativity of Union and Intersection (but not Set difference).

$$R \cup S = S \cup R$$

$$R \cap S = S \cap R$$

(9) Commutativity of Selection and set operations (Union, Intersection, and Set difference).

$$\sigma_p(R \cup S) = \sigma_p(S) \cup \sigma_p(R)$$

$$\sigma_p(R \cap S) = \sigma_p(S) \cap \sigma_p(R)$$

$$\sigma_p(R - S) = \sigma_p(S) - \sigma_p(R)$$

(10) Commutativity of Projection and Union.

$$\Pi_L(R \cup S) = \Pi_L(S) \cup \Pi_L(R)$$

(11) Associativity of Theta join (and Cartesian product).
Cartesian product and Natural join are always associative:

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

$$(R \times S) \times T = R \times (S \times T)$$

If the join condition q involves only attributes from the relations S and T, then Theta join is associative in the following manner:

$$(R \bowtie_p S) \bowtie_{q \wedge r} T = R \bowtie_{p \wedge r} (S \bowtie_q T)$$

For example:

$$(\text{Staff} \bowtie_{\text{Staff.staffNo}=\text{PropertyForRent.staffNo}} \text{PropertyForRent}) \bowtie_{\text{ownerNo}=\text{Owner.ownerNo} \wedge \text{Staff.lName}=\text{Owner.lName}} \text{Owner}$$

$$= \text{Staff} \bowtie_{\text{Staff.staffNo}=\text{PropertyForRent.staffNo} \wedge \text{Staff.lName}=\text{lName}} (\text{PropertyForRent} \bowtie_{\text{ownerNo}} \text{Owner})$$

Note that in this example it would be incorrect simply to 'move the brackets' as this would result in an undefined reference (Staff.lName) in the join condition between PropertyForRent and Owner:

$$\text{PropertyForRent} \bowtie_{\text{PropertyForRent.ownerNo}=\text{Owner.ownerNo} \wedge \text{Staff.lName}=\text{Owner.lName}} \text{Owner}$$

(12) Associativity of Union and Intersection (but not Set difference).

$$(R \cup S) \cup T = S \cup (R \cup T)$$

$$(R \cap S) \cap T = S \cap (R \cap T)$$

Example 21.3 Use of transformation rules

For prospective renters who are looking for flats, find the properties that match their requirements and are owned by owner CO93.

We can write this query in SQL as:

```
SELECT p.propertyNo, p.street
FROM Client c, Viewing v, PropertyForRent p
WHERE c.prefType = 'Flat' AND c.clientNo = v.clientNo AND
      v.propertyNo = p.propertyNo AND c.maxRent >= p.rent AND
      c.prefType = p.type AND p.ownerNo = 'CO93';
```

For the purposes of this example we will assume that there are fewer properties owned by owner CO93 than prospective renters who have specified a preferred property type of Flat. Converting the SQL to relational algebra, we have:

$$\Pi_{p.propertyNo,\ p.street}(\sigma_{c.prefType='Flat' \wedge c.clientNo=v.clientNo \wedge v.propertyNo=p.propertyNo \wedge c.maxRent>=p.rent \wedge c.prefType=p.type \wedge p.ownerNo='CO93'}((c \times v) \times p))$$

We can represent this query as the canonical relational algebra tree shown in Figure 21.4(a). We now use the following transformation rules to improve the efficiency of the execution strategy:

(1) (a) Rule 1, to split the conjunction of Selection operations into individual Selection operations.
    (b) Rule 2 and Rule 6, to reorder the Selection operations and then commute the Selections and Cartesian products.

    The result of these first two steps is shown in Figure 21.4(b).

(2) From Section 4.1.3, we can rewrite a Selection with an Equijoin predicate and a Cartesian product operation, as an Equijoin operation; that is:

$$\sigma_{R.a=S.b}(R \times S) = R \bowtie_{R.a=S.b} S$$

    Apply this transformation where appropriate. The result of this step is shown in Figure 21.4(c).

(3) Rule 11, to reorder the Equijoins, so that the more restrictive selection on (p.ownerNo = 'CO93') is performed first, as shown in Figure 21.4(d).

(4) Rules 4 and 7, to move the Projections down past the Equijoins, and create new Projection operations as required. The result of applying these rules is shown in Figure 21.4(e).

An additional optimization in this particular example is to note that the Selection operation (c.prefType=p.type) can be reduced to (p.type = 'Flat'), as we know that (c.prefType='Flat') from the first clause in the predicate. Using this substitution, we push this Selection down the tree, resulting in the final reduced relational algebra tree shown in Figure 21.4(f).

Figure 21.4 Relational algebra tree for Example 21.3: (a) canonical relational algebra tree; (b) relational algebra tree formed by pushing Selections down; (c) relational algebra tree formed by changing Selection/Cartesian products to Equijoins; (d) relational algebra tree formed using associativity of Equijoins; (e) relational algebra tree formed by pushing Projections down; (f) final reduced relational algebra tree formed by substituting c.prefType = 'Flat' in Selection on p.type and pushing resulting Selection down tree.

21.3.2 Heuristical Processing Strategies

Many DBMSs use heuristics to determine strategies for query processing. In this section we examine some good heuristics that could be applied during query processing.

(1) Perform Selection operations as early as possible.
Selection reduces the cardinality of the relation and reduces the subsequent processing of that relation. Therefore, we should use rule 1 to cascade the Selection operations, and rules 2, 4, 6, and 9 regarding commutativity of Selection with unary and binary operations, to move the Selection operations as far down the tree as possible. Keep selection predicates on the same relation together.

(2) Combine the Cartesian product with a subsequent Selection operation whose predicate represents a join condition into a Join operation.
We have already noted that we can rewrite a Selection with a Theta join predicate and a Cartesian product operation as a Theta join operation:

$$\sigma_{R.a\ \theta\ S.b}(R \times S) = R \bowtie_{R.a\ \theta\ S.b} S$$

(3) Use associativity of binary operations to rearrange leaf nodes so that the leaf nodes with the most restrictive Selection operations are executed first.
Again, our general rule of thumb is to perform as much reduction as possible before performing binary operations. Thus, if we have two consecutive Join operations to perform:

$$(R \bowtie_{R.a\ \theta\ S.b} S) \bowtie_{S.c\ \theta\ T.d} T$$

then we should use rules 11 and 12 concerning associativity of Theta join (and Union and Intersection) to reorder the operations so that the relations resulting in the smaller join is performed first, which means that the second join will also be based on a smaller first operand.

(4) Perform Projection operations as early as possible.
Again, Projection reduces the cardinality of the relation and reduces the subsequent processing of that relation. Therefore, we should use rule 3 to cascade the Projection operations, and rules 4, 7, and 10 regarding commutativity of Projection with binary operations, to move the Projection operations as far down the tree as possible. Keep projection attributes on the same relation together.

(5) Compute common expressions once.
If a common expression appears more than once in the tree, and the result it produces is not too large, store the result after it has been computed once and then reuse it when required. This is only beneficial if the size of the result from the common expression is small enough to either be stored in main memory or accessed from secondary storage at a cost less than that of recomputing it. This can be especially useful when querying views, since the same expression must be used to construct the view each time.

In Section 23.7 we show how these heuristics can be applied to distributed queries. In Section 28.5 we will see that some of these heuristics may require further consideration for the Object-Relational DBMS, which supports queries containing user-defined types and user-defined functions.

21.4 Cost Estimation for the Relational Algebra Operations

A DBMS may have many different ways of implementing the relational algebra operations. The aim of query optimization is to choose the most efficient one. To do this, it uses formulae that estimate the costs for a number of options and selects the one with the lowest cost. In this section we examine the different options available for implementing the main relational algebra operations. For each one, we provide an overview of the implementation and give an estimated cost. As the dominant cost in query processing is usually that of disk accesses, which are slow compared with memory accesses, we concentrate exclusively on the cost of disk accesses in the estimates provided. Each estimate represents the required number of disk block accesses, excluding the cost of writing the result relation.

Many of the cost estimates are based on the cardinality of the relation. Therefore, as we need to be able to estimate the cardinality of intermediate relations, we also show some typical estimates that can be derived for such cardinalities. We start this section by examining the types of statistics that the DBMS will store in the system catalog to help with cost estimation.

21.4.1 Database Statistics

The success of estimating the size and cost of intermediate relational algebra operations depends on the amount and currency of the statistical information that the DBMS holds. Typically, we would expect a DBMS to hold the following types of information in its system catalog:

For each base relation R

- nTuples(R) – the number of tuples (records) in relation R (that is, its cardinality).
- bFactor(R) – the blocking factor of R (that is, the number of tuples of R that fit into one block).
- nBlocks(R) – the number of blocks required to store R. If the tuples of R are stored physically together, then:

  $$nBlocks(R) = [nTuples(R)/bFactor(R)]$$

  We use $[x]$ to indicate that the result of the calculation is rounded to the smallest integer that is greater than or equal to x.

For each attribute A of base relation R

- $nDistinct_A(R)$ – the number of distinct values that appear for attribute A in relation R.
- $min_A(R)$,$max_A(R)$ – the minimum and maximum possible values for the attribute A in relation R.
- $SC_A(R)$ – the *selection cardinality* of attribute A in relation R. This is the average number of tuples that satisfy an equality condition on attribute A. If we assume that the values of A are uniformly distributed in R, and that there is at least one value that satisfies the condition, then:

$$SC_A(R) = \begin{cases} 1 & \text{if A is a key attribute of R} \\ [nTuples(R)/nDistinct_A(R)] & \text{otherwise} \end{cases}$$

We can also estimate the selection cardinality for other conditions:

$$SC_A(R) = \begin{cases} [nTuples(R)*((max_A(R) - c)/(max_A(R) - min_A(R))] & \text{for inequality } (A > c) \\ [nTuples(R)*((c - max_A(R))/(max_A(R) - min_A(R))] & \text{for inequality } (A < c) \\ [(nTuples(R)/nDistinct_A(R))*n] & \text{for A in } \{c_1, c_2, \ldots, c_n\} \\ SC_A(R)*SC_B(R) & \text{for } (A \wedge B) \\ SC_A(R) + SC_B(R) - SC_A(R)*SC_B(R) & \text{for } (A \vee B) \end{cases}$$

For each multilevel index I on attribute set A

- $nLevels_A(I)$ – the number of levels in I.
- $nLfBlocks_A(I)$ – the number of leaf blocks in I.

Keeping these statistics current can be problematic. If the DBMS updates the statistics every time a tuple is inserted, updated, or deleted, at peak times this would have a significant impact on performance. An alternative, and generally preferable, approach is for the DBMS to update the statistics on a periodic basis, for example nightly or whenever the system is idle. Another approach taken by some systems is to make it the users' responsibility to indicate that the statistics should be updated.

Selection Operation ($S = \sigma_p(R)$) 21.4.2

As we have seen in Section 4.1.1, the Selection operation in the relational algebra works on a single relation R, say, and defines a relation S containing only those tuples of R that satisfy the specified predicate. The predicate may be simple, involving the comparison of an attribute of R with either a constant value or another attribute value. The predicate may also be composite, involving more than one condition, with conditions combined using the logical connectives $\wedge$ (AND), $\vee$ (OR), and $\sim$ (NOT). There are a number of different implementations for the Selection operation, depending on the structure of the file in which the relation is stored, and on whether the attribute(s) involved in the predicate have been indexed/hashed. The main strategies that we consider are:

- linear search (unordered file, no index);
- binary search (ordered file, no index);
- equality on hash key;
- equality condition on primary key;
- inequality condition on primary key;
- equality condition on clustering (secondary) index;
- equality condition on a non-clustering (secondary) index;
- inequality condition on a secondary B^+-tree index.

The costs for each of these strategies are summarized in Table 21.1.

Table 21.1 Summary of estimated I/O cost of strategies for Selection operation.

Strategies	Cost
Linear search (unordered file, no index)	[nBlocks(R)/2], for equality condition on key attribute
	nBlocks(R), otherwise
Binary search (ordered file, no index)	$[\log_2(\text{nBlocks}(R))]$, for equality condition on ordered attribute
	$[\log_2(\text{nBlocks}(R))] + [\text{SC}_A(R)/\text{bFactor}(R)] - 1$, otherwise
Equality on hash key	1, assuming no overflow
Equality condition on primary key	$\text{nLevels}_A(I) + 1$
Inequality condition on primary key	$\text{nLevels}_A(I) + [\text{nBlocks}(R)/2]$
Equality condition on clustering (secondary) index	$\text{nLevels}_A(I) + [\text{SC}_A(R)/\text{bFactor}(R)]$
Equality condition on a non-clustering (secondary) index	$\text{nLevels}_A(I) + [\text{SC}_A(R)]$
Inequality condition on a secondary B^+-tree index	$\text{nLevels}_A(I) + [\text{nLfBlocks}_A(I)/2 + \text{nTuples}(R)/2]$

Estimating the cardinality of the Selection operation

Before we consider these options, we first present estimates for the expected number of tuples and the expected number of distinct values for an attribute in the result relation S obtained from the Selection operation on R. Generally it is quite difficult to provide accurate estimates. However, if we assume the traditional simplifying assumptions that attribute values are uniformly distributed within their domain and that attributes are independent, we can use the following estimates:

$$\text{nTuples}(S) = \text{SC}_A(R) \qquad \text{predicate } p \text{ is of the form } (A\ \theta\ x)$$

For any attribute $B \neq A$ of S:

$$\text{nDistinct}_B(S) = \begin{cases} \text{nTuples}(S) & \text{if nTuples}(S) < \text{nDistinct}_B(R)/2 \\ [(\text{nTuples}(S) + \text{nDistinct}_B(R))/3] & \text{if nDistinct}_B(R)/2 \leq \text{nTuples}(S) \leq 2^* \text{nDistinct}_B(R) \\ \text{nDistinct}_B(R) & \text{if nTuples}(S) > 2^*\text{nDistinct}_B(R) \end{cases}$$

It is possible to derive more accurate estimates where we relax the assumption of uniform distribution, but this requires the use of more detailed statistical information, such as histograms and distribution steps (Piatetsky-Shapiro and Connell, 1984). We briefly discuss how Oracle uses histograms in Section 21.6.2.

(1) Linear search (unordered file, no index)

With this approach, it may be necessary to scan each tuple in each block to determine whether it satisfies the predicate, as illustrated in the outline algorithm shown in Figure 21.5. This is sometimes referred to as a *full table scan*. In the case of an equality condition on a

```
//
// Linear search
// Predicate is the search key.
// File is unordered. Blocks are numbered sequentially from 1.
// Returns a result table containing those tuples of R that match predicate.
//
for i = 1 to nBlocks(R) {                          // loop over each block
      block = read_block(R, i);
      for j = 1 to nTuples(block) {                // loop over each tuple in block i
            if (block.tuple[j] satisfies predicate)
            then add tuple to result;
      }
}
```

Figure 21.5
Algorithm for linear search.

key attribute, assuming tuples are uniformly distributed about the file, then on average only half the blocks would be searched before the specific tuple is found, so the cost estimate is:

$$[\text{nBlocks}(R)/2]$$

For any other condition, the entire file may need to be searched, so the more general cost estimate is:

$$\text{nBlocks}(R)$$

(2) Binary search (ordered file, no index)

If the predicate is of the form $(A = x)$ and the file is ordered on attribute A, which is also the key attribute of relation R, then the cost estimate for the search is:

$$[\log_2(\text{nBlocks}(R))]$$

The algorithm for this type of search is outlined in Figure 21.6. More generally, the cost estimate is:

$$[\log_2(\text{nBlocks}(R))] + [SC_A(R)/\text{bFactor}(R)] - 1$$

The first term represents the cost of finding the first tuple using a binary search method. We expect there to be $SC_A(R)$ tuples satisfying the predicate, which will occupy $[SC_A(R)/\text{bFactor}(R)]$ blocks, of which one has been retrieved in finding the first tuple.

(3) Equality on hash key

If attribute A is the hash key, then we apply the hashing algorithm to calculate the target address for the tuple. If there is no overflow, the expected cost is 1. If there is overflow, additional accesses may be necessary, depending on the amount of overflow and the method for handling overflow.

Figure 21.6
Algorithm for binary search on an ordered file.

```
//
// Binary search
// Predicate is the search key.
// File is ordered in ascending value of the ordering key field, A.
// The file occupies nBlocks blocks, numbered sequentially from 1.
// Returns a boolean variable (found) indicating whether a record has been found that
// matches predicate, and a result table, if found.
//
next = 1; last = nBlocks; found = FALSE; keep_searching = TRUE;
while (last >= 1 and (not found) and (keep_searching)) {
      i = (next + last)/2;                          // half the search space
      block = read_block(R, i) ;
      if (predicate < ordering_key_field(first_record(block)))
      then                                          // record is in bottom half of search area
            last = i – 1;
      else if (predicate > ordering_key_field(last_record(block)))
            then                                    // record is in top half of search area
                  next = i + 1;
            else if (check_block_for_predicate(block, predicate, result))
                  then                              // required record is in the block
                        found = TRUE;
                  else                              // record not there
                        keep_searching = FALSE;
}
```

(4) Equality condition on primary key

If the predicate involves an equality condition on the primary key field (A = x), then we can use the primary index to retrieve the single tuple that satisfies this condition. In this case, we need to read one more block than the number of index accesses, equivalent to the number of levels in the index, and so the estimated cost is:

$$\text{nLevels}_{A}(I) + 1$$

(5) Inequality condition on primary key

If the predicate involves an inequality condition on the primary key field A (A < x, A <= x, A > x, A >= x), then we can first use the index to locate the tuple satisfying the predicate A = x. Provided the index is sorted, then the required tuples can be found by accessing all tuples before or after this one. Assuming uniform distribution, then we would expect half the tuples to satisfy the inequality, so the estimated cost is:

$$\text{nLevels}_{A}(I) + [\text{nBlocks}(R)/2]$$

(6) Equality condition on clustering (secondary) index

If the predicate involves an equality condition on attribute A, which is not the primary key but does provide a clustering secondary index, then we can use the index to retrieve the required tuples. The estimated cost is:

$$nLevels_A(I) + [SC_A(R)/bFactor(R)]$$

The second term is an estimate of the number of blocks that will be required to store the number of tuples that satisfy the equality condition, which we have estimated as $SC_A(R)$.

(7) Equality condition on a non-clustering (secondary) index

If the predicate involves an equality condition on attribute A, which is not the primary key but does provide a non-clustering secondary index, then we can use the index to retrieve the required tuples. In this case, we have to assume that the tuples are on different blocks (the index is not clustered this time), so the estimated cost becomes:

$$nLevels_A(I) + [SC_A(R)]$$

(8) Inequality condition on a secondary B^+-tree index

If the predicate involves an inequality condition on attribute A ($A < x$, $A <= x$, $A > x$, $A >= x$), which provides a secondary B^+-tree index, then from the leaf nodes of the tree we can scan the keys from the smallest value up to x (for < or <= conditions) or from x up to the maximum value (for > or >= conditions). Assuming uniform distribution, we would expect half the leaf node blocks to be accessed and, via the index, half the tuples to be accessed. The estimated cost is then:

$$nLevels_A(I) + [nLfBlocks_A(I)/2 + nTuples(R)/2]$$

The algorithm for searching a B^+-tree index for a single tuple is shown in Figure 21.7.

(9) Composite predicates

So far, we have limited our discussion to simple predicates that involve only one attribute. However, in many situations the predicate may be composite, consisting of several conditions involving more than one attribute. We have already noted in Section 21.2 that we can express a composite predicate in two forms: conjunctive normal form and disjunctive normal form:

- A conjunctive selection contains only those tuples that satisfy all conjuncts.
- A disjunctive selection contains those tuples formed by the union of all tuples that satisfy the disjuncts.

Conjunctive selection without disjunction

If the composite predicate contains no disjunct terms, we may consider the following approaches:

Figure 21.7
Algorithm for searching B^+-tree for single tuple matching a given value.

```
//
// B+-Tree search
// B+-Tree structure is represented as a linked list with each non-leaf node structured as:
// a maximum of n elements, each consisting of:
//    a key value (key) and a pointer (p) to a child node (possibly NULL).
//    Keys are ordered: key1 < key2 < key3 < … < keyn-1
// The leaf nodes point to addresses of actual records.
// Predicate is the search key.
// Returns a boolean variable (found) indicating whether record has been found, and
// the address (return_address) of the record, if found.
//
node = get_root_node();
while (node is not a leaf node) {
      i = 1;                              // find the key that is less than predicate
      while (not (i > n or predicate < node[i].key)) {
            i = i + 1;
      }
node = get_next_node(node[i].p);    // node[i].p points to subtree that may contain predicate.
}
// Have found leaf node, so check whether a record exists with this predicate.
i = 1;
found = FALSE;
while (not (found or i > n)) {
      if (predicate = node[i].key)
      then {
            found = TRUE;
            return_address = node[i].p;
      }
      else
            i = i + 1;
}
```

(1) If one of the attributes in a conjunct has an index or is ordered, we can use one of the selection strategies 2–8 discussed above to retrieve tuples satisfying that condition. We can then check whether each retrieved tuple satisfies the remaining conditions in the predicate.

(2) If the Selection involves an equality condition on two or more attributes and a composite index (or hash key) exists on the combined attributes, we can search the index directly, as previously discussed. The type of index will determine which of the above algorithms will be used.

(3) If we have secondary indexes defined on one or more attributes and again these attributes are involved only in equality conditions in the predicate, then if the indexes use record pointers (a record pointer uniquely identifies each tuple and provides the address of the tuple on disk), as opposed to block pointers, we can scan each index for

tuples that satisfy an individual condition. By then forming the intersection of all the retrieved pointers, we have the set of pointers that satisfy these conditions. If indexes are not available for all attributes, we can test the retrieved tuples against the remaining conditions.

Selections with disjunction

If one of the terms in the selection condition contains an $\vee$ (OR), and the term requires a linear search because no suitable index or sort order exists, the entire Selection operation requires a linear search. Only if an index or sort order exists on *every* term in the Selection can we optimize the query by retrieving the tuples that satisfy each condition and applying the Union operation, as discussed below in Section 21.4.5, which will also eliminate duplicates. Again, record pointers can be used if they exist.

If no attribute can be used for efficient retrieval, we use the linear search method and check all the conditions simultaneously for each tuple. We now give an example to illustrate the use of estimation with the Selection operation.

Example 21.4 Cost estimation for Selection operation

For the purposes of this example, we make the following assumptions about the Staff relation:

- There is a hash index with no overflow on the primary key attribute staffNo.
- There is a clustering index on the foreign key attribute branchNo.
- There is a B^+-tree index on the salary attribute.
- The Staff relation has the following statistics stored in the system catalog:

nTuples(Staff)	= 3000			
bFactor(Staff)	= 30	$\Rightarrow$	nBlocks(Staff)	= 100
$nDistinct_{branchNo}$(Staff)	= 500	$\Rightarrow$	$SC_{branchNo}$(Staff)	= 6
$nDistinct_{position}$(Staff)	= 10	$\Rightarrow$	$SC_{position}$(Staff)	= 300
$nDistinct_{salary}$(Staff)	= 500	$\Rightarrow$	SC_{salary}(Staff)	= 6
min_{salary}(Staff)	= 10,000		max_{salary}(Staff)	= 50,000
$nLevels_{branchNo}$(I)	= 2			
$nLevels_{salary}$(I)	= 2		$nLfBlocks_{salary}$(I)	= 50

The estimated cost of a linear search on the key attribute staffNo is 50 blocks, and the cost of a linear search on a non-key attribute is 100 blocks. Now we consider the following Selection operations, and use the above strategies to improve on these two costs:

S1: $\sigma_{staffNo='SG5'}$(Staff)
S2: $\sigma_{position='Manager'}$(Staff)
S3: $\sigma_{branchNo='B003'}$(Staff)
S4: $\sigma_{salary>20000}$(Staff)
S5: $\sigma_{position='Manager' \wedge branchNo='B003'}$(Staff)

S1: This Selection operation contains an equality condition on the primary key. Therefore, as the attribute staffNo is hashed we can use strategy 3 defined above to estimate the cost as 1 block. The estimated cardinality of the result relation is $SC_{staffNo}(Staff) = 1$.

S2: The attribute in the predicate is a non-key, non-indexed attribute, so we cannot improve on the linear search method, giving an estimated cost of 100 blocks. The estimated cardinality of the result relation is $SC_{position}(Staff) = 300$.

S3: The attribute in the predicate is a foreign key with a clustering index, so we can use Strategy 6 to estimate the cost as 2 + [6/30] = 3 blocks. The estimated cardinality of the result relation is $SC_{branchNo}(Staff) = 6$.

S4: The predicate here involves a range search on the salary attribute, which has a B^+-tree index, so we can use strategy 7 to estimate the cost as: 2 + [50/2] + [3000/2] = 1527 blocks. However, this is significantly worse than the linear search strategy, so in this case we would use the linear search method. The estimated cardinality of the result relation is $SC_{salary}(Staff) = [3000*(50000–20000)/(50000–10000)] = 2250$.

S5: In the last example, we have a composite predicate but the second condition can be implemented using the clustering index on branchNo (S3 above), which we know has an estimated cost of 3 blocks. While we are retrieving each tuple using the clustering index, we can check whether it satisfies the first condition (position = 'Manager'). We know that the estimated cardinality of the second condition is $SC_{branchNo}(Staff) = 6$. If we call this intermediate relation T, then we can estimate the number of distinct values of position in T, $nDistinct_{position}(T)$, as: [(6 + 10)/3] = 6. Applying the second condition now, the estimated cardinality of the result relation is $SC_{position}(T) = 6/6 = 1$, which would be correct if there is one manager for each branch.

21.4.3 Join Operation ($T = (R \bowtie_F S)$)

We mentioned at the start of this chapter that one of the main concerns when the relational model was first launched commercially was the performance of queries. In particular, the operation that gave most concern was the Join operation which, apart from Cartesian product, is the most time-consuming operation to process, and one we have to ensure is performed as efficiently as possible. Recall from Section 4.1.3 that the Theta join operation defines a relation containing tuples that satisfy a specified predicate *F* from the Cartesian product of two relations R and S, say. The predicate *F* is of the form R.a θ S.b, where θ may be one of the logical comparison operators. If the predicate contains only equality (=), the join is an Equijoin. If the join involves all common attributes of R and S, the join is called a Natural join. In this section, we look at the main strategies for implementing the Join operation:

- block nested loop join;
- indexed nested loop join;
- sort–merge join;
- hash join.

Table 21.2 Summary of estimated I/O cost of strategies for Join operation.

Strategies	Cost
Block nested loop join	nBlocks(R) + (nBlocks(R) * nBlocks(S)), if buffer has only one block for R and S
	nBlocks(R) + [nBlocks(S)*(nBlocks(R)/(nBuffer – 2))], if (nBuffer – 2) blocks for R
	nBlocks(R) + nBlocks(S), if all blocks of R can be read into database buffer
Indexed nested loop join	Depends on indexing method; for example:
	nBlocks(R) + nTuples(R)*($nLevels_A(I)$ + 1), if join attribute A in S is the primary key
	nBlocks(R) + nTuples(R)*($nLevels_A(I)$ + [$SC_A(R)$/bFactor(R)]), for clustering index I on attribute A
Sort–merge join	nBlocks(R)*[$\log_2$(nBlocks(R)] + nBlocks(S)*[$\log_2$(nBlocks(S)], for sorts
	nBlocks(R) + nBlocks(S), for merge
Hash join	3(nBlocks(R) + nBlocks(S)), if hash index is held in memory
	2(nBlocks(R) + nBlocks(S))*[$\log_{nBuffer-1}$(nBlocks(S)) – 1] + nBlocks(R) + nBlocks(S), otherwise

For the interested reader, a more complete survey of join strategies can be found in Mishra and Eich (1992). The cost estimates for the different Join operation strategies are summarized in Table 21.2. We start by estimating the cardinality of the Join operation.

Estimating the cardinality of the Join operation

The cardinality of the Cartesian product of R and S, R × S, is simply:

nTuples(R) * nTuples(S)

Unfortunately, it is much more difficult to estimate the cardinality of any join as it depends on the distribution of values in the joining attributes. In the worst case, we know that the cardinality of the join cannot be any greater than the cardinality of the Cartesian product, so:

nTuples(T) ≤ nTuples(R) * nTuples(S)

Some systems use this upper bound, but this estimate is generally too pessimistic. If we again assume a uniform distribution of values in both relations, we can improve on this estimate for Equijoins with a predicate (R.A = S.B) as follows:

(1) If A is a key attribute of R, then a tuple of S can only join with one tuple of R. Therefore, the cardinality of the Equijoin cannot be any greater than the cardinality of S:

nTuples(T) ≤ nTuples(S)

(2) Similarly, if B is a key of S, then:

nTuples(T) ≤ nTuples(R)

(3) If neither A nor B are keys, then we could estimate the cardinality of the join as:

$$nTuples(T) = SC_A(R)*nTuples(S)$$

or

$$nTuples(T) = SC_B(S)*nTuples(R)$$

To obtain the first estimate, we use the fact that for any tuple *s* in S, we would expect on average $SC_A(R)$ tuples with a given value for attribute A, and this number to appear in the join. Multiplying this by the number of tuples in S, we get the first estimate above. Similarly, for the second estimate.

(1) Block nested loop join

The simplest join algorithm is a nested loop that joins the two relations together a tuple at a time. The outer loop iterates over each tuple in one relation R, and the inner loop iterates over each tuple in the second relation S. However, as we know that the basic unit of reading/writing is a disk block, we can improve on the basic algorithm by having two additional loops that process blocks, as indicated in the outline algorithm of Figure 21.8.

Since each block of R has to be read, and each block of S has to be read for each block of R, the estimated cost of this approach is:

$$nBlocks(R) + (nBlocks(R) * nBlocks(S))$$

With this estimate the second term is fixed, but the first term could vary depending on the relation chosen for the outer loop. Clearly, we should choose the relation that occupies the smaller number of blocks for the outer loop.

Figure 21.8
Algorithm for block nested loop join.

```
//
// Block nested loop join
// Blocks in both files are numbered sequentially from 1.
// Returns a result table containing the join of R and S.
//
for iblock = 1 to nBlocks(R) {                          // outer loop
      Rblock = read_block(R, iblock);
      for jblock = 1 to nBlocks(S) {                    // inner loop
            Sblock = read_block(S, jblock);
            for i = 1 to nTuples(Rblock) {
                  for j = 1 to nTuples(Sblock) {
                        if (Rblock.tuple[i]/Sblock.tuple[j] match join condition)
                        then    add them to result;
                  }
            }
      }
}
```

Another improvement to this strategy is to read as many blocks as possible of the smaller relation, R say, into the database buffer, saving one block for the inner relation, and one for the result relation. If the buffer can hold nBuffer blocks, then we should read (nBuffer − 2) blocks from R into the buffer at a time, and one block from S. The total number of R blocks accessed is still nBlocks(R), but the total number of S blocks read is reduced to approximately [nBlocks(S)*(nBlocks(R)/(nBuffer − 2))]. With this approach, the new cost estimate becomes:

nBlocks(R) + [nBlocks(S)*(nBlocks(R)/(nBuffer – 2))]

If we can read all blocks of R into the buffer, this reduces to:

nBlocks(R) + nBlocks(S)

If the join attributes in an Equijoin (or Natural join) form a key on the inner relation, then the inner loop can terminate as soon as the first match is found.

(2) Indexed nested loop join

If there is an index (or hash function) on the join attributes of the inner relation, then we can replace the inefficient file scan with an index lookup. For each tuple in R, we use the index to retrieve the matching tuples of S. The indexed nested loop join algorithm is outlined in Figure 21.9. For clarity, we use a simplified algorithm that processes the outer loop a block at a time. As noted above, however, we should read as many blocks of R into the database buffer as possible. We leave this modification of the algorithm as an exercise for the reader (see Exercise 21.19).

This is a much more efficient algorithm for a join, avoiding the enumeration of the Cartesian product of R and S. The cost of scanning R is nBlocks(R), as before. However,

```
//
// Indexed block loop join of R and S on join attribute A
// Assume that there is an index I on attribute A of relation S, and
// that there are m index entries I[1], I[2], … , I[m] with indexed value of tuple R[i].A
// Blocks in R are numbered sequentially from 1.
// Returns a result table containing the join of R and S.
//
for iblock = 1 to nBlocks(R) {
      Rblock = read_block(R, iblock);
      for i = 1 to nTuples(Rblock) {
            for j = 1 to m {
                  if (Rblock.tuple[i].A = I[j])
                  then    add corresponding tuples to result;
            }
      }
}
```

Figure 21.9
Algorithm for indexed nested loop join.

the cost of retrieving the matching tuples in S depends on the type of index and the number of matching tuples. For example, if the join attribute A in S is the primary key, the cost estimate is:

$$nBlocks(R) + nTuples(R)*(nLevels_A(I) + 1)$$

If the join attribute A in S is a clustering index, the cost estimate is:

$$nBlocks(R) + nTuples(R)*(nLevels_A(I) + [SC_A(R)/bFactor(R)])$$

(3) Sort–merge join

For Equijoins, the most efficient join is achieved when both relations are sorted on the join attributes. In this case, we can look for qualifying tuples of R and S by merging the two relations. If they are not sorted, a preprocessing step can be carried out to sort them. Since the relations are in sorted order, tuples with the same join attribute value are guaranteed to be in consecutive order. If we assume that the join is many-to-many, that is there can be many tuples of both R and S with the same join value, and if we assume that each set of tuples with the same join value can be held in the database buffer at the same time, then each block of each relation need only be read once. Therefore, the cost estimate for the sort–merge join is:

$$nBlocks(R) + nBlocks(S)$$

If a relation has to be sorted, R say, we would have to add the cost of the sort, which we can approximate as:

$$nBlocks(R)*[\log_2(nBlocks(R))]$$

An outline algorithm for sort–merge join is shown in Figure 21.10.

(4) Hash join

For a Natural join (or Equijoin), a hash join algorithm may also be used to compute the join of two relations R and S on join attribute set A. The idea behind this algorithm is to partition relations R and S according to some hash function that provides uniformity and randomness. Each equivalent partition for R and S should hold the same value for the join attributes, although it may hold more than one value. Therefore, the algorithm has to check equivalent partitions for the same value. For example, if relation R is partitioned into $R_1, R_2, \ldots, R_M$, and relation S into $S_1, S_2, \ldots, S_M$ using a hash function *h()*, then if B and C are attributes of R and S respectively, and $h(R.B) \neq h(S.C)$, then $R.B \neq S.C$. However, if $h(R.B) = h(S.C)$, it does not necessarily imply that $R.B = S.C$, as different values may map to the same hash value.

The second phase, called the *probing phase*, reads each of the R partitions in turn and for each one attempts to join the tuples in the partition to the tuples in the equivalent S partition. If a nested loop join is used for the second phase, the smaller partition is used

```
//
// Sort-merge join of R and S on join attribute A
// Algorithm assumes join is many-to-many.
// Reads are omitted for simplicity.
// First sort R and S (unnecessary if two files are already sorted on join attributes).
sort(R);
sort(S);
// Now perform merge
nextR = 1; nextS = 1;
while (nextR <= nTuples(R) and nextS <= nTuples(S)) {
      join_value = R.tuples[nextR].A;
// scan S until we find a value less than the current join value
      while (S.tuples[nextS].A < join_value and nextS <= nTuples(S)) {
            nextS = nextS + 1;
      }
// May have matching tuple of R and S.
// For each tuple in S with join_value, match it to each tuple in R with join_value.
// (Assumes M:N join).
      while (S.tuples[nextS].A = join_value and nextS <= nTuples(S)) {
            m = nextR;
            while (R.tuples[m].A = join_value and m <= nTuples(R)) {
                  add matching tuples S.tuples[nextS] and R.tuples[m] to result;
                  m = m + 1;
            }
            nextS = nextS + 1;
      }
// Have now found all matching tuples in R and S with the same join_value.
// Now find the next tuple in R with a different join value.
      while (R.tuples[nextR].A = join_value and nextR <= nTuples(R)) {
            nextR = nextR + 1;
      }
}
```

Figure 21.10
Algorithm for sort–merge join.

as the outer loop, R_i say. The complete partition R_i is read into memory and each block of the equivalent S_i partition is read and each tuple is used to probe R_i for matching tuples. For increased efficiency, it is common to build an in-memory hash table for each partition R_i using a second hash function, different from the partitioning hash function. The algorithm for hash join is outlined in Figure 21.11. We can estimate the cost of the hash join as:

$$3(\text{nBlocks}(R) + \text{nBlocks}(S))$$

This accounts for having to read R and S to partition them, write each partition to disk, and then having to read each of the partitions of R and S again to find matching tuples. This

Figure 21.11
Algorithm for hash join.

```
//
// Hash join algorithm
// Reads are omitted for simplicity.
//
// Start by partitioning R and S.
for i = 1 to nTuples(R) {
      hash_value = hash_function(R.tuple[i].A);
      add tuple R.tuple[i].A to the R partition corresponding to hash value, hash_value;
}
for j = 1 to nTuples(S) {
      hash_value = hash_function(S.tuple[j].A);
      add tuple S.tuple[j].A to the S partition corresponding to hash value, hash_value;
}
// Now perform probing (matching) phase
for ihash = 1 to M {
      read R partition corresponding to hash value ihash;
      RP = Rpartition[ihash];
      for i = 1 to max_tuples_in_R_partition(RP) {
// build an in-memory hash index using hash_function2( ), different from hash_function( )
            new_hash = hash_function2(RP.tuple[i].A);
            add new_hash to in-memory hash index;
      }
// Scan S partition for matching R tuples
      SP = Spartition[ihash];
      for j = 1 to max_tuples_in_S_partition(SP) {
            read S and probe hash table using hash_function2(SP.tuple[j].A);
            add all matching tuples to output;
      }
      clear hash table to prepare for next partition;
}
```

estimate is approximate and takes no account of overflows occurring in a partition. It also assumes that the hash index can be held in memory. If this is not the case, the partitioning of the relations cannot be done in one pass, and a recursive partitioning algorithm has to be used. In this case, the cost estimate can be shown to be:

$$2(\text{nBlocks}(R) + \text{nBlocks}(S))*[\log_{\text{nBuffer}-1}(\text{nBlocks}(S)) - 1] + \text{nBlocks}(R) + \text{nBlocks}(S)$$

For a more complete discussion of hash join algorithms, the interested reader is referred to Valduriez and Gardarin (1984), DeWitt *et al.* (1984), and DeWitt and Gerber (1985). Extensions, including the hybrid hash join, are described in Shapiro (1986), and a more recent study by Davison and Graefe (1994) describe hash join techniques that can adapt to the available memory.

Example 21.5 Cost estimation for Join operation

For the purposes of this example, we make the following assumptions:

- There are separate hash indexes with no overflow on the primary key attributes staffNo of Staff and branchNo of Branch.
- There are 100 database buffer blocks.
- The system catalog holds the following statistics:

nTuples(Staff) = 6000
bFactor(Staff) = 30 ⇒ nBlocks(Staff) = 200
nTuples(Branch) = 500
bFactor(Branch) = 50 ⇒ nBlocks(Branch) = 10
nTuples(PropertyForRent) = 100,000
bFactor(PropertyForRent) = 50 ⇒ nBlocks(PropertyForRent) = 2000

A comparison of the above four strategies for the following two joins is shown in Table 21.3:

J1: Staff $\bowtie_{staffNo}$ PropertyForRent
J2: Branch $\bowtie_{branchNo}$ PropertyForRent

In both cases, we know that the cardinality of the result relation can be no larger than the cardinality of the first relation, as we are joining over the key of the first relation. Note that no one strategy is best for both Join operations. The sort–merge join is best for the first join provided both relations are already sorted. The indexed nested loop join is best for the second join.

Table 21.3 Estimated I/O costs of Join operations in Example 21.5.

Strategies	J1	J2	Comments
Block nested loop join	400,200	20,010	Buffer has only one block for R and S
	4282	N/A[a]	(nBuffer – 2) blocks for R
	N/A[b]	2010	All blocks of R fit in database buffer
Indexed nested loop join	6200	510	Keys hashed
Sort–merge join	25,800	24,240	Unsorted
	2200	2010	Sorted
Hash join	6600	6030	Hash table fits in memory

[a] All blocks of R can be read into buffer.
[b] Cannot read all blocks of R into buffer.

21.4.4 Projection Operation ($S = \Pi_{A_1, A_2, \ldots, A_m}(R)$)

The Projection operation is also a unary operation that defines a relation S containing a vertical subset of a relation R extracting the values of specified attributes and eliminating duplicates. Therefore, to implement Projection, we need the following steps:

(1) removal of attributes that are not required;

(2) elimination of any duplicate tuples that are produced from the previous step.

The second step is the more problematic one, although it is required only if the projection attributes do not include a key of the relation. There are two main approaches to eliminating duplicates: sorting and hashing. Before we consider these two approaches, we first estimate the cardinality of the result relation.

Estimating the cardinality of the Projection operation

When the Projection contains a key attribute, then since no elimination of duplicates is required, the cardinality of the Projection is:

$$nTuples(S) = nTuples(R)$$

If the Projection consists of a single non-key attribute ($S = \Pi_A(R)$), we can estimate the cardinality of the Projection as:

$$nTuples(S) = SC_A(R)$$

Otherwise, if we assume that the relation is a Cartesian product of the values of its attributes, which is generally unrealistic, we could estimate the cardinality as:

$$nTuples(S) \leq \min(nTuples(R), \prod_{i=1}^{m} nDistinct_{A_i}(R))$$

(1) Duplicate elimination using sorting

The objective of this approach is to sort the tuples of the reduced relation using all the remaining attributes as the sort key. This has the effect of arranging the tuples in such a way that duplicates are adjacent and can be removed easily thereafter. To remove the unwanted attributes, we need to read all tuples of R and copy the required attributes to a temporary relation, at a cost of nBlocks(R). The estimated cost of sorting is $nBlocks(R)*[\log_2(nBlocks(R))]$, and so the combined cost is:

$$nBlocks(R) + nBlocks(R)*[\log_2(nBlocks(R))]$$

An outline algorithm for this approach is shown in Figure 21.12.

(2) Duplicate elimination using hashing

The hashing approach can be useful if we have a large number of buffer blocks relative to the number of blocks for R. Hashing has two phases: partitioning and duplicate elimination.

```
//
// Projection using sorting
// Assume projecting relation R over the attributes a1, a2, …, am.
// Returns result relation S.
//
// First, remove unwanted attributes.
for iblock = 1 to nBlocks(R) {
      block = read_block(R, iblock);
      for i = 1 to nTuples(block) {
            copy block.tuple[i].a1, block.tuple[i].a2, …, block.tuple[i].am, to output T
      }
}
// Now sort T, if necessary.
if {a1, a2, …, am} contains a key
then
      S = T;
else {
      sort(T);
// Finally, remove duplicates.
      i = 1; j = 2;
      while (i <= nTuples(T)) {
            output T[i] to S;
// Skip over duplicates for this tuple, if any.
            while (T[i] = T[j]) {
                  j = j + 1;
            }
            i = j; j = i + 1;
      }
}
```

Figure 21.12
Algorithm for Projection using sorting.

In the partitioning phase, we allocate one buffer block for reading relation R, and (nBuffer – 1) buffer blocks for output. For each tuple in R, we remove the unwanted attributes and then apply a hash function *h* to the combination of the remaining attributes, and write the reduced tuple to the hashed value. The hash function *h* should be chosen so that tuples are uniformly distributed to one of the (nBuffer – 1) partitions. Two tuples that belong to different partitions are guaranteed not to be duplicates, because they have different hash values, which reduces the search area for duplicate elimination to individual partitions. The second phase proceeds as follows:

- Read each of the (nBuffer – 1) partitions in turn.
- Apply a second (different) hash function *h2()* to each tuple as it is read.
- Insert the computed hash value into an in-memory hash table.
- If the tuple hashes to the same value as some other tuple, check whether the two are the same, and eliminate the new one if it is a duplicate.
- Once a partition has been processed, write the tuples in the hash table to the result file.

If the number of blocks we require for the temporary table that results from the Projection on R before duplicate elimination is *nb*, then the estimated cost is:

nBlocks(R) + nb

This excludes writing the result relation and assumes that hashing requires no overflow partitions. We leave the development of this algorithm as an exercise for the reader.

21.4.5 The Relational Algebra Set Operations (T = R ∪ S, T = R ∩ S, T = R − S)

The binary set operations of Union (R ∪ S), Intersection (R ∩ S), and Set difference (R − S) apply only to relations that are union-compatible (see Section 4.1.2). We can implement these operations by first sorting both relations on the same attributes and then scanning through each of the sorted relations once to obtain the desired result. In the case of Union, we place in the result any tuple that appears in either of the original relations, eliminating duplicates where necessary. In the case of Intersection, we place in the result only those tuples that appear in both relations. In the case of Set difference, we examine each tuple of R and place it in the result only if it has no match in S. For all these operations, we could develop an algorithm using the sort–merge join algorithm as a basis. The estimated cost in all cases is simply:

$$\text{nBlocks}(R) + \text{nBlocks}(S) + \text{nBlocks}(R)*[\log_2(\text{nBlocks}(R))] + \text{nBlocks}(S)*[\log_2(\text{nBlocks}(S))]$$

We could also use a hashing algorithm to implement these operations. For example, for Union we could build an in-memory hash index on R, and then add the tuples of S to the hash index only if they are not already present. At the end of this step we would add the tuples in the hash index to the result.

Estimating the cardinality of the set operations

Again, because duplicates are eliminated when performing the Union operation, it is generally quite difficult to estimate the cardinality of the operation, but we can give an upper and lower bound as:

$$\max(\text{nTuples}(R), \text{nTuples}(S)) \leq \text{nTuples}(T) \leq \text{nTuples}(R) + \text{nTuples}(S)$$

For Set difference, we can also give an upper and lower bound:

$$0 \leq \text{nTuples}(T) \leq \text{nTuples}(R)$$

Consider the following SQL query, which finds the average staff salary:

```
SELECT AVG(salary)
FROM Staff;
```

This query uses the aggregate function AVG. To implement this query, we could scan the entire Staff relation and maintain a running count of the number of tuples read and the sum of all salaries. On completion, it is easy to compute the average from these two running counts.

Now consider the following SQL query, which finds the average staff salary at each branch:

```
SELECT AVG(salary)
FROM Staff
GROUP BY branchNo;
```

This query again uses the aggregate function AVG but, in this case, in conjunction with a grouping clause. For grouping queries, we can use sorting or hashing algorithms in a similar manner to duplicate elimination. We can estimate the cardinality of the result relation when a grouping is present using the estimates derived earlier for Selection. We leave this as an exercise for the reader.

21.5 Enumeration of Alternative Execution Strategies

Fundamental to the efficiency of query optimization is the **search space** of possible execution strategies and the **enumeration algorithm** that is used to search this space for an optimal strategy. For a given query, this space can be extremely large. For example, for a query that consists of three joins over the relations R, S, and T there are 12 different join orderings:

R ⋈ (S ⋈ T)	R ⋈ (T ⋈ S)	(S ⋈ T) ⋈ R	(T ⋈ S) ⋈ R
S ⋈ (R ⋈ T)	S ⋈ (T ⋈ R)	(R ⋈ T) ⋈ S	(T ⋈ R) ⋈ S
T ⋈ (R ⋈ S)	T ⋈ (S ⋈ R)	(R ⋈ S) ⋈ T	(S ⋈ R) ⋈ T

In general, with n relations, there are $(2(n-1))!/(n-1)!$ different join orderings. If n is small, this number is manageable; however, as n increases this number becomes overly large. For example, if $n = 4$ the number is 120; if $n = 6$ the number is 30,240; if $n = 8$ the number is greater than 17 million, and with $n = 10$ the number is greater than 176 billion. To compound the problem, the optimizer may also support different selection methods (for example, linear search, index search) and join methods (for example, sort–merge join, hash join). In this section, we discuss how the search space can be reduced and efficiently processed. We first examine two issues that are relevant to this discussion: *pipelining* and *linear trees*.

21.5.1 Pipelining

In this section we discuss one further aspect that is sometimes used to improve the performance of queries, namely **pipelining** (sometimes known as **stream-based processing** or **on-the-fly processing**). In our discussions to date, we have implied that the results of intermediate relational algebra operations are written temporarily to disk. This process is known as **materialization**: the output of one operation is stored in a temporary relation for processing by the next operation. An alternative approach is to pipeline the results of one operation to another operation without creating a temporary relation to hold the intermediate

result. Clearly, if we can use pipelining we can save on the cost of creating temporary relations and reading the results back in again.

For example, at the end of Section 21.4.2, we discussed the implementation of the Selection operation where the predicate was composite, such as:

$$\sigma_{\text{position}=\text{'Manager'} \wedge \text{salary}>20000}(\text{Staff})$$

If we assume that there is an index on the salary attribute, then we could use the cascade of selection rule to transform this Selection into two operations:

$$\sigma_{\text{position}=\text{'Manager'}}(\sigma_{\text{salary}>20000}(\text{Staff}))$$

Now, we can use the index to efficiently process the first Selection on salary, store the result in a temporary relation and then apply the second Selection to the temporary relation. The pipeline approach dispenses with the temporary relation and instead applies the second Selection to each tuple in the result of the first Selection as it is produced, and adds any qualifying tuples from the second operation to the result.

Generally, a pipeline is implemented as a separate process or thread within the DBMS. Each pipeline takes a stream of tuples from its inputs and creates a stream of tuples as its output. A buffer is created for each pair of adjacent operations to hold the tuples being passed from the first operation to the second one. One drawback with pipelining is that the inputs to operations are not necessarily available all at once for processing. This can restrict the choice of algorithms. For example, if we have a Join operation and the pipelined input tuples are not sorted on the join attributes, then we cannot use the standard sort–merge join algorithm. However, there are still many opportunities for pipelining in execution strategies.

21.5.2 Linear Trees

All the relational algebra trees we created in the earlier sections of this chapter are of the form shown in Figure 21.13(a). This type of relational algebra tree is known as a **left-deep (join) tree**. The term relates to how operations are combined to execute the query – for example, only the left side of a join is allowed to be something that results from a previous join, and hence the name left-deep tree. For a join algorithm, the left child node is the outer relation and the right child is the inner relation. Other types of tree are the **right-deep tree**, shown in Figure 21.13(b), and the **bushy tree**, shown in Figure 21.13(d) (Graefe and DeWitt, 1987). Bushy trees are also called *non-linear trees*, and left-deep and right-deep trees are known as *linear trees*. Figure 21.13(c) is an example of another linear tree, which is not a left- or right-deep tree.

With linear trees, the relation on one side of each operator is always a base relation. However, because we need to examine the entire inner relation for each tuple of the outer relation, inner relations must always be materialized. This makes left-deep trees appealing, as inner relations are always base relations (and thus already materialized).

Left-deep trees have the advantages of reducing the search space for the optimum strategy, and allowing the query optimizer to be based on dynamic processing techniques, as we discuss shortly. Their main disadvantage is that, in reducing the search space, many alternative execution strategies are not considered, some of which may be of lower cost

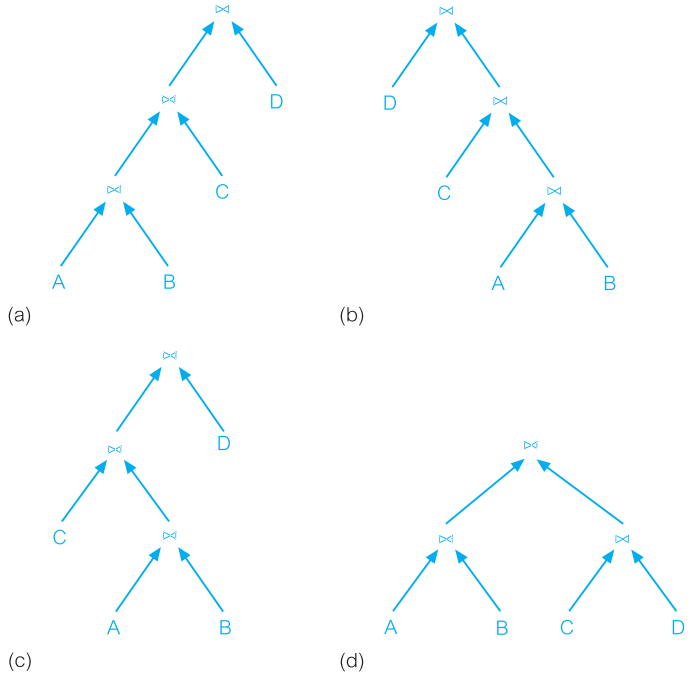


Figure 21.13
(a) Left-deep tree; (b) right-deep tree; (c) another linear tree; (d) (non-linear) bushy tree.

than the one found using the linear tree. Left-deep trees allow the generation of all fully pipelined strategies, that is, strategies in which the joins are all evaluated using pipelining.

Physical Operators and Execution Strategies 21.5.3

The term **physical operator** is sometimes used to represent a specific algorithm that implements a logical database operation, such as selection or join. For example, we can use the physical operator sort–merge join to implement the relational algebra join operation. Replacing the logical operations in a relational algebra tree with physical operators produces an **execution strategy** (also known as a **query evaluation plan** or **access plan**) for the query. Figure 21.14 shows a relational algebra tree and a corresponding execution strategy.

While DBMSs have their own internal implementations, we can consider the following abstract operators to implement the functions at the leaves of the trees:

(1) **TableScan(R)**: All blocks of R are read in an arbitrary order.

(2) **SortScan(R, L)**: Tuples of R are read in order, sorted according to the attribute(s) in list L.

(3) **IndexScan(R, P)**: P is a predicate of the form A θ c, where A is an attribute of R, θ is one of the normal comparison operators, and c is a constant value. Tuples of R are accessed through an index on attribute A.

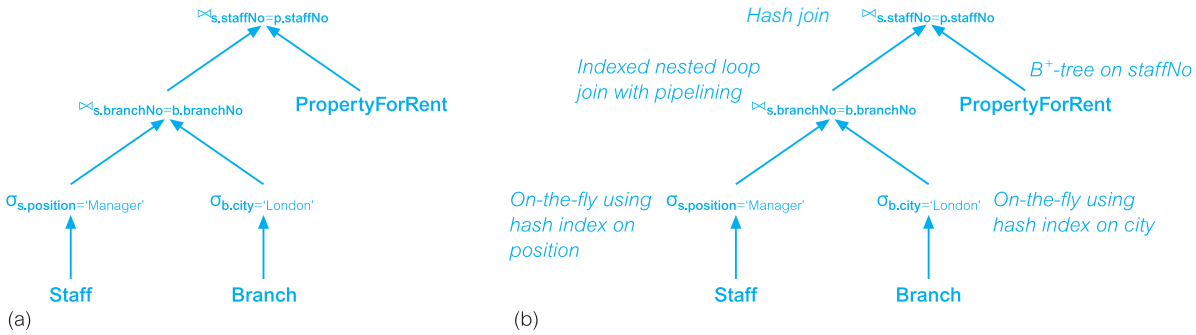


Figure 21.14 (a) Example relational algebra tree; (b) a corresponding execution strategy.

(4) **IndexScan(R, A)**: A is an attribute of R. The entire relation R is retrieved using the index on attribute A. Similar to TableScan, but may be more efficient under certain conditions (for example, R is not clustered).

In addition, the DBMS usually supports a uniform **iterator** interface, hiding the internal implementation details of each operator. The iterator interface consists of the following three functions:

(1) **Open**: This function initializes the state of the iterator prior to retrieving the first tuple and allocates buffers for the inputs and the output. Its arguments can define selection conditions that modify the behavior of the operator.

(2) **GetNext**: This function returns the next tuple in the result and places it in the output buffer. GetNext calls GetNext on each input node and performs some operator-specific code to process the inputs to generate the output. The state of the iterator is updated to reflect how much input has been consumed.

(3) **Close**: When all output tuples have been produced (through repeated calls to GetNext), the Close function terminates the operator and tidies up, de-allocating buffers as required.

When iterators are used, many operations may be active at once. Tuples pass between operators as required, supporting pipelining naturally. However, the decision to pipeline or materialize is dependent upon the operator-specific code that processes the input tuples. If this code allows input tuples to be processed as they are received, pipelining is used; if this code processes the same input tuples more than once, materialization is used.

21.5.4 Reducing the Search Space

As we showed at the start of this section, the search space for a complicated query can be enormous. To reduce the size of the space that the search strategy has to explore, query optimizers generally restrict this space in several ways. The first common restriction applies to the unary operations of Selection and Projection:

Restriction 1: Unary operations are processed on-the-fly: selections are processed as relations are accessed for the first time; projections are processed as the results of other operations are generated.

This implies that all operations are dealt with as part of join execution. Consider, now, the following simplified version of the query from Example 21.3:

```
SELECT p.propertyNo, p.street
FROM Client c, Viewing v, PropertyForRent p
WHERE c.clientNo = v.clientNo AND v.propertyNo = p.propertyNo;
```

From the discussion at the start of this section, there are 12 possible join orderings for this query. However, note that some of these orderings result in a Cartesian product rather than a join. For example:

Viewing ⋈ (Client ⋈ PropertyForRent)

results in the Cartesian product of Client and PropertyForRent. The next reduction eliminates suboptimal join trees that include a Cartesian product:

Restriction 2: Cartesian products are never formed unless the query itself specifies one.

The final typical reduction deals with the shape of join trees and, as discussed in Section 21.5.2, uses the fact that with left-deep trees the inner operand is a base relation and, therefore, already materialized:

Restriction 3: The inner operand of each join is a base relation, never an intermediate result.

This third restriction is of a more heuristic nature than the other two and excludes many alternative strategies, some of which may be of lower cost than the ones found using the left-deep tree. However, it has been suggested that most often the optimal left-deep tree is not much more expensive than the overall optimal tree. Moreover, the third restriction significantly reduces the number of alternative join strategies to be considered to $O(2^n)$ for queries with n relations and has a corresponding time complexity of $O(3^n)$. Using this approach, query optimizers can handle joins with about 10 relations efficiently, which copes with most queries that occur in traditional business applications.

21.5.5 Enumerating Left-Deep Trees

The enumeration of left-deep trees using **dynamic programming** was first proposed for the System R query optimizer (Selinger *et al.*, 1979). Since then, many commercial systems have used this basic approach. In this section we provide an overview of the algorithm, which is essentially a dynamically pruning, exhaustive search algorithm.

The dynamic programming algorithm is based on the assumption that the cost model satisfies the *principle of optimality*. Thus, to obtain the optimal strategy for a query consisting of n joins, we only need to consider the optimal strategies for subexpressions that consist of $(n - 1)$ joins and extend those strategies with an additional join. The remaining

suboptimal strategies can be discarded. The algorithm recognizes, however, that in this simple form some potentially useful strategies could be discarded. Consider the following query:

```
SELECT p.propertyNo, p.street
FROM Client c, Viewing v, PropertyForRent p
WHERE c.maxRent < 500 AND c.clientNo = v.clientNo AND
      v.propertyNo = p.propertyNo;
```

Assume that there are separate B^+-tree indexes on the attributes clientNo and maxRent of Client and that the optimizer supports both sort–merge join and block nested loop join. In considering all possible ways to access the Client relation, we would calculate the cost of a linear search of the relation and the cost of using the two B^+-trees. If the optimal strategy came from the B^+-tree index on maxRent, we would then discard the other two methods. However, use of the B^+-tree index on clientNo would result in the Client relation being sorted on the join attribute clientNo, which would result in a lower cost for a sort–merge join of Client and Viewing (as one of the relations is already sorted). To ensure that such possibilities are not discarded the algorithm introduces the concept of *interesting orders*: an intermediate result has an interesting order if it is sorted by a final ORDER BY attribute, GROUP BY attribute, or any attributes that participate in subsequent joins. For the above example, the attributes c.clientNo, v.clientNo, v.propertyNo, and p.propertyNo are interesting. During optimization, if any intermediate result is sorted on any of these attributes, then the corresponding partial strategy must be included in the search.

The dynamic programming algorithm proceeds from the bottom up and constructs all alternative join trees that satisfy the restrictions defined in the previous section, as follows:

Pass 1: We enumerate the strategies for each base relation using a linear search and all available indexes on the relation. These partial (single-relation) strategies are partitioned into equivalence classes based on any interesting orders, as discussed above. An additional equivalence class is created for the partial strategies with no interesting order. For each equivalence class, the strategy with the lowest cost is retained for consideration in the next pass. If the lowest-cost strategy for the equivalence class with no interesting order is not lower than all the other strategies it is not retained. For a given relation R, any selections involving only attributes of R are processed on-the-fly. Similarly, any attributes of R that are not part of the SELECT clause and do not contribute to any subsequent join can be projected out at this stage (restriction 1 above).

Pass 2: We generate all two-relation strategies by considering each single-relation strategy retained after Pass 1 as the outer relation, discarding any Cartesian products generated (restriction 2 above). Again, any on-the-fly processing is performed and the lowest cost strategy in each equivalence class is retained for further consideration.

Pass *k*: We generate all k-relation strategies by considering each strategy retained after Pass $(k - 1)$ as the outer relation, again discarding any Cartesian products generated and processing any selection and projections on-the-fly. Again, the lowest cost strategy in each equivalence class is retained for further consideration.

Pass *n*: We generate all n-relation strategies by considering each strategy retained after Pass $(n - 1)$ as the outer relation, discarding any Cartesian products generated. After pruning, we now have the lowest overall strategy for processing the query.

Although this algorithm is still exponential, there are query forms for which it only generates $O(n^3)$ strategies, so for $n = 10$ the number is 1000, which is significantly better than the 176 billion different join orders noted at the start of this section.

21.5.6 Semantic Query Optimization

A different approach to query optimization is based on constraints specified on the database schema to reduce the search space. This approach, known as **semantic query optimization**, may be used in conjunction with the techniques discussed above. For example, in Section 6.2.5 we defined the general constraint that prevents a member of staff from managing more than 100 properties at the same time using the following assertion:

```
CREATE ASSERTION StaffNotHandlingTooMuch
    CHECK (NOT EXISTS (SELECT staffNo
                       FROM PropertyForRent
                       GROUP BY staffNo
                       HAVING COUNT(*) > 100))
```

Consider now the following query:

```
SELECT s.staffNo, COUNT(*)
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo
GROUP BY s.staffNo
HAVING COUNT(*) > 100;
```

If the optimizer is aware of this constraint, it can dispense with trying to optimize the query as there will be no groups satisfying the HAVING clause.

Consider now the following constraint on staff salary:

```
CREATE ASSERTION ManagerSalary
    CHECK (salary > 20000 AND position = 'Manager')
```

and the following query:

```
SELECT s.staffNo, fName, lName, propertyNo
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo AND position = 'Manager';
```

Using the above constraint, we can rewrite this query as:

```
SELECT s.staffNo, fName, lName, propertyNo
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo AND salary > 20000 AND position = 'Manager';
```

This additional predicate may be very useful if the only index for the Staff relation is a B^+-tree on the salary attribute. On the other hand, this additional predicate would complicate the query if no such index existed. For further information on semantic query optimization the interested reader is referred to King (1981); Malley and Zdonik (1986); Chakravarthy *et al.* (1990); Siegel *et al.* (1992).

21.5.7 Alternative Approaches to Query Optimization

Query optimization is a well researched field and a number of alternative approaches to the System R dynamic programming algorithm have been proposed. For example, **Simulated Annealing** searches a graph whose nodes are all alternative execution strategies (the approach models the annealing process by which crystals are grown by first heating the containing fluid and then allowing it to cool slowly). Each node has an associated cost and the goal of the algorithm is to find a node with a globally minimum cost. A move from one node to another is deemed to be downhill (uphill) if the cost of the source node is higher (lower) than the cost of the destination node. A node is a *local minimum* if, in all paths starting at that node, any downhill move comes after at least one uphill move. A node is a *global minimum* if it has the lowest cost among all nodes. The algorithm performs a continuous random walk accepting downhill moves always and uphill moves with some probability, trying to avoid a high-cost local minimum. This probability decreases as time progresses and eventually becomes zero, at which point the search stops and the node with the lowest cost visited is returned as the optimal execution strategy. The interested reader is referred to Kirkpatrick *et al*. (1983) and Ioannidis and Wong (1987).

The **Iterative Improvement** algorithm performs a number of local optimizations, each starting at a random node and repeatedly accepting random downhill moves until a local minimum is reached. The interested reader is referred to Swami and Gupta (1988) and Swami (1989). The **Two-Phase Optimization** algorithm is a hybrid of Simulated Annealing and Iterative Improvement. In the first phase, Iterative Improvement is used to perform some local optimizations producing some local minimum. This local minimum is used as the input to the second phase, which is based on Simulated Annealing with a low start probability for uphill moves. The interested reader is referred to Ioannidis and Kang (1990).

Genetic algorithms, which simulate a biological phenomenon, have also been applied to query optimization. The algorithms start with an initial population, consisting of a random set of strategies, each with its own cost. From these, pairs of strategies from the population are matched to generate offspring that inherit the characteristics of both parents, although the children can be randomly changed in small ways (*mutation*). For the next generation, the algorithm retains those parents/children with the least cost. The algorithm ends when the entire population consists of copies of the same (optimal) strategy. The interested reader is referred to Bennett *et al*. (1991).

The **A*** heuristic algorithm has been used in artificial intelligence to solve complex search problems and has also been applied to query optimization (Yoo and Lafortune, 1989). Unlike the dynamic programming algorithm discussed above, the A* algorithm expands one execution strategy at a time, based on its proximity to the optimal strategy. It has been shown that A* generates a full strategy much earlier than dynamic programming and is able to prune more aggressively.

21.5.8 Distributed Query Optimization

In Chapters 22 and 23 we discuss the distributed DBMS (DDBMS), which consists of a logically interrelated collection of databases physically distributed over a computer

network, each under the control of a local DBMS. In a DDBMS a relation may be divided into a number of fragments that are distributed over a number of sites; fragments may be replicated. In Section 23.6 we consider query optimization for a DDBMS. Distributed query optimization is more complex due to the distribution of the data across the sites in the network. In the distributed environment, as well as local processing costs (that is, CPU and I/O costs), the speed of the underlying network has to be taken into consideration when comparing different strategies. In particular, we discuss an extension to the System R dynamic programming algorithm considered above as well as the query optimization algorithm from another well-known research project on DDBMSs known as SDD-1.

21.6 Query Optimization in Oracle

To complete this chapter, we examine the query optimization mechanisms used by Oracle9*i* (Oracle Corporation, 2004b). We restrict the discussion in this section to optimization based on primitive data types. Later, in Section 28.5, we discuss how Oracle provides an extensible optimization mechanism to handle user-defined types. In this section we use the terminology of the DBMS – Oracle refers to a relation as a *table* with *columns* and *rows*. We provided an introduction to Oracle in Section 8.2.

21.6.1 Rule-Based and Cost-Based Optimization

Oracle supports the two approaches to query optimization we have discussed in this chapter: rule-based and cost-based.

The rule-based optimizer

The Oracle *rule-based optimizer* has fifteen rules, ranked in order of efficiency, as shown in Table 21.4. The optimizer can choose to use a particular access path for a table only if the statement contains a predicate or other construct that makes that access path available. The rule-based optimizer assigns a score to each execution strategy using these rankings and then selects the execution strategy with the best (lowest) score. When two strategies produce the same score, Oracle resolves this tie-break by making a decision based on the order in which tables occur in the SQL statement, which would generally be regarded as not a particularly good way to make the final decision.

For example, consider the following query on the PropertyForRent table and assume that we have an index on the primary key, propertyNo, an index on the rooms column, and an index on the city column:

```
SELECT propertyNo
FROM PropertyForRent
WHERE rooms > 7 AND city = 'London';
```

In this case, the rule-based optimizer will consider the following access paths:

- A single-column access path using the index on the city column from the WHERE condition (city = 'London'). This access path has rank 9.

Table 21.4 Rule-based optimization rankings.

Rank	Access path
1	Single row by ROWID (row identifier)
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite key
9	Single-column indexes
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort–merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed columns
15	Full table scan

- An unbounded range scan using the index on the rooms column from the WHERE condition (rooms > 7). This access path has rank 11.
- A full table scan, which is available for all SQL statements. This access path has rank 15.

Although there is an index on the propertyNo column, this column does not appear in the WHERE clause and so is not considered by the rule-based optimizer. Based on these paths, the rule-based optimizer will choose to use the index based on the city column. Rule-based optimization is a deprecated feature now.

The cost-based optimizer

To improve query optimization, Oracle introduced the cost-based optimizer in Oracle 7, which selects the execution strategy that requires the minimal resource use necessary to process all rows accessed by the query (avoiding the above tie-break anomaly). The user can select whether the minimal resource usage is based on *throughput* (minimizing the amount of resources necessary to process *all* rows accessed by the query) or based on *response time* (minimizing the amount of resources necessary to process the *first* row accessed by the query), by setting the OPTIMIZER_MODE initialization parameter. The cost-based optimizer also takes into consideration hints that the user may provide, as we discuss shortly.

Statistics

The cost-based optimizer depends on statistics for all tables, clusters, and indexes accessed by the query. However, Oracle does not gather statistics automatically but makes it the users' responsibility to generate these statistics and keep them current. The PL/SQL package DBMS_STATS can be used to generate and manage statistics on tables, columns, indexes, partitions, and on all schema objects in a schema or database. Whenever possible, Oracle uses a parallel method to gather statistics, although index statistics are collected serially. For example, we could gather schema statistics for a 'Manager' schema using the following SQL statement:

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS('Manager',
                                       DBMS_STATS. AUTO_SAMPLE_SIZE);
```

The last parameter tells Oracle to determine the best sample size for good statistics.

There are a number of options that can be specified when gathering statistics. For example, we can specify whether statistics should be calculated for the entire data structure or on only a sample of the data. In the latter case, we can specify whether sampling should be row or block based:

- *Row sampling* reads rows ignoring their physical placement on disk. As a worst-case scenario, row sampling may select one row from each block, requiring a full scan of the table or index.
- *Block sampling* reads a random sample of blocks but gathers statistics using all the rows in these blocks.

Sampling generally uses fewer resources than computing the exact figure for the entire structure. For example, analyzing 10% or less of a very large table may produce the same relative percentages of unused space.

It is also possible to get Oracle to gather statistics while creating or rebuilding indexes by specifying the COMPUTE STATISTICS option with the CREATE INDEX or ALTER INDEX commands. Statistics are held within the Oracle data dictionary and can be inspected through the views shown in Table 21.5. Each view can be preceded by three prefixes:

- ALL_ includes all the objects in the database that the user has access to, including objects in another schema that the user has been given access to.
- DBA_ includes all the objects in the database.
- USER_ includes only the objects in the user's schema.

Hints

As mentioned earlier, the cost-based optimizer also takes into consideration *hints* that the user may provide. A hint is specified as a specially formatted comment within an SQL statement. There are a number of hints that can be used to force the optimizer to make different decisions, such as forcing the use of:

- the rule-based optimizer;
- a particular access path;

Table 21.5 Oracle data dictionary views.

View	Description
ALL_TABLES	Information about the object and relational tables that a user has access to
TAB_HISTOGRAMS	Statistics about the use of histograms
TAB_COLUMNS	Information about the columns in tables/views
TAB_COL_STATISTICS	Statistics used by the cost-based optimizer
TAB_PARTITIONS	Information about the partitions in a partitioned table
CLUSTERS	Information about clusters
INDEXES	Information about indexes
IND_COLUMNS	Information about the columns in each index
CONS_COLUMNS	Information about the columns in each constraint
CONSTRAINTS	Information about constraints on tables
LOBS	Information about large object (LOB) data type columns
SEQUENCES	Information about sequence objects
SYNONYMS	Information about synonyms
TRIGGERS	Information about the triggers on tables
VIEWS	Information about views

- a particular join order;
- a particular Join operation, such as a sort–merge join.

For example, we can force the use of a particular index using the following hint:

```
SELECT /*+ INDEX(sexIndex) */ fName, lName, position
FROM Staff
WHERE sex = 'M';
```

If there are as many male as female members of staff, the query will return approximately half the rows in the Staff table and a full table scan is likely to be more efficient than an index scan. However, if we know that there are significantly more female than male staff, the query will return a small percentage of the rows in the Staff table and an index scan is likely to be more efficient. If the cost-based optimizer assumes there is an even distribution of values in the sex column, it is likely to select a full table scan. In this case, the hint tells the optimizer to use the index on the sex column.

Stored execution plans

There may be times when an optimal plan has been found and it may be unnecessary or unwanted for the optimizer to generate a new execution plan whenever the SQL statement is submitted again. In this case, it is possible to create a *stored outline* using the CREATE

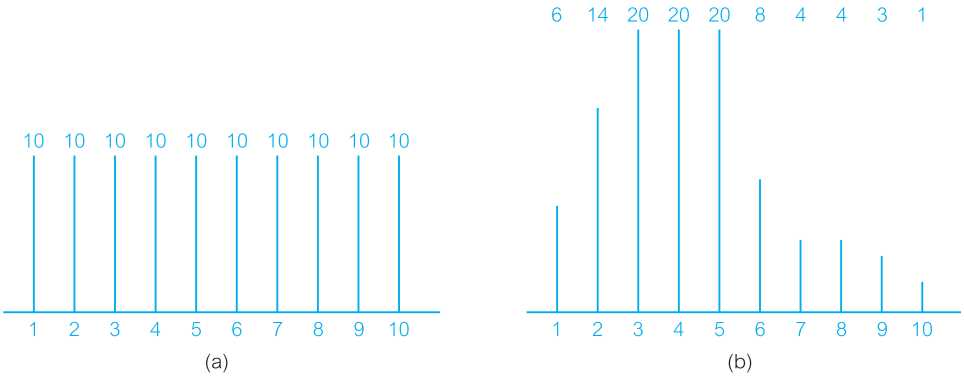


Figure 21.15 Histogram of values in rooms column in the PropertyForRent table: (a) uniform distribution; (b) non-uniform distribution.

OUTLINE statement, which will store the attributes used by the optimizer to create the execution plan. Thereafter, the optimizer uses the stored attributes to create the execution plan rather than generate a new plan.

Histograms 21.6.2

It earlier sections, we made the assumption that the data values within the columns of a table are uniformly distributed. A histogram of values and their relative frequencies gives the optimizer improved selectivity estimates in the presence of non-uniform distribution. For example, Figure 21.15(a) illustrates an estimated uniform distribution of the rooms column in the PropertyForRent table and Figure 21.15(b) the actual non-uniform distribution. The first distribution can be stored compactly as a low value (1) and a high value (10), and as a total count of all frequencies (in this case, 100).

For a simple predicate such as rooms > 9, based on a uniform distribution we can easily estimate the number of tuples in the result as (1/10)*100 = 10 tuples. However, this estimate is quite inaccurate (as we can see from Figure 21.15(b) there is actually only 1 tuple).

A **histogram** is a data structure that can be used to improve this estimate. Figure 21.16 shows two types of histogram:

- a *width-balanced histogram*, which divides the data into a fixed number of equal-width ranges (called *buckets*) each containing a count of the number of values falling within that bucket;
- a *height-balanced histogram*, which places approximately the same number of values in each bucket so that the end-points of each bucket are determined by how many values are in that bucket.

For example, suppose that we have five buckets. The width-balanced histogram for the rooms column is illustrated in Figure 21.16(a). Each bucket is of equal width with two values (1-2, 3-4, and so on), and within each bucket the distribution is assumed to be uniform. This information can be stored compactly by recording the upper and lower

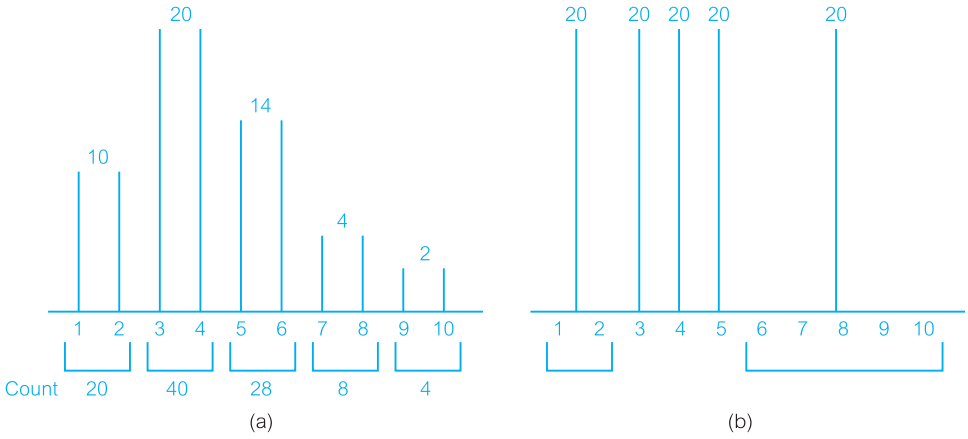


Figure 21.16
Histogram of values in rooms column in the PropertyForRent table: (a) width-balanced; (b) height-balanced.

value within each bucket and the count of the number of values within the bucket. If we consider again the predicate rooms > 9, with the width-balanced histogram we estimate the number of tuples satisfying this predicate as the size of a range element multiplied by the number of range elements, that is 2*1 = 2, which is better than the estimate based on uniform distribution.

The height-balanced histogram is illustrated in Figure 21.16(b). In this case, the height of each column is 20 (100/5). Again, the data can be stored compactly by recording the upper and lower value within each bucket, and recording the height of all buckets. If we consider the predicate rooms > 9, with the height-balanced histogram we estimate the number of tuples satisfying this predicate as: (1/5)*20 = 4, which in this case is not as good as the estimate provided by the width-balanced histogram. Oracle uses height-balanced histograms. A variation of the height-balanced histogram assumes a uniform height within a bucket but possibly slightly different heights across buckets.

As histograms are persistent objects there is an overhead involved in storing and maintaining them. Some systems, such as Microsoft's SQL Server, create and maintain histograms automatically without the need for user input. However, in Oracle it is the user's responsibility to create and maintain histograms for appropriate columns, again using the PL/SQL package DBMS_STATS. Appropriate columns are typically those columns that are used within the WHERE clause of SQL statements and have a non-uniform distribution, such as the rooms column in the above example.

21.6.3 Viewing the Execution Plan

Oracle allows the execution plan that would be chosen by the optimizer to be viewed using the EXPLAIN PLAN command. This can be extremely useful if the efficiency of a query is not as expected. The output from EXPLAIN PLAN is written to a table in the database (the default table is PLAN_TABLE). The main columns in this table are:

- STATEMENT_ID, the value of an optional STATEMENT_ID parameter specified in the EXPLAIN PLAN statement.
- OPERATION, the name of the internal operation performed. The first row would be the actual SQL statement: SELECT, INSERT, UPDATE, or DELETE.
- OPTIONS, the name of another internal operation performed.
- OBJECT_NAME, the name of the table or index.
- ID, a number assigned to each step in the execution plan.
- PARENT_ID, the ID of the next step that operates on the output of the ID step.
- POSITION, the order of processing for steps that all have the same PARENT_ID.
- COST, an estimated cost of the operation (null for statements that use the rule-based optimizer).
- CARDINALITY, an estimated number of rows accessed by the operation.

An example plan is shown in Figure 21.17. Each line in this plan represents a single step in the execution plan. Indentation has been used in the output to show the order of the operations (note the column ID by itself is insufficient to show the ordering).

```
SQL> EXPLAIN PLAN
  2  SET STATEMENT_ID = 'PB'
  3  FOR SELECT b.branchNo, b.city, propertyNo
  4  FROM Branch b, PropertyForRent p
  5  WHERE b.branchNo = p.branchNo
  6  ORDER BY b.city;

Explained.

SQL> SELECT ID||' '||PARENT_ID||'   '||LPAD('   ', 2*(LEVEL – 1))||OPERATION||' '||OPTIONS||
  2  '   '||OBJECT_NAME "Query Plan"
  3  FROM Plan_Table
  4  START WITH ID = 0 AND STATEMENT_ID = 'PB'
  5  CONNECT BY PRIOR ID = PARENT_ID AND STATEMENT_ID = 'PB';

Query Plan
------------------------------------------------------------
0     SELECT STATEMENT
1  0      SORT ORDER BY
2  1         NESTED LOOPS
3  2            TABLE ACCESS FULL PROPERTYFORRENT
4  2            TABLE ACCESS BY INDEX ROWID BRANCH
5  4               INDEX UNIQUE SCAN SYS_C007455

6  rows selected.
```

Figure 21.17 Output from the Explain Plan utility.

Chapter Summary

- The aims of **query processing** are to transform a query written in a high-level language, typically SQL, into a correct and efficient execution strategy expressed in a low-level language like the relational algebra, and to execute the strategy to retrieve the required data.
- As there are many equivalent transformations of the same high-level query, the DBMS has to choose the one that minimizes resource usage. This is the aim of **query optimization**. Since the problem is computationally intractable with a large number of relations, the strategy adopted is generally reduced to finding a near-optimum solution.
- There are two main techniques for query optimization, although the two strategies are usually combined in practice. The first technique uses **heuristic rules** that order the operations in a query. The other technique compares different strategies based on their relative costs, and selects the one that minimizes resource usage.
- Query processing can be divided into four main phases: decomposition (consisting of parsing and validation), optimization, code generation, and execution. The first three can be done either at compile time or at runtime.
- **Query decomposition** transforms a high-level query into a relational algebra query, and checks that the query is syntactically and semantically correct. The typical stages of query decomposition are analysis, normalization, semantic analysis, simplification, and query restructuring. A **relational algebra tree** can be used to provide an internal representation of a transformed query.
- **Query optimization** can apply transformation rules to convert one relational algebra expression into an equivalent expression that is known to be more efficient. Transformation rules include cascade of selection, commutativity of unary operations, commutativity of Theta join (and Cartesian product), commutativity of unary operations and Theta join (and Cartesian product), and associativity of Theta join (and Cartesian product).
- **Heuristics rules** include performing Selection and Projection operations as early as possible; combining Cartesian product with a subsequent Selection whose predicate represents a join condition into a Join operation; using associativity of binary operations to rearrange leaf nodes so that leaf nodes with the most restrictive Selections are executed first.
- **Cost estimation** depends on statistical information held in the system catalog. Typical statistics include the cardinality of each base relation, the number of blocks required to store a relation, the number of distinct values for each attribute, the selection cardinality of each attribute, and the number of levels in each multilevel index.
- The main strategies for implementing the Selection operation are: linear search (unordered file, no index), binary search (ordered file, no index), equality on hash key, equality condition on primary key, inequality condition on primary key, equality condition on clustering (secondary) index, equality condition on a nonclustering (secondary) index, and inequality condition on a secondary B^+-tree index.
- The main strategies for implementing the Join operation are: block nested loop join, indexed nested loop join, sort–merge join, and hash join.
- With **materialization** the output of one operation is stored in a temporary relation for processing by the next operation. An alternative approach is to **pipeline** the results of one operation to another operation without creating a temporary relation to hold the intermediate result, thereby saving the cost of creating temporary relations and reading the results back in again.
- A relational algebra tree where the right-hand relation is always a base relation is known as a **left-deep tree**. Left-deep trees have the advantages of reducing the search space for the optimum strategy and allowing the query optimizer to be based on dynamic processing techniques. Their main disadvantage is that in reducing the search space many alternative execution strategies are not considered, some of which may be of lower cost than the one found using a linear tree.

- Fundamental to the efficiency of query optimization is the **search space** of possible execution strategies and the **enumeration algorithm** that is used to search this space for an optimal strategy. For a given query this space can be very large. As a result, query optimizers restrict this space in a number of ways. For example, unary operations may be processed on-the-fly; Cartesian products are never formed unless the query itself specifies it; the inner operand of each join is a base relation.
- The **dynamic programming** algorithm is based on the assumption that the cost model satisfies the *principle of optimality*. To obtain the optimal strategy for a query consisting of n joins, we only need to consider the optimal strategies that consist of $(n - 1)$ joins and extend those strategies with an additional join. Equivalence classes are created based on *interesting orders* and the strategy with the lowest cost in each equivalence class is retained for consideration in the next step until the entire query has been constructed, whereby the strategy corresponding to the overall lowest cost is selected.

Review Questions

21.1 What are the objectives of query processing?

21.2 How does query processing in relational systems differ from the processing of low-level query languages for network and hierarchical systems?

21.3 What are the typical phases of query processing?

21.4 What are the typical stages of query decomposition?

21.5 What is the difference between conjunctive and disjunctive normal form?

21.6 How would you check the semantic correctness of a query?

21.7 State the transformation rules that apply to:
(a) Selection operations
(b) Projection operations
(c) Theta join operations.

21.8 State the heuristics that should be applied to improve the processing of a query.

21.9 What types of statistics should a DBMS hold to be able to derive estimates of relational algebra operations?

21.10 Under what circumstances would the system have to resort to a linear search when implementing a Selection operation?

21.11 What are the main strategies for implementing the Join operation?

21.12 What are the differences between materialization and pipelining?

21.13 Discuss the difference between linear and non-linear relational algebra trees. Give examples to illustrate your answer.

21.14 What are the advantages and disadvantages of left-deep trees?

21.15 Describe how the dynamic programming algorithm for the System R query optimizer works.

Exercises

21.16 Calculate the cost of the three strategies cited in Example 21.1 if the Staff relation has 10 000 tuples, Branch has 500 tuples, there are 500 Managers (one for each branch), and there are 10 London branches.

21.17 Using the Hotel schema given at the start of the Exercises at the end of Chapter 3, determine whether the following queries are semantically correct:

(a) **SELECT** r.type, r.price
FROM Room r, Hotel h
WHERE r.hotel_number = h.hotel_number **AND** h.hotel_name = 'Grosvenor Hotel' **AND** r.type > 100;

(b) **SELECT** g.guestNo, g.name
FROM Hotel h, Booking b, Guest g
WHERE h.hotelNo = b.hotelNo **AND** h.hotelName = 'Grosvenor Hotel';

(c) **SELECT** r.roomNo, h.hotelNo
FROM Hotel h, Booking b, Room r
WHERE h.hotelNo = b.hotelNo **AND** h.hotelNo = 'H21' **AND** b.roomNo = r.roomNo **AND**
type = 'S' **AND** b.hotelNo = 'H22';

21.18 Again using the Hotel schema, draw a relational algebra tree for each of the following queries and use the heuristic rules given in Section 21.3.2 to transform the queries into a more efficient form. Discuss each step and state any transformation rules used in the process.

(a) **SELECT** r.roomNo, r.type, r.price
FROM Room r, Booking b, Hotel h
WHERE r.roomNo = b.roomNo **AND** b.hotelNo = h.hotelNo **AND**
h.hotelName = 'Grosvenor Hotel' **AND** r.price > 100;

(b) **SELECT** g.guestNo, g.guestName
FROM Room r, Hotel h, Booking b, Guest g
WHERE h.hotelNo = b.hotelNo **AND** g.guestNo = b.guestNo **AND** h.hotelNo = r.hotelNo **AND**
h.hotelName = 'Grosvenor Hotel' **AND** dateFrom >= '1-Jan-04' **AND** dateTo <= '31-Dec-04';

21.19 Using the Hotel schema, assume the following indexes exist:

- a hash index with no overflow on the primary key attributes, roomNo/hotelNo in Room;
- a clustering index on the foreign key attribute hotelNo in Room;
- a B^+-tree index on the price attribute in Room;
- a secondary index on the attribute type in Room.

nTuples(Room)	= 10,000	bFactor(Room)	= 200
nTuples(Hotel)	= 50	bFactor(Hotel)	= 40
nTuples(Booking)	= 100,000	bFactor(Booking)	= 60
$nDistinct_{hotelNo}$(Room)	= 50		
$nDistinct_{type}$(Room)	= 10		
$nDistinct_{price}$(Room)	= 500		
min_{price}(Room)	= 200	max_{price}(Room)	= 50
$nLevels_{hotelNo}$(I)	= 2		
$nLevels_{price}$(I)	= 2	$nLfBlocks_{price}$(I)	= 50

(a) Calculate the cardinality and minimum cost for each of the following Selection operations:

S1: $\sigma_{roomNo=1 \wedge hotelNo='H001'}$(Room)
S2: $\sigma_{type='D'}$(Room)
S3: $\sigma_{hotelNo='H002'}$(Room)
S4: $\sigma_{price>100}$(Room)
S5: $\sigma_{type='S' \wedge hotelNo='H003'}$(Room)
S6: $\sigma_{type='S' \vee price<100}$(Room)

(b) Calculate the cardinality and minimum cost for each of the following Join operations:

J1: Hotel $\bowtie_{hotelNo}$ Room
J2: Hotel $\bowtie_{hotelNo}$ Booking
J3: Room $\bowtie_{roomNo}$ Booking
J4: Room $\bowtie_{hotelNo}$ Hotel
J5: Booking $\bowtie_{hotelNo}$ Hotel
J6: Booking $\bowtie_{roomNo}$ Room

(c) Calculate the cardinality and minimum cost for each of the following Projection operations:

P1: $\Pi_{hotelNo}$(Hotel)
P2: $\Pi_{hotelNo}$(Room)
P3: Π_{price}(Room)
P4: Π_{type}(Room)
P5: $\Pi_{hotelNo,\ price}$(Room)

21.20 Modify the block nested loop join and the indexed nested loop join algorithms presented in Section 21.4.3 to read (nBuffer – 2) blocks of the outer relation R at a time, rather than one block at a time.