28 Chapter 1 Introduction (Database System Concepts)

department *A* invokes a program called *new hire*. This program asks the clerk for the name of the new instructor, her new *ID*, the name of the department (that is, *A*), and the salary.
The typical user interface for na¨ıve users is a forms interface, where the user can fill in appropriate fields of the form. Na¨ıve users may also simply read *reports* generated from the database.
As another example, consider a student, who during class registration period, wishes to register for a class by using a Web interface. Such a user connects to a Web application program that runs at a Web server. The application first verifies the identity of the user, and allows her to access a form where she enters the desired information. The form information is sent back to the Web application at the server, which then determines if there is room in the class (by retrieving information from the database) and if so adds the student information to the class roster in the database.

- **Application programmers** are computer professionalswho write application programs. Application programmers can choose frommany tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reportswith minimal programming effort.
- **Sophisticated users** interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.
- **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, knowledgebase and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems. Chapter 22 covers several of these applications.

1.12.2 Database Administrator

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator (DBA)**. The functions of a DBA include:

- **Schema definition**. The DBA creates the original database schema by executing a set of data definition statements in the DDL.

- **Storage structure and access-method definition**.

- **Schema and physical-organization modification**. TheDBAcarries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.

(Database System Concepts)

• **Granting of authorization for data access**. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.

• **Routine maintenance**. Examples of the database administrator's routine maintenance activities are:

◦ Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
◦ Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
◦ Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

1.13 History of Database Systems

Information processing drives the growth of computers, as it has from the earliest days of commercial computers. In fact, automation of data processing tasks predates computers. Punched cards, invented by Herman Hollerith, were used at the very beginning of the twentieth century to record U.S. census data, and mechanical systemswere used to process the cards and tabulate results. Punched cards were later widely used as a means of entering data into computers.

Techniques for data storage and processing have evolved over the years:

• **1950s and early 1960s**:Magnetic tapes were developed for data storage. Data processing tasks such as payroll were automated, with data stored on tapes.

Processing of data consisted of reading data from one or more tapes and writing data to a new tape. Data could also be input from punched card decks, and output to printers. For example, salary raises were processed by entering the raises on punched cards and reading the punched card deck in synchronization with a tape containing the master salary details. The records had to be in the same sorted order. The salary raises would be added to the salary read from the master tape, and written to a new tape; the new tape would become the new master tape.

Tapes (and card decks) could be read only sequentially, and data sizeswere much larger than main memory; thus, data processing programs were forced to process data in a particular order, by reading and merging data fromtapes and card decks.

• **Late 1960s and 1970s**:Widespread use of hard disks in the late 1960s changed the scenario for data processing greatly, since hard disks allowed direct access to data. The position of data on disk was immaterial, since any location on disk could be accessed in just tens of milliseconds. Data were thus freed from

The Design Process (http://en.wikipedia.org/wiki/Database_design#cite_note-3 "Database design – Wikipedia")

The design process consists of the following steps

1- **Determine the purpose of your database** - This helps prepare you for the remaining steps.
2- **Find and organize the information required** - Gather all of the types of information you might want to record in the database, such as product name and order number.
3- **Divide the information into tables** - Divide your information items into major entities or subjects, such as Products or Orders. Each subject then becomes a table.
4- **Turn information items into columns** - Decide what information you want to store in each table. Each item becomes a field, and is displayed as a column in the table. For example, an Employees table might include fields such as Last Name and Hire Date.
5- **Specify primary keys** - Choose each table's primary key. The primary key is a column that is used to uniquely identify each row. An example might be Product ID or Order ID.
6- **Set up the table relationships** - Look at each table and decide how the data in one table is related to the data in other tables. Add fields to tables or create new tables to clarify the relationships, as necessary.
7- **Refine your design** - Analyze your design for errors. Create the tables and add a few records of sample data. See if you can get the results you want from your tables. Make adjustments to the design, as needed.

Normalization

Normalization In relational database , is the process of organizing data to minimize redundancy . The goal of database normalization is to decompose complex relations(tables) in order to produce smaller, well-structured relations(tables).

Normalization usually involves dividing large, badly-formed tables into smaller, well-formed tables and defining relationships between them. The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships

Free the database of modification anomalies

When an attempt is made to modify (update, insert into, or delete from) a table, undesired side-effects may follow. Not all tables can suffer from these side-effects; rather***, the side-effects can only arise in tables that have not been sufficiently normalized***. An insufficiently normalized table might have one or more of the following characteristics:

- The same information can be expressed on multiple rows; therefore updates to the table may result in logical inconsistencies. For example, each record in an "Employees' Skills" table might contain an Employee ID, Employee Address, and Skill; thus a change of address for a particular employee will potentially need to be applied to multiple records (one for each of his skills). If the update is not carried through successfully—if, that is, the employee's address is updated on some records but not others—then the table is left in an inconsistent state. Specifically, the table provides conflicting answers to the question of what this particular employee's address is. This phenomenon is known as an **update anomaly**.
- There are circumstances in which certain facts cannot be recorded at all. For example, each record in a "Faculty and Their Courses" table might contain a Faculty ID, Faculty Name, Faculty Hire Date, and Course Code—thus we can record the details of any faculty member who teaches at least one course, but we cannot record the details of a newly-hired faculty member who has not yet been assigned to teach any courses except by setting the Course Code to null. This phenomenon is known as an **insertion anomaly**.
- There are circumstances in which the deletion of data representing certain facts necessitates the deletion of data representing completely different facts. The "Faculty and Their Courses" table described in the previous example suffers from this type of anomaly, for if a faculty member temporarily ceases to be assigned to any courses, we must delete the last of the records on which that faculty member appears, effectively also deleting the faculty member. This phenomenon is known as a **deletion anomaly**

Employees' Skills

Employee ID	Employee Address	Skill
426	87 Sycamore Grove	Typing
426	87 Sycamore Grove	Shorthand
519	94 Chestnut Street	Public Speaking
519	96 Walnut Avenue	Carpentry

An **update anomaly**. Employee 519 is shown as having different addresses on different records.

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201
424	Dr. Newsome	29-Mar-2007	?

An **insertion anomaly**. Until the new faculty member, Dr. Newsome, is assigned to teach at least one course, his details cannot be recorded.

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201

DELETE

A **deletion anomaly**. All information about Dr. Giddens is lost when he temporarily ceases to be assigned to any courses.

→ : Levels of normalization

To normalize a database, there are three levels :

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)

1: First Normal Form (1NF)
The role of (1NF) is: No repeating elements or groups of elements, this mean:

- Eliminate duplicative columns from the same table.
- Create separate tables for each group of related data and identify each row with a unique column or set of columns (the primary key).

The first rule dictates that we must not duplicate data within the same row of a table. For example consider the table of figure (7.4) of a bank.

Person ID	Name	City Name	City Number	Account Type	Balance	Account Notes
123	Nader	Baghdad	1	A	4556 $	******
123	Nader	Baghdad	1	B	7654 $	######
123	Nader	Baghdad	1	C	1287 $	&&&&&
150	Muna	Basra	2	A	654 $	******
150	Muna	Basra	2	B	66743 $	######

Figure (7.4) Accounts table of a bank

The table of figure (7.4) consists of repeated information.
The meaning of repeated information is all the information belong for a single person. For person "NADER" there are three records repeated for him.
For "Muna" there are two record repeated for her.

The table must be **separated** from the repeated information into **two tables** as shown in figure (7.5). The two tables are joined by the ***Person ID*** as a key. For table 2 the primary key is the combination of the first two fields 'Person Id + Account Type'

Primary Key

Person ID	Name	City Name	City Number
123	Nader	Baghdad	1
150	Muna	Basra	2

Table 1

The Primary Key

Person ID	Account Type	Balance	Account Notes
123	A	4556 $	******
123	B	7654 $	######
123	C	1287 $	&&&&&
150	A	654 $	******
150	B	66743 $	######

Table 2

Figure (7.5) the new two tables of (1NF)

2: Second Normal Form (2NF)

A table that has a concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column only depends upon one part of the concatenated key, then we say that the entire table has failed Second Normal Form and we must create another table to rectify the failure

We looks at the tables with a primary key made from many fields, for each non key fields that do not depends of the primary key (with all the fields of the primary key), this non key fields must be separated in another table.
Table 1 of figure (7.5) have a primary key with single fields, this mean it is in (2NF).

Table 2 of figure (7.5) , field BALANCE depends on both the fields of the primary key, because each balance we must know the person and the account for it.

But "ACCOUNT NOTS" depends only on the second field (Account Type) because the not of the account is the same for all the persons .

Figure (7.6) shows the (2NF)

Person ID	Name	City Name	City Number
123	Nader	Baghdad	1
150	Muna	Basra	2

Table 1 with no change

Person ID	Account Type	Balance
123	A	4556 $
123	B	7654 $
123	C	1287 $
150	A	654 $
150	B	66743 $

Table 2

Account Type	Account Notes
A	******
B	######
C	&&&&&

Table 3

Table 2 of figure (7.5) is separated into two tables

Figure (7.6) the Second Normal Form (2NF)

3 : Third Normal Form (3NF)

Third normal form (3NF) is to remove columns that are not dependent upon the primary key. This means we look at the non key fields and try to find a relation between them.

Table 2 and table 3 of figure (7.6) are in the (3NF). But table 1 is not.

In table 1 we find that City Number depends on City Name , not on the key. We separate these two fields on a table and link this new table with the exist one as shown in figure (7.7)

Person ID	Name	City Number
123	Nader	1
150	Muna	2

City Number	City Name
1	Baghdad
2	Basra

The new two tables in (3NF) of table1 of figure (7.6)

Figure (7.7)

Example: Find the appropriate tables in 3NF for the following inventory form

Store Inventory form

Store No :
Store Address :
Date of inventory :

Part No.	Description	Location	Quantity

Solution:

First we must find the unnormalized table :

Store no	Store address	Date of inventory	Location	Part No	Desc	Qty

1- The 1NF will be:

Table 1

Store no	Store address	Date of inventory

Primary Key

Table 2

Store no	Location	Part No	Desc	Qty

Primary Key

2- The 2NF : **(non-key fields depends on Key-field)**

Table 1 of 1Nf is the 2NF with no change.

Table 2 has the following problem: when we look at the non-key fields, we find that two fields **(Part No) and (Qty)** depends on the entire primary key, but **(Desc) don't depend on any part of the primary key**. instead (Desc) depends on non-key field (Part No).

The 2Nf will be

Table 1

Store no	Store address	Date of inventory

Primary Key

(table 2 of 1NF will be)

Table 2 on 2NF

Store no	Location	Part No	Qty

Primary Key

Table 3 of 2NF

Part No	Desc

Primary Key

When we splits (Desc) on a new table 3 , we put with it (Part No) to connect it with table 2

3- The 3NF : **(Non-key field depends on non-key field)**

3NF is the same as 2NF.

Problem : look at table 1 of 3NF (or 2NF) , if any store have many inventory bill, each bill with a deferent date. In table 1 , we must sore a record for each date and repeat the store no ***and the address*** . This is a problem.
It is better to split table 1 into two tables as bellow :

Table 1-1

Store no	Store address

Table 1-2

Store no	Date of inventory

Another problem arise : in table 2 how we will know each record belong to which date, i.e. how we shall link table 2 with table 1-2 ?

CHAPTER17 (Database System Concepts)

Database-System Architectures

The architecture of a database system is greatly influenced by the underlying computer system on which it runs, in particular by such aspects of computer architecture as networking, parallelism, and distribution:

• Networking of computers allows some tasks to be executed on a server system and some tasks to be executed on client systems. This division of work has led to *client–server database systems*.

• Parallel processing within a computer system allows database-system activities to be speeded up, allowing faster response to transactions, as well as more transactions per second. Queries can be processed in a way that exploits the parallelism offered by the underlying computer system. The need for parallel query processing has led to *parallel database systems*.

• Distributing data across sites in an organization allows those data to reside where they are generated ormost needed, but still to be accessible from other sites and from other departments. Keeping multiple copies of the database across different sites also allows large organizations to continue their database operations even when one site is affected by a natural disaster, such as flood, fire, or earthquake. *Distributed database systems* handle geographically or administratively distributed data spread across multiple database systems.
We study the architecture of database systems in this chapter, starting with the traditional centralized systems, and covering client–server, parallel, and distributed database systems.

17.1 Centralized and Client–Server Architectures
Centralized database systems are those that run on a single computer system and do not interact with other computer systems. Such database systems span a range from single-user database systems running on personal computers to high-performance database systems running on high-end server systems. Client–server systems, on the other hand, have functionality split between a server system and multiple client systems.

17.1.1 Centralized Systems

A modern, general-purpose computer system consists of one to a few processors and a number of device controllers that are connected through a common bus that provides access to shared memory (Figure 17.1).

The processors have local cache memories that store local copies of parts of the memory, to speed up access to data. Each processor may have several independent **cores**, each of which can execute a separate instruction stream. Each device controller is in charge of a specific type of device (for example, a disk drive, an audio device, or a video display).
The processors and the device controllers can execute concurrently, competing for memory access. Cache memory reduces the contention for memory access, since it reduces the number of times that the processor needs to access the shared memory.

We distinguish two ways in which computers are used: as single-user systems and as multiuser systems. Personal computers and workstations fall into the first category. A typical **single-user system** is a desktop unit used by a single person, usually with only one processor and one or two hard disks, and usually only one person using the machine at a time. A typical **multiuser system**, on the other hand, has more disks and more memory and may have multiple processors. It serves a large number of users who are connected to the system remotely.

Database systems designed for use by single users usually do not provide many of the facilities that a multiuser database provides. In particular, they may not support concurrency control, which is not required when only a single user can generate updates.

Provisions for crash recovery in such systems are either absent or primitive—for example, they may consist of simply making a backup of the database before any update. Some such systems do not support SQL, and they provide a simpler query language, such as a variant of QBE. In contrast,

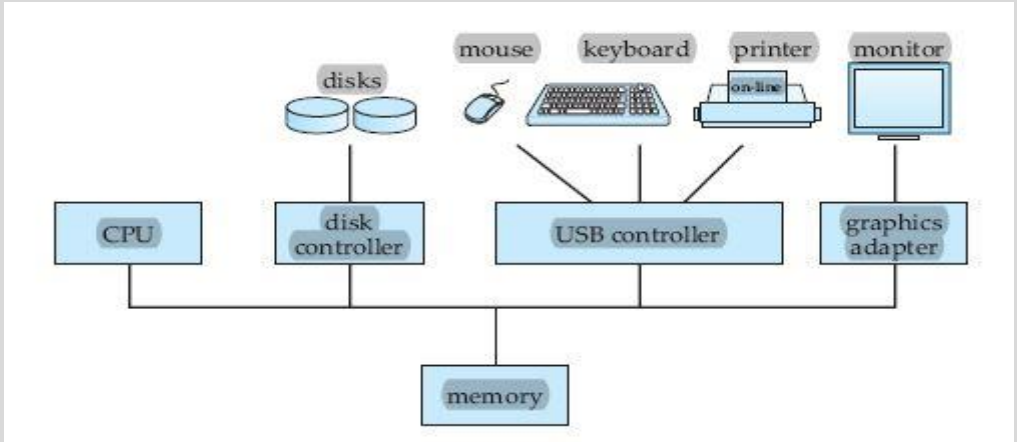


Figure 17.1 A centralized computer system.

17.1 Centralized and Client–Server Architectures 771

database systems designed for multiuser systems support the full transactional features that we have studied earlier.

Although most general-purpose computer systems in use today have multiple processors, they have **coarse-granularity parallelism**, with only a few processors (about two to four, typically), all sharing the main memory. Databases running on such machines usually do not attempt to partition a single query among the processors; instead, they run each query on a single processor, allowing multiple queries to run concurrently. Thus, such systems support a higher throughput; that is, they allow a greater number of transactions to run per second, although individual transactions do not run any faster.

Databases designed for single-processor machines already provide multitasking, allowing multiple processes to run on the same processor in a time-shared manner, giving a view to the user of multiple processes running in parallel. Thus, coarse-granularity parallel machines logically appear to be identical to single processor machines, and database systems designed for time-shared machines can be easily adapted to run on them.

In contrast, machines with **fine-granularity parallelism** have a large number of processors, and database systems running on such machines attempt to parallelize single tasks (queries, for example) submitted by users. We study the architecture of parallel database systems in Section 17.3.

Parallelism is emerging as a critical issue in the future design of database systems. Whereas today those computer systems with multicore processors have only a few cores, future processors will have large numbers of cores.1 As a result, parallel database systems, which once were specialized systems running on specially designed hardware, will become the norm.

17.1.2 Client–Server Systems

As personal computers became faster, more powerful, and cheaper, there was a shift away from the centralized system architecture. Personal computers supplanted terminals connected to centralized systems.

Correspondingly, personal computers assumed the user-interface functionality that used to be handled directly by the centralized systems. As a result, centralized systems today act as **server systems** that satisfy requests generated by *client systems*.

Figure 17.2 shows the general structure of a client–server system.

Functionality provided by database systems can be broadly divided into two parts—the front end and the back end. The back end manages access structures, query evaluation and optimization, concurrency control, and recovery. The front end of a database system consists of tools such as the SQL user interface, forms interfaces, report generation tools, and data mining and analysis tools (see Figure 17.3). The interface between the front end and the back end is through SQL, or through an application program.

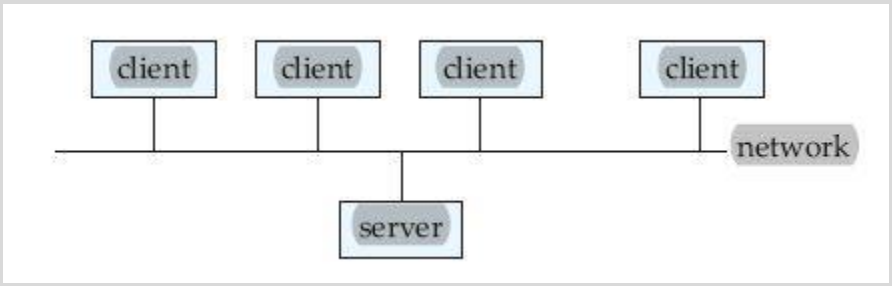


Figure 17.2 General structure of a client–server system.

Standards such as *ODBC* and *JDBC*,whichwe saw inChapter 3,were developed to interface clients with servers. Any client that uses the ODBC or JDBC interface can connect to any server that provides the interface. Certain application programs, such as spreadsheets and statistical-analysis packages, use the client–server interface directly to access data from a back-end server. In effect, they provide front ends specialized for particular tasks. Systems that deal with large numbers of users adopt a three-tier architecture, which we saw earlier in Figure 1.6 (Chapter 1), where the front end is a Web browser that talks to an application server. The application server, in effect, acts as a client to the database server.

Some transaction-processing systems provide a **transactional remote procedure call** interface to connect clients with a server. These calls appear like ordinary procedure calls to the programmer, but all the remote procedure calls from a client are enclosed in a single transaction at the server end. Thus, if the transaction aborts, the server can undo the effects of the individual remote procedure calls.

17.2 Server System Architectures

Server systems can be broadly categorized as transaction servers and data servers.

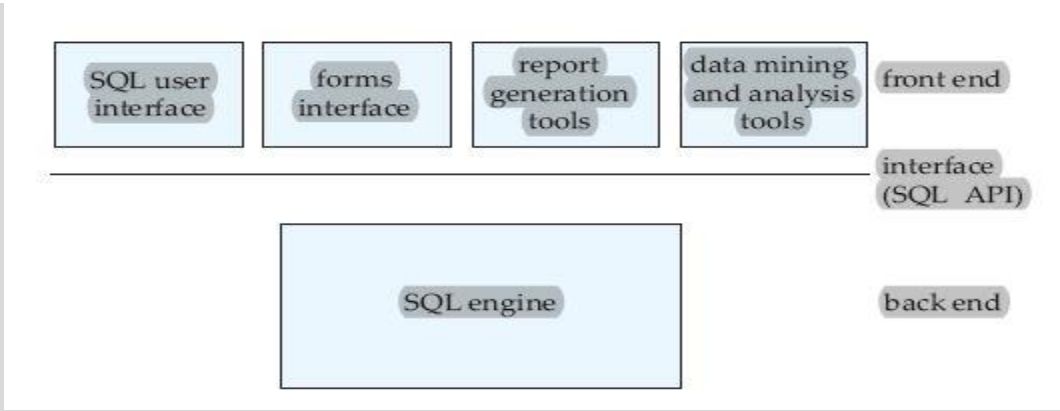


Figure 17.3 Front-end and back-end functionality.

17.2.3 Cloud-Based Servers

Servers are usually owned by the enterprise providing the service, but there is an increasing trend for service providers to rely at least in part upon servers that are owned by a "third party" that is neither the client nor the service provider. One model for using third-party servers is to outsource the entire service
to another company that hosts the service on its own computers using its own
software. This allows the service provider to ignore most details of technology
and focus on the marketing of the service.

Another model for using third-party servers is **cloud computing**, in which the service provider runs its own software, but runs it on computers provided by another company. Under this model, the third party does not provide any of the application software; it provides only a collection of machines. These machines are not "real" machines, but rather simulated by software that allows a single real computer to simulate several independent computers. Such simulated machines are called **virtual machines**. The service provider runs its software (possibly including a database system) on these virtual machines. A major advantage of cloud computing is that the service provider can add machines as needed to meet demand and release them at times of light load. This can prove to be highly cost-effective in terms of both money and energy.

Athird model uses a cloud computing service as a data server; such *cloud-based data storage* systems are covered in detail in Section 19.9. Database applications using cloud-based storage may run on the same cloud (that is, the same set of machines), or on another cloud. The bibliographical references provide more information about cloud-computing systems.

17.3 Parallel Systems

Parallel systems improve processing and I/O speeds by using multiple processors and disks in parallel. Parallel machines are becoming increasingly common, making the study of parallel database systems correspondingly more important. The driving force behind parallel database systems is the demands of applications that have to query extremely large databases (of the order of terabytes—that is, 10^{12} bytes) or that have to process an extremely large number of transactions per second
(of the order of thousands of transactions per second). Centralized and client–server database systems are not powerful enough to handle such applications.

In parallel processing, many operations are performed simultaneously, as opposed to serial processing, in which the computational steps are performed sequentially.

A**coarse-grain** parallelmachine consists of a small number of powerful processors; a **massively parallel** or **fine-grain parallel** machine uses thousands of smaller processors.

Virtually all high-end machines today offer some degree of coarse-grain parallelism: at least two or four processors.

Massively parallel computers can be distinguished from the coarse-grain parallelmachines by the much larger degree of parallelism that they support. Parallel computers with hundreds of processors and disks are available commercially.

There are two main measures of performance of a database system:
(1) **throughput**, the number of tasks that can be completed in a given time interval, and
(2) **response time**, the amount of time it takes to complete a single task from the time it is submitted. A system that processes a large number of small transactions can improve throughput by processing many transactions in parallel. A system that processes large transactions can improve response time as well as throughput by performing subtasks of each transaction in parallel.

17.3.1 Speedup and Scaleup

Two important issues in studying parallelism are speedup and scaleup. Running a given task in less time by increasing the degree of parallelism is called **speedup**. Handling larger tasks by increasing the degree of parallelism is called **scaleup**.
Consider a database application running on a parallel system with a certain number of processors and disks. Now suppose that we increase the size of the system by increasing the number of processors, disks, and other components of the system. The goal is to process the task in time inversely proportional to the number of processors and disks allocated. Suppose that the execution time of a task on the larger machine is *TL* , and that the execution time of the same task on the smaller machine is *TS*. The speedup due to parallelism is defined as *TS/TL*. The parallel system is said to demonstrate **linear speedup** if the speedup is *N* when the larger system has *N* times the resources (processors, disk, and so on) of the smaller system. If the speedup is less than *N*, the system is said to demonstrate **sublinear speedup**. Figure 17.5 illustrates linear and sublinear speedup.
Scaleup relates to the ability to process larger tasks in the same amount of time by providing more resources. Let *Q* be a task, and let *QN* be a task that is *N* times bigger than *Q*. Suppose that the execution time of task *Q* on a given machine
linear speedup
sublinear speedup
resources
speed

17.3.3.4 Hierarchical

The **hierarchical architecture** combines the characteristics of shared-memory, shared-disk, and shared-nothing architectures. At the top level, the system consists of nodes that are connected by an interconnection network and do not share disks or memory with one another. Thus, the top level is a shared-nothing architecture.

Each node of the system could actually be a shared-memory system with a few processors. Alternatively, each node could be a shared-disk system, and each of the systems sharing a set of disks could be a shared-memory system. Thus, a system could be built as a hierarchy, with shared-memory architecture with a few processors at the base, and a shared-nothing architecture at the top, with possibly a shared-disk architecture in the middle. Figure 17.8d illustrates a hierarchical architecture with shared-memory nodes connected together in a shared-nothing architecture. Commercial parallel database systems today run on several of these architectures.

Attempts to reduce the complexity ofprogramming such systems have yielded **distributed virtual-memory** architectures, where logically there is a single shared memory, but physically there are multiple disjoint memory systems; the virtual memory-mapping hardware, coupled with system software, allows each processor to view the disjoint memories as a single virtual memory. Since access speeds differ, depending on whether the page is available locally or not, such anarchitecture is also referred to as a **nonuniform memory architecture** (**NUMA**).

17.4 Distributed Systems

In a **distributed database system**, the database is stored on several computers. The computers in a distributed system communicate with one another through various communication media, such as high-speed private networks or the Internet. They do not share main memory or disks.

The computers in a distributed system may vary in size and function, ranging from workstations up to mainframe systems.

The computers in a distributed system are referred to by a number of different names, such as **sites** or **nodes**, depending on the context in which they are mentioned. We mainly use the term **site**, to emphasize the physical distribution of these systems. The general structure of a distributed system appears in Figure 17.9.

The main differences between shared-nothing parallel databases and distributed databases are that distributed databases are typically geographically separated, are separately administered, and have a slower interconnection. Another major difference is that,

in a distributed database system, we differentiate between local and global transactions. A **local transaction** is one that accesses data only from sites where the transaction was initiated. A **global transaction**, on the other hand, is one that either accesses data in a site different from the one at which the transaction was initiated, or accesses data in several different sites.

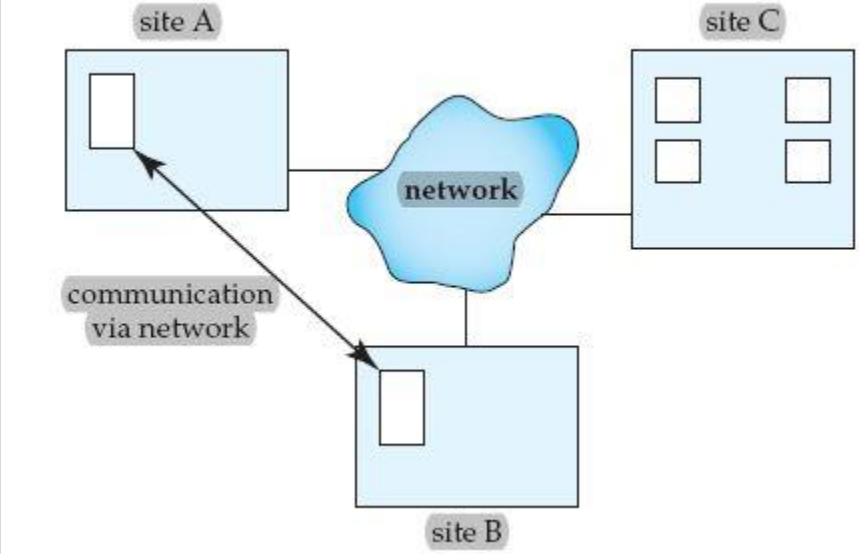


Figure 17.9 A distributed system.

There are several reasons for buildingdistributeddatabase systems, including sharing of data, autonomy, and availability.

• **Sharing data.** Themajor advantage inbuilding a distributeddatabase system is the provision of an environment where users at one site may be able to access the data residing at other sites. For instance, in a distributeduniversity system, where each campus stores data related to that campus, it is possible for a user in one campus to access data in another campus. Without this capability, the transfer of student records fromone campus toanother campus would have to resort to some external mechanism that would couple existing systems.

• **Autonomy.** The primary advantage of sharing data by means of data distribution is that each site is able to retain a degree of control over data that are stored locally. In a centralized system, the database administrator of the central site controls the database. In a distributed system, there is a global database administrator responsible for the entire system. A part of these responsibilities is delegated to the local database administrator for each site.

Depending on the design of the distributed database system, each administratormay have a different degree of **local autonomy**. The possibility of local autonomy is often a major advantage of distributed databases.

• **Availability.** If one site fails in a distributed system, the remaining sites may be able to continue operating. In particular, if data items are **replicated** in several sites, a transaction needing a particular data itemmay find that item in any of several sites. Thus, the failure of a site does not necessarily imply the shutdown of the system.

9 Transactions

9.1 : Definition :A transaction is unit of program execution that accesses and possibly updates various data items.

Often a collection of several operations on the database is considered to be a single unit from the point of view of the user. For example a transfer of funds from a checking account to a saving account is a single operation for the user , but for the database it comprise several operations.
A database system must ensure proper execution of transaction, either the entire transaction is executed or none of it does.
To ensure integrity of the data, the database system maintains the following properties of the transaction:

1- Atomicity : either all operations of the transaction are executed of none are.
2- Consistency : execution of the transaction in isolation to preserve the consistency of the database
3- Isolation : if there are two transaction Ta and Tb ; it appear for Ta that Tb either finished before Ta started , or Tb start execution after Ta finished.
Thus each transaction unaware of other transactions executing concurrently in the system.
4- Durability : after a transaction completes successfully , the change it has made to the database persist even if there are a system failures.

For example , access to the database is accomplished by the two operation:

- Read (X) : is to read X from the database to a local buffer belonging to the transaction.
- Write (X) : is to write X to the database from the local buffer.

Let Ta be a transaction that transfer 50$ from account A to account B as follow:

```
Ta :  Read(A)
      A=A – 50;
      Write (A)
      Read(B)
      B=B + 50;
      Write(B)
```

Let us consider ***Atomicity*** : the database system keeps track (on disk) of the old values of any data on which a transaction performs a write, and if the transaction does not complete its execution, the old value is restored to make it appear as though the transaction never executed.

Let us consider ***Consistency*** : the consistency required here is the sum of A and B be unchanged.

Without this consistency, money could be created or destroyed by the transaction. If the database is consistent before the execution of the transaction then the database must remain consistent after the execution of the transaction.

Let us consider ***Durability*** : we assume that system failure may result of losing data in main memory but data written to disk are never lost. We can guarantee durability by:

1- The update carried by the transaction have been written to disk before the transaction completes.
2- There are sufficient Information about the update to enable the database to reconstruct the update when the system is restarted after the failure.

Let us consider ***Isolation*** : while executing the Ta transaction , the Ta transaction execute the third statement (Write (A)) and before executing the last statement (Write (B)), another transaction Tb read the values of A and B , and calculate the sum (A+B), then the consistency of transaction Tb will discover an error of the values of A and B.

The solution of the problem of concurrently execution is to execute the transaction in sequence, one after another.

9.2 : Implementation of atomicity and durability

The recovery-management system is the one responsible of atomicity and durability. One way to implement them is to use a copy of the database, and a pointer that point at the current copy as shown in figure (9.1).

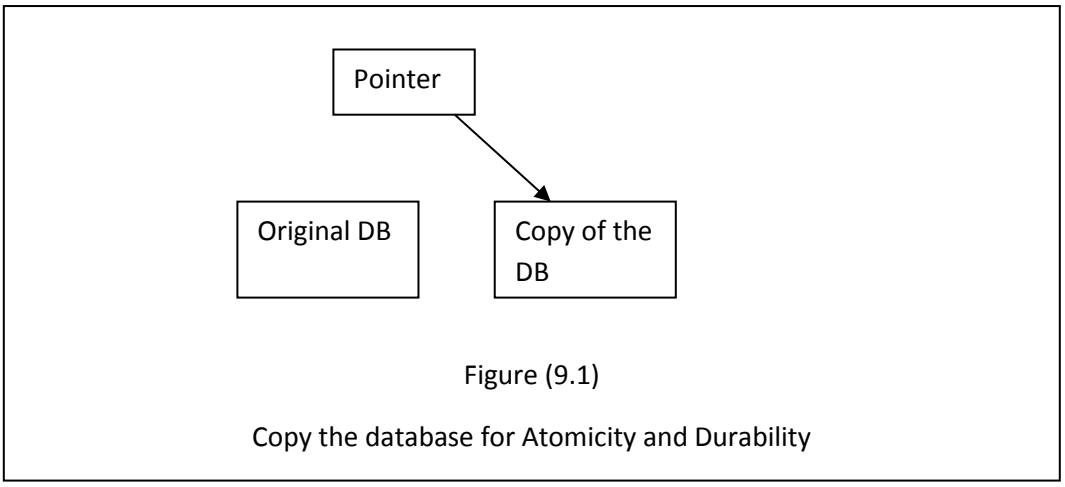


Figure (9.1)

Copy the database for Atomicity and Durability

If a transaction want to update the database , all updates are done on the copy database. If an error occur during the execution of the transaction then the copy is deleted and the system return to the original database.

9.3 : Concurrent Execution

Transaction processing system usually allow multiple transaction to run concurrently. When several transaction run concurrently, database consistency can be destroyed.

The database system must control the interaction among the concurrent transaction to prevent them from destroying the consistency of the database.

As an example , consider a banking system which has several accounts, and a set of transactions that access and updates these accounts.

Let T1 and T2 be two transactions that transfer funds from one account to another, they can be executed one after another T1 then T2 as shown in figure (9.2) or T2 then T1 as shown in figure (9.3).

T1	T2
Read(A) A=A-50 Write(A) Read (B) B=B+50 Write (B)	
	Read (A) temp=A * 0.1 A=A-temp Write (A) Read(B) B=B+temp Write(B)

Figure (9.2)
Execution of T1 then T2

T1	T2
	Read (A) temp=A * 0.1 A=A-temp Write (A) Read(B) B=B+temp Write(B)
Read(A) A=A-50 Write(A) Read (B) B=B+50 Write (B)	

Figure (9.3)
Execution of T2 then T1

When several transactions are executed concurrently , the system may execute one transaction for a little while , then switch to another transaction and execute it for a while then switch back to the first one, and so on as shown in figure (9.4)

T1	T2
Read(A) A=A-50 Write(A)	
	Read (A) temp=A * 0.1 A=A-temp Write (A)
Read (B) B=B+50 Write (B)	
	Read(B) B=B+temp Write(B)

Figure (9.4)
Execution of two transaction

10 : Database Security

10.1 : Introduction

There is a need to secure computer systems ,and securing data must be part of an overall computer security plan. Growing amounts of sensitive data are being retained in databases and more of these databases are being made accessible via the Internet. As more data is made available electronically, it can be assumed that threats and vulnerabilities to the integrity of that data will increase as well.

10.2 : Security objective

The primary objectives of database security are:

- Confidentiality : access control
- Integrity : data corruption
- Availability :

To preserve the data confidentiality, enforcing access control policies on the data, these policies are defined on the database management system (DBMS). Access control is to insure that only authorized users perform authorized activities at authorized time.

There are two points of concern in access control:

- Authentication
- Authorization

To preserve data integrity, we must guaranty that the data cannot be corrupted in an invisible way.

Availability property is to ensure timely and reliable access to the database.

10.3 Access control

Access control is the process by which rights and privileges are assigned to users and database objects. Database objects include tables, views, rows and columns. To ensure proper access to the data, authentication and authorization are applied.

Authentication is the process by which you verify that someone is who they claim they are. This usually involves a user name and a password, but can include any other method of demonstrating identity, such as a smart card, voice recognition or fingerprints.

Authentication is equivalent to showing your driver license or your ID.

Authorization is finding out that the person, once identified , is permitted to have the resource. This is usually determined by finding out if that person has paid admission or has a particular level of security clearness.

Authorization is equivalent to checking your name in a list of names, or checking your ticket for something.

For Example, student A may be given login rights to the university database with authorization includes read only for the course listing data table.

10.4 Database encryption

Application data security has to deal with several security threats and issues beyond hose handle by SQL authorization. For example , data must be protected while they are being transmitted, data may need be protected from intruders.

Encryption is the process of transferring a clear text (plain text) into disguised text (cipher text) by using a key. Decryption is the process to transfer the cipher text back to plain text as shown in figure 10.1

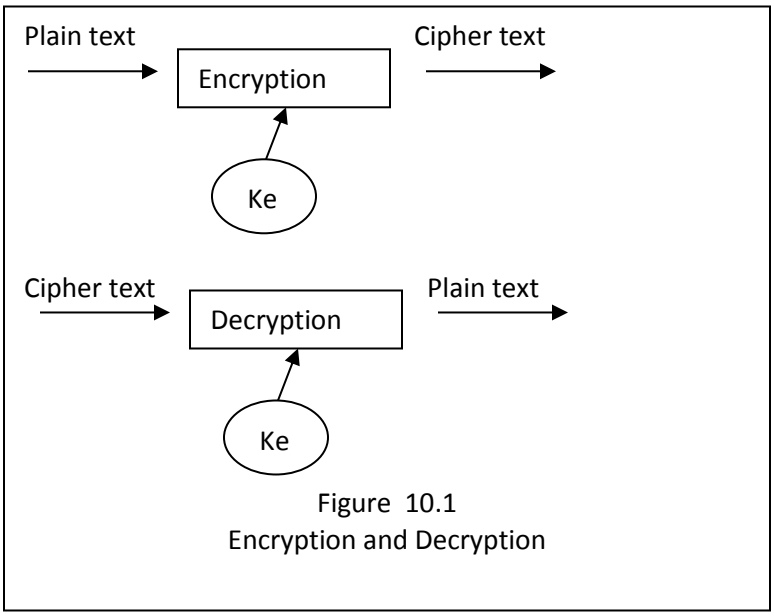


Figure 10.1
Encryption and Decryption

Database encryption refers to the use of encryption techniques to transform a plain text into encrypted database, thus making it unreadable to anyone except those who possess the knowledge of the encryption.

The purpose of database encryption is to ensure the database opacity by keeping the information hidden to any unauthorized persons. Even if someone get through and bypass the access control policies , he/she still unable to read the encrypted database.

There are a vast number of techniques for the encryption of data. An example for simple encryption techniques is to substitute of each character with the next character in the alphabet. ***For example 'Perryridge' becomes 'Qfsszsjehf'.***

A good encryption techniques has the following properties:

1- It is relatively simple for authorized users to encrypt and decrypt data.
2- It depends not on the security of the algorithm, but rather on a parameter of the algorithm called the encryption key.
3- Its encryption key is extremely difficult for an intruder to determine.

10.5 Database encryption level

10.5.1 Application-Level Encryption:

When you encrypt information at the application level, you can protect sensitive data. In many ways the application is the obvious place to encrypt and decrypt data because the application knows exactly which data is sensitive and can apply protection selectively.
You can task a given application with encrypting its own data. This encryption capability is designed into the application itself. By the time the database receives the data, it has already been encrypted and then stored in the database in this encrypted state. As the traffic travels from the application to the database, the data can also be encrypted across the network.

10.5.2 Database Encryption Level:

In this case the information is encrypted in the database. As an example, we'll discuss ***Oracle Advanced Security's transparent data encryption (TDE)***, which automatically encrypts and decrypts the data stored in the database and provides this capability without having to write additional code.
With TDE, the encryption process and associated encryption keys are created and managed by the database. This is transparent to database users who have authenticated to the database. At the operating system, however, attempts to access database files return data in an encrypted state. Therefore, for any operating system level users, the data remains inaccessible. Additionally, because the database is doing the encryption, there is no need to change the application(s), and there is a minimal performance overhead when changes occur in the database. TDE is designed into the database itself: Oracle has integrated the TDE syntax with its data definition language (DDL).
If you encrypt on the database, that means the data is sent to and from the database in unencrypted form. This potentially allows for snooping/tampering between the application and the encryption routines on the database.

11 : Fundamental of relational algebra:

Relational algebra consists of a set of operations that takes one or two relations as input and produce a new relation as their result. In the relational algebra, symbols are used to denote an operation.

- For SELECT we use the sigma letter σ . The relation(table name) is written in parentheses:
  Select * from Loan where B_name="Perryridge:
  will be : $\sigma_{B\_name="Perryridge}$ (Loan)

- For projection we use the pi letter π . projection mean select some fields from the table , not all the fields.
  If we have : Student (S-Id,S-Name,S-Address)
  To display only the names → Select S-Name from student
  will be : $\pi_{S\text{-}Name}$ (Student)

- Selection (σ) and Projection (π) can be used together to select some of the fields with a condition.
  To display only the names whose address in Baghdad from student table:
  ➔ Select S-Name from student where S-Address='Baghdad'

  Will be : $\pi_{S\text{-}Name}$ ($\sigma_{s\text{-}address='Baghdad'}$ (Student))

- Cartesian product (Cross Join) between two tables denoted by X.
  It is used to combine information from any two relations. It will produce a tuple from each possible pair of tuples: one from the first table and one from the second.

  Ex 1 : To Cross Join between Student and Class ; with the condition only for level 2
  ➔ $\sigma_{LVL=2}$ (Student X Class)

  Example 2: we have two tables as shown in figure (11.1)

tables1 :

Loan (loan-no ,branch-name, amount)

loan-no	Branch-name	amount
L-11	Round hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Minus	500

tables2 :

Borrower (customer-name, loan-number)

Customer-name	Loan-number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure (11.1)

If we want to know the names of all customers who have a lone at the Perryridge branch. So if we write : $\sigma_{\text{branch-name=' Perry ridge'}}$ (Borrower X Loan) ; the result of this cross join shown in figure (11.2).

customer-name	Borrower. loan-number	Loan. loan-number	branch-name	amount
Adams	L-16	L-15	Perry ridge	1500
Adams	L-16	L-16	Perry ridge	1300
Curry	L-93	L-15	Perry ridge	1500
Curry	L-93	L-16	Perry ridge	1300
Hayes	L-15	L-15	Perry ridge	1500
Hayes	L-15	L-16	Perry ridge	1300
Jackson	L-14	L-15	Perry ridge	1500
Jackson	L-14	L-16	Perry ridge	1300
Jones	L-17	L-15	Perry ridge	1500
Jones	L-17	L-16	Perry ridge	1300
Smith	L-11	L-15	Perry ridge	1500
Smith	L-11	L-16	Perry ridge	1300
Smith	L-23	L-15	Perry ridge	1500
Smith	L-23	L-16	Perry ridge	1300
Williams	L-17	L-15	Perry ridge	1500
Williams	L-17	L-16	Perry ridge	1300

Figure (11.2)
the result of cross join $\sigma_{\text{branch-name=' Perry ridge'}}$ (Borrower X Loan)

THE RESULT IS NOT RIGHT!!.

The Cross Join links every record from Borrower with all the records of Loan who have Perry ridge in branch-name.
The correct answer will be :

$\sigma_{\text{borrower. loan-number=loan.loan-number}}$ ($\sigma_{\text{branch-name=' Perry ridge'}}$ (Borrower X Loan))

Adams	L-16	L-16	Perry ridge	1300
Hayes	L-15	L-15	Perry ridge	1500

- The natural join operation
  It allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the join symbol (⋈). The natural join operation do the following:
  - Cartesian product of its arguments (ex: two tables)
  - Perform selection forcing equality on those attributes that appear in both tables.
  - Remove duplicate attributes.

  Example 11.1: Consider the borrower and loan tables in figure (11. 1), to find the names of all customers who have a loan at the bank, and find the amount of the loan:

  $\Pi_{\text{customer-name,loan-number,amount}}(\text{borrower} \bowtie \text{loan})$

  Because borrower and loan tables both have the attribute loan-number, the natural join operation considers only pairs of tuples from the two tables that have the same value on loan-number. The result will be as shown in figure (11.3) :

Customer-name	Loan-number	amount
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-11	900
Smith	L-23	2000
Williams	L-17	1000

Figure (11.3)
The result of natural join $\Pi_{\text{customer-name,loan-number,amount}}(\text{borrower} \bowtie \text{loan})$

Example 11.2 : Notes: the natural join usually required that the two relations must have at least one common attribute, but if this constraint is omitted, and the two relations have no common attributes, then the natural join becomes exactly the Cartesian product as shown in figure (11.4)

Car

CarModel	CarPrice
CarA	20,000
CarB	30,000

Boat

BoatModel	BoatPrice
Boat1	10,000
Boat2	40,000

Car ⋈ ***Boat***

CarModel	CarPrice	BoatModel	BoatPrice
CarA	20,000	Boat1	10,000
CarA	20,000	Boat2	40,000
CarB	30,000	Boat1	10,000
CarB	30,000	Boat2	40,000

Figure (11.4)
Natural join becomes a cross join because there are no common attribute

Example 11.3 : find the names of all branches with customers who have an account in the bank and who live in Harrison for the relations shown in figure (11.5).

Customer relation

Customer-name	Customer-street	Customer-city
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Account relation

Account-number	Branch-name	Balance
A-101	Downtown	500
A-215	Minus	700
A-102	Perryridge	400
A-305	Round hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

Depositor relation

Customer-name	Account-number
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Figure (11.5) . Three Tebles

The result will be : $\Pi_{\text{branch-name}}(\sigma_{\text{customer-city}}=\text{'Harrison'}(\text{customer} \bowtie \text{account} \bowtie \text{depositor}))$

Branch-name
Perryridge
Brighton

12 : Query processing

Query processing refers to the number of activities involved in extracting data from a database.
The steps involved in query processing are :

1- Parsing and translation
2- Optimization
3- Evaluation

12.1- Parsing and translation

Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use , but it is not suitable for the internal representation of the query in the system, thus the query must translated into its internal form, and this is the work of the ***PARSER***.
The ***PARSER*** check for :

- The syntax of the query
- The relation names appearing in the query are exist in the database
- Generate the relational-algebra expression.

12.2- Optimization

The **query optimizer** is the component of a database management system that attempts to determine the most efficient way to execute a query.
The optimizer considers the possible query plans for a given input query, and attempts to determine which of those plans will be the most efficient. *[A **query plan** (or **query execution plan**) is an ordered set of steps used to access or modify information in a database.]*
Cost-based query optimizers assign an estimated "cost" to each possible query plan, and choose the plan with the smallest cost.
[Costs are used to estimate the runtime cost of evaluating the query, in terms of the number of I/O operations required, the CPU requirements(CPU time to execute a query), the cost of memory used for the query and the cost of communication (in distributed or client-server DB)]

12.2-1 : Equivalent expression

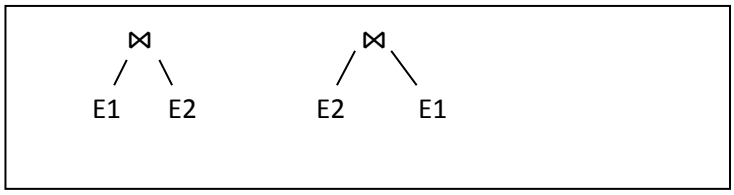
To find the least-costly query evaluation plan, the optimizer generates alternative plan (by generating alternative algebra expression) that produce the same result but with deferent costs.
The rules of equivalence are :

- Rule (1) : Selection operations are commutative

  $\sigma_a(\sigma_b(E)) = \sigma_b(\sigma_a(E))$

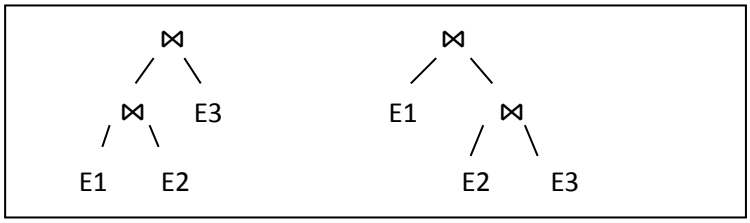
- Rule (2): Natural Join operations are commutative

  $E1 \bowtie E2 = E2 \bowtie E1$



- Rule(3): Natural join operations are associative

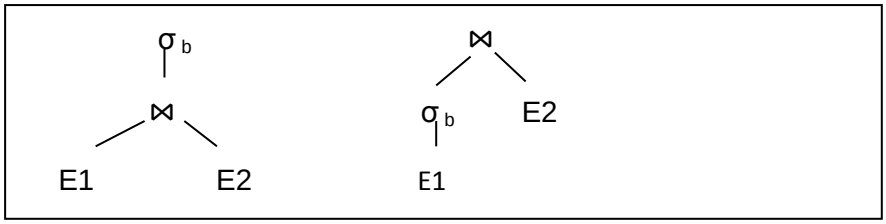
  $(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)$



- Rule(4) :The selection operation distributes over the join operation under the following two condition:

  a- It distribute when all the attributes in selection condition ($_b$) involves only the attributes of one of the expressions (say E1) being joined.

  $\sigma_b (E1 \bowtie E2) = (\sigma_b(E1)) \bowtie E2$



  b- It distributes when selection condition ($_{b1}$) involves only the attributes of E1 And ($_{b2}$) involves only the attributes of E2

  $\sigma_{b1 \wedge b2} (E1 \bowtie E2) = (\sigma_{b1} (E1)) \bowtie (\sigma_{b2} (E2))$

Example 12.1 : consider the relational algebra

$$\Pi_{customer\text{-}name}(\sigma_{branch\text{-}city}\text{='Brooklyn'}(branch \bowtie (account \bowtie depositor)))$$

This expression construct a large intermediate relation , $branch \bowtie account \bowtie depositor$. However we are interesting in only a few tuples of this relation (branch in Brooklyn) and one attribute of the relation (customer-name).

Since we are concerned with only tuples those in Brooklyn in the ***branch*** relation, we do not need to consider those tuples that do not have branch-city="Brooklyn" from the ***branch*** relation.

By reducing the number of tuples of the branch relation that we need to access, we reduce the size of the intermediate result. By using rule (4.a) our query is now represented by the relational expression: (figure 12.1)

$$\Pi_{\text{customer-name}}((\sigma_{\text{branch-city}}=\text{'Brooklyn'}(\text{branch})) \bowtie (\text{account} \bowtie \text{depositor}))$$

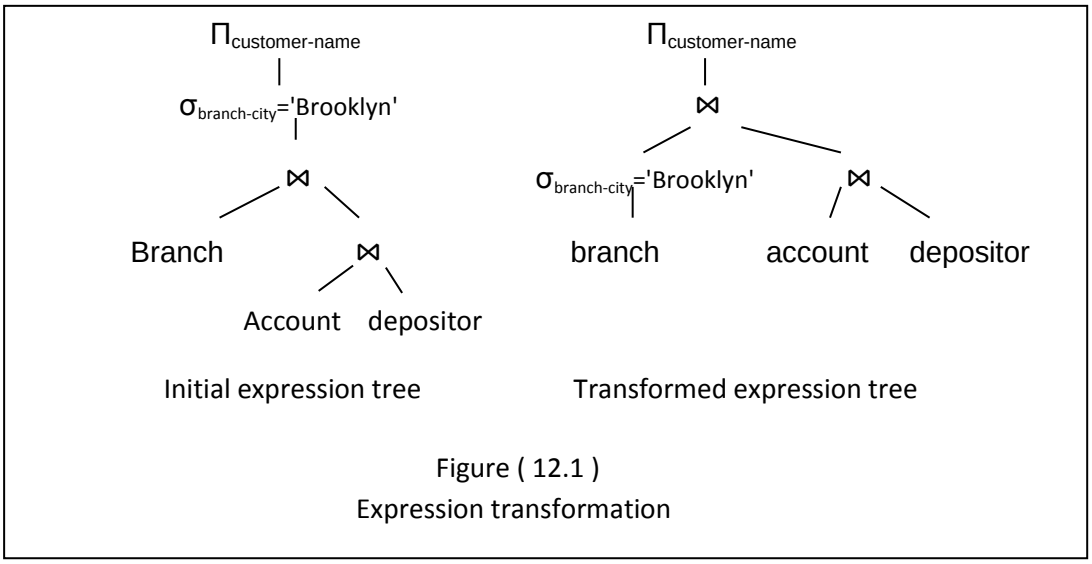


Figure (12.1)
Expression transformation

Example12. 2: We have three relations

Branch (branch-name,branch-city,assets)
Account (account-number,branch-name,balance)
Depositor (customer-name,account-number)

To find customer names in Brooklyn and have balance over 1000$. The relational algebra is:

$$\Pi_{\text{customer-name}}(\sigma_{\text{branch-city}}=\text{'Brooklyn'} \wedge \text{balance}>1000 (\text{branch} \bowtie (\text{account} \bowtie \text{depositor})))$$

We cannot apply rule (4) directly because the condition involves attributes of both the ***branch and account*** relations. We can apply rule (3) to transform the join (branch ⋈ (account ⋈ depositor))

Into (branch ⋈ account) ⋈ depositor) :

$$\Pi_{\text{customer-name}}(\sigma_{\text{branch-city='Brooklyn'} \wedge \text{balance}>1000} ((\text{branch} \bowtie \text{account}) \bowtie \text{depositor}))$$

Then using rule (4.a) to take ***depositor relation*** out from the condition:

$$\Pi_{\text{customer-name}}((\sigma_{\text{branch-city='Brooklyn'} \wedge \text{balance}>1000} (\text{branch} \bowtie \text{account})) \bowtie \text{depositor})$$

Then using rule (4.b) :

$$\Pi_{\text{customer-name}}((\sigma_{\text{branch-city='Brooklyn'}} \text{ branch} \bowtie \sigma_{\text{balance}>1000} \text{ account}) \bowtie \text{depositor})$$

12.2-2 : Disk I/O cost

In the database, the cost to access data from disk is important, since disk accesses are slow compared to in memory operation. Disk access measured by taking into account:

* Number of disk seeks (average-seek-cost)
* Number of blocks transfers from disk (average-block-read-cost)

If the disk subsystem takes an average of

tT – seconds to transfer one block of data
tS – seconds for one seek (block access time)

then the operation of transfers N blocks and performs S seeks would take :

$$N * tT + S * tS \quad seconds$$

When calculating disk I/O cost, some system need one seek to transfer many block. For example if there are 10 blocks need to be transfer from disk to memory with one seek , then the time will be tS + 10 * tT.

12.2-3 :Projection Example

Projections produce a result tuple for every argument tuple. Change in the output size is the change in the length of tuples .

Let's take a relation 'R' : R(a, b, c), the number of tuples in this relation are (20,000 tuples).
Each Tuple (190 bytes size) : header = 24 bytes, a = 8 bytes, b = 8 bytes, c = 150 bytes.
Each Block (1024): header = 24 bytes
We can fit 5 tuples into 1 block

- 5 tuples * 190 bytes(size of the tuple) = 950 bytes can fit into 1 block
- For 20,000 tuples, we would require ***4,000*** blocks (20,000 / 5 tuples per block)

With a projection resulting in elimination of column c (150 bytes), we could estimate that each tuple would decrease to 40 bytes (190 – 150 bytes)
Now, the new estimate will be 25 tuples in 1 block. (25 tuples * 40 byte= 1000)

- 25 tuples * 40 bytes/tuple = 1000 bytes will be able to fit into 1 block
- With 20,000 tuples, the new estimate is 800 blocks (20,000 tuples / 25 tuples per block = ***800*** blocks)

Result is reduction by a factor of 5

12.3- Evaluation

Query evaluation is the process of executing the plan for that query and return the result to query.
The ***query-execution engine*** is the subsystem of the DBMS that execute the query plan.