INTRODUCTION TO DATABASE CONCEPT AND SYSTEMS

INTRODUCTION

The world is replete with information that we seek to capture and represent in digital devices for easy access, update and manipulation. This unit will introduce you to the general basic concepts of database management systems, without being specific to a particular commercially available database product.

The general characteristics of a database system is outlined here, subsequent units will give more insights into the detail features, however, you are required to know what a database system should be, and it used to be in the earlier computer systems.

What is a Database (Db)?

A Database is a collection of data in an organized manner, so that its content can easily be **accessed**, **managed** and **updated**. A database is an aggregation of data records or files such as sales transactions, products catalog and inventories and customer profiles. Database systems are designed to manage medium and large enterprises such as the National Population Commission, Banks, large Telecom companies, etc. A database system must also ensure the safety of the information stored, despite system crashes or attempts at unauthorized access.

Databases are widely used, here are some representative applications:

- ***Banking:*** For customer information, accounts, and loans, and banking transactions.
- ***Airlines:*** For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner—terminals situated around the world accessed the central database system through phone lines and other data networks.
- ***Universities:*** For student information, course registrations, and grades.
- ***Credit card transactions*****:** For purchases on credit cards and generation of monthly statements.
- ***Telecommunication*****:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
- ***Finance*****:** For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds.

- ***Sales*****:** For customer, product, and purchase information.
- ***Manufacturing*****:** For management of supply chain and for tracking production of items in factories, inventories of items in warehouses/stores, and orders for items.
- ***Human resources*****:** For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.

What is a FLAT FILE?

Before the advent of database system there used to be a **flat file system** usually provided by computer hardware manufacturer as part of the Operating System (OS) to store and manage files by tracking their names and location on the storage media. Because there was no data model, the file content and usage were defined by the application program that uses the file, as all files appeared the same. As data become larger and complex, the need to make the data independent of application programs arose and different data models came up. You will learn about data models shortly.

An example of a flat file system is a data file for a FORTRAN or a COBOL program. Another similar illustration of a flat file system is the Microsoft Excel worksheet where only one file or table represents all the records of information.

The **disadvantages** of a flat file system are as follows:

- ***Data redundancy and inconsistency:*** Since different programmers create the files and application programs over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, the address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. In addition, it may lead to data inconsistency; that is, the various copies of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.
- ***Difficulty in accessing data:*** Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because the designers of the original

system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of all customers. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same officer needs to trim that list to include only those customers who have an account balance of =N=10,000 or more. As expected, a program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory. The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

- ***Data isolation:*** Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.
- ***Integrity problems:*** The data values stored in the database must satisfy certain types of consistency constraints. For example, the balance of a bank account may never fall below a prescribed amount (say, =N=1000). Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.
- ***Atomicity problems:*** A computer system, like any other mechanical or electrical device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer =N=5000 from account A to account B. If a system failure occurs during the execution of the program, it is possible that the =N=5000 was removed from account A but was not credited to account B, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be atomic—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.
- ***Concurrent-access anomalies:*** For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data.

Consider bank account A, containing =N=5000. If two customers withdraw funds (say =N=500 and =N=1000 respectively) from account A at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value =N=5000, and write back =N=4500 and =N=4000, respectively. Depending on which one writes the value last, the account may contain either =N=4500 or =N=4000, rather than the correct value of =N=3500. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

- ***Security problems:*** Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult. These difficulties, among others, prompted the development of database systems.

Database Systems

A database system is a collection of interrelated files and a set of programs that allow users to access and modify these files. A database system centralizes data collection and management rather than having discreet flat files spread across, as these are difficult to control and manage.

A database-management system (DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the database, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient. Examples of commercial database management systems are Oracle, Microsoft SQL-Server and Sybase.

The following are the advantages of having a database system:

1. **Minimizes data redundancy:** If each dept maintains students record, it may lead to waste of storage and data redundancy, whereas if all information is kept centrally, by grouping, say all admin information are stored by admin dept, Finance dept stores all

information pertaining to payments, etc then all the information can be related to each other by the Administrator, thereby eliminating redundancy.

2. **Avoids data inconsistency:** If data were not stored centrally, as in a database, if a dept stores a student_dept as SICT_IMT, another, another department could store the same data about the same student as simply IMT, hence data inconsistency arises. However, if the same data is captured centrally, there can be no data inconsistency.
3. **Allows data to be shared:** Different applications can share the same data.
   a) **Standards can be enforced:** DBA can enforce policy standards for the stored data.
   b) **Security can be enforced:** DBA can define access rights and authorization checks for data stored in database
4. **Integrity can be maintained:** A database can check or validate data integrity to ensure that some fields conform to certain standard, e.g. all student must have a Matric number, hence Matric number field must not be blank.
5. **Data independence:** The structure of a database often needs to be modified, by adding new attributes or relationship in order to improve performance. While file-based applications are *data-dependent*, i.e. the way the data is organized on secondary storage (either indexed, Sequential or direct, etc) and data organization must be specified, at times in some applications, such as specifying records storage structure or access strategy, all these are *data dependency*. A good DBMS provides for data independence, such that lets you change the structure of the database without requiring changes to the programs or applications that access the database

Demerits of a Database System

Though the merits of a database system outweigh its demerits, the following are the costs of a database system:

1. **Larger file size:** To support its complex functions, a relational database management system, RDBMS is often a large application occupying a large amount of disk space and internal memory.
2. **Increased complexity:** A typical DBMS is a complex application to understand, especially all its features therefore an in-depth understanding of the features of a DBMS is crucial to designing a successful and functional database system.

3. **Greater impact of failure:** If for some reasons, the database backup and recovery procedure is breached, the failure of a database system could lead to a disaster for an organization.
4. **Rigorous recovery procedure:** The process of recovering a database in an event of failure could be rigorous, except some routines and strictly obeyed standard operating procedures (SOPs) are followed constantly with provision for redundancy database to start once the active one fails.

Data Abstraction

A major purpose of a database system is to provide users with an *abstract* view of the data. That is, the system hides certain details of how the data are stored and maintained.

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database-systems users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

- ***Physical level***: The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.
- ***Logical level***: The next-higher level of abstraction describes what data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.
- ***View level***: The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

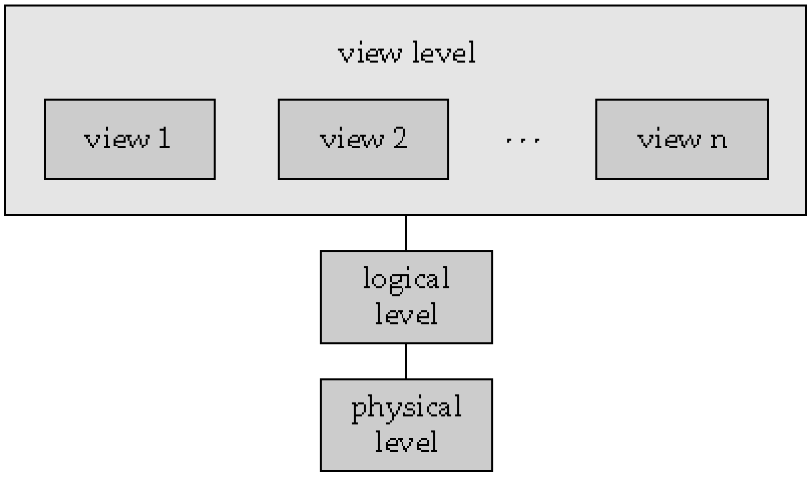


Figure 1.1 The three levels of data abstraction.

Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an instance of the database. The overall design of the database is called the database schema. Schemas are changed infrequently, if at all.

The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an instance of a database schema.

Database systems have several schemas, partitioned according to the levels of abstraction.

The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database. Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit physical data independence if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

We will study languages for describing schemas, after introducing the notion of data models in the next section.

Data Models

Underlying the structure of a database is the **data model** i.e. a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

To illustrate the concept of a data model, we outline two data models: the entity-relationship model (E-R model) and the relational model. Both provide a way to describe the design of a database at the logical level.

The Entity-Relationship Model

The entity-relationship (E-R) data model is based on a perception of a real world that consists of a collection of basic objects, called entities, and of relationships among these objects. An **entity** is a “thing” or “object” in the real world that is distinguishable from other objects. For example, each person is an entity, and bank accounts can be considered as entities.

Entities are described in a database by a set of **attributes**. For example, the attributes *account-number* and *balance* may describe one particular account in a bank, and they form attributes of the *account* entity set. Similarly, attributes *customer-name, customer-street address* and *customer-city* may describe a *customer* entity.

An extra attribute *customer-id* is used to uniquely identify customers (since it may be possible to have two customers with the same name, street address, and city).

A **relationship** is an association among several entities. For example, a *depositor* relationship associates a customer with each account that she has. The set of all entities of the same type and the set of all relationships of the same type are termed an **entity set** and **relationship set**, respectively.

The overall logical structure (schema) of a database can be expressed graphically by an *E-R diagram,* which is built up from the following components:

- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationships among entity sets
- **Lines,** which link attributes to entity sets and entity sets to relationships

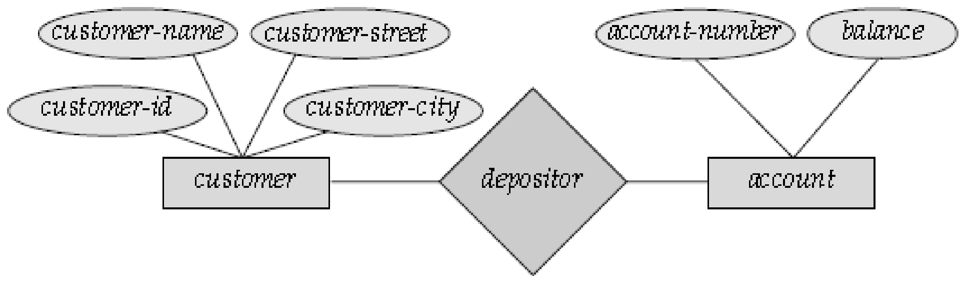


Figure 1.2 A sample E-R diagram.

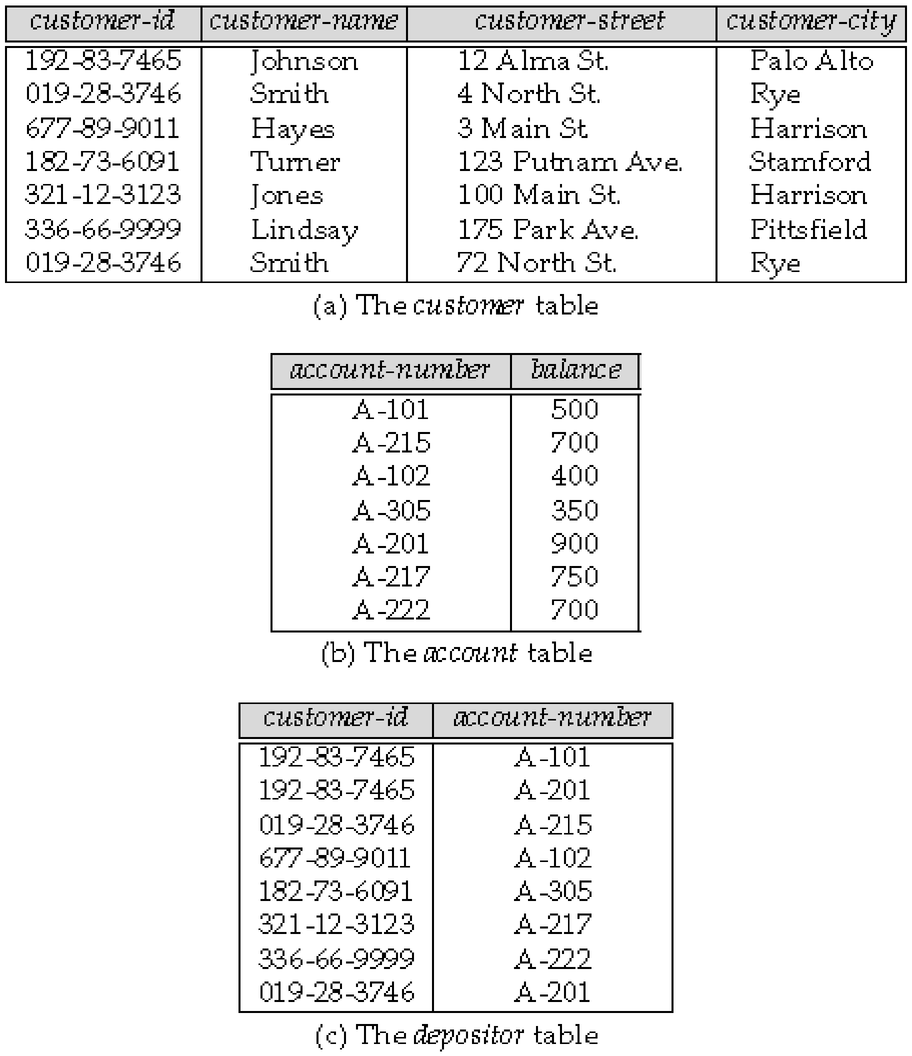
Relational Model

The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Figure 1.3 presents a sample relational database comprising three tables:

One shows details of bank customers, the second shows accounts, and the third shows which accounts belong to which customers. The first table, the customer table, shows, for example, that the customer identified by customer-id 192-83-7465 is named Johnson and lives at 12 Alma St. in Palo Alto. The second table, account, shows, for example, that account A-101 has a balance of $500, and A-201 has a balance of $900. The third table shows which accounts belong to which customers. For example, account number A-101 belongs to the customer whose customer-id is 192-83-7465, namely Johnson, and customers 192-83-7465 (Johnson) and 019-28-3746 (Smith) share account number A-201 (they may share a business venture).

The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type.

The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model. The relational model is at a lower level of abstraction than the E-R model. Database designs are often carried out in the E-R model, and then translated to the relational model. For example, it is easy to see that the tables customer and account correspond to the entity sets of the same name, while the table depositor corresponds to the relationship set depositor.



customer-id	*customer-name*	*customer-street*	*customer-city*
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

account-number	*balance*
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

customer-id	*account-number*
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Figure 1.3 A sample relational database.

Other Data Models

The **object-oriented data model** can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. The **object-relational data model** combines features of the object-oriented data model and relational data model.

Semi-structured data models permit the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast with the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **extensible markup language (XML)** is widely used to represent semi-structured data.

Historically, two other data models, **the network data model** and **the hierarchical data model**, preceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result they are little used now, except in old database code that is still in service in some places.

Database Languages

A database system provides a **data definition language** to specify the database schema and a **data manipulation language** to express database queries and updates. In practice, the data definition and data manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language. There are a number of database query languages in use, either commercially or experimentally. **Query by Example (QBE)** and **Datalog** are two good examples of other database languages. However, we will study the most widely used query language which is the **Structured Query Language (SQL).**

Data-Definition Language

We specify a database schema by a set of definitions expressed by a special language called a data-definition language (DDL). For instance, the following statement in the SQL language defines the *account* table:

```
create table account
(account-number char(10),
balance integer)
```

Execution of the above DDL statement creates the account table. In addition, it updates a special set of tables called the **data dictionary** or **data directory**.

A data dictionary contains **metadata**—that is, data about data. The schema of a table is an example of metadata. A database system consults the data dictionary before reading or modifying actual data. We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain consistency constraints. For example, suppose the balance on an account should not fall below =N=100. The DDL provides facilities to specify such constraints. The database systems check these constraints every time the database is updated.

Data-Manipulation Language

Data manipulation is

- The retrieval of information stored in the database
- The insertion of new information into the database
- The deletion of information from the database
- The modification of information stored in the database

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model.

A query is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. This query in the SQL language finds the name of the customer whose customer-id is 192-83-7465:

```
select customer.customer-name
from customer
where customer.customer-id = 192-83-7465
```

The query specifies that those rows from the table customer where the customer-id is 192-83-7465 must be retrieved, and the customer-name attribute of these rows must be displayed. If the query were run on the table in Figure 1.3, the name Johnson would be displayed. Note that queries may involve information from more than one table.

Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users as follows:

- **Naive users** are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer =N=5000 from account A to account B invokes a program called transfer. This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred. As another example, consider a user who wishes to find her account balance over the World Wide Web. Such a user may access a form, where she enters her account number. An application program at the Web server then retrieves the account balance, using the given account number, and passes this information back to the user. The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read reports generated from the database.

- **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program. There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language. These languages, sometimes called fourth-generation languages, often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth-generation language.

- **Sophisticated users** interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a query processor, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category. Online analytical processing (OLAP) tools simplify analysts' tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region). The tools also permit the analyst to select specific regions, look at data in more detail (for example, sales by city within a region) or look at the data in less detail (for example, aggregate products together by category). Another class of tools for analysts is data mining tools, which help them find certain kinds of patterns in data. You will study OLAP tools and data mining next session.

- **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, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

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:

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

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

3. **Schema and physical-organization modification:** The DBA carries 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.

4. **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.

5. **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.

Transaction Management

Often, several operations on the database form a single logical unit of work. For example, a funds transfer in which one account (say A) is debited and another account (say B) is credited. Clearly, it is essential that either both the credit and debit occur, or that neither occur. That is, the funds transfer must happen in its entirety or not at all. This all-or-none requirement is called **atomicity**. In addition, it is essential that the execution of the funds transfer preserve the consistency of the database. That is, the value of the sum A + B must be preserved. This correctness requirement is called **consistency**. Finally, after the successful execution of a funds transfer, the new values of accounts A and B must persist, despite the possibility of system failure. This persistence requirement is called **durability**.

A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. However, during the execution of a transaction, it may be necessary temporarily to allow inconsistency, since either the debit of A or the credit of B must be done before the other. This temporary inconsistency, although necessary, may lead to difficulty if a failure occurs.

It is the programmer's responsibility to define properly the various transactions, so that each preserves the consistency of the database. For example, the transaction to transfer funds from account A to account B could be defined to be composed of two separate programs: one that debits account A, and another that credits account B. The execution of these two programs one after the other will indeed preserve consistency. However, each program by itself does not transform the database from a consistent state to a new consistent state. Thus, those programs are not transactions.

Ensuring the *atomicity and durability* properties is the responsibility of the database system itself—specifically, of **the transaction-management component**. In the absence of failures, all transactions complete successfully, and atomicity is achieved easily. However, because of various types of failure, a transaction may not always complete its execution successfully. If we are to ensure the atomicity property, a failed transaction must have no effect on the state of the database. Thus, the database must be restored to the state in which it was before the transaction in question started executing. The database system must therefore perform failure recovery, that is, detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

Finally, when several transactions update the database concurrently, the consistency of data may no longer be preserved, even though each individual transaction is correct. It is the responsibility of the **concurrency-control manager** to control the interaction among the concurrent transactions, to ensure the consistency of the database.

Database System Structure

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the **storage manager** and the **query processor** components.

The storage manager is important because databases typically require a large amount of storage space. Corporate databases range in size from hundreds of gigabytes to, for the largest databases, terabytes of data. A gigabyte is 1000 megabytes (1 billion bytes), and a terabyte is 1 million megabytes (1 trillion bytes). Since the main memory of computers cannot store this much information, the information is stored on disks. Data are moved between disk storage and main memory as needed.

Since the movement of data to and from disk is slow relative to the speed of the central processing unit, it is imperative that the database system structure the data so as to minimize the need to move data between disk and main memory. The query processor is important because it helps the database system simplify and facilitate access to data. High-level views help to achieve this goal; with them, users of the system are not to be burdened unnecessarily with the physical details of the implementation of the system. However, quick processing of updates and queries is important. It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

Storage Manager

A storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

1. **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.

2. **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.

3. **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.

4. **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

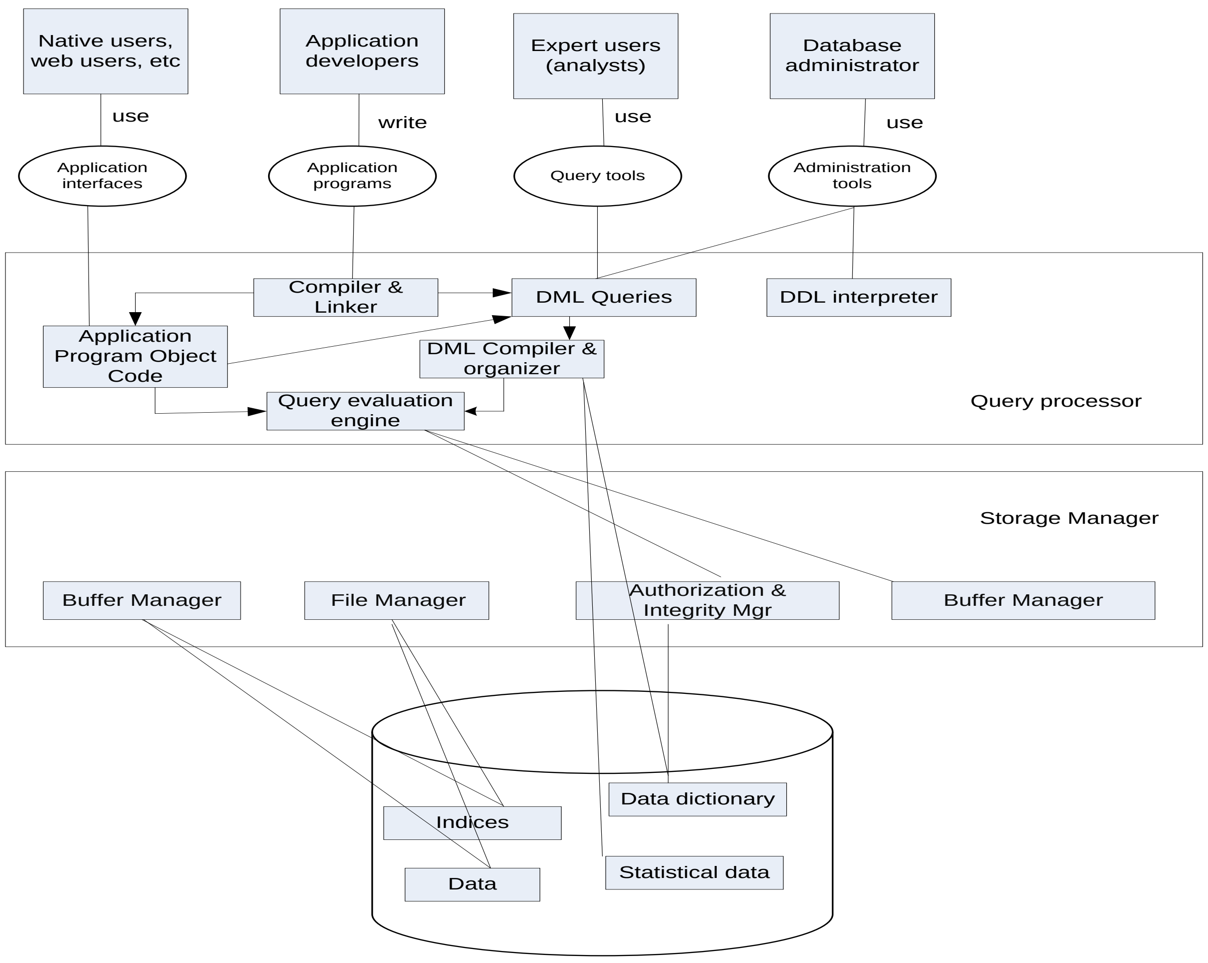


Figure 1.4: Database system structure

The storage manager implements several data structures as part of the physical system implementation as follows:

- Data files, which store the database itself.
- Data dictionary, which stores metadata about the structure of the database, in particular the schema of the database.
- Indices, which provide fast access to data items that hold particular values.

Application Architectures

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between client machines, on which remote database users work, and server machines, on which the database system runs.

Database applications are usually partitioned into two or three parts, as in Figure 1.5. In a two-tier architecture, the application is partitioned into a component that resides at the client machine, which invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

In contrast, in a three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface. The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the World WideWeb.

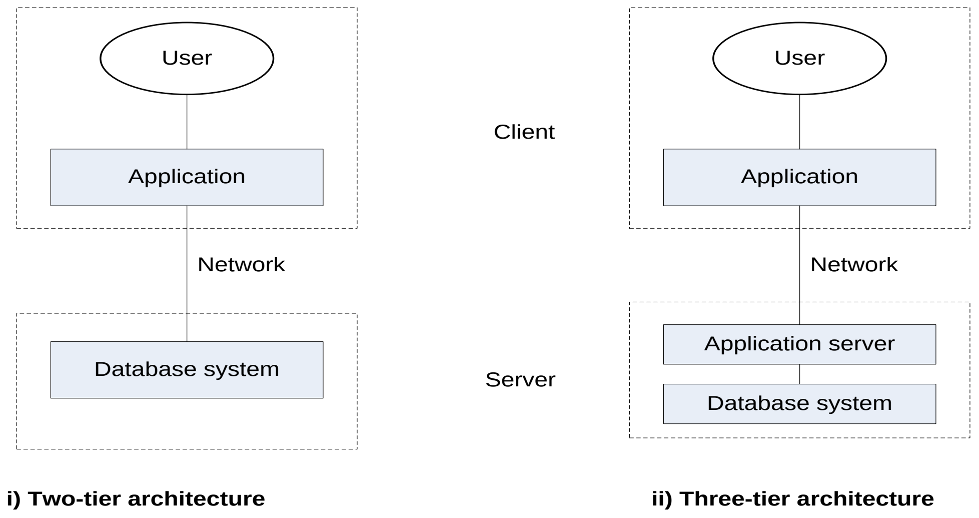


Figure 1.5: 2-Tier and 3-Tier Architectures

REFERENCES / FURTHER READINGS

1. Abraham Silberschatz, Henry F. Korth and S. Sudarshan S (2006). *Database System Concepts, 5th Edition*, McGraw-Hill International Companies Inc. 1221 Avenues of the Americas, New York, NY 10020.
2. Rajesh Narang (2009). *Database Management Systems*, PHI Learning Private Ltd. New Delhi 110001.
3. Narayan S. Umaneth & Richard W. Scamell (2007). *Data Modelling & Database Design*. Thomson Learning Inc., U.S.A.
4. Date C.J. (1990). *An Introduction to Database Systems, Volume I, 5th Edition.* (The Systems Programming Series). Addison-Wesley Publishing Company, Inc., U.S.A.
5. Colin Ritchie (2008). *Database Principles and Design, Third Edition*. Centage Learning EMEA High Holborn House, 50-51 Belford Row London WC1R 4LR.
6. Phillip J. Pratt & Joseph J. Adamski (2005). *Concepts of Database Management, Fifth Edition*. Thomson Learning Inc., U.S.A.