DHABALESWAR INSTITUTE OF POLYTECHNIC,ATHGARH,CUTTACK

# DATABASE MANAGEMENT SYSTEMS LECTURE NOTES

4TH SEMESTER(SUMMER)

DEPT. OF COMPUTER SCIENCE & ENGG.

NAME OF THE TEACHER : RASMAN KUMAR SANTI

# UNIT-1

## Introduction to Database Management System

As the name suggests, the database management system consists of two parts. They are:

1. Database and
2. Management System

### What is a Database?

To find out what database is, we have to start from data, which is the basic building block of any DBMS.

**Data**: Facts, figures, statistics etc. having no particular meaning (e.g. 1, ABC, 19 etc).
**Record**: Collection of related data items, e.g. in the above example the three data items had no meaning. But if we organize them in the following way, then they collectively represent meaningful information.

| Roll | Name | Age |
|---|---|---|
| 1 | ABC | 19 |

**Table** or **Relation**: Collection of related records.

| Roll | Name | Age |
|---|---|---|
| 1 | ABC | 19 |
| 2 | DEF | 22 |
| 3 | XYZ | 28 |

The columns of this relation are called **Fields, Attributes** or **Domains**. The rows are called **Tuples** or **Records.**
**Database**: Collection of related relations. Consider the following collection of tables:

**T1**

| Roll | Name | Age |
|---|---|---|
| 1 | ABC | 19 |
| 2 | DEF | 22 |
| 3 | XYZ | 28 |

**T2**

| Roll | Address |
|---|---|
| 1 | KOL |
| 2 | DEL |
| 3 | MUM |

**T3**

| Roll | Year |
|---|---|
| 1 | I |
| 2 | II |
| 3 | I |

**T4**

| Year | Hostel |
|---|---|
| I | H1 |
| II | H2 |

We now have a collection of 4 tables. They can be called a "related collection" because we can clearly find out that there are some common attributes existing in a selected pair of tables. Because of these common attributes we may combine the data of two or more tables together to find out the complete details of a student. Questions like "Which hostel does the youngest student live in?" can be answered now, although

***Age*** and ***Hostel*** attributes are in different tables.

A database in a DBMS could be viewed by lots of different people with different responsibilities.

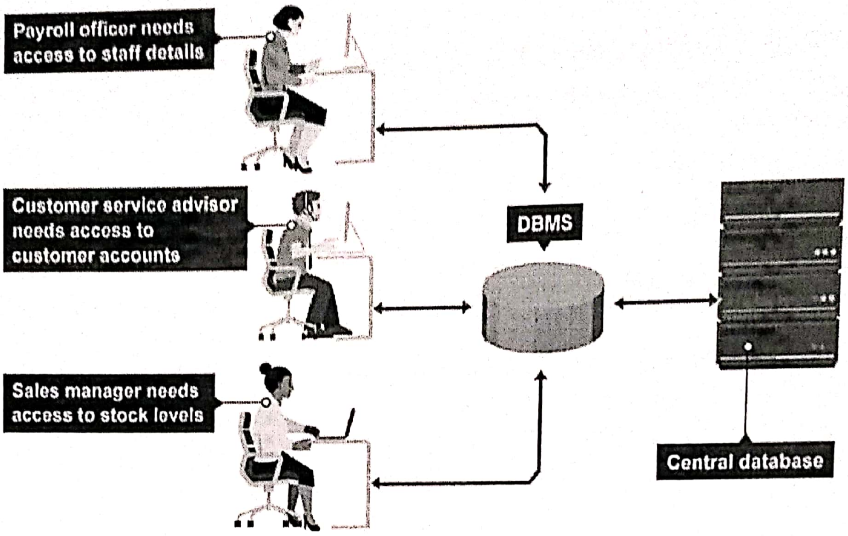

Payroll officer needs access to staff details
Customer service advisor needs access to customer accounts
Sales manager needs access to stock levels
DBMS
Central database

Figure 1.1: Empolyees are accessing Data through DBMS

For example, within a company there are different departments, as well as customers, who each need to see different kinds of data. Each employee in the company will have different levels of access to the database with their own customized **front-end** application.

In a database, data is organized strictly in row and column format. The rows are called **Tuple** or **Record.** The data items within one row may belong to different data types. On the other hand, the columns are often called **Domain** or **Attribute.** All the data items within a single attribute are of the same data type.

## What is Management System?

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database. 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*. By **data,** we mean known facts that can be recorded and that have implicit meaning.

The management system is important because without the existence of some kind of rules and regulations it is not possible to maintain the database. We have to select the particular attributes which should be included in a particular table; the common attributes to create relationship between two tables; if a new record has to be inserted or deleted then which tables should have to be handled etc. These issues must be resolved by having some kind of rules to follow in order to maintain the integrity of the database.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and technique form the focus of this book. This chapter briefly introduces the principles of database systems.

## Database Management System (DBMS) and Its Applications:

A Database management system is a computerized record-keeping system. It is a repository or a container for collection of computerized data files. The overall purpose of DBMS is to allow he users to define, store, retrieve and update the information contained in the database on demand. Information can be anything that is of significance to an individual or organization.

**Databases touch all aspects of our lives. Some of the major areas of application are as follows:**

1. Banking
2. Airlines
3. Universities
4. Manufacturing and selling
5. Human resources

***Enterprise Information***

- *Sales*: For customer, product, and purchase information.
- *Accounting*: For payments, receipts, account balances, assets and other accounting information.
- *Human resources*: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
- *Manufacturing*: For management of the supply chain and for tracking production of items in factories, inventories of items inwarehouses and stores, and orders for items.

*Online retailers*: For sales data noted above plus online order tracking,generation of recommendation lists, and
maintenance of online product evaluations.

***Banking and Finance***

- *Banking*: For customer information, accounts, loans, and banking transactions.
- *Credit card transactions*: For purchases on credit cards and generation of monthly statements.
- *Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.

- *Universities*: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
- *Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- *Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

## Purpose of Database Systems

Database systems arose in response to early methods of computerized management of commercial data. As an example of such methods, typical of the 1960s, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including programs to:

✓ Add new students, instructors, and courses

✓ Register students for courses and generate class rosters

✓ Assign grades to students, compute grade point averages (GPA), and generate transcripts

System programmers wrote these application programs to meet the needs of the university.
New application programs are added to the system as the need arises. For example, suppose that a university decides to create a new major (say, computer science).As a result, the university creates a new department and creates new permanent files (or adds information to existing files) to record information about all the instructors in the department, students in that major, course offerings, degree requirements, etc. The university may have to write new application programs to deal with rules specific to the new major. New application programs may also have to be written to handle new rules in the university. Thus, as time goes by, the system acquires more files and more application programs.

This typical **file-processing system** is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) were introduced, organizations usually stored information in such systems. Keeping organizational information in a file-processing system has a number of major disadvantages:

**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 structures and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the Mathematics department. 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 student address may be reflected in the Music department records but not elsewhere in the system.

**Difficulty in accessing data**. Suppose that one of the university clerks needs to find out the names of all students who live within a particular postal-code area. The clerk 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* students.

The university clerk has now two choices: either obtain the list of all students and extract the needed information manually or ask a 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 clerk needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist. Again, the clerk 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**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. 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 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 $500 from the account balance of department *A* to the account balance of department *B*. If a system failure occurs during the execution of the program, it is possible that the $500 was removed from the balance of department *A* but was not credited to the balance of department *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. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider department *A*, with an account balance of $10,000. If two department clerks debit the account balance (by say $500 and $100, respectively) of department *A* at almost exactly the same time, the result of the concurrent executions may leave the budget 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 $10,000, and write back $9500 and $9900, respectively. Depending on which one writes the value last, the account balance of department *A* may contain either $9500 or $9900, rather than the correct value of $9400. 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.

As another example, suppose a registration program maintains a count of students registered for a course, in order to enforce limits on the number of students registered. When a student registers, the program reads the current count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at (say) 39. The two program executions may both read the value 39, and both would then write back 40, leading to an incorrect increase of only 1, even though two students successfully registered for the course and the count should be 41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able to register, leading to a violation of the limit of 40 students.

**Security problems.** Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult.

These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems.

## Advantages of DBMS:

**Controlling of Redundancy:** Data redundancy refers to the duplication of data (i.e storing same data multiple times). In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided. It also eliminates the extra time for processing the large volume of data. It results in saving the storage space.

**Improved Data Sharing :** DBMS allows a user to share the data in any number of application programs.

**Data Integrity :** Integrity means that the data in the database is accurate. Centralized control of the data helps in permitting the administrator to define integrity constraints to the data in the database. For example: in customer database we can can enforce an integrity that it must accept the customer only from Noida and Meerut city.

**Security :** Having complete authority over the operational data, enables the DBA in ensuring that the only mean of access to the database is through proper channels. The DBA can define authorization checks to be carried out whenever access to sensitive data is attempted.

**Data Consistency :** By eliminating data redundancy, we greatly reduce the opportunities for inconsistency. For example: is a customer address is stored only once, we cannot have disagreement on the stored values. Also updating data values is greatly simplified when each value is stored in one place only. Finally, we avoid the wasted storage that results from redundant data storage.

**Efficient Data Access :** In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing

**Enforcements of Standards :** With the centralized of data, DBA can establish and enforce the data standards which may include the naming conventions, data quality standards etc.

**Data Independence :** Ina database system, the database management system provides the interface between the application programs and the data. When changes are made to the data representation, the meta data obtained by the DBMS is changed but the DBMS is continues to provide the data to application program in the previously used way. The DBMs handles the task of transformation of data wherever necessary.

**Reduced Application Development and Maintenance Time :** DBMS supports many important functions that are common to many applications, accessing data stored in the DBMS, which facilitates the quick development of application.

### Disadvantages of DBMS

1) It is bit complex. Since it supports multiple functionality to give the user the best, the underlying software has become complex. The designers and developers should have thorough knowledge about the software to get the most out of it.

2) Because of its complexity and functionality, it uses large amount of memory. It also needs large memory to run efficiently.

3) DBMS system works on the centralized system, i.e.; all the users from all over the world access this database. Hence any failure of the DBMS, will impact all the users.

4) DBMS is generalized software, i.e.; it is written work on the entire systems rather specific one. Hence some of the application will run slow.

## View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. 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.

## Data Abstraction

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-system users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

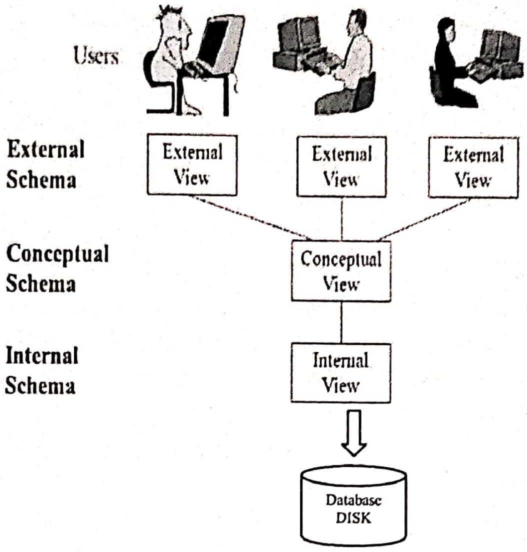


**Figure 1.2 : Levels of Abstraction in a DBMS**

• **Physical level (or Internal View / Schema)**: 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 (or Conceptual View / Schema)**: 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. This is referred to as **physical data independence.** Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

• **View level (or External View / Schema):** 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 1.2 shows the relationship among the three levels of abstraction.

An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programming languages support the notion of a structured type. For example, we may describe a record as follows:

```
type instructor = record
        ID : char (5);
        name : char (20);
        dept name : char (20);
        salary : numeric (8,2);
    end;
```

This code defines a new record type called *instructor* with four fields. Each field has a name and a type associated with it. A university organization may have several such record types, including

- *department*, with fields *dept_name*, *building*, and *budget*
- *course*, with fields *course_id*, *title*, *dept_name*, and *credits*
- *student*, with fields *ID*, *name*, *dept_name*, and *tot_cred*

At the physical level, an *instructor*, *department*, or *student* record can be described as a block of consecutive storage locations. The compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest-level storage details from database programmers. Database administrators, on the other hand, may be aware of certain details of the physical organization of the data.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction.

Finally, at the view level, computer users see a set of application programs that hide details of the data types. At the view level, several views of the database are defined, and a database user sees some or all of these views. In addition
to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database. For example, clerks in the university registrar office can see only that part of the database that has information about students; they cannot access information about salaries of instructors.

## 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**, which 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.

## Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical, and view levels.

The data models can be classified into four different categories:

- **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. Tables are also known as **relations**. 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.

**Entity-Relationship Model.** The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects.

An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design.

**Object-Based Data Model.** Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that 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 Model.** The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to 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, 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 used little 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.

## Data-Manipulation Language

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

- Retrieval of information stored in the database
- Insertion of new information into the database
- Deletion of information from the database
- Modification of information stored in the database

There are basically two types:

- **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.
- **Declarative DMLs** (also referred to as **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data. A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

**Data-Definition Language (DDL)**
We specify a database schema by a set of definitions expressed by a special language called a **data-definition language (DDL)**. The DDL is also used to specify additional properties of the 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 university requires that the account balance of a department must never be negative.
The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database.
However, arbitrary predicates may be costly to test. Thus, database systems implement integrity constraints that can be tested with minimal overhead.

- **Domain Constraints**. A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

- **Referential Integrity**. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the *dept name* value in a *course* record must appear in the *dept name* attribute of some record of the *department* relation. Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.

- **Assertions.** An assertion is any condition that the database must always satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated.

- **Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

The DDL, just like any other programming language, gets as input some instructions (statements) and generates some output. The output of the DDL is placed in the **data dictionary**,which contains **metadata**—that is, data about data. The data dictionary is considered to be a special type of table that can only be accessed and updated by the database system itself (not a regular user). The database system consults the data dictionary before reading or modifying actual data.

### Data Dictionary

We can define a data dictionary as a DBMS component that stores the definition of data characteristics and relationships. You may recall that such "data about data" were labeled metadata. The DBMS data dictionary provides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray of the company's entire data set, and is a crucial element in the data administration function.

The two main types of data dictionary exist, integrated and stand alone. An integrated data dictionary is included with the DBMS. For example, all relational DBMSs include a built in data dictionary or system catalog that is frequently accessed and updated by the RDBMS. Other DBMSs especially older types, do not have a built in data dictionary instead the DBA may use third party stand alone data dictionary systems.

Data dictionaries can also be classified as active or passive. An active data dictionary is automatically updated by the DBMS with every database access, thereby keeping its access information up-to-date. A passive data dictionary is not updated automatically and usually requires a batch process to be run. Data dictionary access information is normally used by the DBMS for query optimization purpose.

The data dictionary's main function is to store the description of all objects that interact with the database. Integrated data dictionaries tend to limit their metadata to the data managed by the DBMS. Stand alone data dictionary systems are more usually more flexible and allow the DBA to describe and manage all the organization's data, whether or not they are computerized. Whatever the data dictionary's format, its existence provides database designers and end users with a much improved ability to communicate. In addition, the data dictionary is the tool that helps the DBA to resolve data conflicts.

Although, there is no standard format for the information stored in the data dictionary several features are common. For example, the data dictionary typically stores descriptions of all:

- Data elements that are define in all tables of all databases. Specifically the data dictionary stores the name, datatypes, display formats, internal storage formats, and validation rules. The data dictionary tells where an element is used, by whom it is used and so on.
- Tables define in all databases. For example, the data dictionary is likely to store the name of the table creator, the date of creation access authorizations, the number of columns, and so on.
- Indexes define for each database tables.. For each index the DBMS stores at least the index name the attributes used, the location, specific index characteristics and the creation date.
- Define databases: who created each database, the date of creation where the database is located, who the DBA is and so on.
- End users and The Administrators of the data base
- Programs that access the database including screen formats, report formats application formats, SQL queries and so on.
- Access authorization for all users of all databases.
- Relationships among data elements which elements are involved: whether the relationship are mandatory or optional, the connectivity and cardinality and so on.

## Database Administrators and Database Users

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

### 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.

**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 $50 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.

**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.

# UNIT-2

## Relational Algebra and Calculus

### PRELIMINARIES

In defining relational algebra and calculus, the alternative of referring to fields by position is more convenient than referring to fields by name: Queries often involve the computation of intermediate results, which are themselves relation instances, and if we use field names to refer to fields, the definition of query language constructs must specify the names of fields for all intermediate relation instances. This can be tedious and is really a secondary issue because we can refer to fields by position anyway. On the other hand, field names make queries more readable.

Due to these considerations, we use the positional notation to formally define relational algebra and calculus. We also introduce simple conventions that allow intermediate relations to 'inherit' field names, for convenience.

We present a number of sample queries using the following schema:

Sailors (*sid:* integer, *sname:* string, *rating:* integer, *age:* real)
Boats (*bid:* integer, *bname:* string, *color:* string)
Reserves (*sid:* integer, *bid:* integer, *day:* date)

The key fields are underlined, and the domain of each field is listed after the field name. Thus *sid* is the key for Sailors, *bid* is the key for Boats, and all three fields together form the key for Reserves. Fields in an instance of one of these relations will be referred to by name, or positionally, using the order in which they are listed above.

In several examples illustrating the relational algebra operators, we will use the in-stances $S1$ and $S2$ (of Sailors) and $R1$ (of Reserves) shown in Figures 4.1, 4.2, and 4.3, respectively,

## RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the relational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as the result. This property makes it easy to compose operators to form a complex query—a relational algebra expression is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions. We describe the basic operators of the algebra (selection, projection, union, cross-product, and difference), as well as some additional operators that can be defined in terms of the basic operators but arise frequently enough to warrant special attention, in the following sections.Each relational query describes a step-by-step procedure for computing the desired answer, based on the order in which operators are applied in the query. The procedural nature of the algebra allows us to think of an algebra expression as a recipe, or a plan, for evaluating a query, and relational systems in fact use algebra expressions to represent query evaluation plans.

### Selection and Projection

Relational algebra includes operators to *select* rows from a relation ($\sigma$) and to *project* columns ($\pi$). These operations allow us to manipulate data in a single relation. Consider the instance of the Sailors relation shown in Figure 4.2, denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the $\sigma$ operator. The expression,

$$\sigma_{rating>8}(S2)$$

evaluates to the relation shown in Figure 4.4. The subscript *rating>8* specifies the selection criterion to be applied while retrieving tuples.

| *sid* | *sname* | *rating* | *age* |
|---|---|---|---|
| *28* | *Yuppy* | *9* | *35.0* |
| *58* | *Rusty* | *10* | *35.0* |

Figure 4.4 $\sigma_{rating>8}(S2)$

| *sname* | *rating* |
|---|---|
| *yuppy* | *9* |
| *Lubber* | *8* |
| *Guppy* | *5* |
| *Rusty* | *10* |

Figure 4.5 $\pi_{sname,rating}(S2)$

The selection operator $\sigma$ specifies the tuples to retain through a *selection condition.* In general, the selection condition is a boolean combination (i.e., an expression using the logical connectives $\wedge$ and $\vee$) of *terms* that have the form *attribute* op *constant* or *attribute1* op *attribute2*, where op is one of the comparison operators <, <=, =, =, >=, or >. The reference to an attribute can be by position (of the form *.i* or *i*) or by name (of the form *.name* or *name*). The schema of the result of a selection is the schema of the input relation instance

The projection operator $\pi$ allows us to extract columns from a relation; for example, we can find out all sailor names and ratings by using $\pi$. The expression $\pi_{sname,rating}(S2)$

Suppose that we wanted to find out only the ages of sailors. The expression

$$\pi_{age}(S2)$$

a single tuple with *age=35.0* appears in the result of the projection. This follows from the definition of a relation as a *set* of tuples. In practice, real systems often omit the expensive step of eliminating *duplicate tuples*, leading to relations that are multisets. However, our discussion of relational algebra and calculus assumes that duplicate elimination is always done so that relations are always sets of tuples.

We can compute the names and ratings of highly rated sailors by combining two of the preceding queries. The expression

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

| *age* |
|---|
| 35.0 |
| 55.5 |

Figure 4.6 $\pi_{age}(S2)$

| *sname* | *rating* |
|---|---|
| yuppy | 9 |
| Rusty | 10 |

Figure 4.7 $\pi_{sname,rating}(\sigma_{rating>8}(S2))$

**Set Operations**

The following standard operations on sets are also available in relational algebra: *union* ($\cup$), *intersection* ($\cap$), *set-difference* ($-$), and *cross-product* ($\times$).

- Union: $R \cup S$ returns a relation instance containing all tuples that occur in *either* relation instance $R$ or relation instance $S$ (or both). $R$ and $S$ must be *union-compatible*, and the schema of the result is defined to be identical to the schema of $R$.

- Intersection: $R \cap S$ returns a relation instance containing all tuples that occur in *both* $R$ and $S$. The relations $R$ and $S$ must be union-compatible, and the schema of the result is defined to be identical to the schema of $R$.

- Set-difference: $R - S$ returns a relation instance containing all tuples that occur in $R$ but not in $S$. The relations $R$ and $S$ must be union-compatible, and the schema of the result is defined to be identical to the schema of $R$.

- Cross-product: $R \times S$ returns a relation instance whose schema contains all the fields of $R$ (in the same order as they appear in $R$) followed by all the fields of $S$ (in the same order as they appear in $S$). The result of $R \times S$ contains one tuple $\langle r, s \rangle$ (the concatenation of tuples $r$ and $s$) for each pair of tuples $r \in R$, $s \in S$.

The cross-product opertion is sometimes called Cartesian product.

We now illustrate these definitions through several examples. The union of $S1$ and $S2$ is shown in Figure 4.8. Fields are listed in order; field names are also inherited from $S1$. $S2$ has the same field names, of course, since it is also an instance of Sailors.In general, fields of $S2$ may have different names; recall that we require only domains to match. Note that the result is a *set* of tuples. Tuples that appear in both $S1$ and $S2$ appear only once in $S1 \cup S2$. Also, $S1 \cup R1$ is not a valid operation because the two relations are not union-compatible. The intersection of $S1$ and $S2$ is shown in Figure 4.9, and the set-difference $S1 - S2$ is shown in Figure 4.10.

| *sid* | *sname* | *rating* | *age* |
|---|---|---|---|
| *22* | *Dustin* | *7* | *45.0* |
| *31* | *Lubber* | *8* | *55.5* |
| *58* | *Rusty* | *10* | *35.0* |
| *28* | *Yuppy* | *9* | *35.0* |
| *44* | *Guppy* | *5* | *35.0* |

Figure 4.8 $S1 \cup S2$

| *sid* | *sname* | *rating* | *age* |
|---|---|---|---|
| *31* | *Lubbe* | *8* | *55.5* |
| *58* | *Rusty* | *10* | *35.0* |

Figure 4.9 $S1 \cap S2$

| *sid* | *sname* | *rating* | *age* |
|---|---|---|---|
| 22 | Dustin | 7 | 45.0 |

Figure 4.10 $S1 - S2$

The result of the cross-product $S1 \times R1$ is shown in Figure 4.11 The fields in $S1 \times R1$ have the same domains as the corresponding fields in $R1$ and $S1$. In Figure 4.11 *sid* is listed in parentheses to emphasize that it is not an inherited field name; only the corresponding domain is inherited.

| *(sid)* | *sname* | *rating* | *age* | *(sid)* | *bid* | *day* |
|---|---|---|---|---|---|---|
| 22 | *Dustin* | *7* | *45.0* | 22 | *101* | *10/10/96* |
| 22 | *Dustin* | *7* | *45.0* | 58 | *103* | *11/12/96* |
| 31 | *Lubber* | *8* | *55.5* | 22 | *101* | *10/10/96* |
| 31 | *Lubber* | *8* | *55.5* | 58 | *103* | *11/12/96* |
| 58 | *Rusty* | *10* | *35.0* | 22 | *101* | *10/10/96* |
| 58 | *Rusty* | *10* | *35.0* | 58 | *103* | *11/12/96* |

Figure 4.11 $S1 \times R1$

## Renaming

We introduce a renaming operator $\rho$ for this purpose. The expression $\rho(R(F), E)$ takes an arbitrary relational algebra expression $E$ and returns an instance of a (new) relation called $R$. $R$ contains the same tuples as the result of $E$, and has the same schema as $E$, but some fields are renamed. The field names in relation $R$ are the same as in $E$, except for fields renamed in the *renaming list* $F$.

For example, the expression $\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$ returns a relation that contains the tuples shown in Figure 4.11 and has the following schema: C(*sid1:* integer, *sname:* string, *rating:* integer, *age:* real, *sid2:* integer, *bid:* integer,*day:* dates).

It is customary to include some additional operators in the algebra, but they can all be defined in terms of the operators that we have defined thus far. (In fact, the renaming operator is only needed for syntactic convenience, and even the $\cap$ operator is redundant; $R \cap S$ can be defined as $R - (R - S)$.) We will consider these additional operators,and their definition in terms of the basic operators, in the next two subsections.

# SCHEMA REFINEMENT

Ve now present an overview of the problems that schema refinement is intended to iddress and a refinement approach based on decompositions. Redundant storage of nformation is the root cause of these problems. Although decomposition can eliminate edundancy, it can lead to problems of its own and should be used with caution.

## 5.1.1 Problems Caused by Redundancy

toring the same information redundantly, that is, in more than one place within a atabase, can lead to several problems:

**Redundant storage:** Some information is stored repeatedly.

**Update anomalies:** If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.

**Insertion anomalies:** It may not be possible to store some information unless some other information is stored as well.

**Deletion anomalies:** It may not be possible to delete some information without losing some other information as well.

nsider a relation obtained by translating a variant of the Hourly_Emps entity set m Chapter 2:

Hourly_Emps(<u>*ssn*</u>, *name*, *lot*, *rating*, *hourly_wages*, *hours_worked*)

his chapter we will omit attribute type information for brevity, since our focus is on grouping of attributes into relations. We will often abbreviate an attribute name a single letter and refer to a relation schema by a string of letters, one per ibute. For example, we will refer to the Hourly Emps schema as *SNLRWH* (*W* notes the *hourly wages* attribute).

key for Hourly_Emps is *ssn*. In addition, suppose that the *hourly_wages* attribute etermined by the *rating* attribute. That is, for a given *rating* value, there is only permissible *hourly wages* value. This IC is an example of a *functional endency*. It leads to possible redundancy in the relation Hourly Emps, as trated in Figure 15.1.

e same value appears in the *rating* column of two tuples, the IC tells us that the e value must appear in the *hourly_wages* column as well. This redundancy has eral negative consequences:

| *ssn* | *name* | *lot* | *rating* | *hourly wages* | *hours worked* |
|---|---|---|---|---|---|
| 123-22-3666 | Attishoo | 48 | 8 | 10 | 40 |
| 231-31-5368 | Smiley | 22 | 8 | 10 | 30 |
| 131-24-3650 | Smethurst | 35 | 5 | 7 | 30 |
| 434-26-3751 | Guldu | 35 | 5 | 7 | 32 |
| 612-67-4134 | Madayan | 35 | 8 | 10 | 40 |

An Instance of the Hourly Emps Relation

- Some information is stored multiple times. For example, the rating value 8 corresponds to the hourly wage 10, and this association is repeated three times. In addition to wasting space by storing the same information many times, redundancy leads to potential inconsistency. For example, the *hourly wages* in the first tuple could be updated without making a similar change in the second tuple, which is an example of an *update anomaly.* Also, we cannot insert a tuple for an employee unless we know the hourly wage for the employee's rating value, which is an example of an *insertion anomaly.*

- If we delete all tuples with a given rating value (e.g., we delete the tuples for Smethurst and Guldu) we lose the association between that *rating* value and its *hourly_wage* value (a *deletion anomaly*).

Let us consider whether the use of *null* values can address some of these problems. Clearly, *null* values cannot help eliminate redundant storage or update anomalies. It appears that they can address insertion and deletion anomalies.

Ideally, we want schemas that do not permit redundancy, but at the very least we want to be able to identify schemas that do allow redundancy. Even if we choose to accept a schema with some of these drawbacks, perhaps owing to performance considerations, we want to make an informed decision.

### 5.1.2 Use of Decompositions

ntuitively, redundancy arises when a relational schema forces an association between ttributes that is not natural. Functional dependencies (and, for that matter, other Cs) can be used to identify such situations and to suggest refinements to the schema. he essential idea is that many problems arising from redundancy can be addressed by placing a relation with a collection of `smaller' relations. Each of the smaller relations ntains a (strict) subset of the attributes of the original relation. We refer to this process s *decomposition* of the larger relation into the smaller relations.

e can deal with the redundancy in Hourly Emps by decomposing it into two relations:

Hourly_Emps2(*ssn*, *name*, *lot*, *rating*, *hours_worked*)
Wages(*rating*, *hourly_wages*)

ne instances of these relations corresponding to the instance of_Hourly Emps ation in Figure 15.1 is shown in Figure 15.2.

| *ssn* | *name* | *lot* | *rating* | *hours worked* |
|---|---|---|---|---|
| 123-22-3666 | Attishoo | 48 | 8 | 40 |
| 231-31-5368 | Smiley | 22 | 8 | 30 |
| 131-24-3650 | Smethurst | 35 | 5 | 30 |
| 434-26-3751 | Guldu | 35 | 5 | 32 |
| 612-67-4134 | Madayan | 35 | 8 | 40 |

| *rating* | *hourly wages* |
|---|---|
| 8 | 10 |
| 5 | 7 |

Instances of Hourly Emps2 and Wages

e that we can easily record the hourly wage for any rating simply by adding a e to Wages, even if no employee with that rating appears in the current instance lourly_Emps. Changing the wage associated with a rating involves updating a le Wages tuple. This is more efficient than updating several tuples (as in the nal design), and it also eliminates the potential for inconsistency. Notice that the tion and deletion anomalies have also been eliminated.

### 15.1.3 Problems Related to Decomposition

Unless we are careful, decomposing a relation schema can create more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

To help with the first question, several *normal forms* have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition (i.e., a particular collection of smaller relations to replace the given relation).

With respect to the second question, two properties of decompositions are of particular interest. The *lossless-join* property enables us to recover any instance of the decom-posed relation from corresponding instances of the smaller relations. The *dependency-preservation* property enables us to enforce any constraint on the original relation by simply enforcing some constraints on each of the smaller relations. That is, we need not perform joins of the smaller relations to check whether a constraint on the original relation is violated.

## FUNCTIONAL DEPENDENCIES

A functional dependency (FD) is a kind of IC that generalizes the concept of a *key*. Let *R* be a relation schema and let *X* and *Y* be nonempty sets of attributes in *R*. We say that an instance *r* of *R* satisfies the FD *X ! Y* [1] if the following holds for every pair of tuples $t_1$ and $t_2$ in *r*:

If *t1:X* = *t2:X*, then *t1:Y* = *t2:Y* .

We use the notation *t1:X* to refer to the projection of tuple $t_1$ onto the attributes in *X*, in a natural extension of our TRC notation (see Chapter 4) *t:a* for referring to attribute *a* of tuple *t*. An FD *X ! Y* essentially says that if two tuples agree on the values in attributes *X*, they must also agree on the values in attributes *Y*.

Figure 15.3 illustrates the meaning of the FD *AB ! C* by showing an instance that satisfies this dependency. The first two tuples show that an FD is not the same as a key constraint: Although the FD is not violated, *AB* is clearly not a key for the relation. The third and fourth tuples illustrate that if two tuples differ in either the *A* ield or the *B* field, they can differ in the *C* field without violating the FD. On the other iand, if we add a tuple *ha*1; *b*1; *c*2; *d*1*i* to the instance shown in this figure, the esulting instance would violate the FD; to see this violation, compare the first tuple n the figure with the new tuple.

| *A* | *B* | *C* | *D* |
|---|---|---|---|
| a1 | b1 | c1 | d1 |
| a1 | b1 | c1 | d2 |
| a1 | b2 | c2 | d1 |
| a2 | b1 | c3 | d1 |

An Instance that Satisfies *AB ! C*

ecall that a *legal* instance of a relation must satisfy all specified ICs, including all pecified FDs. As noted in Section 3.2, ICs must be identified and specified based n the semantics of the real-world enterprise being modeled. By looking at an stance of a relation, we might be able to tell that a certain FD does *not* hold. owever, we can never deduce that an FD *does* hold by looking at one or more stances of the relation because an FD, like other ICs, is a statement about *all* ssible legal instances of the relation.

primary key constraint is a special case of an FD. The attributes in the key play e role of *X*, and the set of all attributes in the relation plays the role of *Y*. Note, wever, that the definition of an FD does not require that the set *X* be minimal; the ditional minimality condition must be met for *X* to be a key. If *X ! Y* holds, where is the set of all attributes, and there is some subset *V* of *X* such that *V ! Y* holds, en *X* is a *super key*; if *V* is a strict subset of *X*, then *X* is not a key.

the rest of this chapter, we will see several examples of FDs that are not key nstraints.

## EASONING ABOUT FUNCTIONAL DEPENDENCIES

e discussion up to this point has highlighted the need for techniques that allow us carefully examine and further re ne relations obtained through ER design (or, for t matter, through other approaches to conceptual design). Before proceeding h the main task at hand, which is the discussion of such schema refinement hniques, we digress to examine FDs in more detail because they play such a tral role in schema analysis and refinement.

en a set of FDs over a relation schema *R*, there are typically several additional s that hold over *R* whenever all of the given FDs hold. As an example, consider:

Workers(<u>ssn</u>, *name*, *lot*, *did*, *since*)

We know that *ssn ! did* holds, since *ssn* is the key, and FD *did ! lot* is given to hold. Therefore, in any legal instance of Workers, if two tuples have the same *ssn* value, they must have the same *did* value (from the first FD), and because they have the same *did* value, they must also have the same *lot* value (from the second FD). Thus, the FD *ssn ! lot* also holds on Workers.

We say that an FD *f* is implied by a given set *F* of FDs if *f* holds on every relation instance that satisfies all dependencies in *F*, that is, *f* holds whenever all FDs in *F* hold. Note that it is not sufficient for *f* to hold on some instance that satisfies all dependencies in *F*; rather, *f* must hold on *every* instance that satisfies all dependencies in *F*.

### 15.4.1 Closure of a Set of FDs

The set of all FDs implied by a given set *F* of FDs is called the closure of F and is denoted as $F^+$. An important question is how we can infer, or compute, the closure of a given set *F* of FDs. The answer is simple and elegant. The following three rules, called Armstrong's Axioms, can be applied repeatedly to infer all FDs implied by a set *F* of FDs. We use *X*, *Y*, and *Z* to denote *sets* of attributes over a relation schema *R*:

- Reflexivity: If *X* *Y*, then *X ! Y*.
- Augmentation: If *X ! Y*, then *XZ ! YZ* for any Z.
- Transitivity: If *X ! Y* and *Y ! Z*, then *X ! Z*.

Armstrong's Axioms are sound in that they generate only FDs in $F^+$ when applied to a set *F* of FDs. They are complete in that repeated application of these rules will generate all FDs in the closure $F^+$. (We will not prove these claims.) It is convenient to use some additional rules while reasoning about $F^+$:

- Union: If *X ! Y* and *X ! Z*, then *X ! YZ*.
- Decomposition: If *X ! YZ*, then *X ! Y* and *X ! Z*.

These additional rules are not essential; their soundness can be proved using Armstrong's Axioms.

use a more elaborate version of the Contracts relation:

Contracts (<u>*contractid*</u>, *supplierid*, *projectid*, *deptid*, *partid*, *qty*, *value*)

We denote the schema for Contracts as *CSJDPQV*. The meaning of a tuple in this relation is that the contract with *contractid* C is an agreement that supplier S (*supplierid*) will supply Q items of part P (*partid*) to project J (*projectid*) associated with lepartment *D* (*deptid*); the value *V* of this contract is equal to *value*.

he following ICs are known to hold:

1. The contract id *C* is a key: *C ! CSJDPQV*.
2. A project purchases a given part using a single contract: *JP ! C*.
3. A department purchases at most one part from a supplier: *SD ! P*.

everal additional FDs hold in the closure of the set of given FDs:

om *JP ! C*, *C ! CSJDPQV* and transitivity, we infer *JP ! CSJDPQV*.

om *SD ! P* and augmentation, we infer *SDJ ! JP*.

om *SDJ ! JP*, *JP ! CSJDPQV* and transitivity, we infer *SDJ ! CSJDPQV*. (Inci-
ntally, while it may appear tempting to do so, we *cannot* conclude *SD ! CSDPQV*,
nceling *J* on both sides. FD inference is not like arithmetic multiplication!)

e can infer several additional FDs that are in the closure by using augmentation or
composition. For example, from *C ! CSJDPQV*, using decomposition we can infer:

*C ! C*, *C ! S*, *C ! J*, *C ! D*, etc.

ally, we have a number of trivial FDs from the reflexivity rule.

## ORMAL FORMS

en a relation schema, we need to decide whether it is a good design or whether we
d to decompose it into smaller relations. Such a decision must be guided by an
lerstanding of what problems, if any, arise from the current schema. To provide such
dance, several normal forms have been proposed. If a relation schema is in one of
se normal forms, we know that certain kinds of problems cannot arise.

normal forms based on FDs are *first normal form (1NF)*, *second normal form*
*F)*, *third normal form (3NF)*, and *Boyce-Codd normal form (BCNF)*. These forms
e increasingly restrictive requirements: Every relation in BCNF is also in 3NF,
ry relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF. A relation is
rst normal form if every field contains only atomic values, that is, not lists or sets.
requirement is implicit in our definition of the relational model. Although some of
newer database systems are relaxing this requirement, in this chapter we will
ume that it always holds. 2NF is mainly of historical interest. 3NF and BCNF are

# INTENTION LOCK MODES

**(DBMS | Unit – IV (Part B) | Prepared by M V Kamal, Associate Professor)**

A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity).

Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, *Q*—must traverse a path in the tree from the root to *Q*. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is in Figure

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | true | true | true | true | false |
| IX | true | true | false | false | false |
| S | true | false | true | false | false |
| SIX | true | false | false | false | false |
| X | false | false | false | false | false |

**Figure: Compatibility matrix**

The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction *Ti* that attempts to lock a node *Q* must follow these rules:

1. Transaction *Ti* must observe the lock-compatibility function of Figure above.
2. Transaction *Ti* must lock the root of the tree first, and can lock it in anymode.
3. Transaction *Ti* can lock a node *Q* in S or IS mode only if *Ti* currently has the parent of *Q* locked in either IX or IS mode.
4. Transaction *Ti* can lock a node *Q* in X, SIX, or IX mode only if *Ti* currently has the parent of *Q* locked in either IX or SIX mode.
5. Transaction *Ti* can lock a node only if *Ti* has not previously unlocked any node (that is, *Ti* is two phase).
6. Transaction *Ti* can unlock a node *Q* only if *Ti* currently has none of the children of *Q* locked.

**Locking: Top-Down and Bottom-up**

Observe that the multiple-granularity protocol requires that locks be acquired in ***top-down*** (root-to-leaf) order, whereas locks must be released in ***bottom-up*** (leaf-to-root) order.

As an illustration of the protocol, consider the tree of Figure (above) and these transactions:

• Suppose that transaction *T*21 reads record *ra*2 in file *Fa* . Then, *T*21 needs to lock the database, area *A*1, and *Fa* in IS mode (and in that order), and finally to lock *ra*2 in S mode.

• Suppose that transaction *T*22 modifies record *ra*9 in file *Fa* . Then, *T*22 needs to lock the database, area *A*1, and file *Fa* (and in that order) in IX mode, and finally to lock *ra*9 in X mode.

• Suppose that transaction *T*23 reads all the records in file *Fa* . Then, *T*23 needs to lock the database and area *A*1 (and in that order) in IS mode, and finally to lock *Fa* in S mode.

• Suppose that transaction *T*24 reads the entire database. It can do so after locking the database in S mode.

# Precedence graph

*Reference 'encyclopedia'*

A **precedence graph**, also named **conflict graph** and **serializability graph**, is used in the context of **concurrency control in databases**.

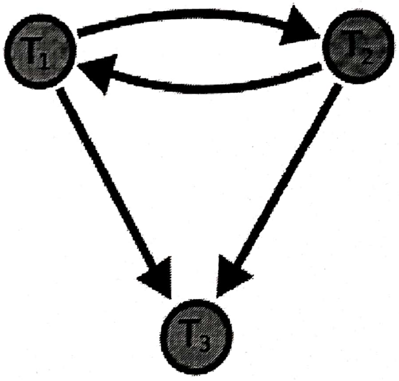
The precedence graph for a schedule S contains:

- A node for each committed transaction in S
- An arc from $T_i$ to $T_j$ if an action of $T_i$ precedes and conflicts with one of $T_j$'s actions.

## Precedence graph example

Example 1:

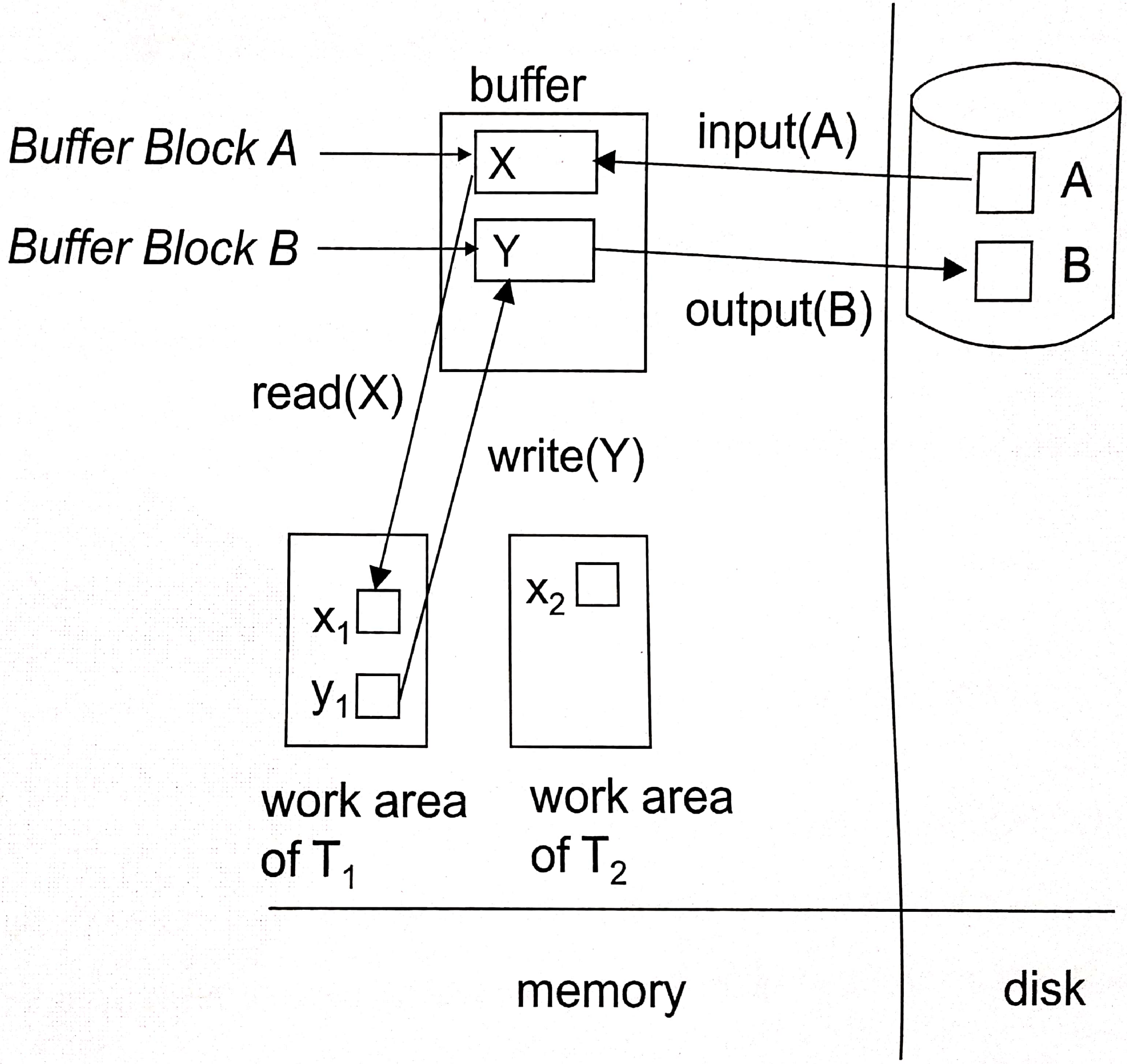
$$D = \left[\begin{array}{ccc} T1 & T2 & T3 \\ & R(B) & \\ R(C) & W(A) & \\ W(C) & & \\ R(D) & & \\ & & W(B) \\ W(D) & & W(A) \end{array}\right]$$



Example 2

A precedence graph of the schedule D, with 3 transactions. As there is a cycle (of length 2; with two edges) through the committed transactions T1 and T2, this schedule (history) is *not* Conflict serializable.

# Example of Data Access with Concurrent transactions



# Recovery Algorithm

- **Logging** (during normal operation):
    - <$T_i$ **start**> at transaction start
    - <$T_i$, $X_j$, $V_1$, $V_2$> for each update, and
    - <$T_i$ **commit**> at transaction end
- **Transaction rollback (during normal operation)**
    - Let $T_i$ be the transaction to be rolled back
    - Scan log backwards from the end, and for each log record of $T_i$ of the form <$T_i$, $X_j$, $V_1$, $V_2$>
        - perform the undo by writing $V_1$ to $X_j$,
        - write a log record <$T_i$ , $X_j$, $V_1$>
            - such log records are called **compensation log records**
    - Once the record <$T_i$ **start**> is found stop the scan and write the log record <$T_i$ **abort**>