A Primer on Database Management Systems and the Relational Model

Himadri Barman

A **database** is a collection of related data. By **data**, we mean known facts that can be recorded and that have implicit meaning.

The preceding definition of database is quite general. However, the common use of the term database is usually restricted. A database has the following restricted properties:

- A database represents some aspects of the real world, sometimes called the miniworld or the Universe of Discourse (UOD). Changes to the miniworld are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning.
- A database is designed, built, and populated with data for a specific purpose. This means the existence of a *structure*.
- It has an extended group of users and some preserved applications in which these users are interested. This gives the idea of *sharing*.

A database may be generated and maintained manually or it may be computerized. A computerized database may be created and maintained either by a group of application programs written specifically for that task or by a database management system. A database management system (DBMS) is a collection of programs that enables users to create and maintain a database. The DBMS is hence a general-purpose software system that facilitates the process of defining, constructing, and manipulating databases for various applications.

Defining a database involves specifying the data types, structures and constraints for the data to be stored in the database. Constructing the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. Manipulating a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.

Actors on the Scene:

Many persons are involved in the design, use, and maintenance of a large database with a few hundred users or more. We call them “actors on the scene” – those who work to maintain the database system environment, but who are not actively interested in the database itself.

- **Database Administrator**: There is a need for a person to oversee and manage resources when many persons use the same set of resources. In a database environment, the primary resource is the database itself and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the Database Administrator (DBA). In large organizations, the DBA has a support staff to carry out the designated functions.

  The responsibilities of the DBA are –
  Authorizing access to the database
  Coordinating and managing the use of the database
  Acquiring software and hardware resources as required

  The DBA is also accountable for the following problems –
  Breach of security
  Poor system response time

Downloaded from http://himadri.cmsdu.org

- **Database Designers**: The people responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. These tasks are mostly undertaken before the database is actually implemented and populated with data. It is the responsibility of the database designers to communicate with all prospective database users, in order to understand their requirements, and to come up with a design that meets these requirements. Database designers typically interact with each potential group of users and develop a **view** of the database that meets the data and processing requirements of this group. These views are then analyzed and *integrated* with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

- **End Users**: These are the people whose jobs require access to the database for querying, updating and generating reports; the database primarily exists for their use. There are several categories of end users –

  Casual end users occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle or high–level managers or other occasional browsers.

  Naïve or parametric end users make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates called **canned transactions** that have been carefully programmed and tested. The tasks performed are varied.

  Sophisticated end users include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS so as to implement their applications to meet their complex requirements.

  Stand-alone users maintain personal databases by using ready–made program packages that provide easy–to–use menu or graphic-based interfaces.

  A typical DBMS provides multiple facilities to access a database. Naïve end users need to learn very little about the facilities provided by the DBMS; they have to understand only the types of standard transactions designed and implemented for their use. Casual users learn only a few facilities that they may use repeatedly. Sophisticated users try to learn most of the DBMS facilities in order to achieve their complex requirements. Stand-alone users typically become very efficient in using a specific software package.

- **System Analysts and Application Programmers (Software Designers)**: System analysts determine the requirements of end users, and to develop specifications for canned transactions that meet these requirements. Application programmers implement these specifications as programs; they then test, debug, document, and maintain these canned transactions. Such analysts and programmers should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

Advantages of a DBMS:

- ☑ **Controlling Redundancy** – Redundancy is the storing of the same data multiple times. This leads to several problems. First, there is the need to perform a single logical update – such as entering data on a new student – multiple times; once for each file where student data is recorded. This leads to duplication of effort. Second, storage space is wasted when the same data is stored repeatedly, and this problem may me serious for large databases. Third, files that represent the same data may become inconsistent. This may happen because an update is applied to some of the files but not to others. Even, if an update – such as adding a new student - is applied to all the appropriate files, the data

Downloaded from http://himadri.cmsdu.org

concerning the student may still be inconsistent since each group applies updates independently.

In the database approach, the views of the different users groups are integrated during database design. For consistency, we should have a database design that stores each logical data item – such as student's name or birth date – in only one place in the database. This does not permit inconsistency and saves storage space. However, in some cases, controlled redundancy may be useful for improving the performance of queries.

- ☑ **Restricting Unauthorized Access –** When multiple users share a database, it is likely that some users will not be authorized to access all information in the database. In addition, some users may be permitted only to retrieve data, whereas others are allowed both to retrieve and update it. Hence, the type of access operation – retrieval or update - must also be controlled. Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database. A DBMS must provide a *security and authorization subsystem*, which the DBS uses to create accounts and to specify account restrictions.

- ☑ **Providing Multiple User Interfaces** – Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include query languages for casual users; programming language interfaces for application programmers; forms and command codes for parametric users; and menu-driven interfaces and natural interfaces for stand-alone users.

- ☑ **Representing Complex Relationships among Data** – A database may include numerous varieties of data that are interrelated in many ways. A DBMS must have the capability to represent complex relationships among the data as well as to retrieve and update related data easily and efficiently.

- ☑ **Enforcing Integrity Constraints –** Most database applications have certain integrity constraints that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints. The simplest type of integrity constraint involves specifying a data type for each data item. A more complex type of constraint that occurs frequently involves specifying that a record in one file must be related to records in other files. Another type of constraint specifies uniqueness on data item values. These constraints are derives from the meaning or *semantics* of the data and the miniworld it represents. It is the database designers' responsibility to identify integrity constraints during database design. Some constraints can be specified to the DBMS and automatically enforced. Other constrains may have to be checked by update programs or at the time of data entry.

- ☑ **Providing Backup and Recovery** – A DBMS must provide facilities for recovering from hardware or software failures. The *backup and recovery subsystem* of the DBMS is responsible for recovery.

- ☑ **Permitting Inferencing and Actions using Rules** – Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts. Such systems are called *deductive systems.*

- ☑ **Providing Persistent Storage for Program Objects and Data Structures –** Databases can be used to provide *persistent storage* for program objects and data structures. This is one of the main reasons for the emergence of *object-oriented database systems.* An object is said to be *persistent*, if it survives the termination of program execution and can later be directly retrieved by another program.

Downloaded from http://himadri.cmsdu.org

Implications of the Database Approach / Additional Advantages of a DBMS:

- ☑ **Potential for Enforcing Standards** - The database approach permits the DBA to define and enforce standards among database users in a large organization. This facilitates communication and cooperation among various departments, projects, and users within the organization. Standards can be defined for names and formats of data elements, display formats, report structures, terminology and so on.

- ☑ **Reduced Application Development Time** - Once, a database is up and running, substantially less time is generally required to create new applications using DBMS facilities.

- ☑ **Flexibility** – Modern DBMSs allow certain types of changes to the structure of the database without affecting the stored data and the existing application programs, greatly introducing flexibility.

- ☑ **Availability of Up-to-date Information** – A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can immediately see this update. This is made possible by the concurrency control and recovery subsystems of the DBMS.

- ☑ **Economies of Scale** – The DBMS approach permits consolidation of data and applications, thus reducing the amount of wasteful overlap between activities of data-processing personnel in different projects or departments. This enables the whole organization to invest in more powerful processors, storage devices, or communication gear, rather than having each department purchase its own (weaker) equipment. This reduces overall costs of operation and management.

When use of a DBMS is not advised:

In spite of the various advantages offered by a DBMS, there are situations where the overhead costs incurred in a DBMS are deemed unnecessary or infeasible. The overhead costs of using a DBMS are due to the following –

- High initial investment in hardware, software and training
- Generality that a DBMS provides for defining and processing data
- Overhead for providing security, concurrency control, recovery, and integrity functions

Additional problems may arise if the database designers and DBA do not properly design the database or if the database systems applications are not implemented properly. Hence, it may be more desirable to use regular files under the following circumstances –

- The database and applications are simple, well defined, and not expected to change
- There are stringent real-time requirements for some programs that may not be met because of DBMS overhead
- Multiple user access to data is not required

Data Model:

One fundamental characteristic of the database approach is that it provides for some level of data abstraction by hiding details of data storage that are not needed by most database users. A data model – a collection of concepts that can be used to describe the structure of a database – provides the necessary means to achieve the abstraction. By structure of a database, we mean the data types, relationships and constraints that should hold on the data. Most data models also include a set of basic operations for specifying retrieval and updates on the database. In addition to the basic operations provided by the data model, it is becoming more common to include concepts in the data model to specify the dynamic aspect or behaviour of a database application. This allows the database designer to specify a set of valid user-defined operations that are allowed on the database objects.

Downloaded from http://himadri.cmsdu.org

Data models are categorized according to the type of concepts they use to describe the database structure. These are –

- **High-level or Conceptual Data Models** provide concepts that are close to the way many users perceive data. It uses concepts such as entities (a real-world object or concept), attributes (some property of interest that furthers an entity) and relationships among two or more entities representing an interaction among the entities. ER Model is an example of this type of models.

- **Low-level or Physical Data Models** provide concepts that describe the details of how data is stored in the computer. These concepts are generally meant for computer specialists, not for typical end users.

- **Representational or Implementation Data Models**, which provide concepts that may be understood by end users but which are not too far removed from the way data is organized within the computer. These hide some details of data storage but can be implemented on a computer system in a direct way. These are the models used most frequently in traditional commercial DBMSs, and they include the widely used **Relational Data Model**, as well as the so-called legacy data models – **Network** and **Hierarchical Data Models**. Representational Data Models represent data by using record structures and hence are sometimes called **Record-based Data Models**.

Schemas, Instances and Database State:

The description of a database is called the **Database Schema**, which is specified during database design and is not expected to change frequently. Most data models have certain conventions for displaying the schema as diagrams. A displayed schema is called a **Schema Diagram**. Each object in the schema is called a **Schema Construct**. A schema diagram displays only some aspects of a schema, such as the names of record types and data items, and some types of constraints.

The actual data in a database may change quite frequently. The data in the database at a particular moment in time is called a **Database State** or a **Snapshot**. It is also called the *current set* of **Occurrences** or **Instances** in the database. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record, or change the value of a data item in a record, we change one state of the database into another state.

When we define a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of database when the database is first populated or loaded with the initial data. At any point ii time, the database has a *current state*.

The DBMS stores the description of the schema constructs and constraints – also called the **Meta Data** – in the DBMS catalog so that the DBMS software can refer to the schema whenever it need to. The schema is sometimes called the intension of the database state and a database state an extension of the schema.

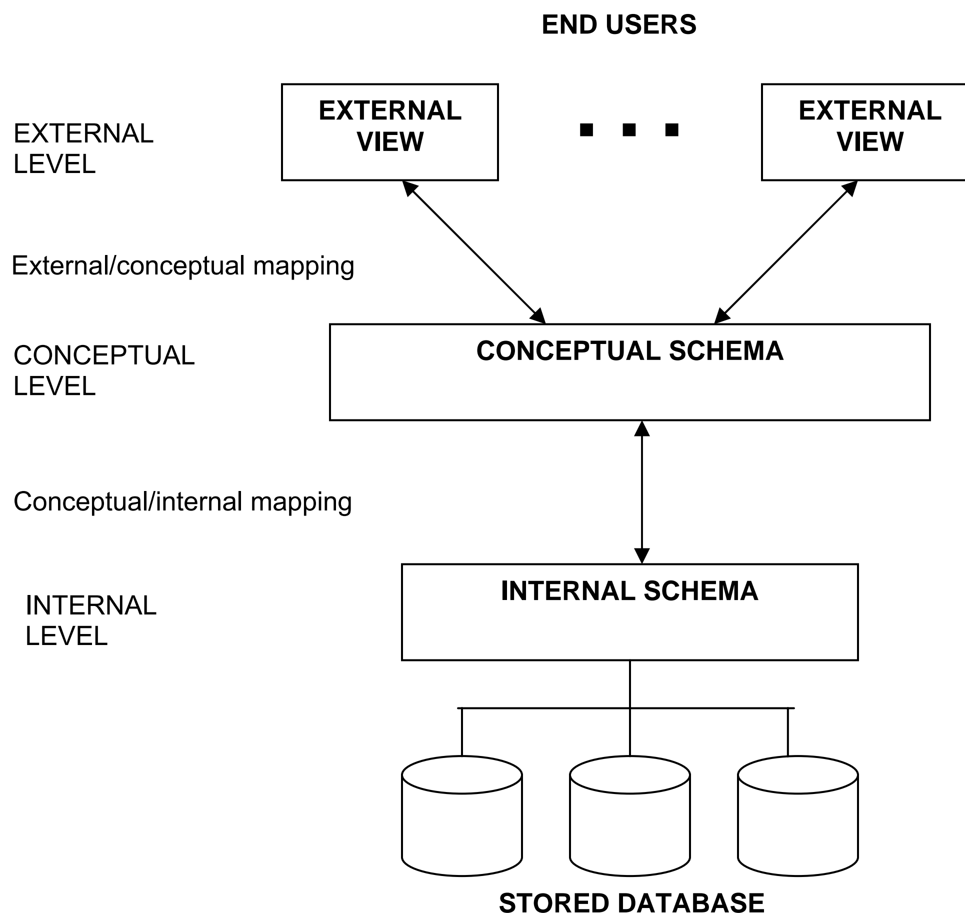
DBMS Architecture and Data Independence:

An architecture for database systems, called the three-schema architecture has been proposed to help achieve and visualize the important characteristics of the database approach. The goal is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

Downloaded from http://himadri.cmsdu.org

2. The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
3. The **external** or **view level** includes a number of **external schemas** or **user views**. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.



The process of transforming requests and results between different levels are called **mappings**.

Two types of data independence are defined –
Logical Data Independence is the capacity to change the conceptual schema without having to change the external schemas or application programs. We may change the conceptual schema to expand the database or to reduce the database. In the latter case, external schemas that refer only to the remaining data should not be affected. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.

Physical Data Independence is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized. If the same data as before remains in the database, we should not have to change the conceptual schema.

Downloaded from http://himadri.cmsdu.org

The Database System Environment:

A DBMS is a complex software system. What follows is a brief overview of the types of software components that constitute a DBMS and the types of Computer system software with which the DBMS interacts.

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **Operating System (OS)**, which schedules disk input/output. A higher-level **Stored Data Manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog. The stored data manager may use basic OS services for carrying out low-level data transfer, such as handling buffers in main memory. Once the data is in main memory buffers, it can be processed by other DBMS modules, as well as by application programs.

The **DDL Compiler** processes schema definitions and stores description of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the name of files, data items, storage details of each file, mapping information among schemas, and constraints, in addition to many other types of information that are needed by the DBMS modules. DBMS software modules then look up at the catalog information as needed.

The **Run-time Database Processor** handles database access at run time; it receives retrieval or update operations and carries them out on the database. Access to disk goes through the stored data manager. The **Query Compiler** handles high-level queries that are entered interactively. It parses, analyzes and compiles or interprets a query by creating database access code, and then generates calls to the run-time processor for executing the code.

The **Pre-compiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the **DML compiler** for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the run time data processor.

Classification of DBMS:

Several criteria are normally used to classify DBMSs. The first is the **data model** on which the DBMS is based. The two types of data models in many current commercial DBMSs are the **Relational Data Model** and the **Object Data Model**. Many legacy applications still run on the database systems based on the **Hierarchical** and **Network Data Models**. The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs called the **Object-Relational DBMSs**. A new innovation is the **Functional Data Model**.

The second criterion used to classify DBMSs is the **number of users** supported by the system. **Single-user Systems** support only one user at a time and are mostly used with personal computers. **Multi-user Systems**, which include the majority of DBMSs, support multiple users concurrently.

A third criterion is the **number of sites** over which the database is distributed. A DBMS is **Centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database themselves reside totally at a single computer site. A **Distributed DBMS (DDBMS)** can have the actual database and DBMS software distributed over many sites, connected by a computer network.

A fourth criterion is the **cost** of the DBMS. Accordingly, we have **Low-end Systems** and **High-end Systems**.

Downloaded from http://himadri.cmsdu.org

Finally, a DBMS can either be **General-purpose** or **Special-purpose**. **On-line Transaction Processing (OLTP) Systems** are special-purpose DBMSs.

The E-R Model:

An entity-relationship model (or E-R model) is a detailed, logical representation of the data for an organization or for a business area. An E-R model is normally expressed as an entity-relationship diagram (or E-R diagram), which is a graphical representation of an E-R model.

Entities and their Attributes:

The basic object that the ER model represents is an entity and is regarded to have an independent existence. An entity may be an object with a physical existence, e.g. – a particular person, a car, a house or an employee. It may also be an object with a conceptual existence, e.g. – a company, a job or a university course.

Attributes are the particular properties that describe an entity. For example, an employee may be described by the employee's name, age, address and salary. A particular entity will have a value for each of its attributes.

Types of Attributes:

- ✓ Single versus composite
- ✓ Single valued versus multi-valued
- ✓ Stored versus derived

Composite attributes can be divided into smaller sub-parts, which represent more basic attributes with independent meanings. Attributes that are not divisible are called single or atomic attributes.

Most attributes have a single value for a particular entity; such attributes are called single valued. On the other hand if an attribute has a number of values allowed for each individual entity, it is called multi-valued.
E.g., Single valued – age (particular cases)
Multi-valued – degrees of a person (in general)

In some cases two or more attribute values are related. For example date of birth (DOB) and age of a person. The age attribute is hence called a derived attribute, as it is derivable from the DOB attribute, which in turn is called a stored attribute.

Null values:

In some cases, a particular entity may not have an applicable value for an attribute. For such situations a special value called null is created. NULL can also be used if we do not know the value of an entity. The meaning of the former type of null is 'not applicable' whereas the meaning of the later is 'unknown'.

Weak Entities:

A database usually contains groups of entities that are similar; for example, a company employing hundreds of employees may want store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has its own value(s) for each attributes. The collection of all entities of a particular entity type in the database at any point in time is usually referred to using the same name as the entity type.

There is a subtle difference between *entity types* and *entity instances*. An *entity type* is a collection of entities that share common properties or characteristics. An *entity instance* is a single occurrence of an entity type.

Entity types that do not have key attributes of their own are called weak entity types. Strong entity types always have a key attribute.

Downloaded from http://himadri.cmsdu.org

Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with some of their attribute values.

Relationships:

Relationships are the glue that holds together the various components of an E-R model. A *relationship type* is a meaningful association between (or among) entity types. The phrase "meaningful association" implies that the relationship allows us to answer the questions that could not be answered given only the entity types. A *relationship instance* is an association between entity instances. Attributes may be associated with many-to-many (or one-to-one) relationship.

Relational Constraints:

Various restrictions on data can be specified on a relational database schema in the form of constraints. These include domain constraints, key constraints, entity integrity constraints and referential integrity constraints.

Another class of constraints are the 'data dependencies' (which include functional dependencies and multivalued dependencies) and are used mainly for database design by normalization.

Domain Constraint:

It specifies that the value of each attribute A must be an atomic value from the domain *dom*(A).

Key Constraint:

A relation is defined as a set of tuples and as such no two tuples can have the same combination of values for all their attributes. Usually, there are other subsets of attributes of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for their attributes. Suppose that we denote one such subset of attributes by SK; then for any two distinct tuples t1 and t2 in the relation state r of R, we have the constraint t1 [SK] ≠ t2 [SK].

Any such set of attributes SK is called a super key of the relation schema R. A super key SK specifies a uniqueness constraint that no two distinct tuples in a relation state r of R can have the same value for SK. Every relation has at least one default super key – the set of all the attributes. It may have redundant attributes.

Entity Integrity Constraint:

It specifies that no primary key value can be null.

Referential Integrity Constraint:

It states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation.

To define referential integrity more formally, we first define a *foreign key*. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relational schemas R1 and R2. A set of attributes FK in relation schema R1, is a foreign key of R1, that references relation R2 if it satisfies the following two rules –

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R2; the attributes FK are said to refer to the relation R2.
2. A value of FK in a tuple t1 of the current state r1(R1) either occurs as a value of PK for some tuple t2 in the current state r2(R2) or is null. In the first case, we have t1[FK] = t2[PK], and we say that the tuple t1 references or refers to the tuple t2. R1 is called the referencing relation and R2 is the referenced relation.

Downloaded from http://himadri.cmsdu.org

Key:

A key is a super key with the property that removing any attribute from it will leave a set of attributes that is not a super key. In other words a key is a minimal super key, that is, we can't remove any attributes from it and still have the uniqueness constraint hold.

Functional Dependencies:

A functional dependency is a constraint between two sets of attributes in a database. A functional dependency or FD, denoted by X → Y, between two sets of attributes X and Y that are subsets of R specifies constraints on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t1 and t2 in r that have t1 [x] = t2 [x], we must also have t1 [y] = t2 [y]. This means that the values of the y component of a tuple in r depend on, or are determined by the values of the x component; or alternatively, the values of the x component of a tuple uniquely (functionally) determine the values of the y component. The set of attributes x is called the LHS of the FD and y is called the RHS of the FD. The attribute(s) on the left side of the dependency are known as determinant(s). We also say that there is a functional dependence from x to y or that y is functionally dependent on x.

E.g.,

J	**K**	**L**
X	1	0
X	1	6
Y	4	1
Y	4	9
Z	3	5

J → K (K is F.D. on J)
J $\not\rightarrow$ L (L not F.D. on J)

The attribute on the LHS of the arrow is called a determinant.

Note: A functional dependency is a property of the semantics or meaning of the attributes. Whenever the semantics of two sets of attributes in R indicate that a functional dependency should hold, we specify the dependency as a constraint.

Normalisation:

The normalisation process, as first proposed by E.F. Codd, takes a relation schema through a series of tests to certify whether it satisfies a certain normal form. The process that proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary can thus be considered as 'relational design by analysis'. Initially Codd proposed three normal forms, which he called first, second and third normal form. A stronger definition of 3NF, called Boyce-Codd Normal form (BCNF) was proposed later by Boyce and Codd. All these normal forms are based on functional dependencies among the attributes of a relation.

Normalisation of data can be looked upon as a process of analyzing the given relational schema based on their functional dependencies and primary keys to achieve the desirable properties of (1) Minimising redundancies and (2) Minimising insertion, deletion and updation anomalies.

Unsatisfactory relation schemas that don't meet certain conditions – the normal form tests – are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database designers with:

- ✓ A formal framework
- ✓ A series of normal form tests.

In short, normalization is a formal process for deciding which attributes should be grouped together in a relation.

Downloaded from http://himadri.cmsdu.org

First Normal Form:
It states that the domain of an attribute must include only atomic values and that the value of any attribute in a tuple must be a single value from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple.1NF is now considered to be part of the formal definition of a relation in the basic relation model.

Full Functional Dependency:
A Functional Dependency X —› Y is a full functional dependency if the removal of any attribute **A** from X means that the dependency does not hold any more.

Partial Dependency:
A Functional Dependency X —› Y is a partial dependency if some attribute **A** Є X can be removed from X and the dependency still holds.

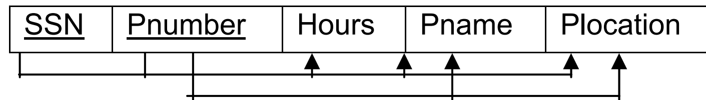
Prime and non – prime attributes:
An attribute of a relation schema R is called a prime attribute of R if it is a member of some candidate key of R. An attribute is called non-prime if it is not prime.

Second Normal Form:
A relation schema R is in 2NF if every non-prime attribute **A** in R is fully functionally dependent on the primary key of R, besides being in 1NF.

The test for 2NF involves resting for functional dependencies whose left – hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all.

If a relation schema is not in 2NF, it can be 'second normalised' into a number of 2NF relations in which non-prime attributes are associated only with the part of the primary key on which they are fully functionally dependent.

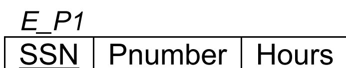
Emp_proj



2NF Normalisation

In the above example of the relation *Emp_proj*, {SSN, Pnumber} is the primary key. There is partial dependency because Pnumber also determines Hours, Pname, Plocation. So, we break up the relation and construct the following two relations which are both in 2NF.

E_P1



E_P2

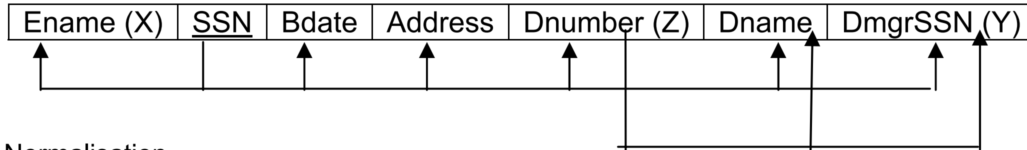
Transitive Dependency:
A functional dependency X → Y in a relation schema R is a transitive dependency if there is a set of attributes Z that is neither a candidate key nor a subset of any key of R, and both X → Z and Z → Y hold.

Note: This is the general definition of transitive dependency. Because, we are concerned only with primary keys in this section, we allow transitive dependency where X is the primary key but Z may be (a subset of) a candidate key.

Downloaded from http://himadri.cmsdu.org

Third Normal Form:
According to Codd's original definition, a relation schema R is in 3NF if it satisfies 2NF and no non prime attribute of R is transitively dependent on the primary key.

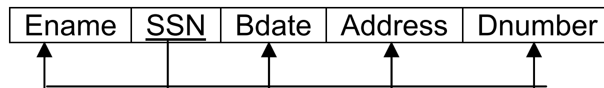
ED



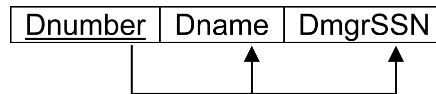
3NF Normalisation

Here the dependency SSN → DmgrSSN is transitive through Dnumber because dependencies (SSN → Dnumber) and (Dnumber → DmgrSSN) hold and Dnumber is neither a key itself nor a subset of the key in the relation.

ED1



ED2



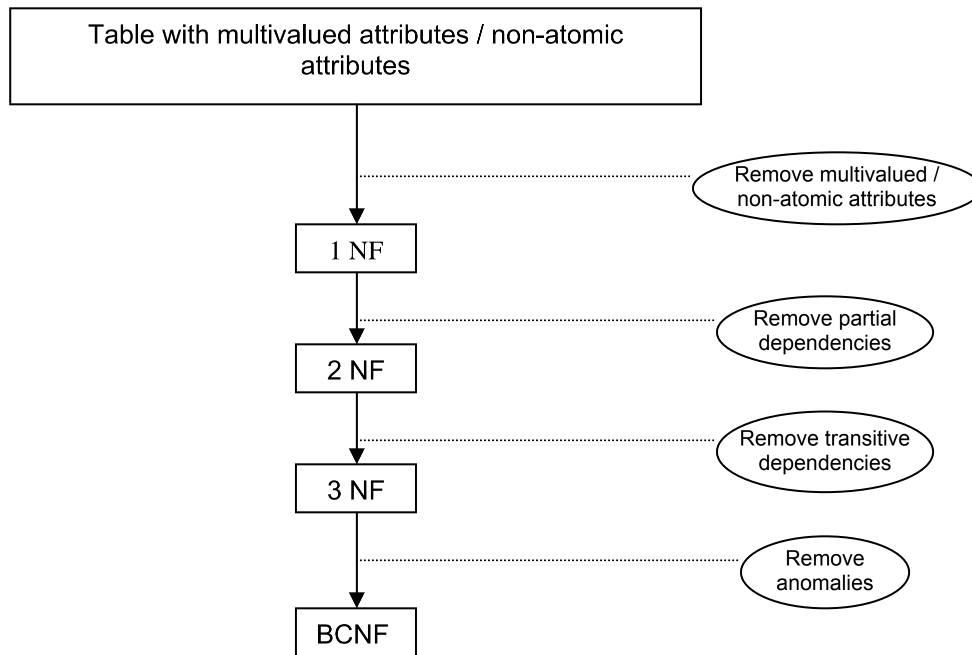
Boyce-Codd Normal Form:
BCNF was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF, because every relation in BCNF is also in 3NF, however a relation in 3NF is not necessarily in BCNF.

The formal definition of BCNF differs slightly from 3NF. A relation schema R is in BCNF if whenever a non-trivial functional dependency X → A holds in R, then X is a super key of R. The only difference between the definitions of BCNF and 3NF is that the condition of 3NF, which allows A to be prime is absent from BCNF. BCNF simply says that every determinant must be a candidate key.

In practice, most relation schemas that are in 3NF are also in BCNF. Only if X→A holds in a relation schema R with X not being a super key and A being a prime attribute, R will be in 3NF but not in BCNF.

Downloaded from http://himadri.cmsdu.org

The whole normalisation process as a series of steps is depicted below:



References:

- Fundamentals of Database Systems: *S. B. Navathe, R Elmasri*
- Modern Database Management: *Jeffrey A. Hoffer, Mary B. Prescott, Fred R. McFadden*

Downloaded from http://himadri.cmsdu.org