ANNAMACHARYA UNIVERSITY:: RAJAMPET

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

LECTURE NOTES

Name of the Faculty: M.NANDINI

Class: II B.Tech II Sem.

Branch: AI&DS

Name of the Course: DATABASE MANAGEMENT SYSTEMS

Subject Code: 24ACSE41T

Academic Year: 2025-2026

DEPARMENTOFAIDS

DBMS

M. NANDINI

UNIT-1:INTRODUCTION TO DBMS AND BASICS OF SQL

INTRODUCTION: Database-SystemApplications,Characteristics, Purpose Of Database Systems, View Of Data, Database Languages, Data Storage And Querying, Transaction Management, Data Base Architecture, Database Users And Administrators.

Basic Sql: Simple Database Schema , Data Types, Table Definitions(Create, Alter), Diferent DML Operations(Insert,Delete,Update)

Introductionto Database Management System

Asthename suggests,thedatabasemanagement system consists of twoparts.Theyare:

1. Database and
2. ManagementSystem

WhatisaDatabase?

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

Data:Facts,figures, statisticsetc. havingno particularmeaning(e.g.1, ABC,19 etc).
Record:Collectionofrelateddataitems,e.g.intheaboveexamplethethreedataitemshadnomeaning. Butifweorganizetheminthefollowingway,thentheycollectivelyrepresentmeaningfulinformation.

Roll	Name	Age
1	ABC	19

Tableor**Relation**:Collectionofrelatedrecords.

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

Thecolumnsofthisrelationarecalled**Fields**,**Attributes**or**Domains**.Therowsare called **Tuples** or **Records**.

Database:Collectionofrelatedrelations.Considerthefollowingcollectionoftables:

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

T1

Year	Hostel
I	H1
II	H2

T2

Roll	Address
1	KOL
2	DEL
3	MUM

T3

T4

Roll	Year
1	I
2	II
3	I

Wenowhaveacollectionof4tables.Theycanbecalleda“relatedcollection”becausewecan clearly find out that there are some common attributes existing in a selected pair of tables. Becauseofthesecommon attributes wemaycombinethedataoftwo or moretables 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.

AdatabaseinaDBMS couldbe viewedbylotsofdifferentpeople withdifferent responsibilities.

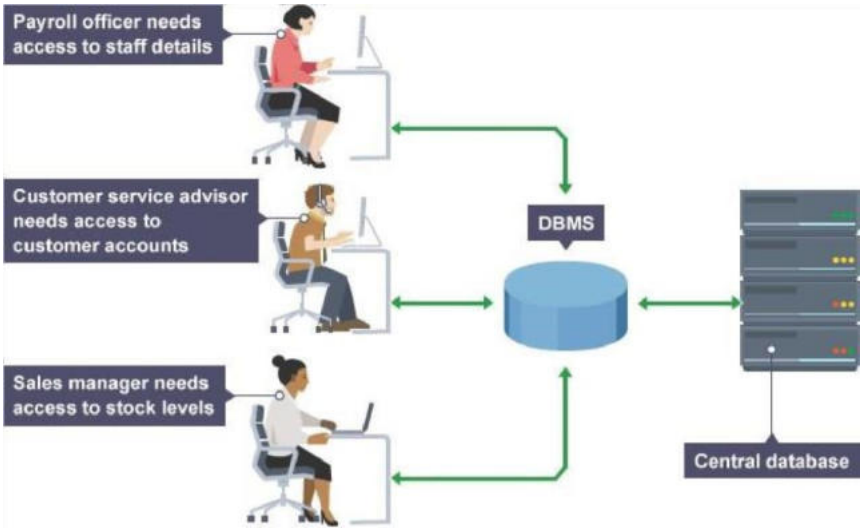


Figure1.1:EmpolyeesareaccessingDatathroughDBMS

Forexample,withinacompanytherearedifferentdepartments,aswellascustomers,whoeachneedto 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 strictlyin 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.

WhatisManagementSystem?

A **database-managementsystem**(DBMS)is a collection ofinterrelateddataand aset ofprograms to access thosedata.This is acollection ofrelated datawith an implicitmeaningand henceis adatabase. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. Theprimarygoal ofaDBMS is to provideawayto storeand retrievedatabaseinformation thatisboth*convenient* and*efficient*.By**data,**wemeanknownfactsthat canberecorded andthathave implicit meaning.

The management system is important because without the existence of some kind of rules and regulations it is not possibleto maintain thedatabase. Wehaveto select theparticularattributes which shouldbeincludedinaparticulartable;thecommonattributestocreaterelationshipbetweentwotables; if a new record has to be inserted or deleted then which tables should have to be handled etc. These issues must beresolvedbyhavingsomekind ofrulesto followin ordertomaintain theintegrityof 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 safetyof 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.

Becauseinformationissoimportantinmost organizations,computerscientistshavedevelopeda 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.

DatabaseManagementSystem(DBMS)andItsApplications:

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. Manufacturingandselling

5. Humanresources

EnterpriseInformation

- *Sales*:Forcustomer,product,andpurchase information.
- *Accounting*:Forpayments,receipts,accountbalances,assetsandother accountinginformation.
- *Human resources*: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
- *Manufacturing*: Formanagement of thesupplychain and fortrackingproduction ofitems in factories, inventories of items inwarehouses and stores, and orders for items.

Onlineretailers:Forsalesdatanotedaboveplusonlineordertracking,generationofrecommendation lists, and
maintenanceofonline productevaluations.

BankingandFinance

- *Banking*:Forcustomer information,accounts,loans,andbankingtransactions.
- *Creditcardtransactions*:Forpurchasesoncredit cardsandgenerationof monthlystatements.
- *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*:Forreservationsandscheduleinformation.Airlineswereamongthefirsttouse 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.

PurposeofDatabaseSystems

Databasesystemsaroseinresponsetoearlymethodsofcomputerizedmanagementofcommercialdata. As an example of such methods, typical of the 1960s, consider part of a university organization that, among otherdata,keeps information about all instructors, students, departments, and courseofferings. 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:

- ✓ Addnewstudents,instructors,andcourses
- ✓ Registerstudentsforcoursesandgenerateclassrosters
- ✓ Assign gradestostudents,computegradepointaverages (GPA),and generatetranscripts Systemprogrammerswrotetheseapplicationprogramstomeettheneedsoftheuniversity.

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 theinstructors in the department, students inthat major, course offerings, degree requirements,etc.Theuniversitymayhavetowritenewapplicationprogramstodealwithrulesspecific 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.

Thistypical**file-processingsystem**issupportedbyaconventionaloperatingsystem.Thesystemstores permanent recordsin various files, and it needs different application programs to extract recordsfrom, 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 fileprocessing 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 maybewritteninseveral programminglanguages. Moreover,thesameinformationmaybeduplicated inseveralplaces(files).Forexample,ifastudenthasadoublemajor(say,musicandmathematics)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,achangedstudentaddressmaybereflectedintheMusicdepartmentrecordsbutnotelsewhere in the system.

Difficulty in accessing data. Suppose that one of the university clerks needs to find out the names of allstudentswholivewithinaparticularpostal-codearea.Theclerkasksthedataprocessingdepartment togeneratesuchalist.Becausethedesignersoftheoriginalsystemdidnotanticipatethisrequest,there isnoapplicationprogramonhandtomeetit.Thereis,however,anapplicationprogramtogeneratethe list of *all* students.

The university clerk has now two choices: either obtain the list of all students and extract the needed informationmanuallyoraskaprogrammertowritethenecessaryapplicationprogram.Bothalternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the sameclerkneedstotrimthatlisttoincludeonlythosestudentswhohavetakenatleast60credithours. 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.

Dataisolation.Becausedataarescatteredinvariousfiles,andfilesmaybeindifferentformats,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**.Supposetheuniversitymaintainsanaccountforeachdepartment,andrecordsthebalance 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 difficulttochangetheprogramstoenforcethem.Theproblemiscompoundedwhenconstraintsinvolve several data items from different files.

Atomicity problems. A computer system, like any other device, is subject to failure. In many applications,itiscrucialthat,ifafailureoccurs,thedataberestoredtotheconsistentstatethatexisted prior to the failure.

Consider a program to transfer $500 from the account balance of department *A* to the account balance ofdepartment*B*.Ifasystemfailureoccursduringtheexecutionoftheprogram,itispossiblethatthe

$500wasremovedfromthebalanceofdepartment*A*butwasnotcreditedtothebalanceofdepartment *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-accessanomalies. Forthesakeofoverallperformanceofthe systemandfasterresponse, manysystemsallowmultipleuserstoupdatethedatasimultaneously.Indeed,today,thelargestInternet retailersmayhavemillionsofaccessesperdaytotheirdatabyshoppers.Insuchanenvironment, interaction of concurrent updates is possible and may result in inconsistent data. Consider department *A*,withanaccountbalanceof$10,000.Iftwodepartmentclerksdebittheaccountbalance(bysay$500 and $100, respectively) of department *A* at almost exactly the same time, the result of the concurrent executionsmayleavethebudgetinanincorrect(orinconsistent)state.Supposethattheprograms executing on behalf of each withdrawal read the old balance, reduce thatvalue by the amount beingwithdrawn,andwritetheresultback.Ifthetwoprogramsrunconcurrently,they may bothreadthe value$10,000,andwriteback$9500and$9900,respectively.Dependingonwhichonewritesthevalue last, the account balance of department *A* may contain either $9500 or $9900, rather than the correctvalue of $9400. To guard against this possibility, the system must maintain some form of supervision. Butsupervisionisdifficulttoprovidebecausedatamaybeaccessedbymanydifferentapplication 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 onetothecount,andstoresthecountbackinthedatabase.Supposetwostudentsregisterconcurrently, 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 registeredforthecourseandthecountshouldbe41.Furthermore,supposethecourseregistrationlimit 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 onlythat 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.

Advantagesof DBMS:

Controlling of Redundancy: Data redundancy refers to the duplication of data (i.e storing same data multiple times). In a database system, byhaving 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.

ImprovedDataSharing:DBMSallowsausertosharethedatainanynumberofapplicationprograms.

DataIntegrity:Integritymeansthatthedatainthedatabaseisaccurate.Centralizedcontrolofthedata 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.Forexample:isacustomeraddressisstoredonlyonce,wecannothavedisagreementon thestoredvalues.Alsoupdatingdatavaluesisgreatlysimplifiedwheneachvalueisstoredinoneplace only. Finally, we avoid the wasted storage that results from redundant data storage.

EfficientDataAccess: Inadatabasesystem,thedataismanagedbytheDBMSandallaccessto the data is through the DBMS providing a key to effective data processing

EnforcementsofStandards:Withthecentralizedofdata,DBAcanestablishandenforcethedata standards which may include the naming conventions, data quality standards etc.

Data Independence : Ina database system, the database management system provides the interfacebetween theapplicationprogramsandthedata.Whenchangesaremadetothedatarepresentation,the meta data obtained by the DBMS is changed but the DBMS is continues to provide the data to application program in the previouslyused 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 manyapplications, accessing data stored in the DBMS, which facilitates the quick development of application.

DisadvantagesofDBMS

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) DBMSsystemworksonthecentralizedsystem,i.e.;alltheusersfromallovertheworld 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 someof the application will run slow.

ViewofData

A database system is a collection of interrelated data and a set of programs that allow users to access and modifythesedata.A majorpurposeofadatabasesystem is to provideuserswith an *abstract* view of the data. That is, the system hides certain details of how the data are stored and maintained.

DataAbstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiencyhas led designers to usecomplex datastructures to represent datain thedatabase. Since many database-system usersare not computer trained, developers hide the complexityfrom users through several levels of abstraction, to simplify users' interactions with the system:

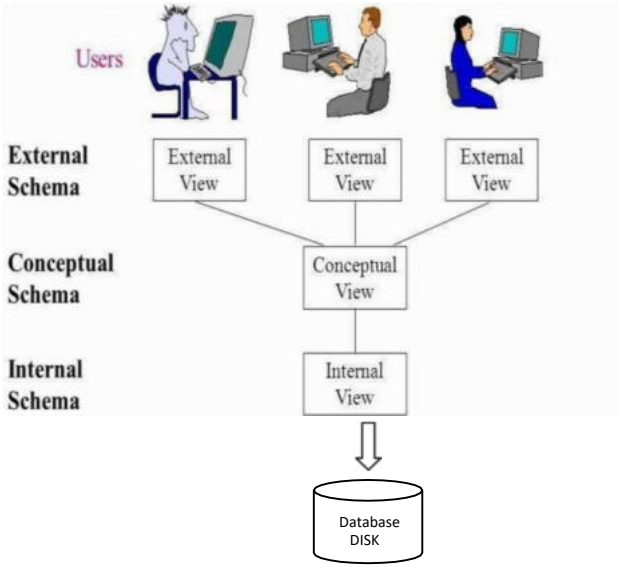


Figure1.2:LevelsofAbstractioninaDBMS

- **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**.Databaseadministrators,whomustdecidewhatinformationtokeepinthe database, use the logical level of abstraction.

- **Viewlevel(orExternalView/Schema):**Thehighestlevelofabstractiondescribesonly part of the entire database. Even though the logical level uses simpler structures, complexityremainsbecauseofthevarietyofinformationstoredinalargedatabase.Many usersofthedatabasesystem do not need all this information; instead, theyneed to access onlyapartofthedatabase.Theviewlevelofabstractionexiststosimplifytheirinteraction 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. Manyhigh-level programminglanguages 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**;

Thiscodedefinesanewrecordtypecalled*instructor*withfourfields.Eachfieldha

sanameandatype associated with it. A university organization may have several such record types, including

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

At the physical level, an *instructor*, *department*, or *student* record can be described as a block of consecutivestoragelocations.Thecompilerhides thislevelofdetailfromprogrammers.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.

Atthelogicallevel,eachsuchrecordisdescribedbyatypedefinition,asinthepreviouscodesegment, and the interrelationship of these record types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, databaseadministrators usuallywork at this level of abstraction.

Finally, at theviewlevel, computerusserssee aset ofapplication programs that hidedetails ofthe data types.Attheviewlevel,severalviewsofthedatabasearedefined,andadatabaseuserseessomeorall of these views. In addition to hiding details of the logical level of the database, the views also provide asecuritymechanismtopreventusersfromaccessingcertainpartsofthedatabase.Forexample,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.

InstancesandSchemas

Databaseschangeovertimeasinformationisinsertedanddeleted.Thecollectionofinformationstored 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 analogyto a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program.

Eachvariablehasaparticularvalueatagiveninstant.Thevaluesofthevariablesinaprogramatapoint 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 logicalschema.Thephysicalschemaishiddenbeneaththelogicalschema,andcanusuallybechanged easilywithoutaffectingapplicationprograms.Applicationprogramsaresaidtoexhibit**physicaldata independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

DataModels

Underlyingthestructureofadatabaseisthe**datamodel**:acollectionofconceptualtoolsfordescribing 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.

Thedata modelscan be classified intofour different categories:

• **RelationalModel**.Therelationalmodelusesacollectionoftablestorepresentbothdataandthe relationshipsamongthosedata.Eachtablehasmultiplecolumns,andeachcolumnhasauniquename. Tables are also known as **relations**. The relational model is an example of a record-based model. Recordbasedmodelsaresonamedbecausethedatabaseisstructuredinfixed-formatrecordsofseveral types.Eachtablecontainsrecordsofaparticulartype.Eachrecordtypedefinesafixednumberoffields, orattributes.Thecolumnsofthetablecorrespondtotheattributesoftherecordtype.Therelationaldata modelisthemostwidelyuseddatamodel,andavastmajorityofcurrentdatabasesystemsarebasedon the relational model.

Entity-RelationshipModel.Theentity-relationship(E-R)datamodelusesacollectionofbasic 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. Theentityrelationship model is widely used in database design.

Object-BasedDataModel.Object-orientedprogramming(especiallyinJava,C++,orC#)hasbecome 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-structuredDataModel.Thesemi-structureddatamodelpermitsthespecificationofdatawhere 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**ExtensibleMarkupLanguage(XML)**iswidelyusedtorepresentsemi-structureddata.

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 modelingdata.Asaresulttheyareusedlittlenow, exceptinolddatabasecodethatisstillinservicein some places.

DatabaseLanguages

Adatabasesystemprovidesa**data-definitionlanguage**tospecifythedatabaseschemaand a **datamanipulation language** to express database queries and updates. In practice, the datadefinitionanddata-manipulationlanguagesarenottwoseparatelanguages;insteadthey simply form parts of a single database language, such as the widely used SQL language.

Data-ManipulationLanguage

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:

- Retrievalofinformation storedinthedatabase
- Insertionofnew informationintothe database
- Deletionofinformationfromthe database
- Modificationofinformation storedinthedatabase

There arebasicallytwotypes:

- **Procedural DMLs**require a user to specify *what*data are needed and*how*to get thosedata.
- **DeclarativeDMLs**(alsoreferredtoas**nonproceduralDMLs**)requireausertospecify *what*data areneeded*without* specifyinghow toget thosedata.

DeclarativeDMLsareusuallyeasiertolearnandusethanareproceduralDMLs.However,sinceauser doesnothavetospecifyhowtogetthedata,thedatabasesystemhastofigureoutanefficientmeansof 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-DefinitionLanguage(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.

Wespecifythestoragestructureandaccessmethodsusedbythedatabasesystembyasetofstatements 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.

Thedatavaluesstoredinthedatabasemustsatisfycertain**consistencyconstraints**.

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 pertainingto the database. However, arbitrarypredicates maybe costlyto 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(forexample, integertypes, charactertypes, date/time types). Declaringan attribute tobeofaparticulardomainactsasaconstraintonthevaluesthatitcantake.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.

- **ReferentialIntegrity**.Therearecaseswherewewishtoensurethatavaluethatappears inonerelationforagivensetofattributesalsoappearsinacertainsetofattributesinanother relation (referential integrity). For example, the department listed for each course must be onethatactuallyexists.Moreprecisely,the*deptname*valueina*course*recordmustappear inthe*deptname*attributeofsomerecordofthe*department*relation.Databasemodifications 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**.Anassertionisanyconditionthatthedatabasemustalwayssatisfy.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 beexpressedasanassertion.Whenanassertioniscreated,thesystemtestsitforvalidity.If theassertionisvalid,thenanyfuturemodificationtothedatabaseisallowedonlyifitdoes not cause that assertion to be violated.

- **Authorization**.Wemaywanttodifferentiateamongtheusersasfarasthetypeofaccess 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 allowsreading,butnotmodification,ofdata;**insertauthorization**,whichallowsinsertion of new

data, but not modification of existing data; **update authorization**, which allows modification,butnotdeletion,ofdata;and **delete authorization**,whichallowsdeletionof 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 canonlybeaccessedandupdatedbythedatabasesystemitself(notaregularuser).Thedatabasesystem consults the data dictionary before reading or modifying actual data.

DataDictionary

WecandefineadatadictionaryasaDBMScomponentthatstoresthedefinitionofdatacharacteristics and relationships. You mayrecall 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 resemblesandX-rayofthecompany'sentiredataset,andisacrucialelementinthedataadministration function.

Thetwomaintypesofdatadictionaryexist,integratedandstandalone.Anintegrateddatadictionaryis includedwiththeDBMS.Forexample,allrelationalDBMSsincludeabuiltindatadictionaryorsystem 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 dictionaryis automatically updatedbytheDBMSwitheverydatabaseaccess,therebykeepingitsaccessinformationup-to-date.A passivedatadictionaryisnotupdatedautomaticallyandusuallyrequiresabatchprocesstoberun.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 abilityto communicate. In addition, the data dictionaryis 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.
- Tablesdefineinalldatabases.Forexample,thedatadictionaryislikelytostorethenameof 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.
- Endusersand TheAdministratorsofthedata base
- Programs that access the database including screen formats, report formats application formats, SQL queries and so on.
- Access authorizationforallusersofalldatabases.

- Relationships among data elements which elements are involved: whether the relationship are mandatory or optional, the connectivity and cardinality and so on.

DatabaseAdministratorsandDatabaseUsers

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.

DatabaseUsersandUserInterfaces

Therearefourdifferenttypesofdatabase-systemusers,differentiatedbythewaytheyexpecttointeract 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 byinvokingone of the application programsthathavebeen writtenpreviously.Forexample,abanktellerwhoneedstotransfer$50from 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.
Asanotherexample,considerauserwhowishestofindheraccountbalanceovertheWorldWideWeb. 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 usercanfillinappropriatefieldsoftheform.Naiveusersmayalsosimplyread *reports*generatedfrom the database.
Application programmers are computer professionals who write application programs. Application programmerscanchoosefrommanytoolstodevelopuserinterfaces.**Rapidapplicationdevelopment (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 thedatamanipulationlanguage.Theselanguages,sometimescalled*fourth-generationlanguages*,often includespecialfeaturestofacilitatethegenerationofformsandthedisplayofdataonthescreen.Most major commercial database systems include a fourth generation language.
Sophisticatedusersinteractwiththesystemwithoutwritingprograms.Instead,theyformtheirrequests inadatabasequerylanguage.Theysubmiteachsuchquerytoa**queryprocessor**,whosefunctionisto 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), orbyproduct, orbyacombination ofregion and product (that is, total salesof eachproductineachregion).Thetoolsalsopermittheanalysttoselectspecificregions,lookatdatain 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.

DatabaseArchitecture:

We are now in a position to provide a single picture (Figure 1.3) of the various components of a database system and the connections among them.

Thearchitectureofadatabasesystemisgreatlyinfluencedbytheunderlyingcomputersystemonwhich the database system runs. Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed toexploitparallelcomputerarchitectures.Distributeddatabasesspanmultiplegeographicallyseparated machines.

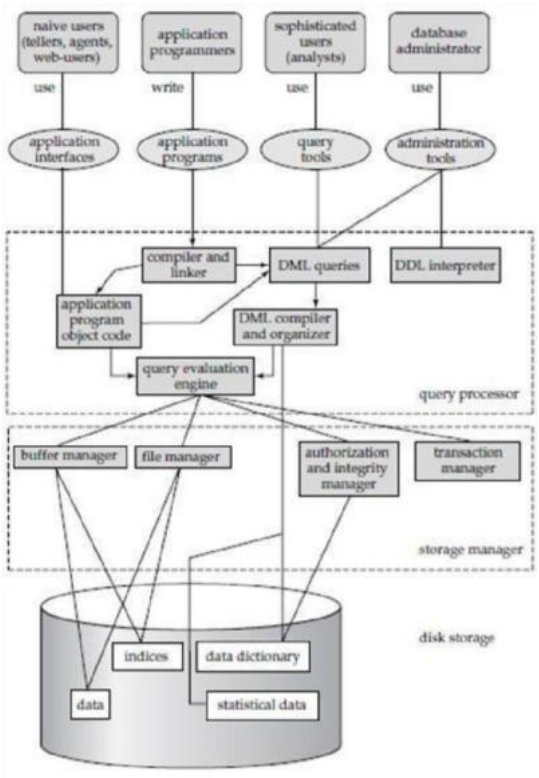


Figure1.3:DatabaseSystemArchitecture

A database system is partitioned into modules that deal with each of the responsibilities of theoverallsystem.Thefunctionalcomponentsofadatabasesystemcanbebroadlydivided into the **storage manager** and the **query processor** components. The storage manager is important because databases typically require a large amount of storage space. The query processorisimportantbecauseithelpsthedatabasesystemsimplifyandfacilitateaccessto data.

Itisthejobofthedatabasesystemtotranslateupdatesandquerieswritteninanonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

Database applications are usually partitioned into two or three parts, as in Figure 1.4. In a twotierarchitecture,theapplicationresidesattheclientmachine,whereitinvokesdatabase system functionalityat theservermachinethrough querylanguage statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server. In contrast, in a three-tier architecture, the client machine acts as merely a front end

and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface.

The application server in turn communicates with a database system to access data. The businesslogicoftheapplication,whichsayswhatactionstocarryoutunderwhatconditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tierapplicationsaremoreappropriateforlargeapplications,andforapplicationsthat run on the WorldWideWeb.

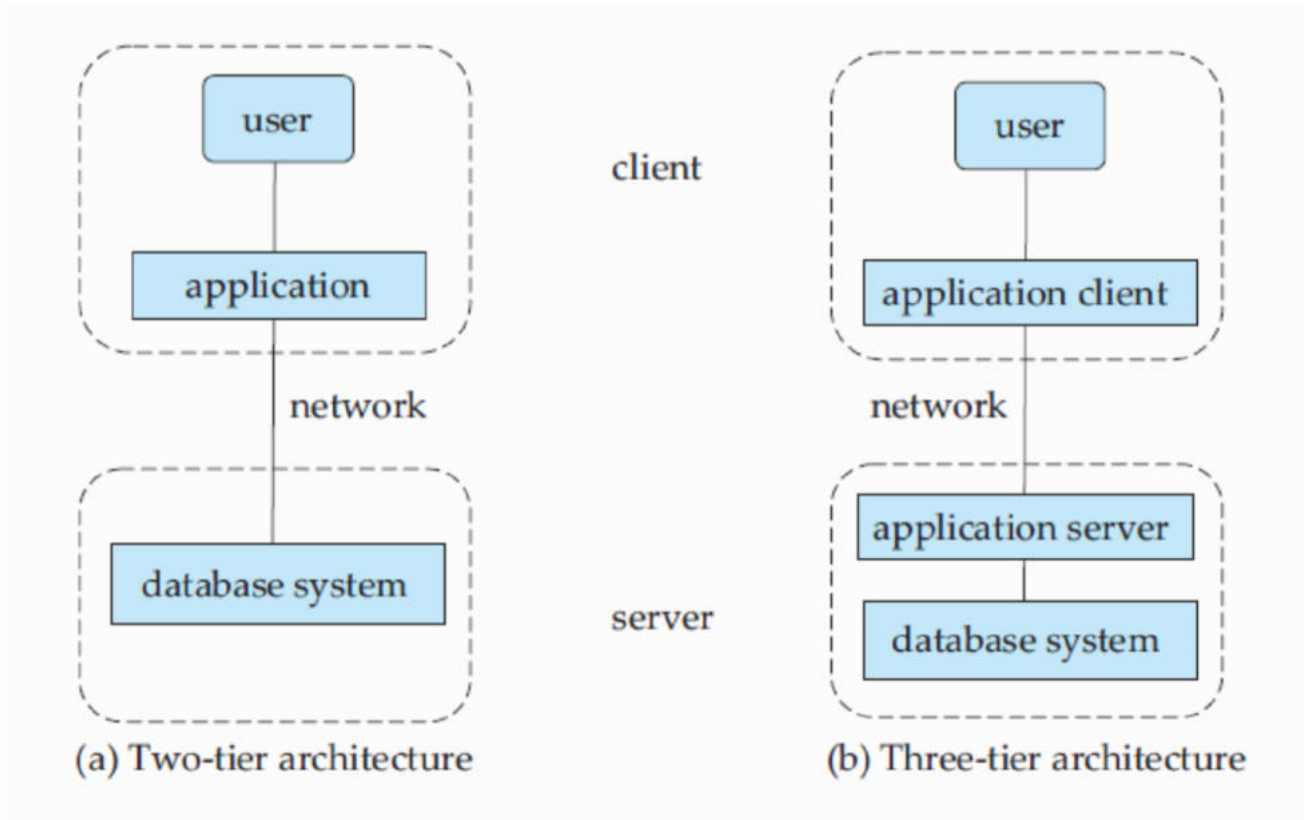


Figure1.4:Two-tierandthree-tierarchitectures.

QueryProcessor:

Thequeryprocessorcomponentsinclude

·**DDLinterpreter,**whichinterpretsDDLstatementsandrecordsthedefinitionsinthe datadictionary.·**DMLcompiler,**whichtranslatesDMLstatementsinaquerylanguage intoanevaluationplanconsistingoflow-levelinstructionsthatthequeryevaluationengine understands.

A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**, that is, it picks the lowest cost evaluation plan from among the alternatives.

Query evaluation engine, which executes low-level instructions generated by the DML compiler.

StorageManager:

A *storage manager* is a program module that provides the interface between the lowlevel datastoredinthedatabaseandtheapplicationprogramsandqueriessubmittedtothesystem. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-levelfile-systemcommands.Thus,thestoragemanagerisresponsibleforstoring,retrieving, and updating data in the database. The storage manager components include:

· **Authorizationandintegritymanager**,whichtestsforthesatisfactionofintegrityconstraints and checks the authority of users to access data.

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

· **Filemanager**,whichmanagestheallocationofspaceondiskstorageandthedatastructures used to represent information stored on disk.

· **Buffer manager**, which is responsible for fetching data from disk storage into main memory,anddecidingwhatdatatocacheinmainmemory.Thebuffermanagerisacritical partofthedatabasesystem, sinceit enablesthedatabaseto handledatasizes that aremuch larger than the size of main memory.

TransactionManager:

A **transaction** is a collection of operations that performs a single logical function in a databaseapplication.Eachtransactionisaunitofbothatomicityandconsistency.Thus,we require that transactions do not violate anydatabase-consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. **Transaction - manager** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

A database schema is the structural blueprint of a database, which includes definitions for tables, columns, constraints (like primary and foreign keys), and the **datatypes** assigned to eachcolumn.Datatypesdefinethekindofvaluesacolumncanhold(e.g.,integer,text,date).

Common data types generally fall into these main categories, although specific names and sizes vary between different Database Management Systems (DBMS) like MySQL, PostgreSQL, or SQL Server:

CommonDataTypeCategories

NumericTypes:Usedforstoringnumbers.o

): INT(orINTEGERForwholenumbers(integers).

DECIMAL(p,) o orNUMERIC(p,s):Forexactprecisionnumbers,suchascurrencyorfinancialdata, s where 'p' is the total digits and 's' is the digits after the decimal point.

o FLOAT orREAL:Forfloating-point(approximate)numbersthatcontaindecimals.

BOOLEAN ~~BIT~~:Forlogicalvalues,typically TRUE or FALSE o or(or 1or 0).

String/CharacterTypes:Used forstoringtext,letters, numbers,and symbols.

: A CHAR(size) fixed-length string. It always uses the full specified length, which is efficient for

o data of a consistent size (e.g., a 2-character state abbreviation).

VARCHAR(size) :Avariable-

- a small overhead.

o lengthstring.Itonlyusesthestorageneededfortheactualdata entered, plus

o TEXT orMEMO: Forstoringlargeamountsof text data.

DateandTimeTypes:Usedforworkingwithdatesand

DATEtimes.o: Storesadate only. o: Storesatime only.

TIME o DATETIME orTIMESTAMP :Storesbothdateandtime,oftenusedforrecordingwhena row was created or modified.

BinaryTypes:Usedforstoringbinarydatasuchasimages,audio,orotherfiles.o

BLOB (Binary Large Object) orBINARY :/VARBINARY Stores raw binary byte strings.

ImportanceinSchemaDesign

Choosingthecorrectdatatypeiscrucialduringdatabasedesign(datamodeling)forseveralreasons:

- **Data Integrity**: Ensures only valid data types can be entered into a column (e.g., preventing textfrom being stored in a numeric age field).

- **StorageEfficiency**:Usingtheappropriatedatatypehelps managediskspaceeffectively.
- **Performance**: Proper data types and indexing improve the speed and efficiency of data retrieval(queries).

DDLANDDMLCOMMANDS:

DDL (Data Definition Language) commands are used to define or modify the database structure,whileDML(DataManipulationLanguage)commandsareusedtomanagethedata within those structures.

DDLCommands(DataDefinitionLanguage)

DDLcommandsaffecttheschemaorstructureofthedatabaseandareauto-committed,meaningthe changes are permanent and cannot be rolled back.

_ **CREATE**:Usedtocreatenewdatabaseobjects,suchastables,indexes,views,orthedatabaseitself.

o *Example*: CREATE TABLE Students (StudentID int, Name varchar(255));

_ **ALTER**: Usedto modifythestructureof anexistingdatabaseobject, likeadding,deleting,ormodifyingcolumnsinatable. o *Example*: ALTER TABLE Students ADD Email varchar(255);

_ **DROP** : Used to permanently delete a database object (e.g., a table, index, or database) and all the data within it.

o *Example*:DROPTABLEStudents;

_ **TRUNCATE** : Used to quickly delete all records from a table but keeps the table structure intact for future use. It is a DDL command because, like DROP , its operations are non-transactional and cannot be rolled back.

o *Example*:TRUNCATETABLEStudents;

_ :**RENAME**Usedtochangethenameofadatabase object.

o *Example*: RENAME TABLE Students TO Student_Data;

DMLCommands(DataManipulationLanguage)

DML commands are used to manage the data stored within the database objects. These commands are transactional, meaning changes can be rolled back (undone) if necessary before being committed permanently.

- **INSERT**: Usedto addnew rows(records) ofdataintoatable.
  - *Example*: INSERTINTOStudents(StudentID,Name)VALUES(1,'John Doe');
- **UPDATE**: Usedto modifyexisting datawithina table.
  - *Example*:

```
UPDATEStudentsSETName='JaneDoe'WHEREStudentID=
1;
```

- **DELETE**: Usedtoremoveexistingrecords(rows)froma table.
  - *Example*: DELETE FROMStudentsWHERE StudentID =1;
- **SELECT**: While often categorized separately as Data Query Language (DQL), SELECT is widely considered part of DML as it interacts with the data within the tables.
  - *Example*:SELECT*FROM Students;

UNITII: DATABASE DESIGN

Database Design: Database Design And Er Diagrams, Entities, Atributes And Entity Sets, Relationships And Relationship Sets, Additional Features Of The Er Model, Conceptual Design With The Er Model, Case Study: The Internet Shop.

The Relational Model: Introduction To The Relational Model, IntegrityConstraintsOverRelations,EnforcingIntegrityConstraints, Querying Relational Data, Logical Data Base Design: Er To Relational.

WhatisERModeling?

Agraphicaltechniqueforunderstandingandorganizingthedataindependent of the actual database implementation

Weneedtobefamiliarwiththefollowingtermstogofurther.

Entity

Anythingthathasanindependentexistenceandaboutwhichwecollectdata. It is also known as entity type.

InERmodeling,notationforentityisgivenbelow.

Entityinstance

Entityinstanceisaparticularmemberoftheentitytype. Example for entity instance : A particular employee **Regular Entity**

Anentitywhichhasitsownkeyattributeisaregularentity. Example for regular entity : Employee. **Weakentity**

Anentitywhichdependsonotherentityforitsexistenceanddoesn'thaveany key attribute of its own is a weak entity.

Exampleforaweakentity:Inaparent/childrelationship,aparentis considered as a strong entity and the child is a weak entity.

InERmodeling,notationforweakentityisgivenbelow.

Attributes

Properties/characteristicswhichdescribeentitiesarecalledattributes. In ER modeling, notation for attribute is given below.

DomainofAttributes

Thesetofpossiblevaluesthatanattributecantakeiscalledthedomainofthe attribute.Forexample,theattributedaymaytakeanyvaluefromtheset {Monday,Tuesday...Friday}.Hencethissetcanbetermedasthedomainof the attribute day. **Key attribute**

The attribute (or combination of attributes) which is unique for every entity instance is called key attribute.

E.gtheemployee_idofanemployee,pan_card_numberofapersonetc.Ifthe key attribute consists of two or more attributes in combination, it is called a composite key.

InERmodeling,notationforkeyattributeisgivenbelow.

Simpleattribute

Ifanattributecannotbedividedintosimplercomponents,itisasimple attribute.

Exampleforsimpleattribute:employee_idofanemployee.

Compositeattribute

Ifanattributecanbesplitintocomponents,itiscalledacompositeattribute.

Example for composite attribute : Name of the employee which can be split into First_name, Middle_name, and Last_name.

SinglevaluedAttributes

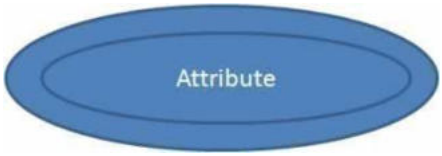
Ifanattributecantakeonlyasinglevalueforeachentityinstance,itisasingle valued attribute.

exampleforsinglevaluedattribute:ageofastudent.Itcantakeonlyone value for a particular student.

Multi-valuedAttributes

If an attribute can take more than one value for each entity instance, it is a multi-valued attribute. Multi-valued example for multi valued attribute : telephone number of an employee, a particular employee may have multiple telephone numbers.

InERmodeling,notationformulti-valuedattributeisgivenbelow.



StoredAttribute

Anattributewhichneedtobestoredpermanentlyisastoredattribute Example for stored attribute : name of a student

DerivedAttribute

An attribute which can be calculated or derived based on other attributes is a derived attribute.

Exampleforderivedattribute:ageofemployeewhichcanbecalculatedfrom date of birth and current date.

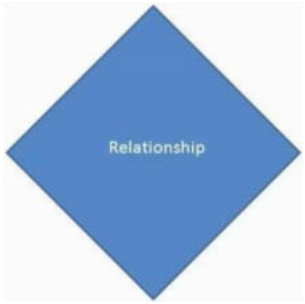
InERmodeling,notationforderivedattributeisgivenbelow.

Relationships

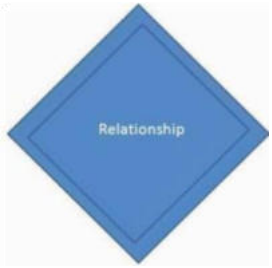
Associationsbetweenentitiesarecalledrelationships

Example:Anemployeeworksforanorganization.Here"worksfor"isa relation between the entities employee and organization.

InERmodeling,notationforrelationshipisgivenbelow.



Howeverin ER Modeling,Toconnecta weakEntitywithothers, youshould use a weak relationship notation as given below



Degreeofa Relationship

Degree of a relationship is the number of entity types involved. The n-ary relationship is the generalformfordegreen. Special cases are unary, binary, and ternary ,where the degree is 1, 2, and 3, respectively.

Example for unary relationship : An employee ia a manager of another employee Example

for binary relationship : An employee works-for department.

Exampleforternaryrelationship:customerpurchaseitemfromashopkeeper

CardinalityofaRelationship

Relationship cardinalities specify how many of each entity type is allowed. Relationships can have four possible connectivities as given below.

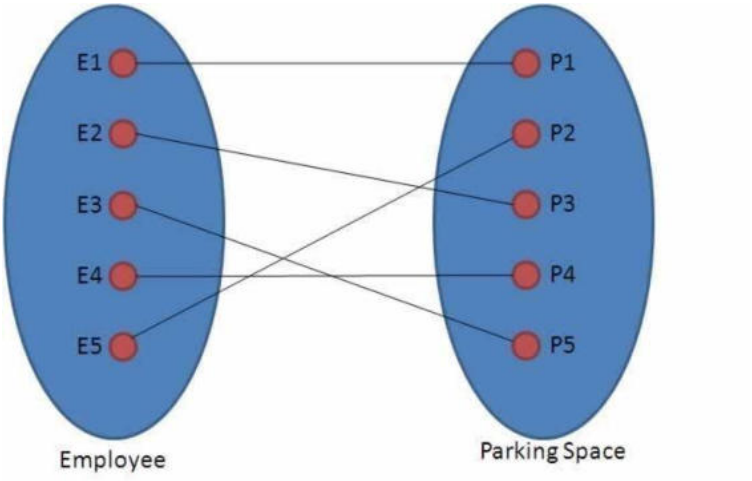
1. Onetoone(1:1)relationship
2. Onetomany(1:N)relationship

3. Manytoone(M:1)relationship

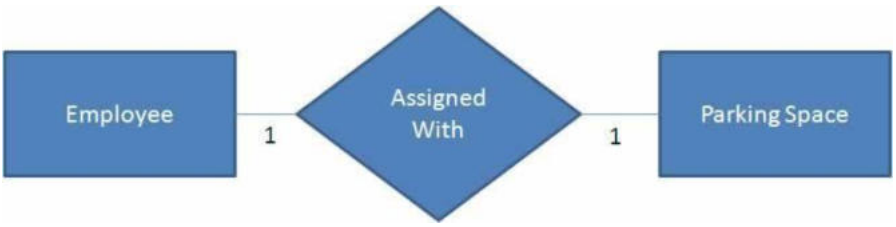
4. Manytomany(M:N)relationship

Theminimumandmaximumvaluesofthisconnectivityiscalledthecardinality of the relationship

Example for Cardinality – One-to- One (1:1) Employee is assigned with a parking space.

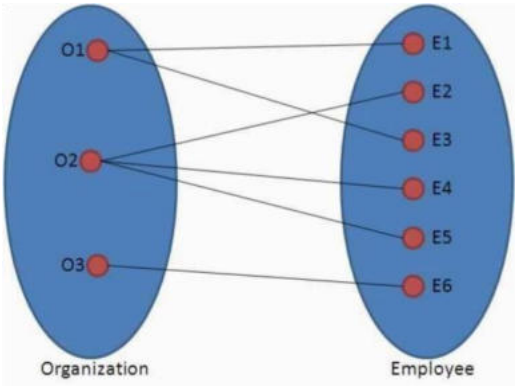


Oneemployeeisassignedwithonlyoneparkingspaceandoneparkingspace isassignedtoonlyoneemployee.Henceitisa1:1relationshipandcardinality is One-To-One (1:1) InERmodeling,thiscanbementionedusingnotationsasgivenbelow



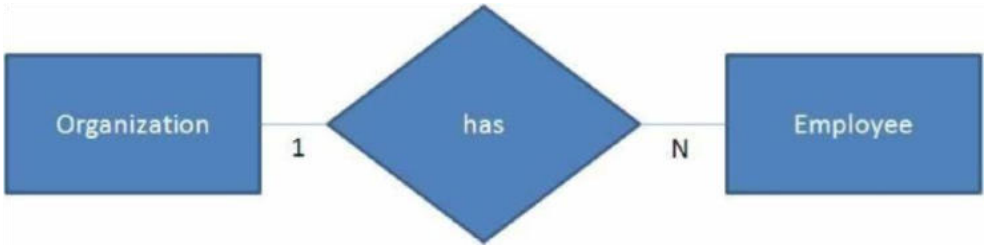
ExampleforCardinality–One-to-Many(1:N)

Organizationhasemployees



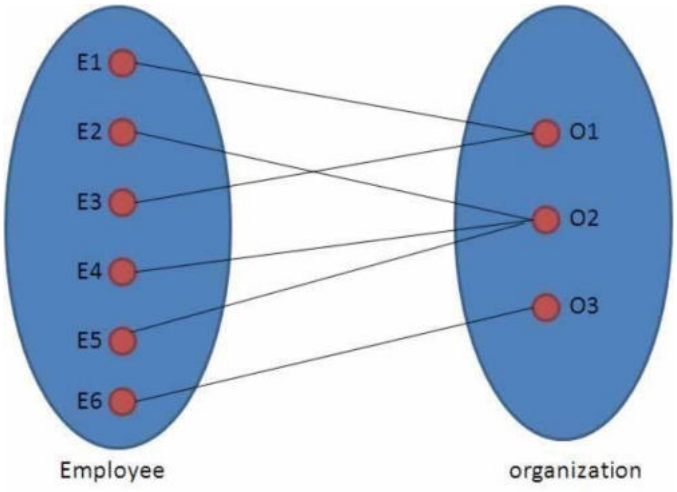
Oneorganizationcanhavemanyemployees,butoneemployeeworksinonly one organization. Hence it is a 1:N relationship and cardinality is One-To- Many (1:N)

InERmodeling,thiscanbementionedusingnotationsasgivenbelow



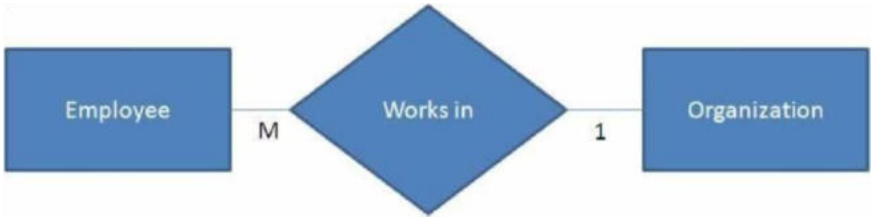
ExampleforCardinality–Many-to-One(M:1)

ItisthereverseoftheOnetoManyrelationship.employeeworksin organization



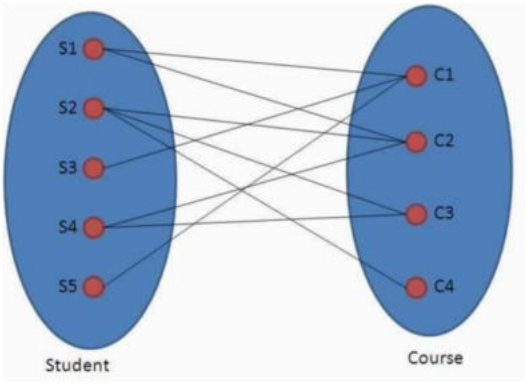
OneemployeeworksinonlyoneorganizationButoneorganizationcanhave many employees. Hence it is a M:1 relationship and cardinality is Many-to- One (M :1)

InERmodeling,thiscanbementionedusingnotationsasgivenbelow.

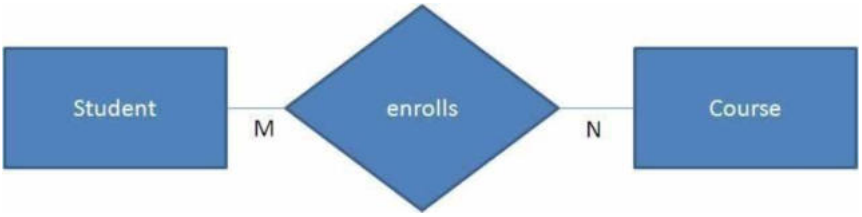


Cardinality–Many-to-Many(M:N)

Studentsenrollsforcourses



One student can enroll for many courses and one course can be enrolled by many students. Hence it is a M:N relationship and cardinality is Many-to- Many (M:N) InERmodeling,thiscanbementionedusingnotationsasgivenbelow



RelationshipParticipation

1.Total

Intotalparticipation,everyentityinstancewillbeconnectedthroughthe relationship to another instance of the other participating entity types

2.Partial

Exampleforrelationshipparticipation

Considertherelationship-Employeeisheadofthedepartment.

Hereallemployeeswillnotbetheheadofthedepartment.Onlyoneemployee will be the head of the department. In other words, only few instances of employee entity participate in theabove relationship. So employee entity's participation is partial in the said relationship.

However each department will be headed by some employee. So department entity's participation is total in the said relationship.

Advantages and Disadvantages of ER Modeling (Merits and

Demerits of ER Modeling) Advantages

1. ERModelingissimpleandeasilyunderstandable.Itisrepresentedin businessuserslanguageanditcanbeunderstoodbynon-technicalspecialist.
2. IntuitiveandhelpsinPhysicalDatabasecreation.
3. Canbegeneralizedandspecializedbasedonneeds.
4. Canhelpindatabasedesign.
5. Givesahigherleveldescriptionofthesystem.

Disadvantages

1. Physical design derived from E-R Model may have some amount ofambiguitiesorinconsistency.2.Sometimediagramsmayleadto misinterpretations

ADVANCEDFEATURESOFERMODEL:

Beyondbasicentities,attributes,andrelationships,ERmodelsgainpowerwithEnhancedER (EER) features, including Specialization/Generalization, Aggregation, Inheritance, WeakEntities,andUnionTypes(Categories),allowingformorecomplexdatamodeling, better abstraction, hierarchycreation, and handling of real-worldcomplexities like IS-A relationships (e.g., "Employee ISA Person") and treating relationships as entities.

KeyAdditionalFeatures(EERModel)

1. **Specialization&Generalization**:

- **Specialization**:Creatingsubclasses(e.g.,Student,Professor)fromasuperclass(e.g.,Person) basedonspecificcharacteristics.
- **Generalization**: The reverse, abstracting common attributes to form a superclass from multiple subclasses.
- **AttributeInheritance**: Subclasses automaticallyinherit attributes fromtheirsuperclass.

2. **Aggregation**:

- Treatsarelationshipsetasanabstractentity,allowingrelationshipstorelationships(e.g., a "Sponsor" relationship between "Employee" and "Project" might have an "Amount" attribute, forming an aggregate entity).

3. **WeakEntities**:

- Entitiesthatdependonastrongentityfortheirexistence(e.g., DependentreliesonEmployee) anduseapartialkey.

4. **UnionTypes(Categories)**:

- Model entities that can belong to one of several different entity sets (e.g., a Registered_User could be a Customer OR an Admin).

5. **EnhancedConstraints**:

- More detailed rules for generalization/specialization, like Disjointness (a subclass can only belong to one superclass) and Overlap (a subclass can belong to multiple superclasses). Theseextensionsprovidearicher,moreexpressivewaytodesigndatabasesthat accuratelyreflectcomplexreal-worlddatastructures,movingbeyondbasicERmodels.

THE INTERNET SHOP

AnERDiagram(Entity-RelationshipDiagram)foranonlineshopmodelskeycomponents like **Customers**, **Products**, **Orders**, and **Payments**, showing how they connect: Customers place Orders,OrderscontainProducts(viaanOrder_Item/Cartlink),andOrdersarelinkedtoPayments,with attributes like customer details, product descriptions, order dates, and payment methods defining each entity and relationship, forming the database blueprint.

KeyEntities&Attributes

- **Customer**: `customer_id` (PK),name,email,address, password.
- **Product**: `product_id` (PK),name,description,price,stock_quantity,image_url.
- **Order**: `order_id` (PK), `customer_id` (FK),order_date,total_amount,status, shipping_address.
- **Payment**: `payment_id` `order_id` (PK), (FK),amount,payment_method,transaction_id, payment_date.
- **Cart/Cart_Item**:Representsitemsacustomerintendstobuy(oftenatemporarylinkbetweenCustomer and Product).

KeyRelationships(Crow'sFootNotationExample)

- **CustomerPlaces Order**:OneCustomercan placeManyOrders (1:M).
- **Order Contains Products**: One Order contains Many Products (M:N, linked through an `Order_Item` entity).
- **ProductBelongs to Category**: OneCategoryhasManyProducts (1:M).
- **OrderLinkedto Payment**:OneOrder hasOnePayment (1:1or1:M ifmultiplepayment attempts).

HowItWorksintheDiagram

1. **Customers**registerandlogin.
2. Theybrowse**Products**,whichhavedetailsand categories.
3. Productsareadded toa**Cart**.
4. Thecustomerplacesan**Order**,linkingthemtotheitemsintheir cart.
5. The**Order**isprocessed,a**Payment**ismade,andshippingdetailsarecaptured.

This ER Diagram acts as a blueprint for the database, ensuring efficient storage and retrieval of ecommerce data.

RelationalModel

The relational model is today the primary data model for commercial data processing applications.Itattaineditsprimarypositionbecauseofitssimplicity,whicheasesthejob of the programmer, compared to earlier data models such as the network model or the hierarchical model. In this, we first study the fundamentals of the relational model. A substantial theory exists for relational databases.

StructureofRelational Databases:

Arelationaldatabaseconsistsofacollectionof**tables**,eachofwhichisassignedaunique name.Forexample,considerthe*instructor*tableofFigure:1.5,whichstoresinformation about instructors. The table has four column headers: *ID*, *name*, *dept name*, and *salary*. Each row of this table records information about an instructor, consisting of the instructor's *ID*, *name*, *dept name*, and *salary*. Similarly, the *course* table of Figure 1.6 stores informationabout courses,consisting ofa *courseid*,*title*,*deptname*,and*credits*, for each course. Note that each instructor is identified by the value of the column *ID*, while each course is identified by the value of the column *course id*.

Figure 1.7 shows a third table, *prereq*, which stores the prerequisite courses for each course. Thetablehas two columns, *courseid* and *prereq id*. Each row consists of apair of course identifiers such that the second course is a prerequisite for the first course.

Thus,arowinthe*prereq*tableindicatesthattwocoursesare*related*inthesensethatone course is a prerequisite for the other. As another example, we consider the table *instructor*, a row in the table can be thought of as representing the relationship between a specified *ID* and the corresponding values for *name*,*dept name*, and *salary* values.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

*Figure1.5:The*instructor*relation (2.1)*

Ingeneral,arowinatablerepresentsa*relationship* amongasetofvalues.Sinceatable isacollectionofsuchrelationships,thereisaclosecorrespondencebetweentheconcept of *table* and the mathematical concept of *relation*, fromwhich the relational data model takes its name. In mathematical terminology, a *tuple* is simply a sequence (or list) of values. A relationship between *n* values is represented mathematically by an *n-tuple* of values, i.e., a tuple with *n* values, which corresponds to a row in a table.

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure:1.6:The course*relation(2.2)*

course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

*Figure:1.7:The*prereq *relation. (2.3)*

Thus,intherelationalmodeltheterm**relation** isused toreferto atable, whiletheterm **tuple**isusedtorefertoarow.Similarly,theterm**attribute**referstoacolumnofatable.
ExaminingFigure1.5,wecanseethattherelation *instructor*hasfourattributes:

***ID*,*name*,*dept name*,and*salary*.**

We use the term **relation instance** to refer to a specific instance of a relation, i.e., containing a specific set of rows. The instance of *instructor* shown in Figure 1.5 has 12 tuples, corresponding to 12 instructors.

In this topic, we shall be using a number of different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a university.Theydonotincludeallthedataanactualuniversitydatabasewouldcontain, in order to simplify our presentation.

The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 1.5, or are unsorted, as in Figure1.8, does not matter; the relations in the two figures arethe same, since both contain the same set of tuples. For ease of exposition, we will mostly show the relations sorted by their first attribute. For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute.
Thus, the domain of the *salary*attributeofthe*instructor*relationisthesetofallpossiblesalaryvalues,whilethe domain of the *name* attribute is the set of all possible instructor names.

Werequirethat,forallrelations*r*,thedomainsofallattributesof*r*beatomic.Adomain is **atomic** if elements of the domain are considered to be indivisible units.

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

*Figure:1.8:Unsorteddisplay ofthe*instructor*relation.* ***(2-4)***

Forexample,supposethetable*instructor*hadanattribute*phonenumber*,whichcanstore a set of phone numbers corresponding to the instructor. Then the domain of *phone number* wouldnotbeatomic,sinceanelementofthedomainisasetofphonenumbers, and it has subparts, namely the individual phone numbers in the set.
The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the *phone number* attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code and a local number, we would be treating it as a nonatomic value.Ifwetreateachphonenumberasasingleindivisibleunit,thentheattribute*phone number* would have an atomic domain.

The**null**valueisaspecialvaluethatsignifiesthatthevalueisunknownordoesnotexist. For example, suppose as before that we include the attribute *phone number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or thatthetelephonenumberisunlisted.Wewouldthenhavetousethenullvaluetosignify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus should be eliminated if at all possible. We shall assume null values are absent initially.

DatabaseSchema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given instant in time. The concept of a relation corresponds to the programminglanguage notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

In general, a relation schema consists of a list of attributes and their corresponding domains. The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time;

dept_name	building	budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

*Figure1.9:The*department *relation.**(2-5)***

similarly the contents of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change. Although it is important to know the difference between a relation schema and a relation instance, we oftenusethesamename,suchas*instructor*,torefertoboththeschemaandtheinstance. Where required, we explicitly refer to the schema or to the instance, for example "the *instructor*schema,"or"aninstanceofthe*instructor*relation."However,whereitisclear whether we mean the schema or the instance, we simply use the relation name.
Considerthe*department*relationofFigure1.9.Theschemaforthatrelationis

***department*(*deptname*,*building*,*budget*)**

Note that the attribute *dept name* appears in both the *instructor* schema and the *department* schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations.

Forexample,supposewewishtofindtheinformationaboutalltheinstructorswhowork intheWatsonbuilding.Welookfirstatthe*department*relationtofindthe*deptname*of all the departments housed in Watson. Then, for each such department, we look in the *instructor* relation to find the information about the instructor associated with the corresponding *dept name*.

Let us continue with our university database example. Each course in a university may beofferedmultipletimes,acrossdifferentsemesters,orevenwithinasemester.Weneed a relation to describe each individual offering, or section, of the class. The schema is ***section* (*course id*, *sec id*, *semester*, *year*, *building*, *room number*, *time slot id*)**

Figure 1.10 shows a sample instance of the *section* relation. We need a relation to describe the association between instructors and the class sections that they teach. The relation schema to describe this association is ***teaches* (*ID*, *course id*, *sec id*, *semester*, *year*)**

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

*Figure1.10:The*section*relation.(2-6)*

Figure 1.11 shows a sample instance of the *teaches* relation. As you can imagine, there aremanymorerelationsmaintainedinarealuniversitydatabase.Inadditiontothose relationswehavelistedalready,*instructor*,*department*,*course*,*section*,*prereq*,and *teaches*,weusethefollowingrelationsinthistext:

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

*Figure:1.11:The*teaches*relation.(2-7)*

- *student*(*ID*,*name*,*deptname*,*totcred*)
- *advisor*(*sid*,*iid*)
- *takes*(*ID*,*courseid*,*secid*,*semester*,*year*,*grade*)
- *classroom*(*building*,*roomnumber*,*capacity*)
- *timeslot*(*timeslotid*,*day*,*starttime*,*endtime*)

Keys

Wemusthaveawaytospecifyhowtupleswithinagivenrelationaredistinguished.This is expressed in terms of their attributes. That is, the values of the attribute values of a

tuplemustbesuchthattheycan*uniquelyidentify*thetuple.Inotherwords,notwotuples in a relation are allowed to have exactly the same value for all attributes.

A**superkey**isasetofoneormoreattributesthat,takencollectively,allowustoidentify uniquelyatupleintherelation.Forexample,the *ID*attributeoftherelation*instructor*is sufficient to distinguish one instructor tuple from another. Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name. Formally, let *R* denote the set of attributes in the schemaofrelation*r*.Ifwesaythatasubset*K*of*R*isa*superkey*for*r*,wearerestricting consideration to instances of relations *r* in which no two distinct tuples have the same values on all attributes in *K*. That is, if t_1 and t_2 are in *r* and $t_1 = t_2$, then $t_1.K = t_2.K$.

A superkey may contain extraneous attributes. For example, the combination of *ID* and *name*is asuperkeyfor the relation *instructor*. If *K*is a superkey, then so is anysuperset of*K*.Weareofteninterestedinsuperkeysforwhichnopropersubsetisasuperkey.Such minimal superkeys are called **candidate keys**.

Itispossiblethatseveraldistinctsetsofattributescouldserveasacandidatekey.Suppose that a combination of *name* and *dept name* is sufficient to distinguish among members of the *instructor* relation. Then, both *{ID}* and *{name*, *dept name}* are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, *{ID*, *name}*, does not form a candidate key, since the attribute *ID* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. A key (whether primary, candidate, or super) is a property of the entire relation, rather than of theindividualtuples.Anytwoindividualtuplesintherelationareprohibitedfromhaving thesamevalueonthekeyattributesatthesametime.Thedesignationofakeyrepresents a constraint in the real-world enterprise being modeled.

Primarykeys must bechosen with care. Aswe noted,thename of a person is obviously not sufficient, because there may be many people with the same name. In the United States, the social-security number attribute of a person would be a candidate key. Since non-U.S.residentsusuallydonothavesocial-securitynumbers,internationalenterprises must generate their own unique identifiers.

Analternativeistousesomeuniquecombinationofotherattributesasakey.Theprimary keyshouldbechosensuchthatitsattributevaluesarenever,orveryrarely,changed.For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social-security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

It is customary to list the primary key attributes of a relation schema before the other attributes;forexample,the*deptname*attributeof*department*islistedfirst,sinceitisthe primary key. Primary key attributes are also underlined. A relation, say r_1, may include among its attributes the primarykeyof another relation, say r_2. This attribute is called a **foreign key** from r_1, referencing r_2.

Therelation$r_1$isalsocalledthe**referencingrelation**oftheforeignkeydependency,and r_2 is called the **referenced relation** of the foreign key. For example, the attribute *dept name* in *instructor* is a foreign key from *instructor*, referencing *department*, since *dept name*istheprimarykeyof*department*.Inanydatabaseinstance,givenanytuple,sayt_a, fromthe *instructor* relation, there must be some tuple, say t_b, in the *department* relation suchthatthevalueofthe*deptname*attributeof$t_a$isthesameasthevalueoftheprimary key, *dept name*, of t_b.

Nowconsider the *section* and *teaches* relations. It would be reasonable to require thatif asectionexistsforacourse,itmustbetaughtbyatleastoneinstructor;however,itcould possibly be taught by more than one instructor. To enforce this constraint, we would require that if a particular (*course id*, *sec id*, *semester*, *year*) combination appears in *section*, then the same combination must appear in *teaches*. However, this set of values does not form a primary key for *teaches*, since more than one instructor may teach one such section. As a result, we cannot declare a foreign key constraint from *section* to *teaches* (although we can define a foreign key constraint in the other direction, from *teaches* to *section*).

The constraint from *section* to *teaches* is an example of a **referential integrity constraint**; a referential integrity constraint requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

SchemaDiagrams

A database schema, along with primary key and foreign key dependencies, can be depictedby**schemadiagrams**.Figure1.12showstheschemadiagramforouruniversity organization. Each relation appears as a box, with the relation name at the top in blue, and the attributes listed inside the box. Primary key attributes are shown underlined. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

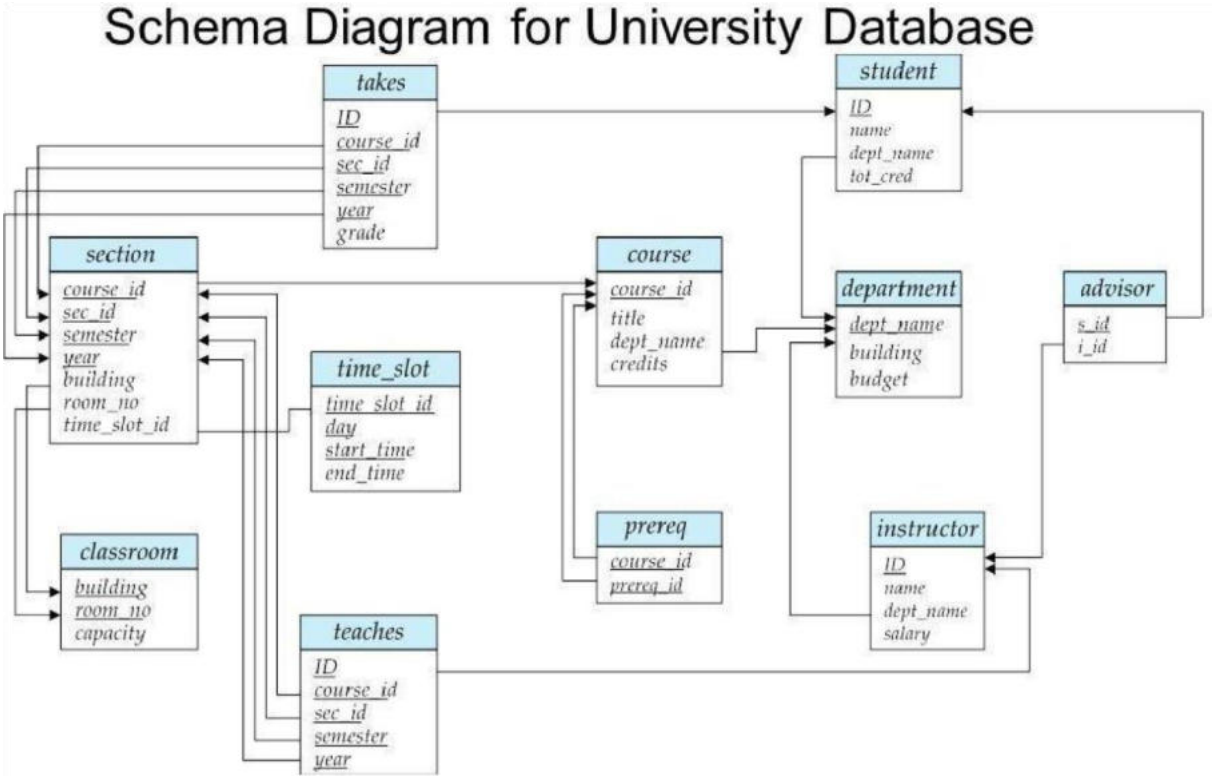


Figure1.12:Schemadiagramfortheuniversitydatabase.

Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams. We will study a different diagrammatic representation called the entity-relationship diagram.

UNITIII:SOLandPL/SOL

SQLANDPL/SQL:IntroductiontoSQL,DataDefinition Commands, Data Manipulation Commands, Basic SQl Querying(Select and project) Select Queries, **Virtual Tables**: Creating View, Altering View, Updating View, Destroying View, Relational Set Operators, aL loin Operators, Sub Queries and Correlated âueries, implementation of different integrity constraints, Aggregate Functions, Procedural SQl: Stored Procedures, Stored Functions, Triggers, Cursors

DATA DEFINITION COMMANDS, DATA MANIPULATION COMMANDS, BASIC SQL QUERYING(SELECT AND PROJECT) SELECT QUERIES

DDL (Data Definition Language) commands are used to define or modify the database structure, while DML (Data Manipulation Language) commands are used to manage the data within those structures.

DDLCommands(DataDefinitionLanguage)

DDLcommandsaffecttheschemaorstructureofthedatabaseand are autocommitted, meaning the changes are permanent and cannot be rolled back.

- **CREATE**:Usedtocreatenewdatabaseobjects,suchastables, indexes,views,orthedatabaseitself.
  - *Example*:

```
CREATETABLEStudents(StudentIDint,Namevarchar(255));
```

- **ALTER** :Usedtomodifythestructureofanexistingdatabase object, like adding, deleting, or modifying columns in a table.
  - *Example*:

```
ALTERTABLEStudentsADDEmailvarchar(255);
```

- :**DROP** Usedtopermanentlydeleteadatabaseobject(e.g.,atable, index, or database) and all the data within it.
  - *Example*:

```
DROPTABLEStudents;
```

- **TRUNCATE** :Usedtoquicklydeleteallrecordsfromatablebut keepsthetablestructureintactforfutureuse.ItisaDDLcommand because,likeDROP,itsoperationsarenon-transactionalandcannot berolledback.
  - *Example*: `TRUNCATETABLEStudents;`
- **RENAME**:Usedtochangethenameofadatabaseobject.
  - *Example*: `RENAMETABLEStudentsTOStudent_Data;`

DMLCommands(DataManipulationLanguage)

DML commands are used to manage the data stored within the database objects. These commands are transactional, meaning changes can be rolled back (undone) if necessary before being committed permanently.

- **INSERT**:Usedtoaddnewrows(records)ofdataintoa table.
  - *Example*: `INSERTINTOStudents(StudentID,Name)VALUES(1,'JohnDoe');`
- **UPDATE** :Usedtomodifyexistingdatawithinatable.
  - *Example*: `UPDATEStudentsSETName='JaneDoe'WHEREStudentID=1;`
- **DELETE** :Usedtoremoveexistingrecords(rows)fromatable.
  - *Example*: `DELETEFROMStudentsWHEREStudentID=1;`

SELECT

`SELECT`

• :WhileoftencategorizedseparatelyasDataQuery Language (DQL), iswidelyconsideredpartofDMLasit interactswiththedatawithin thetables. ○ *Example*:

```
SELECT*FROMStudents;
```

IntroductiontoPL/SQL

PL/SQL (Procedural Language/Structured Query Language) is an extension of SQL (Structured Query Language) designed by

Oracle.Itenablesdeveloperstowritecodethatcombinesthepower of SQL queries with the procedural capabilities of a programming

language.PL/SQLismainlyusedtointeractwithOracledatabases, providingawaytoexecuteSQLstatementsinaproceduralmanner, whichallowsformorecontroloverdataprocessing,includinglogic flow and error handling.

StructureofPL/SQLinDBMS

PL/SQLprogramscanbestructuredintodifferentblocks.Thebasic structure of a PL/SQL block is as follows:

1. DeclarationSection(optional):
This section is used to declare variables, constants, cursors, and exceptions that will be used in the PL/SQL block.

```
DECLARE
    variable_namedatatype;
    CONSTANT_NAMECONSTANTdatatype:=value; BEGIN
```

2. ExecutionSection(mandatory):

ThisisthesectionwheretheactualSQLstatementsandprocedural code are written. It is the core of the PL/SQL block and is mandatory.SQLcommandslikeSELECT,INSERT,UPDATE,or

DELETEareexecutedhere,alongwithprocedurallogiclikeloops and conditions.

Syntax:

```
BEGIN
   --SQLandPL/SQLcode END;
```

3. ExceptionHandlingSection(optional):

- This section is used to handle runtime errors or exceptions thatoccurduringtheexecutionofthePL/SQLblock.It allows programmersto gracefully handle errors instead of having the program crash or stop.

Syntax:

```
EXCEPTION
   WHENexception_nameTHEN
      -- Handle error
   WHENOTHERSTHEN
      --Handleallothererrors END;
```

ExampleofaSimplePL/SQLBlock:

```
DECLARE v_emp_nameVARCHAR2(50);
    v_emp_salaryNUMBER(10,2);
BEGIN
    SELECT employee_name, salary INTO v_emp_name, v_emp_salary
    FROM employees
    WHEREemployee_id=100;

    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_emp_name);
    DBMS_OUTPUT.PUT_LINE('Salary:'||v_emp_salary); EXCEPTION
    WHENNO_DATA_FOUNDTHEN
        DBMS_OUTPUT.PUT_LINE('No employee found with the given ID.');
    WHENOTHERSTHEN
                DBMS_OUTPUT.PUT_LINE('An unexpected error
occurred.');
END;
```

Topic1:

Decisionmakingstatements Or

Conditionalstatements Or Controlstatements:

Decision making or conditional or Control statements are those statements which are in charge of executing a statement out of multiple given statements based on some condition. The condition willreturneithertrueorfalse.Basedonwhattheconditionreturns, the associated statement is executed.

ConditionalStatementsavailableinPL/SQLaredefinedbelow:

1. **IFTHEN**
2. **IFTHENELSE**
3. **NESTED-IF-THEN**
4. **IFTHENELSIF-THEN-ELSE**

1. IFTHEN if then the statement is the most simple decision-making

statement.Itisusedtodecide whether acertainstatement orblock of statementswill be executedor not i.e ifa certaincondition is true then a block of statement is executed otherwise not.

Syntax:

IFconditionTHEN

-- Code to execute if the condition is true END IF; **Example:**

```
DECLARE
    v_salaryNUMBER := 3000; BEGIN
    IF v_salary > 2500 THEN DBMS_OUTPUT.PUT_LINE('Salary is above
        2500');
    ENDIF; END;
```

2. IFTHENELSE

The if statement alone tells us that if a condition is true it will execute ablockof statements and ifthe condition is false it won't. But what if we want to do something else if the condition is false. Here comestheelsestatement.

Wecanusetheelsestatement with ifstatement toexecute ablockofcodewhenthecondition isfalse. **Syntax:**

```
IFconditionTHEN
    --Codetoexecuteiftheconditionistrue ELSE
    --Codetoexecuteiftheconditionisfalse END IF;
```

Example:

```
DECLARE v_ageNUMBER:=20;
    BEGIN
    IF v_age >= 18 THEN DBMS_OUTPUT.PUT_LINE('Adult');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Minor'); END
    IF;
END;
```

3. NESTED-IF-THEN

A **nested IF-THEN** statement occurs when an IF-THEN statement is placed inside another IF-THEN statement.

Syntax:

```
IF condition1 THEN
   --Code to execute if condition1 is true IF condition2
   THEN

      --Code to execute if condition2 is true END IF;
END IF;
```

Here, the second IF statement (the nested one) will only be evaluated if the first IF statement is true.

Example:

```
declare
num1 number:=10;
num2 number:=20;
num3 number:=20;


begin if num1 < num2 then
dbms_output.put_line('num1 small num
2');      if      num1      <      num3      then
  dbms_output.put_line('num1 small num3 also');
  end if;
```

```
end if;

dbms_output.put_line('afterendif')
; end;
```

4. if...then...elsif...elsestatement

It is used to check multiple conditions. Sometimes it is required to test more than one condition in that case if...then...else statement cannot be used. For this purpose, if...then...elsif...else statement is suitable in which all the conditions are tested one by one and whichever condition is found to be

TRUE, that block of code is executed.Andifalltheconditionsresultin**FALSE**thenthe else part is executed.

Syntax:

```
if<test_condition1>then body of action

elsif<test_condition2>then body of
 action elsif<test_condition3>then
            body of action

...
...
...
else
        bodyofaction
```

endif;

setserveroutputon;

Example:

```
DECLARE
        a int;
        bint;
BEGIN
        a := &a; b := &b; if(a>b)then
        dbms_output.put_line('aisgreaterthanb'); elsif(b>a)
        then
                dbms_output.put_line('bisgreaterthana');
        else dbms_output.put_line('Bothaandbareequal');
        endif;
END;
```

Topic2:

PL/SQLLoops:

LoopsStatements(or)IterativeStatements

Loops in PL/SQL provides a way of repeating a particular part of any program or any code statement as many times as required. In PL/SQLwehavethreedifferentloopoptionstochoosefromwhen we want to execute a statement repeatedly in our code block.

Theyare:

1. BasicLoop
2. While Loop
3. For Loop

1. BasicLoop

Basicloop or simpleloop ispreferred in PL/SQLcode whenthere is no surety about how many times the block of code is to be repeated. When we use the basic loop the code block will be executed at least once.

Whileusingit,followingtwothingsmustbeconsidered:

- SimpleloopalwaysbeginswiththekeywordLOOPandends with a keyword END LOOP.
- Abasic/simpleloopcanbeterminatedatanygivenpointby usingtheexitstatementorbyspecifyingcertainconditionby using the statement exit when.

Syntax:

```
LOOP
  --Statementstobeexecuted repeatedly
  EXITWHENcondition;--Optionalexitcondition END LOOP;
```

Example:

```
DECLARE counterNUMBER:=1;
  BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Counter:'||counter); counter := counter
    + 1;
    EXITWHENcounter>5; END
  LOOP;
END;
/
```

2. WhileLoop

Itisanentrycontrolledloopwhichmeansthatbeforeenteringina whileloopfirsttheconditionistested,iftheconditionis**TRUE**the statementoragroupofstatementsgetexecutedandifthecondition is **FALSE** the control will move out of the while loop.

Syntax:

```
WHILE<test_condition>LOOP
        <action>E
ND LOOP;
```

Example:

```
setserveroutputon; DECLARE numint:=1;
BEGIN
        while(num<=10)LOOP
            dbms_output.put_lin
        e('||no); num

    := num+2;
    ENDLOOP;
END;
```

3. For Loop

ThisloopisusedwhensomestatementsinPL/SQLcodeblockare to be repeated for a fixed number of times.

When we use the for loop we are supposed to define a counter variable which decides how many time the loop will be executed based on a starting and ending value provided at the beginning of the loop. The for loop automatically increments the value of the counter variable by 1 at the end of each loop cycle.

Syntax:

```
FORcounter_variableINstart_value..end_valueLOOP statement to be executed
ENDLOOP;
```

Example:

```
setserveroutputon; DECLARE inumber(2);
BEGIN
```

```
    FORiIN1..10 LOOP

    dbms_output.put_line(i); END
        LOOP;

END;
```

Topic3:

Triggersin PL/SQL

A **trigger** in PL/SQL is a type of stored procedure that automatically executes (or "fires") in response to a specific event occurring in the database. Triggers are used to enforce business rules, validate data, maintain audit trails, or automatically modify data in a table based on changes made to that table.

Triggers are defined to execute either **before** or **after** a specific event on a database object such as an **INSERT**, **UPDATE**, or

DELETEoperation.Theevent canbeachangein atableorview, and triggers are bound to that table or view.

TypesofTriggers

1. BEFORETrigger
2. AFTERTrigger
3. INSTEADOFTrigger
4. Level Triggers

Thereare2differenttypesofleveltriggers,theyare:

1. ROWLEVEL TRIGGERS

2. STATEMENTLEVELTRIGGERS

TriggerSyntax:

CREATE[ORREPLACE]TRIGGERtrigger_name

{BEFORE|AFTER|INSTEADOF}

{INSERT|UPDATE|DELETE}

ON table_name

[FOREACHROW]

DECLARE

--Variabledeclarations(optional) BEGIN

--Triggercode END;

1. BEFORETrigger:Thistriggerfiresbeforethetriggeringevent (INSERT, UPDATE, DELETE) is executed on the table.

Syntax:

CREATEORREPLACETRIGGERtrigger_name BEFORE

{INSERT | UPDATE | DELETE}

ON table_name

[FOREACHROW]

DECLARE

--Variabledeclarations(optional) BEGIN

--Triggercode END; **Example:**

CREATEORREPLACETRIGGERbefore_employee_insert BEFORE INSERT

ON employees

FOREACHROW

```
BEGIN
    --Setthehire_dateto thecurrentsystem datebeforethenew record is inserted
   :NEW.hire_date:=SYSDATE; END;
```

2. AFTER trigger in PL/SQL is a type of trigger that executes **after** a specified event (such as INSERT, UPDATE, or DELETE) has been performed on a table. This trigger is typically used for actionsthatneedtooccur afterthedatahasbeenmodified,suchas logging changes, auditing data, or updating related tables.

Syntax:

```
CREATEORREPLACETRIGGERtrigger_name
AFTER{INSERT|UPDATE|DELETE}
ON table_name
[FOREACHROW]
DECLARE
   --Variabledeclarations(optional) BEGIN
   --Triggercode END;
```

Example:

```
CREATEORREPLACETRIGGERafter_employee_insert AFTER
INSERT ON employees
FOREACHROW
BEGIN
   --Insertarecordintotheaudit_logtableafteranewemployeeis inserted
   INSERTINTOaudit_log(log_id,action_type,employee_id, action_date)
```

VALUES (audit_log_seq.NEXTVAL, 'INSERT',
:NEW.employee_id,SYSDATE); END;

3. INSTEADOFTriggerin PL/SQL

An**INSTEADOFtrigger**inPL/SQLisusedtodefinethebehavior that should occur **instead of** the usual DML (Data Manipulation

Language)operationsuchasINSERT,UPDATE,orDELETE.Thistype oftriggeristypicallyusedon**views**ratherthantables.Thepurpose of an INSTEAD OF trigger is to replace the default action with a custom action when an operation is performed on a view.

Syntax:

```
CREATEORREPLACETRIGGERtrigger_name INSTEAD OF
{INSERT | UPDATE | DELETE}
ON view_name
[FOREACHROW]
DECLARE
  --Variabledeclarations(optional) BEGIN
  --Triggercode END;
```

Example:

```
CREATE OR REPLACE TRIGGER update_employee_department
INSTEADOFUPDATEONemployee_department_view FOR EACH
ROW
BEGIN
```

```
    --Updatetheemployeestable
    UPDATE employees
    SETemployee_name=:NEW.employee_name WHERE employee_id
    = :OLD.employee_id;


    --Updatethedepartmentstable
    UPDATE departments
    SETdepartment_name=:NEW.department_name WHERE department_id
    = :OLD.department_id;
END;
```

4. LevelTriggersinPL/SQL

Triggers are special procedures in PL/SQL that automatically execute in response to certain events on a particular table or view. Basedonthescopeoftheiroperation,triggersarecategorizedinto **Row-Level Triggers** and **Statement-Level Triggers**.

1. Row-LevelTriggers

- A **Row-Level Trigger** is executed once for each row affected by a DML (INSERT, UPDATE, DELETE) statement.
- Theyareusedwhenyouneedtoworkwithortrackchanges to specific rows in a table.

ExampleofaRow-Level Trigger:

```
CREATEORREPLACETRIGGERrow_level_trigger_example AFTER INSERT
OR UPDATE ON employees
FOREACHROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Row-Level Trigger fired for
Employee ID: ' || :NEW.employee_id);
END;
/
```

2. Statement-LevelTriggers

- A **Statement-Level Trigger** is executed once for the entire DML statement, regardless of the number of rows affected.
- These triggers are used when the operation affects the table as a whole, not specific rows.

ExampleofaStatement-LevelTrigger:

```
CREATE OR REPLACE TRIGGER statement_level_trigger_example
AFTERDELETEONemployees
BEGIN
        DBMS_OUTPUT.PUT_LINE('Statement-Level Trigger fired after a
DELETE operation on employees.');
END;
/
```

Topic4:

Cursorin PL/SQL

A **cursor** in PL/SQL is a pointer to a context area that stores the result set of a query. It allows row-by-row processing of query results.

CursorAttributes

PL/SQLprovidescursorattributesforobtaininginformationabout the cursor state:

- **%FOUND**: Returns TRUE if a row was fetched, otherwise FALSE.
- **%NOTFOUND**: Returns TRUE if no row was fetched, otherwise FALSE.
- **%ISOPEN**: Returns TRUE if the cursor is open, otherwise FALSE.
- **%ROWCOUNT**: Returns the number of rows fetched so far.

PL/SQLprovidestwotypesofcursors:

1. **ImplicitCursors**
2. **ExplicitCursors**

1. ImplicitCursor

- Automatically created by PL/SQL for SELECT INTO, INSERT, UPDATE, and DELETE statements that affect a single row or multiple rows.
- Youdon'tneedtodeclareorcontrol it.

ExampleofImplicitCursor:

```
DECLARE emp_countNUMBER;
   BEGIN

   SELECTCOUNT(*)INTOemp_countFROMemployees;
   DBMS_OUTPUT.PUT_LINE('Total Employees: ' || emp_count);
```

```
END;
/
```

2. ExplicitCursor

- Definedandcontrolledexplicitlybytheprogrammer.
- Usefulforqueriesthatreturnmultiplerows.
- Consistsoffourmainsteps:

1. **Declaration**:Declarethecursor.
2. **Opening**: Open the cursor to execute the query andestablish the result set.
3. **Fetching**:Retrieverowsfrom theresultsetoneby one.
4. **Closing**:Closethecursortorelease resources.

ExampleofExplicitCursor:

```
DECLARE
   CURSOREmp_cursorIS
      SELECTemployee_id,nameFROMemployees;  v_emp_id
   employees.employee_id%TYPE;                    v_name
   employees.name%TYPE;

BEGIN
   -- Open the cursor
   OPENemp_cursor;
   --Loopthroughrows LOOP
      FETCHemp_cursorINTOv_emp_id,v_name;
```

```
        --Exittheloopwhennomorerowsareavailable EXIT
        WHEN emp_cursor%NOTFOUND;
        -- Process the fetched data DBMS_OUTPUT.PUT_LINE('Employee ID: ' ||
        v_emp_id ||
',Name:'||v_name); END
    LOOP;
    -- Close the cursor
    CLOSEemp_cursor;
END;
/
```

ExceptionHandlinginPL/SQL

Anexceptionisanerror.Anerroraccruedduringtheprogram execution is called exception.

TypesofExceptions

1. **Pre-defined or System Exceptions**: These are built-in exceptions in PL/SQL, such as NO_DATA_FOUND, TOO_MANY_ROWS, and ZERO_DIVIDE.

2. **User-Defined Exceptions**: These are explicitly defined by the programmer to handle specific business logic errors. **SyntaxforExceptionHandling**

Syntax:

```
DECLARE
   <exception_name>EXCEPTION; BEGIN
   ...
   RAISE<exception_name>;
   ...
END;
```

***1.*System-definedExceptions**

System-defined exceptions are predefined in programming languages to handle runtime errors.

```
DECLARE
    --Declarationstatements; BEGIN
    --SQL statements;
    --Proceduralstatements;
EXCEPTION
     --Exceptionhandlingstatements;
END;
```

Exception	**Raisedwhen....**

LOGIN_DENIED	Atthetimewhenuserloginisdenied.
TOO_MANY_ROWS	When a select query returns more than one thedestinationvariablecantakeonlysingle
VALUE_ERROR	Whenanarithmetic,valueconversion,trunc constraint error occurs.

Example:

```
setserveroutputon; DECLARE
        a int;
        bint;
        cint;
BEGIN
        a := &a;
        b:= &b; c
        := a/b;
        dbms_output.put_line('RESULT='||c);
EXCEPTION
        whenZERO_DIVIDEthen
                dbms_output.put_line('Divisionby0isnotpossible');
```

END;

User-definedException

In any program, there is a possibility that a number of errors can occur that may not be considered as exceptions by oracle. In that

case,anexceptioncanbedefinedbytheprogrammerwhilewriting thecodesuchtypeofexceptionsarecalledUser-definedexception. User defined exceptions are in general defined to handle special caseswhereourcodecangenerateexceptionduetoourcodelogic. Also, in your code logic, you can explicitly specify to genrate an exceptionusingtheRAISEkeywordandthenhandleitusingthe EXCEPTION block. **Syntax:**

```
DECLARE
        <exceptionname>EXCEPTION
BEGIN
        <sqlsentence>
        If <test_condition>
                        THENRAISE<exception_name
                >;
        END IF;
        EXCEPTION
                        WHEN<exception_name>THEN
```

--some action

END;

Example:

DECLARE

v_salary NUMBER := 4000; -- Employee salary e_low_salaryEXCEPTION;--User-definedexception

BEGIN

--Checkifsalaryisbelowthethreshold IF v_salary

< 5000 THEN

RAISEe_low_salary;--Raisetheuser-definedexception END IF;

DBMS_OUTPUT.PUT_LINE('Salaryisacceptable.'); EXCEPTION WHEN e_low_salary THEN

DBMS_OUTPUT.PUT_LINE('Error:Salaryisbelowthe minimum threshold.');

WHENOTHERSTHEN

DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');

END;

/

Proceduresin PL/SQL

A **procedure** in PL/SQL is anamed blockof code that performs a specific task. It is similar to a function but does not necessarily return a value. Procedures are

used to encapsulate and reuse business logic, reducing redundancy and enhancing maintainability.

Parametersin Procedures

In PL/SQL, **parameters** in procedures allow you to pass and retrieve values between the calling program and the procedure. There are three types of parameters:

1. **INParameter**:
    - Usedtopassinputvaluestotheprocedure.
    - Acts as a constant and cannot be modified within the procedure.
2. **OUT Parameter**:
    - Usedtoreturnavaluefromtheproceduretothecalling program.
    - Mustbeassignedavaluewithintheprocedure.
3. **INOUTParameter**:
    - Used to pass a value into the procedure, modify it within the procedure, and return it to the calling program.

Parametersin Procedures

In PL/SQL, **parameters** in procedures allow you to pass and retrieve values between the calling program and the procedure. There are three types of parameters:

1. **INParameter**:
    - Usedtopassinputvaluestotheprocedure.

- Acts as a constant and cannot be modified within the procedure.

2. **OUT Parameter**:
   - Usedtoreturnavaluefromtheproceduretothecalling program.
   - Mustbeassignedavaluewithintheprocedure.
3. **INOUTParameter**:
   - Used to pass a value into the procedure, modify it within the procedure, and return it to the calling program.

SyntaxforParametersinProcedures

```
CREATEORREPLACEPROCEDUREprocedure_name( parameter1
   IN datatype,    -- Input parameter parameter2 OUT datatype,
   -- Output parameter
   parameter3INOUTdatatype    --Inputandoutputparameter
)IS
   --Declarations
BEGIN
   --Executablestatements EXCEPTION
   --Exceptionhandling END procedure_name;
/
```

Example:ProceduREwithParameters Creating the Procedure

```
CREATEORREPLACEPROCEDUREmanage_salary( p_emp_id IN
    NUMBER, -- Input parameter
    p_salary IN OUT NUMBER,  --Inputandoutputparameter p_bonus OUT
    NUMBER       -- Output parameter
) IS
BEGIN
    --Calculatebonusas10%ofsalary p_bonus := p_salary
    * 0.10;

    --Updatesalarybyaddingbonus p_salary := p_salary
    + p_bonus;

    DBMS_OUTPUT.PUT_LINE('SalaryupdatedforEmployee ID: ' || p_emp_id);
END;
/
```

Droppinga Procedure

Toremoveanexistingprocedure,usetheDROPPROCEDURE statement.

Syntax:

```
DROPPROCEDUREprocedure_name;
```

Example:

```
DROPPROCEDUREmanage_salary;
```

PL/SQLFunctions

PL/SQLfunctions arereusableblocks ofcode that canbeusedto perform specific tasks. They are similar to **procedures** but must always return a value.

AfunctioninPL/SQLcontains:

- **Function Header**: The function header includes the function name and an optional parameter list. It is the first part of the function and specifies the name and parameters.
- **Function Body**: The function body contains the executable statements that implement the specific logic. It can include declarative statements, executable statements, and exception- handling statements.

CreateFunctioninPL/SQL

To createaprocedurein PL/SQL,usethe**CREATEFUNCTION statement.**

Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
(parameter_name type [, …])

— This statement is must for functions
RETURN return_datatype

{IS | AS}

BEGIN
— program code
```

[EXCEPTION
exception_section;

END[function_name]; Example

CREATEORREPLACEFUNCTIONfactorial(xNUMBER) RETURN NUMBER

IS

fNUMBER; BEGIN

IFx=0THEN f :=

1;

ELSE

f:=x*factorial(x-1);
END IF;

RETURN f;

END;

/

Droppinga Function

Toremoveanexistingfunction,usetheDROPFUNCTIONstatement.

Syntax:

DROPFUNCTIONfunction_name;

Example:

DROPFUNCTIONcalculate_bonus;

UNITIV:INTRODUCTIONTOSCHEMAREFINEMENT

introduction to Schema Refinement; Purpose of Normalization or Schema refinement Problems Caused by Redundancy, Decompositions, Problems Related to Decomposition, Functional Dependencies, Reasoning about FDS, Normal Forms: 1NF, 2NF, 3NW, BCNE, Properties of Decomposition: Losless Join Decomposition, Dependency Preserving Decomposition, Multivalued Dependencies, 4 NF.

SCHEMAREFINEMENT

We now present an overview of the problems that schema refinement is intended to address andarefinementapproachbasedondecompositions.Redundantstorageofinformationis the root cause of these problems. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

ProblemsCausedbyRedundancy

Storing the same information redundantly, that is, in more than one place within adatabase, can lead to several problems:

- **Redundantstorage:**Someinformationisstoredrepeatedly.
- **Updateanomalies:**Ifonecopyofsuchrepeateddataisupdated,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.

Consider a relation obtained by translating a variant of the Hourly Emps entity set from Chapter 2:

HourlyEmps(*ssn*, *name*, *lot*, *rating*, *hourly wages*, *hours worked*)

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

The key for HourlyEmps is *ssn*. In addition, suppose that the *hourly wages* attribute is determined by the *rating* attribute. That is, for a given *rating* value, there is only one permissible *hourly wages* value. This IC is an example of a *functional dependency*. It leads to possible redundancy in the relation HourlyEmps, as illustrated in Figure 15.1.

If the same value appears in the *rating* column of two tuples, the IC tells us that the same value must appear in the *hourly wages* column as well. This redundancy has several negative consequences:

ssn	*name*	*lot*	*rating*	*hourlywages*	*hoursworked*
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

AnInstanceoftheHourlyEmpsRelation

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.

Use of Decompositions

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

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

HourlyEmps2(*ssn*,*name*,*lot*,*rating*,*hoursworked*)Wages(*rating*, *hourlywages*)

Theinstancesoftheserelationscorrespondingtotheinstanceof HourlyEmpsrelation in Figure 15.1 is shown in Figure 15.2.

ssn	*name*	*lot*	*rating*	*hoursworked*

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

InstancesofHourlyEmps2 andWages

Notethatwecaneasilyrecordthehourlywageforanyratingsimplybyaddingatuple to Wages, even if no employee with that rating appears in the current instance of HourlyEmps.Changingthewageassociatedwitharatinginvolvesupdatingasingle Wages tuple. This is more efficient than updating several tuples (as in the original design),anditalsoeliminatesthepotentialforinconsistency.Noticethattheinsertion and deletion anomalies have also been eliminated.

ProblemsRelatedtoDecomposition

Unlesswearecareful,decomposingarelationschemacancreatemoreproblemstha n it solves. Two important questions must be asked repeatedly:

1. Doweneedtodecomposearelation?

2. Whatproblems(ifany)doesagivendecompositioncause?

To help with thefirstquestion,several *normalforms* have beenproposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problemscannotarise.Consideringthenormalformofagivenrelationschemacan help us to decide whether or not to decompose it further. If we decide that a relation schemamustbedecomposedfurther,wemustchooseaparticulardecomposition(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 particularinterest.The*losslessjoin*propertyenablesustorecoveranyinstanceofthe decom-posed relation from corresponding instances of the smaller relations. The *dependencypreservation*propertyenablesustoenforceanyconstraintontheorigina l relation by simply enforcing some constraints on each of the smaller relations. That is,weneednotperformjoinsofthesmallerrelationstocheckwhetheraconstrainton the original relation is violated.

FUNCTIONALDEPENDENCIES

AfunctionaldependencyFDisakindofICthatgeneralizestheconceptofa *key*.

Let*R*bearelationschemaandlet*X*and*Y* benonemptysetsofattributesin*R*.We 1 saythataninstance*r*of*R* satisfiestheFD*X!Y*ifthefollowingholdsforeverypairoftuples *t*1 and*t*2 in*r*: If *t*1*:X* = *t*2*:X*, then *t*1*:Y* = *t*2*:Y* .

We usethenotation *t*1*:X*torefer totheprojectionoftuple *t*1 ontotheattributesin *X*, inanaturalextensionofourTRCnotation(seeChapter4)*t:a*forreferringtoattribute *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 keyconstraint:AlthoughtheFDisnotviolated,*AB*isclearlynotakeyfortherelation. Thethirdandfourthtuplesillustratethatiftwotuplesdifferineitherthe*Afi*eldorthe *B fi*eld, they can differ in the *C fi*eld without violating the FD. On the other hand, if we add a tuple *ha*1*; b*1*; c*2*; d*1*i* to the instance shown in this figure, the resulting instance would violate the FD; to see this violation, compare the first tuple in 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

AnInstancethatSatisfies *AB !C*

Recall that a *legal* instance of a relation must satisfy all specified ICs, including all specifiedFDs.AsnotedinSection3.2,ICsmustbeidentifiedandspecifiedbasedon the semantics of the real-world enterprise being modeled. By looking at an instance of a relation, we might be able to tell that a certain FD does *not* hold. However, we can never deduce that an FD *does* hold by looking at one or more instances of the

relationbecauseanFD,likeotherICs,isastatementabout*all*possiblelegalinstances of the relation.

Aprimarykeyconstraintisaspecialcaseofan FD.Theattributesinthekeyplaythe roleof*X*,andthesetofallattributesintherelationplaystheroleof*Y*.Note,however, thatthedefinitionofanFDdoesnotrequirethattheset *X*beminimal;theadditional minimalityconditionmust bemet for*X* to be a key. If *X ! Y* holds, where*Y* is the set of all attributes, andthere is some subset *V* of *X* such that *V !Y* holds,then *X*is a *super key*; if *V* is a strict subset of *X*, then *X* is not a key.

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

REASONINGABOUTFUNCTIONALDEPENDENCIES

The discussion up to this point has highlighted the need for techniques that allow us to carefully examine and further re ne relations obtained through ER design (or, for that matter,through other approachestoconceptual design). Before proceedingwith themaintaskathand,whichisthediscussionofsuchschemarefinementtechniques, we digress to examine FDs in more detail because they play such a central role in schema analysis and refinement.

GivenasetofFDsoverarelationschema*R*,therearetypicallyseveraladditionalFDs that hold over *R* whenever all of the given FDs hold. As an example, consider:

Workers(*ssn*,*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 (fromthe 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 dependenciesin*F*;rather,*f*mustholdon*every*instancethatsatisfiesalldependencie s in *F*.

ClosureofaSetofFDs

Theset ofall FDsimpliedbyagivensetFofFDsiscalledtheclosureofFandis +denotedas*F*.Animportantquestionishowwecaninfer,orcompute,

theclosure of agiven set *F*of FDs.Theanswerissimpleandelegant.Thefollowing 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 *XY*,then*X!Y*.

Augmentation:If*X!Y*,then*XZ!YZ*foranyZ. 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

+
tousesomeadditionalruleswhilereasoningabout*F*:

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

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

Theseadditionalrulesarenotessential;theirsoundnesscanbeprovedusingArmstrong's Axioms.

useamoreelaborateversionoftheContracts relation:

Contracts(*contractid*,*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 department *D* (*deptid*); the value *V* of this contract is equal to *value*.

ThefollowingICsareknownto hold:

1. Thecontractid *C*isakey:*C! CSJDPQV*.
2. Aprojectpurchasesagivenpartusingasinglecontract:*JP!C*.
3. Adepartmentpurchasesatmostonepartfromasupplier:*SD!P*.

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

From*JP!C*,*C!CSJDPQV*andtransitivity,weinfer*JP!CSJDPQV*.

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

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

We can infer several additional FDs that arein the closure by using augmentation or

decomposition.Forexample,from*C!CSJDPQV*,usingdecompositionwecaninfer:

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

Finally,wehaveanumberoftrivialFDsfromthereflexivityrule.

M.NANDINI

NORMALFORMS

Givenarelationschema,weneedtodecidewhetheritisagooddesignorwhetherwen eedto decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any, arise from the current schema. To provide such guidance, several normal forms have been proposed. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise.

The normal forms based on FDs are fi*rst normal form (1NF)*, *second normal form (2NF)*,*thirdnormalform(3NF)*,and*Boyce-Coddnormalform(BCNF)*.Theseforms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF, everyrelationin 3NF is also in 2NF, andeveryrelation in 2NF isin 1NF. A relation is in first normal form if every field contains only atomic values, that is, not lists or sets. This requirement is implicit in our de nition of the relational model. Although some ofthenewer databasesystems arerelaxingthisrequirement,inthischapterwe willassumethatitalwaysholds.2NFismainlyofhistoricalinterest.3NFandBCNF are important from a database design standpoint.
While studying normal forms, it is important to appreciate the role played by FDs. Considerarelationschema *R*withattributes *ABC*.Intheabsenceof anyICs,anyset of ternary tuples is a legal instance and there is no potential for redundancy. On the other hand, suppose that we have the FD *A ! B*. Now if several tuples have the same *A* value, they must also have the same *B* value. This potential redundancy can be predicted using the FD information. If more detailed ICs are specified, we may be able to detect more subtle redundancies as well.

Wewillprimarilydiscussredundancythatisrevealedby FDinformation.InSection 15.8, we discuss more sophisticated ICs called *multivalued dependencies* and *join dependencies* and normal forms based on them.

15.5.1Boyce-CoddNormalForm

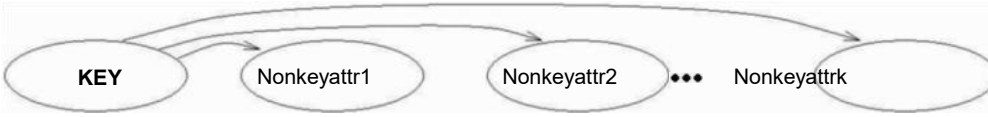
Let *R* be a relation schema, *X* be a subset of the attributes of *R*, and let *A* be an attribute of *R*. *R* is in Boyce-Codd normal form if for every FD *X ! A* that holds over *R*, one of the following statements is true:

- *A 2 X*; that is, it is a trivial FD, or
- *X* is a super key.

Note that if we are given a set *F* of FDs, according to this definition, we must + consider each dependency *X ! A* in the closure *F* to determine whether *R* is in BCNF. However, we can prove that it is sufficient to check whether the left side of each dependency in *F* is a super key (by computing the attribute closure and seeing if it includes all attributes of *R*).

Intuitively, in a BCNF relation the only nontrivial dependencies are those in which a key determines some attribute(s). Thus, each tuple can be thought of as an entity or relationship, identified by a key and described by the remaining attributes. Kent puts this colorfully, if a little loosely: \Each attribute must describe [an entity or relationship identified by] the key, the whole key, and nothing but the key." If we use

ovalstodenoteattributesorsetsofattributesanddrawarcstoindicateFDs,arelation in BCNF has the structure illustrated in Figure, considering just one key for simplicity. (If there are several candidate keys, each candidate key can play the role of KEY in the figure, with the other attributes being the ones not in the chosen candidate key.)



FDsinaBCNF Relation

BCNF ensures that no redundancy can be detected using FD information alone. It is thus the most desirable normal form (from the point of view of redundancy) if we take into account only FD information. This point is illustrated in Figure 15.8.

X	*Y*	*A*

x	*y*1	*a*
x	*y*2	?

InstanceIllustratingBCNF

This figure shows (two tuples in) an instance of a relation with three attributes *X*, *Y*, and *A*. There are two tuples with the same value in the *X* column. Now suppose that

weknowthatthisinstancesatisfiesanFD*X!A*.Wecanseethatoneofthetupleshas thevalue*a*inthe*A*column.Whatcanweinferaboutthevalueinthe*A*columninthe secondtuple?UsingtheFD,wecanconcludethatthesecondtuplealsohasthevalue *a*inthiscolumn.(Notethatthisisreallytheonlykindofinferencewecanmakeabout values in the fields of tuples by using FDs.)

Butisn'tthissituationanexampleofredundancy?Weappeartohavestoredthevalue *a*twice.CansuchasituationariseinaBCNFrelation?No!IfthisrelationisinBCNF, because *A* is distinct from *X* it follows that *X* must be a key. (Otherwise, the FD *X ! A* would violateBCNF.)If*X*isakey,then*y*1=*y*2 ,whichmeansthatthetwotuplesareidentical. Since arelationisdefinedtobea*set*oftuples,wecannothavetwo copiesofthesametupleandthe situation shown in Figure 15.8 cannot arise.

Thus,ifarelationisinBCNF,everyfieldofeverytuplerecordsapieceofinformation thatcannotbeinferred(usingonlyFDs)fromthevaluesinallothereldsin(alltuples of) the relation instance.

ThirdNormalForm

Let *R* be a relation schema, *X* be a subset of the attributes of *R*, and *A* be an attribute of *R*. *R* is in third normal form if for every FD *X ! A* that holds over *R*, one of the following statements is true:

A2X; thatis,it isatrivialFD,or

-

- *X*isasuperkey,or

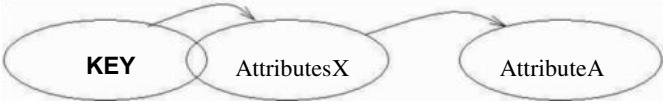
*A*ispartofsomekeyfor*R*.

The de nition of 3NF is similar to that of BCNF, with the only difference being the third condition. Every BCNF relation is also in 3NF. To understand the third condition, recall that a key for a relation is a *minimal* set of attributes that uniquely determinesallotherattributes.
*A*mustbepartofakey(anykey,iftherearéseveral).
Itisnotenoughfor*A*tobepartofasuperkey,becausethelatterconditionissatisfied by each and every attribute! Finding all keys of a relation schema is known to be an NP-complete problem, and so is the problem of determining whether a relation schema is in 3NF.

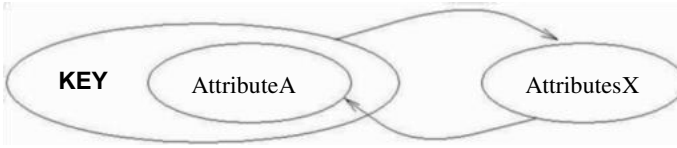
Partial dependencies are illustrated in Figure 15.9, and transitive dependencies are
illustratedinFigure.NotethatinFigure15.10,thesetXofattributesmayormaynot have some attributes in common with KEY; the diagram should be interpreted as indicating only that *X* is not a subset of KEY.

Case1:AnotinKEY

Figure15.9 PartialDependencies



Case 1:Anot in KEY



Case 2:Ais in KEY

TransitiveDependencies

The motivation for 3NF is rather technical. By making an exception for certain dependencies involving key attributes, we can ensure that every relation schema can bedecomposedintoacollectionof3NFrelationsusingonlydecompositionsthatha ve certaindesirableproperties(Section15.6).SuchaguaranteedoesnotexistforBCNF relations; the 3NF definition weakens the BCNF requirements just enough to make this guarantee possible. We may therefore compromise bysettlingfor a 3NF design. As we shall see in Chapter 16, we may sometimes accept this compromise (or even settle for a non-3NF schema) for other reasons as well.

DECOMPOSITIONS

AswehaveseenarelationinBCNFisfreeofredundancy(tobeprecise,redundancy thatcanbedetectedusingFDinformation),andarelationschemain3NFcomesclose.

Ifarelationschemaisnotinoneofthesenormalforms,theFDsthatcauseaviolation can give us insight into the potential problems. The main technique for addressing such redundancy-related problems is decomposing a relation schema into relation schemas with fewer attributes.

Adecomposition of a relation schema *R* consists of replacingtherelation schema by two (or more) relation schemas that each contain a subset of the attributes of *R* and
togetherincludeallattributesin*R*.Intuitively,wewanttostoretheinformationinany given instance of *R* by storing projections of the instance. This section examines the use of decompositions through several examples.

We begin with the Hourly Emps example from Section 15.3.1. This relation has attributes *SNLRWH* and two FDs: *S ! SNLRWH* and *R ! W*. Since *R* is not a key and *W* is not part of any key, the second dependency causes a violation of 3NF.

ThealternativedesignconsistedofreplacingHourlyEmpswithtworelationshavin gattributes *SNLRH* and *RW*. *S ! SNLRH* holds over *SNLRH*, and *S* is a key. *R ! W*holds over *RW*, and *R* isakeyfor*RW*.Theonlyotherdependenciesthatholdovertheseschemasarethoseob tained by augmentation. Thus both schemas are in BCNF.

Our decision to decompose *SNLRWH* into *SNLRH* and *RW*, rather than, say, *SNLR* and *LRWH*, was not just a good guess. It was guided by the observation that the
dependency*R!W*causedtheviolationof3NF;themostnaturalwaytodealwiththis violationistoremovetheattribute*W*fromthisschema.Tocompensateforremoving *W* from the main schema, we can add a relation *RW*, because each *R* value is associated with at most one *W* value according to the FD *R ! W*.

A very important question must be asked at this point: If we replace a legal instance *r* of relation schema *SNLRWH* with its projections on *SNLRH* (*r*1) and *RW*(*r*2), can we recover *r* from *r*1 and *r*2? The decision to decompose *SNLRWH* into *SNLRH* and *RW* is equivalent to saying that we will store instances *r*1 and *r*2 instead of *r*. However

it is the instance *r* that captures the intended entities or relationships. If we cannot compute*r*from*r*1 and*r*2,ourattempttodealwithredundancyhaseectivelythrown out the baby with the bathwater. We consider this issue in more detail below.

Lossless-JoinDecomposition

sition of *R* intotwoschemaswithattributesets*X*and*Y*issaidtobealosslessjoindecomposition withrespectto Fifforeveryinstance*r*of*R*thatsatises the dependenciesin*F*,*X*(*r*) *./ Y*(*r*)=*r*.

This de nition can easily be extended to cover a decomposition of *R* into more than tworelations.Itiseasytoseethat*rX* (*r*)*./Y* (*r*)alwaysholds.Ingeneral,though,the otherdirectiondoesnothold.Ifwetakeprojectionsofarelationandrecombinethem using natural join, we typically obtain some tuples that were not in the original relation. This situation is illustrated in Figure 15.11.

S	*P*	*D*

S P P D

	s1 p1	s1 p1	d1
s2 p2	s2 p2	d2	s3 p1
s3	p1	d3	

Instance*r*

S	*P*

SP(*r*)

P	*D*(*r*)
p1	d1
p2	d2
p1	d3

./P D(*r*)

D

s1	p1	d1
s2	p2	d2
s3	p1	d3
s1	p1	d3
s3	p1	d1

Byreplacingtheinstance*r*showninFigure15.11withtheinstances*SP*(*r*)and*PD*(*r*), we lose some information. In particular, suppose that the tuples in *r* denote relationships.Wecannolongertellthattherelationships(*s*1*;p*1*;d*3)and(*s*3*;p*1*;d*1) do not hold. The decomposition of schema *SPD* into *SP* and *PD* is therefore a `lossy' decomposition if the instance *r* shown in the figure is legal, that is, if this instance could arise in the enterprise being modeled. (Observe the similarities between this example and the Contracts relationship set in Section 2.5.3.)

All decompositions used to eliminate redundancy must *be lossless.* The following simple test is very useful:

et*R*bearelationand *F*beasetofFDsthatholdover*R*.Thedecompositionof *R*intorelationswithattributesets *R*1and*R*2islosslessif

+andonlyif*F*containseithertheFD*R*1*R*2*!R*1ortheFD *R*1\ *R*2*!R*2.

Inother words,theattributescommonto*R*1and*R*2mustcontainakeyforeither*R*1or *R*2.Ifarelationisdecomposedintotworelations,thistestisanecessaryand 2 sufficien t condition for the decomposition to be lossless-join. If a relation is decomposed into more than two relations, an efficient (time polynomial in the size of the dependency set)algorithmisavailabletotestwhetherornotthedecompositionislossless,butwe will not discuss it.

Consider the Hourly Emps relation again. It has attributes *SNLRWH*, and the FD *R ! W*causesaviolationof3NF.Wedealtwiththisviolationbydecomposingtherelation into *SNLRH* and *RW*. Since *R* is common to both decomposed relations, and *R ! W* holds, this decomposition is lossless-join.

Thisexampleillustratesageneralobservation:

IfanFD*X!Y*holdsoverarelation*R*and*X\Y*isempty,thedecompositionof *R*into*R*−*Y*and*XY*is lossless.

*X*appearsinboth*R*−*Y*(since*X\Y*isempty)and*XY*,anditisakeyfor*XY*.Thus,the above observation follows from the test for a lossless-join decomposition.

Anotherimportantobservationhastodowithrepeateddecompositions.Supposeth atarelation *R* is decomposed into *R1* and *R2* through a lossless-join decomposition, and that *R1* is decomposed into *R11* and *R12* through another lossless-join decomposition. Then the decomposition of *R* into *R11*, *R12*, and *R2* is lossless-join; by joining *R11* and *R12* we can recover *R1*, and by then

Let*R*bearelationand *F*beasetofFDsthatholdover*R*.Thedecompositionof joining *R1* and *R2*, we can recover *R*.

Dependency-PreservingDecomposition

onsidertheContractsrelationwithattributes*CSJDPQV*fromSection15.4.1.The given FDs are *C ! CSJDPQV*, *JP ! C*, and *SD ! P*. Because *SD* is not a key the dependency *SD ! P* causes a violation of BCNF.

We can decompose Contracts into two relations with schemas *CSJDQV* and *SDP* to addressthisviolation;thedecompositionislosslessjoin.Thereisonesubtleproblem, however.Wecanenforcetheintegrityconstraint*JP!C*easilywhenatupleisinserted into Contracts by ensuring that no existing tuple has the same *JP* values (as the inserted tuple) but different *C* values. Once we decompose Contracts into *CSJDQV* and *SDP*, enforcing this constraint requires an expensive join of the two relations whenever a tuple is inserted into *CSJDQV*. We say that this decomposition is not dependency-preserving.

Intuitively,a*dependencypreservingdecomposition*allowsustoenforceallFDsbye xamining a single relation instance on each insertion or modification of a tuple. (Note that deletions cannot cause violation of FDs.) To de ne dependency-preserving decompositions precisely, we have to introduce the concept of a projection of FDs.

Let*R*bearelationschemathatisdecomposedintotwoschemaswithattributesets*X* and*Y*,andlet *F*beasetofFDsover+*R*.TheprojectionofFon *X*istheset of FDsintheclosure*F*(notjust *F*!)thatinvolveonlyattributesin*X*.Wewilldenote + the projection of *F* on attributes *X* as *FX* . Note that a dependency *U ! V* in *F*is in *FX* only if *all* the attributes in *U* and *V* are in *X*.

Thedecompositionofrelationschema *R*withFDs*F*intoschemaswithattributesets

+ +
X and *Y* is dependency-preserving if (*FX [FY*) = *F* . That is, if we take the ConsidertheContractsrelationwithattributes*CSJDPQV*fromSection15.4.1.The dependencies in *FX* and *FY* and compute the closure of their union, we get back all dependencies in the closure of *F*. Therefore, we need to enforce only the +dependenciesin*FX*and*FY*;allFDsin*F* are then sure to be satisfied. To enforce *FX* , we need to examine only relation *X* (on inserts to that relation). To enforce *FY* , we need to examine only relation *Y*.

NORMALIZATION

Having covered the concepts needed to understand the role of normal forms and de-
compositionsindatabasedesign,wenowconsideralgorithmsforconvertingrelatio
ns to BCNF or 3NF. If a relation schema is not in BCNF, it is possible to obtain a lossless-join decomposition into a collection of BCNF relation schemas. Unfortunately, there may not be any dependency-preserving decomposition into a collection of BCNF relation schemas

DecompositionintoBCNF

We now present an algorithmfor decomposing a relation schema *R* into a collection of BCNF relation schemas:

1. SupposethatRisnotinBCNF.Let*XR*,*A*beasingleattributein*R*,and*X!A* beanFDthatcausesaviolationofBCNF.Decompose*R*into*R−>A*and*X* _*A*.

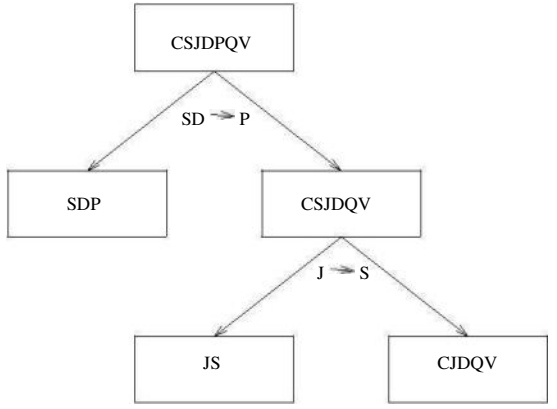
2. Ifeither*R−>A*or*XA*isnotinBCNF,decomposethemfurtherbyarecursi ve application of this algorithm.

*R−>A*denotestheset ofattributesotherthan *A*in*R*,and *XA*denotestheunionof attributesin*X*and*A*.Since*X!A*violatesBCNF,itisnotatrivialdependency;further, *A* is a single attribute. Therefore, *A* is not in *X*; that is, *X* *A* is empty. Thus, each decomposition carried out in Step is lossless-join.

Thesetofdependenciesassociatedwith*R−>A*and*XA*istheprojectionof *F*onto their attributes. If one of the new relations is not in BCNF, we decompose it further in Step. Since a decomposition results in relations with strictly fewer attributes, this
processwillterminate,leavinguswithacollectionofrelationschemasthatareallin BCNF.

Consider the Contracts relation with attributes *CSJDPQV* and key *C*. We are given FDs *JP ! C* and *SD ! P*. By using the dependency *SD ! P* to guide the decomposition, we get the two schemas *SDP* and *CSJDQV*. *SDP* is in BCNF. Suppose that we also have the constraint that eachprojectdealswithasinglesupplier: *J!S*.Thismeansthattheschema *CSJDQV* isnotin BCNF. So we decompose it further into *JS* and *CJDQV*. *C ! JDQV* holds over *CJDQV*; the only other FDs that hold are those obtained from this FD by augmentation, and therefore all FDscontainakeyintheleftside.Thus,eachoftheschemas *SDP*,*JS*,and*CJDQV*isin BCNF, and this collection of schemas also represents a lossless-join decomposition of *CSJDQV*.

The steps in this decomposition process can be visualized as a tree, as shown in Figure. The root is the original relation *CSJDPQV*, and the leaves are the BCNF relations that are the result of the decomposition algorithm, namely, *SDP*, *JS*, and *CSDQV*. Intuitively, each internal node is replaced by its children through a single decomposition step that is guided by the FD shown just below the node.



Decompositionof*CSJDQV*into*SDP*,*JS*,and*CJDQV*

RedundancyinBCNFRevisited

The decomposition of *CSJDQV* into *SDP*, *JS*, and *CJDQV* is not dependencypreserving. Intuitively, dependency *JP ! C* cannot be enforced without a join. One way to deal with this situation is to add a relation with attributes *CJP*. In e ect, this solution amounts to storing some information redundantly in order to make the dependency enforcement cheaper.

This is a subtle point: Each of the schemas *CJP*, *SDP*, *JS*, and *CJDQV* is in BCNF, yet there is some redundancy that can be predicted by FD information. In particular, if we join the relation instances for *SDP* and *CJDQV* and project onto the attributes *CJP*, we must get exactly the instance stored in the relation with schema *CJP*. We sawinSection15.5.1thatthereisnosuchredundancywithinasingleBCNFrelation. The current example shows that redundancy can still occur across relations, even though there is no redundancy within a relation.

MinimalCoverforaSetofFDs

Aminimal cover foraset*F*ofFDsisaset*G*ofFDssuchthat:

1. Everydependencyin *G*isoftheform*X!A*,where*A*isasingle attribute.

+ +

2. Theclosure *F*isequaltotheclosure*G*.

3. Ifweobtainaset*H*ofdependenciesfrom*G*bydeletingoneormoredependencies, ++orby deletingattributesfromadependencyin*G*,then*F6=H*.

Intuitively, a minimal cover for a set *F* of FDs is an equivalent set of dependencies that is *minimal*intworespects: (1) Everydependencyisassmall aspossible;thatis, each attribute on the left side is necessary and the right side is a single attribute.

+

(2)Everydependencyinitisrequiredinorderfortheclosuretobeequalto *F*. As an

example, let *F* be the set of dependencies:

A!B, *ABCD!E*, *EF!G*, *EF!H*, and *ACDF!EG*.

First,letusrewrite*ACDF!EG*sothat everyright sideisasingleattribute:

*ACDF!E*and*ACDF!G*.

Nextconsider*ACDF!G*.Thisdependencyisimpliedbythefollowing FDs:

A!B, *ABCD!E*,and*EF!G*.

Therefore,wecandeleteit.Similarly,wecandelete*ACDF!E*.Nextconsider*ABCD !E*.Since*A!B*holds,wecanreplaceitwith*ACD!E*.(Atthispoint,thereadershould verifythateachremainingFDisminimalandrequired.)Thus,aminimalcoverfor *F* is the set:

A!B, *ACD!E*, *EF!G*,and *EF!H*.

Theprecedingexamplesuggestsageneralalgorithmforobtainingaminimalcovero f a set *F* of FDs:

1. PuttheFDsinastandardform:Obtainacollection*G*ofequivalentFDswitha single attribute on the right side (using the decomposition axiom).

2. Minimizetheleftsideofeach FD:ForeachFDin*G*,checkeachattributein + theleft sidetoseeifitcanbedeletedwhilepreservingequivalenceto*F*.

3. DeleteredundantFDs:CheckeachremainingFDin*G*toseeifitcanbe + deletedwhilepreservingequivalenceto *F*.

M.NANDINI

Note that the order in which we consider FDs while applying these steps could produce different minimal covers; there could be several minimal covers for a given set of FDs.

Moreimportant,itisnecessarytominimizetheleftsidesofFDs*before*checkingforredundant FDs.Ifthesetwostepsarereversed,thenalsetofFDscouldstillcontainsomeredundantFDs (i.e., not be a minimal cover), as the following example illustrates. Let *F* be the set of dependencies, each of which is already in the standard form:

ABCD!E,*E !D*, *A !B*, and*AC!D*.

ObservethatnoneoftheseFDsisredundant;ifwecheckedforredundantFDsrst,we would get the same set of FDs *F*. The left side of *ABCD ! E* can be replaced by + *AC* while preserving equivalence to *F* , and we would stop here if we checked for redundantFDsin*F*beforeminimizingtheleftsides.However,thesetofFDswehave is not a minimal cover:

AC!E, *E !D*, *A !B*,and *AC!D*.

Fromtransitivity,thefirsttwoFDsimplythelastFD,whichcanthereforebe deleted + while preserving equivalence to *F* . The importantpointtonoteisthat*AC!D*becomesredundantonlyafterwereplace*ABCD ! E* with *AC ! E*. If we minimize left sides of FDs first and then check for redundant FDs,weareleftwiththerstthreeFDsintheprecedinglist,whichisindeedaminimal cover for *F*.

Dependency-PreservingDecompositioninto3NF

Returningtotheproblemofobtainingalossless-join,dependencypreservingdecom- position into 3NF relations, let *R* be a relation with a set *F* of FDs that is a minimal cover, and let *R*1*; R*2*; : : : ; Rn* be a lossless-join decomposition of *R*. For 1 *i n*, supposethateach*Ri*isin3NFandlet*Fi*denotetheprojectionof*F*ontotheattributes of *Ri*. Do the following:

Identifytheset*N*ofdependenciesin*F*thatarenotpreserved,thatis,notincluded in the closure of the union of *Fi*s.

ForeachFD*X!A*in*N*,createarelationschema*XA*andadditthedecomposition of *R*.

Obviously, every dependency in *F* is preserved if we replace *R* by the *Ri*s plus the schemas of the form *XA* added in this step. The *Ri*s are given to be in 3NF. We can showthateachoftheschemas *XA*isin3NFasfollows:Since *X!A*isintheminimal cover*F*,*Y!A*doesnotholdforany*Y*thatisastrictsubsetof*X*.Therefore,*X*isakey for *XA*. Further, if any other dependencies hold over *XA*, the right side can involve onlyattributesin*X*because*A*isasingleattribute(because*X!A*isanFDinaminimal cover). Since *X* is a key for *XA*, none of these additional dependencies causes a violation of 3NF (although they might cause a violation of BCNF).

As anoptimization,if theset *N*containsseveral FDswiththesame left side, say, *X! A*1*;X!A*2*;:::;X!An*,wecanreplacethemwithasingleequivalent FD*X!A*1 *::: An*. Therefore, we produce one relation schema *XA*1 *: : : An*, instead of several schemas *XA*1*; : : : ; XAn*, which is generally preferable.
ConsidertheContractsrelationwithattributes*CSJDPQV*andFDs*JP!C*,*SD!P*,and *J!S*.If
wedecompose*CSJDPQV*into*SDP*and*CSJDQV*,then*SDP*isinBCNF,but*CSJDQ V*isnot even in 3NF. So we decompose it further into *JS* and *CJDQV*.
The relation schemas *SDP*, *JS*, and *CJDQV* are in 3NF (in fact, in BCNF), and the decompositionislossless-join. However,thedependency *JP ! C*isnotpreserved.Thisproblem can be addressed by adding a relation schema *CJP* to the decomposition.

ThissetofFDsisnot aminimal cover,andsowe mustndone.Werst replace*C !CSJDPQV*withtheFDs:

C!S, *C!J*, *C!D*, *C!P*, *C !Q*, and*C!V*.

The FD *C!P*isimpliedby*C!S*,*C!D*,and*SD!P*; sowecandeleteit.The FD *C ! S* is implied by *C ! J* and *J ! S*; so we can delete it. This leaves us with a minimal cover:

C!J, *C!D*, *C!Q*,*C!V*, *JP!C*, *SD!P*, and *J !S*.

Using the algorithm for ensuring dependency-preservation, we obtain the relational schema *CJ*,*CD*, *CQ*, *CV*,*CJP*,*SDP*,and*JS*.Wecanimprovethisschemabycombiningrelationsfor which *C* is the key into *CDJPQV*. In addition, we have *SDP* and *JS* in our decomposition. Since one of these relations (*CDJPQV*) is a super key, we are done.

Comparing this decomposition with the one that we obtained earlier in this section, we findthat theyare quiteclose, with the onlydifference beingthat one ofthem has

*CDJPQV*insteadof*CJP*and*CJDQV*.Ingeneral,however,therecouldbesignificant differences.

Database designers typically use a conceptual design methodology (e.g., ER design) to arrive at an initial database design. Given this, the approach of repeated
decompositionstorectifyinstancesofredundancyislikelytobethemostnaturaluse of FDs and normalization techniques. However, a designer can also consider the alternative designs suggested by the synthesis approach.

OTHERKINDSOFDEPENDENCIES*

Ds are probably the most common and important kind of constraint from the point of view ofdatabasedesign.However,thereareseveralotherkindsofdependencies.Inpartic ular,there is a well-developed theory for database design using *multivalued dependencies* and *join dependencies*.Bytakingsuchdependenciesintoaccount,wecanidentifypotentialr edundancy problems that cannot be detected using FDs alone.

Thissectionillustratesthekindsofredundancythatcanbedetectedusingmultivalue d dependencies. Our main observation, however, is that simple guidelines (which can becheckedusingonlyFDreasoning)cantelluswhetherweevenneedtoworryabout complexconstraintssuchasmultivaluedandjoindependencies.Wealsocommento n the role of *inclusion dependencies* in database design.

MultivaluedDependencies

Supposethatwehavearelationwithattributes*course*,*teacher*,and*book*,whichwed enoteas *CTB*. The meaning of a tuple is that teacher *T* can teach course *C*, and book *B* is a recommended text for the course. There are no FDs; the key is *CTB*. However, the recommended texts for a course are independent of the instructor. The instance shown in Figure 15.13 illustrates this situation.

course	*teacher*	*book*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics

Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

BCNFRelationwith RedundancyThatIsRevealed byMVDs

Therearethreepointstonotehere:

Therelationschema *CTB*isinBCNF;thuswewouldnotconsiderdecomposing it further if we looked only at the FDs that hold over *CTB*.

There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.

- Theredundancycanbeeliminatedbydecomposing *CTB*into*CT*and*CB*.

The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be expressed in terms of FDs. This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation usingtwobinaryrelationshipsets,Instructorswithattributes *CT*andTextwithattributes *CB*. Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate. (See Section 2.5.3 for a further discussion of ternary versus binary relationships.) Given the subjectivity of ER design, however, we might create a ternary relationship. A careful analysis of the MVD information would then reveal the problem.

Let *R* be a relation schema and let *X* and *Y* be subsets of the attributes of *R*. Intuitively, the multivalued dependency *X !! Y* is said to hold over *R* if, in every legal instance *r* of *R*, each *X* value is associated with a set of *Y* values and this set is independent of the values in the other attributes.

Formally, if the MVD *X!!Y* holds over *R* and *Z=R−XY*, the following must be true for every legal instance *r* of *R*:

> If *t*1 2 *r*, *t*2 2 *r* and *t*1*:X=t*2*:X*, then there must be some *t*3 2 *r* such that *t*1*:XY =t*3*:XY* and *t*2*:Z=t*3*:Z*.

Figure illustrates this de nition. If we are given the first two tuples and told that the MVD *X!!Y* holds over this relation, we can infer that the relation instance must also contain the third tuple. Indeed, by interchanging the roles of the rst two tuples treating

thefirsttupleas*t*2andthesecondtupleas*t*1|wecandeducethatthetuple*t*4mustalso be in the relation instance.

X	*Y*	*Z*	
a	*b* 1	*c*1	|tuple *t*1
a	*b*2	*c*2	|tuple *t*2
a	*b* 1	*c*2	|tuple *t*3
a	*b* 2	*c*1	|tuple *t*4

IllustrationofMVD Definition

ThistablesuggestsanotherwaytothinkaboutMVDs:If*X!!Y* holdsover *R*,then*Y*

Z (*X*=*x*(*R*))=*Y* (*X*=*x*(*R*)) *Z* (*X*=*x*(*R*))ineverylegalinstanceof *R*,foranyvalue *x*that appearsintheXcolumnof*R*.Inotherwords,considergroupsoftuplesin *R*withthe same *X*-value, for each *X*-value. In each such group consider the projection onto the attributes *YZ*. This projection must be equal to the crossproduct of the projections onto*Y*and*Z*.Thatis,foragiven*X*-value,the*Y*valuesand*Z*-valuesareindependent.

(Fromthisdenitionitiseasytoseethat*X!!Y*mustholdwhenever*X!Y*holds.Ifthe FD *X ! Y* holds, there is exactly one *Y*-value for a given *X*-value, and the conditions in the MVD de nition hold trivially. The converse does not hold, as Figure 15.14 illustrates.)

Returning to our *CTB* example, the constraint that course texts are independent of instructors can be expressed as *C !! T*. In terms of the de nition of MVDs, this constraint can be read as follows:

\If (there is a tuple showing that) *C* is taught by teacher *T*,
and (thereisatupleshowingthat)*C*hasbook*B*astext,
then(thereisatuple showing that) *C* is taught by *T* and has text *B*.
GivenasetofFDsandMVDs,ingeneralwecaninferthatseveraladditionalFDsand MVDs hold. A sound and complete set of inference rules consists of the three ArmstrongAxiomsplusveadditionalrules.Threeoftheadditionalrulesinvolveonl y MVDs:

MVDComplementation:If *X!!Y*,then*X!!R−XY*.

MVD Augmentation: If *X !! Y* and *WZ*, then *WX !! YZ*.

MVDTransitivity:If *X!!Y* and*Y*

!!Z,then*X!!*(*Z−Y*).

As an example of the use of these rules, since we have *C !! T* over *CTB*, MVD complementation allows us to infer that *C !! CT B − CT* as well, that is, *C !! B*. The remaining two rules relate FDs and MVDs:

Replication:If *X!Y*,then*X!!Y*.

Coalescence: If *X !! Y* andthereis a *W*suchthat *W\ Y* is empty, *W! Z*, and *Y Z*, then *X ! Z*.

ObservethatreplicationstatesthateveryFDisalsoanMVD.

FourthNormalForm

Fourthnormalformisadirectgeneralizationof BCNF. Let *R* bearelationschema, *X* and *Y* benonemptysubsetsoftheattributesof *R*, and *F* beasetofdependencies that includesbothFDsandMVDs. *R* issaidtobeinfourthnormalform(4NF)ifforevery MVD *X !! Y* that holds over *R*, one of the following statements is true:

Y X or *XY*=*R*, or

- *X* is a Superkey.

In readingthis definition, it is important to understandthat the de nition ofa keyhas notchangedthekeymustuniquelydetermineallattributesthroughFDsalone. *X!!Y* is a trivial MVD if *Y X R* or *XY* = *R*; such MVDs always hold.

Therelation *CTB* isnotin4NFbecause *C!!T* isanontrivialMVDand *C* isnotakey. We can eliminate the resulting redundancy by decomposing *CTB* into *CT* and *CB*; each of these relations is then in 4NF.

To use MVD information fully, we must understand the theory of MVDs. However, the following result due to Date and Fagin identifies conditions detected using only FD information!|under which we can safely ignoreMVDinformation.That is, using MVDinformation in addition to the FDinformation will notreveal anyredundancy. Therefore, if these conditions hold, we do not even need to identify all MVDs.

> IfarelationschemaisinBCNF,andatleastoneofitskeysconsistsofasingle attribute, it is also in 4NF.

An important assumption is implicit in any application of the preceding result: *The set of FDs identified thus far is indeed the set of all FDs that hold over the relation.*

M.NANDINI

Thisassumptionisimportantbecausetheresultreliesontherelationbeinginbcnf, which in turn depends on the set of FDs that hold over the relation.

Figureshowsthreetuplesfromaninstanceof*ABCD*thatsatisfiesthegivenMVD*B !!C*.FromthedefinitionofanMVD,giventuples *t*1and*t*2,itfollows

B	*C*	*A*	*D*	
b	*c*1	*a*1	*d*1	|tuple *t*1
b	*c*2	*a*2	*d*2	|tuple *t*2
b	*c*1	*a*2	*d*2	|tuple *t*3

ThreeTuples fromaLegalInstanceof *ABCD*

thattuple*t*3mustalsobeincludedintheinstance.Considertuples*t*2and*t*3.Fromthe givenFD*A!BCD*andthefactthatthesetupleshavethesame*A*-value,wecandeduce that *c*1 = *c*2.Thus, we seethat the FD *B !C* must holdover *ABCD*whenever the FD *A!BCD*andtheMVD*B!!C*hold.If*B!C*holds,therelation*ABCD*isnotinBCNF (unless additional FDs hold that make *B* a key)!

Join Dependencies

Ajoindependencyisafurthergeneralizationof MVDs. Ajoindependency(JD) *./fR*1*;:::;Rng*issaidtoholdoverarelation*R*if*R*1*;:::;Rn* isalossless- join decomposition of *R*.

An MVD *X !! Y* over a relation *R* can be expressed as the join dependency *./ f*XY, X(R−Y)*g*. As an example, in the *CTB* relation, the MVD *C !! T* can be expressed as the join dependency *./ f*CT, CB*g*.

UnlikeFDsandMVDs,thereisnosetofsoundandcompleteinferencerulesfor JDs.

FifthNormalForm

Arelationschema*R*issaidtobeinfthnormalform(5NF)ifforevery JD *./fR*1*;: : : ; Rng* that holds over *R*, one of the following statements is true:

- *Ri=R*for some *i*,or

  The JD is implied by the set of those FDs over *R* in which the left side is a key for *R*.

The second condition deserves some explanation, since we have not presented inference rules for FDs andJDs taken together. Intuitively, we must be able to show that the decomposition of *R* into *fR*1*; : : : ; Rng* is losslessjoin whenever the key dependencies (FDs in which the left side is a key for *R*) hold. *./ fR*1*; : : : ; Rng* is a trivial JD if *Ri = R* for some *i*; such a JD always holds.

The following result, also due to Date and Fagin, identifies conditions again, detected using only FD information under which we can safely ignore JD information.

Ifarelationschemaisin3NFandeachofitskeysconsistsofasingleattribute, it is also in 5NF.

The conditions identified in this result are sufficient for a relation to be in 5NF, but not necessary. The result can be very useful in practice because it allows us to conclude that a relation is in 5NF *without ever identifying the MVDs and JDs that may holdover the relation*.

UNITV:TRANSACTIONSANDCONCURRENCYCONTROL

ACIDProperties:ConsistencyandIsolation,Atomicityand Durability, Transactions and Schedules
Serializability, Recoverability, Testing for serializability, Concurrent Execution of Transactions, Lock-Based ConcurrencyControl,PerformanceofLocking,Transaction Support in SQL. Deadlocks, Recovery and Atomicity..

Transactions are a set of operations used to perform a logical set of work. A transaction usually means that the data in the database has changed. One of the major uses of DBMS is to protect the user's data fromsystemfailures.Itisdonebyensuring thatallthedataisrestored to a consistent state when the computer is restarted after a crash. The transaction is any one execution of the user program in a DBMS. Executing the same program multiple times will generate multiple transactions.

Example –

Transaction to be performed to withdraw cash from an ATM vestibule.

Example: Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low- level tasks:

X'accont:

Open_Account(X)
Old_Balance = X.balance

New_Balance=Old_Balance-800 X.balance =

New_Balance Close_Account(X)

Y'sAccount

Open_Account(Y)

Old_Balance=Y.balance

New_Balance=Old_Balance+800 Y.balance =

New_Balance Close_Account(Y)

Operations ofTransaction:

Followingarethemainoperationsoftransaction:

Read(X): Readoperationisusedtoreadthevalueof Xfromthe database and stores it in a buffer in main memory.

Write(X): Writeoperationisusedtowritethevaluebacktothedatabase from the buffer.

Let'stakeanexampletodebittransactionfromanaccountwhichconsists of following operations:

R(X); X=

X - 500;W(X);

Let'sassumethevalueofXbeforestartingofthetransactionis4000.

- The first operation reads X's value fromdatabase and stores it in a buffer.
- Thesecondoperationwilldecreasethevalueof Xby500.So buffer will contain 3500. ○ The third operation will write the buffer's value to the database. So X's final value will be 3500.

Butitmaybepossiblethatbecauseofthefailureofhardware,software orpower,etc.thattransactionmayfailbeforefinishedalltheoperations in the set.

For example: If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

Tosolvethisproblem,wehavetwoimportantoperations:

Commit: Itisusedtosavethework donepermanently.

Rollback: Itisusedtoundotheworkdone.

Topic II:ACID Properties:

A **transaction**is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations. In order to maintain consistency in a database, before and after the transaction,certainpropertiesarefollowed.Theseare called **ACID** properties.

Atomicity:

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—**Abort**:Ifatransactionaborts,changes madetothedatabasearenot visible.

—**Commit**: If a transaction commits, changes made are visible. Atomicity is also known as the 'All or nothing rule'.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	T2
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of**T1** but before completion of**T2**.(say,after**write(X)**butbefore**write(Y)**),thentheamounthas been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the database state.

Consistency:

Thismeansthatintegrityconstraintsmustbemaintainedso thatthe database is consistent before and after the transaction. It refers to thecorrectnessofadatabase.Referringtotheexampleabove,

thetotalamountbeforeandafterthetransactionmustbe maintained.

Total **before T** occurs = **500 + 200 = 700**.

Total**afterToccurs=400+300=700**. Therefore, the database is**consistent**. Inconsistency occurs incase **T1** completes but **T2** fails. As a result, T is incomplete.

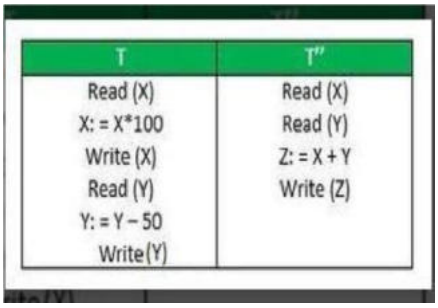
Isolation:

This property ensures that multiple transactions can occur concurrentlywithoutleadingtotheinconsistencyofthedatabasestate. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions

concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

Let **X**= 500, **Y** = 500.

Considertwotransactions**T**and**T".**



T	T''
Read (X)	Read (X)
X: = X*100	Read (Y)
Write (X)	Z: = X + Y
Read (Y)	Write (Z)
Y: = Y – 50	
Write(Y)	

Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result, interleaving of operations takes place due to which **T''** reads the correct value of **X** but the incorrect value of **Y** and sum computed by

T'': (X+Y = 50, 000+500=50, 500)

isthusnotconsistentwiththesumatendofthetransaction:

T:(X+Y=50,000+450=50,450).

Thisresultsindatabaseinconsistency,duetoalossof50units.Hence, transactionsmusttakeplaceinisolationandchangesshouldbe visible only after they have been made to the main memory.

Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

Someimportantpoints:

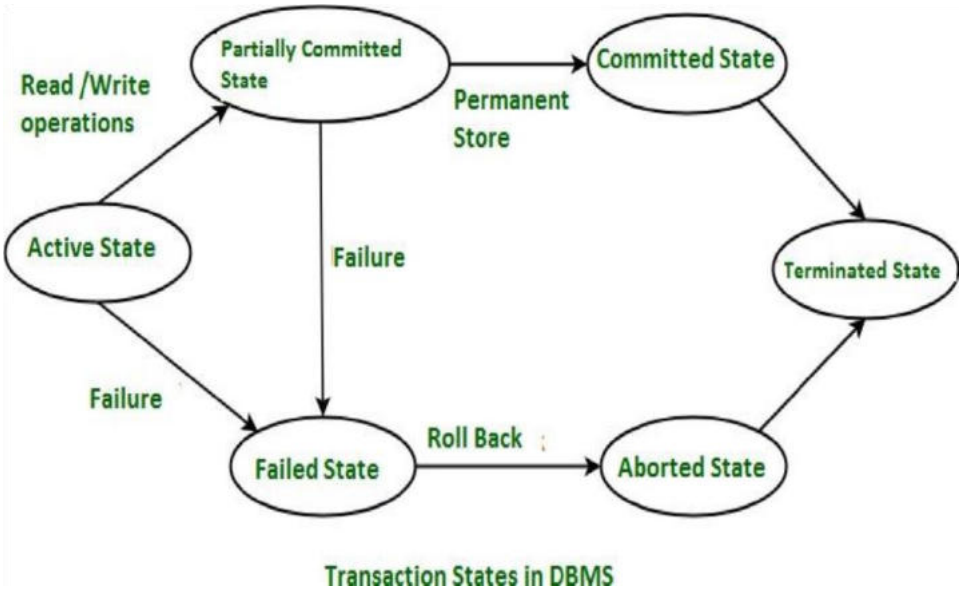
Property	**Responsibilityformaintainingproperties**
Atomicity	TransactionManager
Consistency	Applicationprogrammer
Isolation	ConcurrencyControlManager
Durability	RecoveryManager

The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit,produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

Topic III: Transactionstates:

States through which a transaction goes during its lifetime. These are thestateswhich tellaboutthecurrentstateof theTransaction and also

tellhowwewillfurtherdotheprocessinginthetransactions.These statesgoverntheruleswhichdecidethefateofthetransactionwhether it will commit or abort. They also use **Transaction log.** Transaction log is a file maintain by recovery management component to record all the activities of the transaction. After commit is done transaction log file is removed.



Transaction States in DBMS

ThesearedifferenttypesofTransactionStates:

1. **Active State –**

When the instructions of the transaction are running then the transaction is in active state. If all the 'read and write' operations are performed without any error then it goes to the "partially committed state"; if any instruction fails, it goes to the "failed state".

2. **Partially Committed –**

Aftercompletionofallthereadandwriteoperationthechangesare made in main memory or local buffer. If the changes are made permanent on the DataBase then the state will change to “committed state” and in case of failure it will go to the “failed state”.

3. **Failed State –**

When any instruction of the transaction fails, it goes to the “failed state” or if failure occurs in making a permanent change of data on Data

Base.

4. **Aborted State –**

After having any type of failure the transaction goes from “failed state” to “aborted state” and since in previous states, the changes are only made to local buffer or main memory and hence these changes are deleted or rolled-back.

5. **Committed State –**

It is the state when the changes are made permanent on the Data

Baseandthetransactioniscompleteandthereforeterminatedinthe “terminated state”.

6. **Terminated State –**

Ifthereisn’tanyroll-backorthetransactioncomesfromthe

"committed state", then the system is consistent and ready for new transactionandtheoldtransactionisterminated.

Topic4:ConcurrentExecution:

ConcurrencyControlisthe managementproceduretheatisrequiredfor controlling concurrent execution of the operations that take place on a database.

Butbeforeknowingaboutconcurrencycontrol,weshouldknowabout concurrent execution.

ConcurrentExecutioninDBMS

- Inamulti-usersystem,multipleuserscanaccessandusethesame databaseatonetime,whichisknownastheconcurrentexecution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed shouldbedoneinaninterleavedmanner,andnooperationshould affect the other executing operations, thus maintaining the consistencyofthedatabase.Thus,onmakingtheconcurrent executionofthetransactionoperations,thereoccurseveral challenging problems that need to be solved.

ProblemswithConcurrentExecution:

Inadatabasetransaction,thetwomainoperations areREADandWRITEoperations.So,thereisaneedtomanagethese two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

Problem1:LostUpdateProblems(W-WConflict)

Theproblemoccurs*whentwodifferentdatabasetransactionsperform theread/writeoperationsonthesamedatabaseitemsinaninterleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.*

Forexample:

ConsiderthebelowdiagramwheretwotransactionsTXandTY,are performed on the same account A where the balance of account A is $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- Attimet1,transaction$T_X$readsthevalueofaccountA,i.e.,$300 (only read).
- Attimet2,transaction$T_X$deducts$50fromaccountAthatbecomes $250 (only deducted and not updated/write).

- Alternately, at time t3, transaction T_Y reads the value of account A that will be $300 only because T_X didn't update the value yet.
- Attimet4,transaction$T_Y$adds$100toaccountAthatbecomes $400(onlyaddedbutnotupdated/write).
- Attime t6, transaction$T_X$writesthe valueofaccountA that will be updated as $250 only, as T_Y didn't update the value yet.
- Similarly, at time t7, transaction T_Y writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by T_X is lost, i.e., $250 is lost.

Hencedatabecomesincorrect,anddatabasesetstoinconsistent.

Problem2:DirtyReadProblems(W-RConflict)

The dirty read problem occurs *when one transaction updates an item ofthedatabase,andsomehowthetransactionfails,andbeforethedata gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.*

Forexample:

Consider two transactions TXand TYin the below diagram performing read/write operations on account A where the available balance in account A is $300:

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- Attimet1,transaction$T_X$readsthevalueofaccountA,i.e.,$300.
- Attimet2,transaction$T_X$adds$50toaccountAthatbecomes $350.
- Attime t3,transaction$T_X$writesthe updatedvalueinaccountA, i.e., $350. ○ Then at time t4, transaction T_Y reads account A that will be read as $350.
- Then at time t5, transaction T_X rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction T_Y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Problem3:UnrepeatableReadProblem(W-RConflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

Forexample:

Consider two transactions, TXand TY, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- Attimet1,transaction$T_X$readsthevaluefromaccountA,i.e., $300.
- Attimet2,transaction$T_Y$readsthevaluefromaccountA,i.e., $300.
- Attimet3,transaction$T_Y$updatesthevalueofaccountAby adding $100 to the available balance, and then it becomes $400.
- Attimet4,transaction$T_Y$writestheupdatedvalue,i.e.,$400.
- After that, at time t5, transaction T_X reads the available value of account A, and that will be read as $400.
- ItmeansthatwithinthesametransactionT_X,itreadstwodifferent valuesofaccountA,i.e.,$300initially,andafterupdationmade bytransaction T_Y, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Chapter2:RecoverySystem Topic1:

Storage Structure:

A database system provides an ultimate view of the stored data. However, data in the form of bits, bytes get stored in different storage devices.

In this section, we will take an overview of various types of storage devices that are used for accessing and storing data.

StorageStructure

Wehavealreadydescribedthestoragesystem.Inbrief,thestorage structure can be divided into two categories −

- **Volatile storage** − As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto thechipsetitself.Forexample,mainmemoryandcachememory areexamples of volatile storage. Theyarefast but can store only a small amount of information.
- **Non-volatile storage** − These memories are made to survive systemcrashes.Theyarehugeindatastoragecapacity,butslower in accessibility. Examples may include hard-disks, magnetic tapes,flash memory,andnonvolatile(batterybackedup)RAM.

Topic2:RecoveryAndAtomicity Recovery and Atomicity

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

WhenaDBMSrecoversfromacrash,itshouldmaintainthefollowing

–

- Itshouldcheckthestatesofallthetransactions,whichwerebeing executed.

- Atransactionmaybeinthemiddleofsomeoperation;theDBMS must ensure the atomicity of the transaction in this case.
- Itshouldcheckwhetherthetransactioncanbecompletednowor it needs to be rolled back.
- NotransactionswouldbeallowedtoleavetheDBMSinaninconsistent state.

Therearetwotypesoftechniques,whichcanhelpaDBMSin recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintainingshadowpaging,wherethechangesaredoneona volatile memory, and later, the actual database is updated.

Topic3:LogBasedRecovery

Thelogisasequenceofrecords.Logofeachtransactionismaintained in some stable storage so that if any failure occurs, then it can be recovered from there. If any operation is performed on the database, then it will be recorded in the log. But the process of storing the logs shouldbedonebeforetheactualtransactionisappliedinthedatabase.

Let'sassumethereisatransactionto modifytheCityofastudent.The following logs are written for this transaction.

Whenthetransactionisinitiated,thenitwrites'start'log.

<Tn,Start>

WhenthetransactionmodifiestheCityfrom 'Noida'to'Bangalore', then another log is written to the file.

<Tn,City,'Noida','Bangalore'>

When the transaction is finished, then it writes another log to indicate the end of the transaction.

Therearetwoapproachestomodifythedatabase:

1. **Deferreddatabasemodification:**

- Thedeferredmodificationtechniqueoccursifthetransaction does not modify the database until it has committed.
- In thismethod, allthelogs are created and storedin the stable storage,andthedatabaseisupdatedwhenatransactioncommits.

2. **Immediatedatabasemodification:**

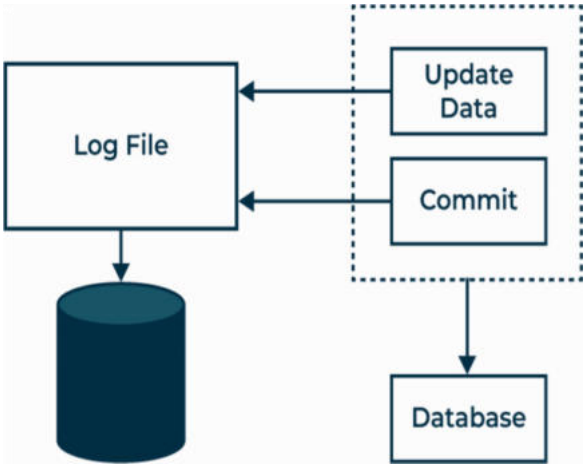
- The Immediate modification technique occurs if database modification occurs while the transaction is still active. ○

Inthistechnique,thedatabaseismodifiedimmediatelyafter every operation. It follows an actual database modification.

Topic4:

Log-BasedRecoveryinDBMS

Log-based recovery is a technique used in database management systems(DBMS)toensuredatabaseconsistencyanddurabilityincase of system failures. It relies on maintaining a **transaction log**, which records all changes made to the database, and uses this log to recover the database to a consistent state.



PhasesofLog-BasedRecovery

1. **LogAnalysis**:
   - Determines which transactions were committed and which were active at the time of the failure.
   - Markstransactionsas:
     - **Committed**:Changeswillberedone.
     - **Active/Uncommitted**:Changeswillbeundone.
2. **RedoPhase**:
   - Reapplieschangesofcommittedtransactionstoensuredurability.
   - Scansthelogforwardfromthelastcheckpointtoapplythe **afterimage**.
3. **UndoPhase**:
   - Rollsbackthechangesofuncommittedtransactionstomaintain atomicity.
   - Scansthelogbackwardfromtheendandappliesthe**before image**.

ExampleofLog-BasedRecovery Scenario:

- AdatabasecontainsadataitemAwithaninitialvalueof50.
- Twotransactions,**T1**and**T2**,updatethevalueofA:
  - **T1**changesAto60.
  - **T2**changesAto70.
  - Asystemcrashoccur safter**T1commits**butbe fore**T2commits**. **Log Entries:**

Log Sequenc e Number (LSN)	**Transactio n ID**	**Operatio n**	**Dat a Item**	**Befor e Image**	**After Imag e**	**Status**
1	T1	Update	A	50	60	Committed
2	T2	Update	A	60	70	Uncommitte d

RecoveryProcess:

1. **AnalysisPhase**:
   - Thelogisanalyzedtodeterminethat:

- **T1**iscommitted.
- **T2**isactive/uncommittedatthetimeofthecrash.

2. **RedoPhase**:
   - **T1'schanges**(A=60)arereappliedbecause**T1**was committed.
   - **T2's changes** (A = 70) are not redone because **T2** was not committed.
3. **UndoPhase**:
   - Rollsback**T2'schanges**usingthe**beforeimage**inthelog, restoring A to 60.

FinalState:

- Afterrecovery,A=60,reflectingthecommittedchangesof**T1** andignoringtheuncommittedchangesof**T2**.

Topic4:

RIESRecoveryTechniqueinDBMS

ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) is a widely used recovery technique in database management systems (DBMS) for ensuring database consistency and fault tolerance. It efficiently handles failures by supporting atomicity anddurability,twokeypropertiesofdatabasetransactionsintheACID model.

StepsofARIESRecoveryTechnique ARIESperformsrecoveryinthreephases:

1. **AnalysisPhase:**
   - Scans thelog to identifytransactions thatwere activeat thetime of failure.
   - Determinesthepointwhererecoveryshouldbegin.
   - Reconstructstransactionanddirtypageinformation.
2. **RedoPhase:**
   - Redoesallchangesmadebycommittedtransactionssincethelast checkpoint.

- Ensuresallcommittedchangesareappliedtothedatabase.

3. UndoPhase:

- Revertstheeffectsofuncommittedtransactionstomaintain database consistency.
- Usesthelogtorollbackuncommittedchangesinreverseorder.

KeyFeaturesofAriesRecoveryAlgorithminDBMS

1. **Write-Ahead Logging (WAL)**: This make sure that all changesare logged before they are applied to the database.
2. **Checkpointing**:Tocreateastablepointinthedatabasefromwhich recovery can start.
3. **ThreePhasesofRecovery**:ToensuredatabaserecoveryAnalysis, Redo, and Undo phases are crucial.