DBMS

R20

UNIT I

Introduction: Database systems applications, Purpose of Database Systems, view of Data, Database Languages, Relational Databases, Database Design, Data Storage and Querying, Transaction Management, Database Architecture, Data Mining and Information Retrieval, Specialty Databases, Database users and Administrators,

Introduction to Relational Model: Structure of Relational Databases, Database Schema, Keys, Schema Diagrams, Relational Query Languages, Relational Operations

Lecture Notes

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

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

1.2. Database-System Applications

Databases are widely used. Here are some representative applications:

Enterprise Information

Sales: For customer, product, and purchase information.

Accounting: For payments, receipts, account balances, assets and other accounting information.

Human resources: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.

Manufacturing: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.

Banking and Finance

Banking: For customer information, accounts, loans, and banking transactions.

Credit card transactions :For purchases on credit cards and generation of monthly statements.

Finance: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.

Universities: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).

Airlines: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.

Telecommunication: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

As the list illustrates, databases form an essential part of every enterprise today, storing not only types of information that are common to most enterprises, but also information that is specific to the category of the enterprise.

Over the course of the last four decades of the twentieth century, use of databases grew in all enterprises. In the early days, very few people interacted directly with database systems, although without realizing it, they interacted with databases indirectly— through printed reports such as credit card statements, or through agents such as bank tellers and airline reservation agents. Then auto- mated teller machines came along and let users interact directly with databases. Phone interfaces to computers (interactive voice- response systems) also allowed users to deal directly with databases a caller could dial a number, and press phone keys to enter information or to select alternative options, to find flight arrival/departure times, for example, or to register for courses in a university.

The Internet revolution of the late 1990s sharply increased direct user access to databases. Organizations converted many of their phone interfaces to databases into Web interfaces, and made a variety of services and information available online. For instance, when you access an online book store and browse a book or music collection, you are accessing data stored in a database. When you enter an order online, your order is stored in a database. When you access a bank Web site and retrieve your bank balance and transaction information, the information is retrieved from the bank's database system. When you access a Web site, information about you may be retrieved from a database to select which advertisements you should see. Furthermore, data about your Web accesses may be stored in a database.

Thus, although user interfaces hide details of access to a database, and most people are not even aware they are dealing with a database, accessing databases forms an essential part of almost everyone's life today.

The importance of database systems can be judged in another way today, database system vendors like Oracle are among the largest software companies in the world, and database systems form an important part of the product line of Microsoft and IBM.

Purpose of Database Systems

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

Add new students, instructors, and courses

Register students for courses and generate class rosters

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

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

introduction to file-processing system has a number of major disadvantages:

Data redundancy and inconsistency. Since different programmers create the files and application programs over a long period, the various files are likely to have different structures and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics) the address and telephone number of that student may appear in a file that consists of student records of students in the Music

department and in a file that consists of student records of students in the Mathematics department. This redundancy leads to higher storage and access cost. In addition, it may lead to data inconsistency; that is, the various copies of the same data may no longer agree. For example, a changed student address may be reflected in the Music department records but not elsewhere in the system.

Difficultyinaccessingdata.Supposethatoneoftheuniversityclerksneedsto find out the names of all students who live within a particular postal- code area. The clerk asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of all students. The university clerk has now two choices: either obtain the list of all students and extract the needed information manually or ask a programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same clerk needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist. Again, the clerk has the preceding two options, neither of which is satisfactory.

The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

Data isolation. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

Integrity problems. The data values stored in the database must satisfy certain types of consistency constraints. Suppose the university maintains an account for each department, and records the balance amount in each ac- count. 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 pro- grams. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

Atomicity problems. A computer system, like any other device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $500 from the account balance of department A to the account balance of department B. If a system failure occurs during the execution of the program, it is possible that the $500 was removed from the balance of department A but was not credited to the balance of department B, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be atomic

— it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

Concurrent-access anomalies. For the sake of overall performance of the sys- tem and faster response, many systems allow multiple users to update the data simultaneously. Indeed, today, the largest Internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider department A, with an account balance of
$10,000. If two department clerks debit the account balance (by say $500 and $100, re- spectively) of department A at almost exactly the same time, the result of the concurrent executions may leave the budget in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $10,000, and write back
$9500 and $9900, respectively. Depending on which one writes the value last, the account balance of department A may contain either $9500or$9900, rather than the correct value of $9400. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

As another example, suppose a registration program maintains a count of students registered for a course, in order to enforce limits on the number of students registered. When a student registers, the program reads the current count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at (say) 39. The two program executions may both read the value 39, and both would then write back 40, leading to an incorrect increase of only 1, even though two studentssuccessfullyregisteredforthecourseandthecountshouldbe41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able to register, leading to a violation of the limit of 40 students.

Security problems. Not every user of the database system should be able to access all the data. For example, in a university, payroll personnel need to see only that part of the database that has financial information. They do not need access to information about academic records. But, since application programs are added to the file-processing system in an ad hoc manner, enforcing such security constraints is difficult. These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems. In most of this book, we use a university organization as a running example of a typical data-processing application.

1.3. View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained.

Data Abstraction

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

Physical level. The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.

Logical level. The next-higher level of abstraction describes what data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as physical data independence. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

View level. The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

Figure 1.1 shows the relationship among the three levels of abstraction .An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programming.

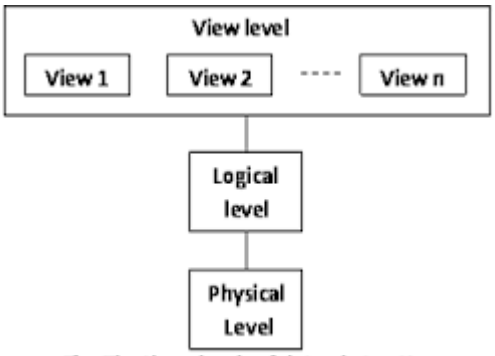


Fig: *The three levels of data abstraction.*

Languages support the notion of a structured type. For example, we may describe a record as follows:1

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

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

department, with fields dept name, building, and budget

course,withfieldscourseid,title,deptname,andcredits

student,withfieldsID,name,deptname,andtotcred

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

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

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

Instances and Schemas

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

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

Database systems have several schemas, partitioned according to the levels of abstraction. The physical schema describes the database design at the physical level, while the logical schema describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called subschema as, that describe different views of the database.

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs.

Application programs are said to exhibit physical data independence if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

Data Models

Underlying the structure of a database is the data model: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. A data model provides a way to describe the design of a database at the physical, logical, and view levels. There are a number of different data models that we shall cover in the text.

The data models can be classified into four different categories:

Relational Model. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as relations. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a par-ticular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

Entity-Relationship Model. The entity-relationship(E-R) data model uses a collection of basic objects, called entities ,and relationships among these objects An entity is a “thing” or “object” in the real world that is distinguishable from other objects.

Object-Based Data Model. Object-oriented programming(especially in Java, C++, or C#) has become the dominant software- development methodology. This led to the development of

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

Semi structured Data Model.

The semi structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item particular type must have the same set of attributes The Extensible Markup Language (XML)is widely used to represent semi structured data.

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

1.4. Database Languages

A database system provides a data-definition language to specify the database schema and a data-manipulation language to express database queries and up-dates. In practice, the data-definition and data- manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

Data-Manipulation Language

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

Retrieval of information stored in the database

Insertion of new information into the database

Deletion of information from the database

Modification of information stored in the database There is basically **two types:**

Procedural DMLs require a user to specify what data are needed and how to get those data.

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

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data.

A query is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a query language. Although technically incorrect, it is common practice to use the terms query language and data-manipulation languages The levels of abstraction that apply not only to defining or structuring data, but also to manipulating data. At the physical level, we must define algorithms that allow efficient access

to data. At higher levels of abstraction, we emphasize ease of use. The goal is to allow humans to interact efficiently with the system.

Data-Definition Language

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 specifyadditionalpropertiesofthedata.Wespecifythestoragestructureand access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schemas, which are usually hidden from theusers.

Thedatavaluesstoredinthedatabasemustsatisfycertainconsistencycon- straints. 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 per- tainingto the database. However, arbitrary predicates may be costly to test. Thus, database systems implement integrity constraints that can be tested with minimaloverhead:

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

ReferentialIntegrity.Therearecaseswherewewishtoensurethatavalue that appears in one relation for a given set of attributes also appears in a cer- tain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists. More precisely, the dept name value in a course record must appear in the dept name attribute of some record of the department relation. Database modifications can cause violations of referential integrity. When a referential-integrity con- straintis violated, the normal procedure is toreject the action that caused theviolation.

Assertions. An assertion is any condition that the database must alwayssatisfy. Domain constraints and referential-integrity constraints are special forms of assertions. However, there are many constraints that we cannot expressbyusingonlythesespecialforms.Forexample,“Everydepartment musthaveatleastfivecoursesofferedeverysemester”mustbeexpressed as an assertion. When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modificationto thedatabaseisallowedonlyifitdoesnotcausethatassertiontobeviolated.

Authorization. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are

expressed in terms of authorization, the most common being: read authorization, which allows reading, but not modification, of data; insert authorization, which allows insertion of new data, but not mod- ificationof existing data; update authorization, which allows modification, but not deletion, of data; and delete authorization, which allows deletion of data. We may assign the user all, none, or a combinationofthesetypesofauthorization.TheDDL,justlikeanyotherdb languages.

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

Relational Databases

A relational database is based on the relational model and uses a collection of tables to represent both data and the relationships among those data. It also in- cludesa DML andDDL.

Tables

Each table has multiple columns and each column has a unique name. Figure 1.2 presents a sample relational database comprising two tables: one shows details of university instructors and the other shows details of the various universitydepartments.

The first table, the instructor table, shows, for example, that an instructor named Einstein with ID 22222 is a member of the Physics department and has an annual salary of $95,000. The second table, department, shows, for example, that the Biology department is located in the Watson building and has a budgetof$90,000. Of course, a real-world university would have many more departments and instructors. We use small tables in the text toillustrate concepts. A larger example for the same schema is available online.

The relational model is an example of a record-based model. Record-basedmodelsaresonamedbecausethedatabaseisstructured in fixed-format records of several types. Each table contains records of aparticular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the recordtype.

It is not hard to see how tables may be stored in files. For instance, a special character (such as a comma) may be used to delimit the different attributes of a record, and another

special character (such as a new-line character) may be used to delimit records. The relational model hides such low-level implementation details from database developers andusers.

We also note that it is possible to create schemas in the relational model that have problems such as unnecessarily duplicated information. For example, sup- pose we store thedepartment budget as an attribute ofthe instructor record. Then, whenever the value ofa particular budget (saythat one for the Physics depart- ment) changes,thatchange must to be reflected in the records of all instructors associated with the Physics department.

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

The instructor table

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

The department table

Figure 1.2 A sample relational database.

Data-ManipulationLanguage

The SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table. Here is an example of an SQLquerythatfindsthenamesofallinstructorsintheHistorydepartment:

```
select instructor.name from instructor
where instructor.dept name = 'History';
```

The query specifies that those rows from the table instructor where the dept name is History must be retrieved, and the name attribute of these rows must bedisplayed. More specifically, the result of executing thisquery is a table with a single column labeled name, and a set of rows, each of which contains thenameofaninstructorwhosedeptname,isHistory.Ifthequeryisrunonthe table in Figure 1.2, the result will consist of two rows, one with the name ElSaidandtheotherwiththenameCalifieri.

Queries may involve information from more than one table. For instance, thefollowing query finds the instructor ID and department name of all instructorsassociatedwithadepartmentwithbudgetofgreaterthan$95,000.

```
select instructor.ID, department.dept name from instructor, department
where instructor.deptnam= department.dept name and department.budget>95000;
```

If the above query were run on the tables in Figure 1.2, the system would find that there are two departments with budget of greater than $95,000 — ComputerScience and Finance; there are five instructors in these departments. Thus, the result will consist of a table with two columns (ID, dept name) and five rows: (12121, Finance), (45565, Computer Science), (10101, Computer Science),(83821,ComputerScience),and(76543,Finance).

Data-DefinitionLanguage

SQL provides a rich DDL that allows one to define tables, integrity constraints, assertions, etc.

For instance, the following SQL DDL statement defines the department table:

```
create table department (deptnamechar (20), buildingchar
(15),
budget numeric(12,2));
```

Execution of the above DDL statement creates the department table with three columns: dept name, building, and budget, each of which has a specific data type associated with it. In addition, the DDL statement updates the data dictionary, which contains metadata .The schema of a table is an example of metadata.

Database Access from ApplicationPrograms

SQL is not as powerful as a universal Turing machine; that is, there are some computations that are possible using a general-purpose programming language but are not possible using SQL. SQL also does not support actions such as input from users, output to displays, or communication over the network. Such computations and actions must be written in a host language, such as C, C++, or Java, with embedded SQL queries that access the data in the database. Application programs are programs that are used to interact with the database in this fashion. Examples in a university system are programs that allow students to register for courses, generate class rosters, calculate student GPA, generate payroll checks ,etc.

Relational Databases

A relational database is based on the relational model and uses a collection of tables to represent both data and the relationships among those data. It also includes a DML and DDL.

Tables

Each table has multiple columns and each column has a unique name. Figure 1.2 presents a sample relational database comprising two tables: one shows details of university instructors and the other shows details of the various university departments.

The first table, the instructor table, shows, for example, that an instructor named Einstein with ID 22222 is a member of the Physics department and has an annual salary of $95,000. The second table, department, shows, for example, that the Biology department is located in the Watson building and has a budget of

$90,000. Of course, a real-world university would have many more departments and instructors. We use small tables in the text to illustrate concepts. A larger example for the same schema is available online.

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

It is not hard to see how tables may be stored in files. For instance, a special character (such as a comma) may be used to delimit the different attributes of a record, and another special character (such as a new-line character) may be used to delimit records. The relational model hides such low-level implementation details from database developers and users.

We also note that it is possible to create schemas in the relational model that have problems such as unnecessarily duplicated information. For example, sup- pose we store the department budget as an attribute of the instructor record. Then, whenever the value of a particular budget (say that one for the Physics department) changes ,that change must to be reflected in the records of all instructors associated with the Physics department.

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

The instructor table

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

The department table

Data-ManipulationLanguage

The SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table. Here is an example of an SQLquerythatfindsthenamesofallinstructorsintheHistorydepartment:

select instructor.name from instructor

where instructor.dept name = 'History';

The query specifies that those rows from the table instructor where the dept name is History must be retrieved, and the name attribute of these rows must be displayed. More specifically, the result of executing thisquery is a table with a single column labeled name, and a set of rows, each of which contains thenameofaninstructorwhosedeptname,isHistory.Ifthequeryisrunon the table in Figure 1.2, the result will consist of two rows, one with the name ElSaidandtheotherwiththenameCalifieri.

Queries may involve information from more than one table. For instance, thefollowing query finds the instructor ID and department name of all instructorsassociatedwithadepartmentwithbudgetofgreaterthan$95,000.

select instructor.ID, department.dept name from instructor, department
where instructor.deptnam= department.dept name and department.budget>95000;

If the above query were run on the tables in Figure 1.2, the system would find that there are two departments with budget of greater than $95,000 — ComputerScience and Finance; there are five instructors in these departments. Thus, the result will consist of a table with two columns (ID, dept name) and five rows: (12121, Finance), (45565, Computer Science), (10101, Computer Science),(83821,ComputerScience),and(76543,Finance).

SQL provides a rich DDL that allows one to define tables, integrity constraints, assertions, etc.

For instance, the following SQL DDL statement defines the department table:

create table department (deptnamechar (20), buildingchar
(15),
budget numeric(12,2));

Execution of the above DDL statement creates the department table with three columns: dept name, building, and budget, each of which has a specific data type associated with it. In addition, the DDL statement updates the data dictionary, which contains metadata .The schema of a table is an example of metadata.

Database Access from Application Programs

SQL is not as powerful as a universal Turing machine; that is, there are some computations that are possible using a general-purpose programming language but are not possible

using SQL. SQL also does not support actions such as input from users, output to displays, or communication over the network. Such com- putationsand actions must be written in a host language, such as C, C++, or Java, with embedded SQL queries thataccess the data in the database. Application programs areprograms that are used to interact with the database in this fashion.Examplesin a university system are programs that allow students to register for courses, generate class rosters, calculate student GPA, generate payroll checks,etc.

To access the database, DML statements need to be executed from thehost language. There are two ways to dothis:

By providing an application program interface (set of procedures) that can be used to send DML and DDL statements to the database and retrieve theresults.

The Open Database Connectivity (ODBC) standard for usewith the C language is a commonly used application program interface standard. The Java Database Connectivity (JDBC) standard provides corresponding features to the Javalanguage.

By extending the host language syntax toembed DML calls within the host language program. Usually, a special character prefaces DML calls, and a preprocessor, called the DML precompiler, converts the DML statements to normal procedure calls in the hostlanguage.

1.5 .Database Design

Database systems are designed to manage large bodies of information. These large bodies of information do not exist in isolation. They are partof the operation of some enterprise whose end product may be information from the database or may be some device or service for which the database plays only a supportingrole.

Databasedesignmainlyinvolvesthedesignofthedatabaseschema. The design of a complete database application environment that meets the needs of the enterprise being modeled requires attention to a broader set of issues. In this text, we focus initially on the writing of databasequeriesandthedesignofdatabaseschema.

DesignProcess

A high-level data model provides the database designer with a conceptual frame- work in which to specify the data requirements of the database users, and how the database will be structured to fulfill these requirements. The initial phase of database design, then, is to characterize fully the data needs of the prospective database users.The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of userrequirements.

Next, the designer chooses a data model, and by applying theconcepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this conceptual-design phase provides a detailed overview of the enterprise. The designer reviews theschema toconfirm that all data requirements areindeed satisfied and are not in conflict with oneanother. The designercanalso examine the design to remove anyredundantfeatures. The focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.

In terms of the relational model, the conceptual-design process involves de- cisions on what attributes we want to capture in the database and how to group theseattributestoformthevarioustables.The"what"partisbasically abusinessdecision,andweshallnotdiscussitfurtherinthistext.The"how" part is mainly a computer-science problem. There are principally two ways to tackle the problem. The first one is to use the entity-relationship model (Section 1.6.3); the other is to employ a set of algorithms(collectively known as normalization) that takes as input the set of all attributes and generates a set of tables (Section1.6.4).

A fully developed conceptual schema indicates the functional requirements oftheenterprise.Inaspecificationoffunctionalrequirements,usersdescribe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure it meets functionalrequirements.

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases. In the logical-design phase, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The designer uses the resulting system-specific database schema in the subsequent physical-design phase, in which the physical features of the database are specified.

Database Design for a University Organization

To illustrate the design process, let us examine how a database for a university could be designed. The initial specification of user requirements may be based on interviews with the database users, and on the designer's own analysis of the organization. The description that arises from this design phase serves as the basis for specifying the conceptual structure of the database. Here are the major characteristics of theuniversity.

The university is organized into departments. Each department is identifiedby a unique name (dept name), is located in a particular building, and has abudget.

Each department has a list of courses it offers. Each course has associated with it a course id, title, dept name, and credits, and may also have have associatedprerequisites.

Instructors are identified by their unique ID. Each instructor has name, associateddepartment (dept name), andsalary.

Students are identified by their unique ID. Each student has a name, an associatedmajordepartment(deptname),andtotcred(totalcredithours

the student earned thus far).The university maintains a list of classrooms, specifying the name of the

building, room number, and room capacity.

The university maintains a list of all classes (sections) taught. Each section is identified by a course id, sec id, year, and semester, and has associated with it a semester, year, building, room number, and times lot id(the times lot when the classmeets).

The department has a list of teaching assignments specifying, for each in- structor, the sections the instructor is teaching.

The university has a list of all student course registrations, specifying ,for each student, the courses and the associated sections the student has taken(registered for).

A real university database would be much more complex than the preceding design. However we use this simplified model to help you understand conceptual ideas without getting lost in details of a complex design.

The Entity-Relationship Model

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

Entities are described in a database by a set of attributes. For example, the attributes dept name, building, and budget may describe one particular department in a university, and they form attributes of the department entity set. Similarly, attributes ID, name, and salary may describe an instructorentity.2

The extra attribute ID is used to identify an instructor uniquely(since it may be possible to have two instructors with the same name and the same salary). A unique instructor identifier must be assigned to each instructor. In the United States, many organizations use the social-security number of a person (a unique number the U.S. government assigns to every person in the United States) as a unique identifier.

A relationship is an association among several entities. For example, a member relationship associates an instructor with her department. The set of all entities of the same type and the set of all relationships of the same type are termed an entity set and relationship set, respectively.

The overall logical structure (schema) of a database can be expressed graph- ically by an entity-relationship (E-R) diagram. There are several ways in which to draw these diagrams. One of the most

popular is to use the Unified Modeling Language (UML). In the notation we use, which is based on UML, an E-R diagram is represented as follows:

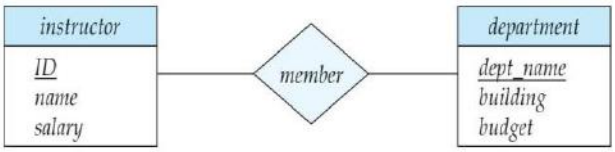


Figure 1.3 A sample E-R diagram.

Entity sets are represented by a rectangular box with the entity set name in the header and the attributes listed below it.

Relationship sets are represented by a diamond connecting a pair of related entity sets. The name of the relationship is placed inside the diamond.

As an illustration, consider part of a university database consisting of instructors and the departments with which they are associated. Figure 1.3 shows the corresponding E-R diagram. The E-R diagram indicates that there are two entity sets, instructor and department, with attributes as outlined earlier. The diagram also shows a relationship member between instructor and department.

In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is mapping cardinalities, which express the number of entities to which another entity can be associated via a relationship set. For example, if each instructor must be associated with only a single department, the E-R model can express that constraint.

Normalization

Another method for designing a relational database is to use a process commonly known as normalization. The goal is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. The approach is to design schemas that are in an appropriate normal form. To determine whether a relation schema is in one of the desirable normal forms, we need additional information about the real-world enterprise that we are modeling with the database.

To understand the need for normalization, let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may haveare:

Repetition ofinformation

Inability to represent certaininformation

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

Figure1.4 The facultytable.

We shall discuss these problems with the help of a modified database design forour universityexample.

Suppose that instead of having the two separate tables instructor and depart- ment, we have a single table, faculty, that combines the information fromthe two tables (as shown in Figure 1.4). Notice that thereare two rows in faculty that contain repeated information about the History department, specifically, that department's building and budget. The repetitionof information in our alterna- tivedesign is undesirable. Repeating information wastes space. Furthermore, it complicates updating the database. Suppose that we wish to change the budget amount of the History department from $50,000 to $46,800. This change must be reflected in the two rows; contrastthis with the original design, where this requires an update to only a single row. Thus, updates are more costly under the alternative design than under the original design. When we perform the update in the alternative database, we must ensure that every tuple pertaining to the His- tory department is updated, or elseour database will show two different budget values for the History department.

Now,letusshiftourattentiontotheissueof"inabilitytorepresentcertain information."Supposewearecreatinganewdepartmentintheuniversity.In the alternative design

above, we cannot represent directly the information concerning a department (dept name, building, budget) unless that department has at least one instructor at the university. This is because rows in the faculty table require values for ID, name, and salary. This means that wecannotrecord information about the newly created department until the first instructor ishiredforthenewdepartment.

One solution to this problem is to introduce null values. The null value indicates that the value does not exist (or is not known). An unknown value maybe either missing (the value does exist, but we do not have that information) or not known (we do not know whether or not the value actually exists). Asweshall see later, null values are difficult tohandle, and it is preferable not to resort to them. If we are not willing to deal with null values, then we can create a particular item of department information only when the department has at least one instructor associated with the department. Furthermore, we would have to delete this information when the last instructor in the department departs. Clearly, thissituation is undesirable, since, under our original database design, the department information would be available regardless of whether or not there is an instructor associated with the department, and without resorting to nullvalues.

An extensive theory of normalization has been developed that helps formally define what database designs are undesirable, and how to obtain desirable de- signs.

1.5. Data Storage and Querying

A database system is partitioned into modules that deal with each of the re- sponsibilitiesof the overall system. The functional components ofa database system can be broadly divided into the storage manager and the query processorcomponents.

The storage manager is important because databases typically require a large amount of storage space. Corporate databases range in size from hundreds of gigabytes to, for the largest databases, terabytes of data. A gigabyte is approxi- mately 1000 megabytes (actually 1024) (1 billion bytes), and a terabyte is 1 million megabytes(1 trillion bytes). Since the main memory of computers cannot store this much information, the information is stored on disks. Data are moved be- tween disk storage and main memory as needed. Since the movement of data to and from disk is slow relative to the speed of the central processing unit, it is imperative that the database system structure the data so as to minimize the need to move data between disk and main memory.

The query processor is important because it helps the database system to simplify and facilitate access to data. The query processor allows database users to obtain good performance while being able to work at the view level and not be burdened with understanding the physical-level details of theimplementation of the system. It is the job ofthe database system to translate updates and queries written inanonprocedural language, at the logical level, into an efficient sequence of operations at the physicallevel.

Storage Manager

The storage manager is the component of a database system that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. The storage manager is respon- sible for the interaction with the file manager. The raw data are stored on the disk using the file system provided by the operating system. The storage man- ager translates the various DML statements into low-level file-system commands.

Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

Authorization and integrity manager, which tests for the satisfaction of integrityconstraintsandchecks theauthorityofuserstoaccessdata.

Transactionmanager,whichensuresthatthedatabaseremainsinaconsis- tent (correct) state despite system failures, and that concurrent transaction executions proceed withoutconflicting.

File manager, which managesthe allocation of space on disk storage and thedatastructuresusedtorepresentinformationstoredondisk.

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

The storage manager implements several data structures as part of the phys- icalsystemimplementation:

Datafiles,whichstorethedatabaseitself.

Data dictionary, which stores metadata about the structure of thedatabase, in particular the schema of thedatabase.

Indices, which can provide fast access to data items. Like the index in thistextbook, a database index provides pointers to those data items that hold aparticular value. For example, we could use an index to find the instructor record with a particular ID, or all instructor records with a particular name. Hashing is an alternative to indexing that isfaster in some but not allcases.

The Query Processor

The query processor components include:

DDL interpreter,whichinterpretsDDL statementsandrecordsthedefinitionsin the datadictionary.

DML compiler, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engineunderstands.

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 DMLcompiler.

1.6. TransactionManagement

Often, several operations on the database form a single logical unit of work. An example is a funds transfer, as in Section 1.2, in which one department account (say A) is debited and another department account (say B) is credited. Clearly, it is essential that either both the credit and debit occur, or that neither occur. That is, the funds transfer must happen in its entirety or not at all. This all-or-none requirementis called atomicity. In addition, it is essential that the execution of the funds transfer preserve the consistency of the database. That is, the value of the sum of the balances of A and B must be preserved. This correctness requirement is called consistency. Finally, after the successful execution of a funds transfer, the new values of the balances of accounts A and B must persist, despite the possibility of system failure. This persistence requirement is calleddurability.

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

It is the programmer's responsibility to define properly the various transac- tions, so that each preserves the consistency of the database. For example, the transaction to transfer funds from the account ofdepartment A to the account of department B could be defined to be composed of two separate programs: one that debits account A, and another that credits account B. The execution of these two programs one after the other will indeed preserve

consistency. However, each program by itself does not transform the database from a consistent state to a new consistent state. Thus, those programs are not transactions.

Ensuring the atomicity and durability properties is the responsibility ofthedatabasesystemitself—specifically,oftherecoverymanager.In the absence of failures, all transactions complete successfully, and atomicity is achieved easily.

However, because of various types of failure, a transactionmay not always com- pleteits execution successfully. If we are to ensure the atomicity property, afailed transaction must have no effect on the state of the database. Thus, the database must be restored to the state in which it was before the transaction in question started executing. The database system must therefore perform failure recovery, that is, detect system failures and restore thedatabase to the state that existed prior to the occurrence of thefailure.

Finally, when several transactions update the database concurrently, the con- sistency of data may no longer be preserved, even though eachindividual transac- tionis correct. It is the responsibility of the concurrency-control manager to con- trolthe interaction among the concurrent transactions, to ensure the consistency of the database. The transaction manager consists of the concurrency-control manager and the recoverymanager.

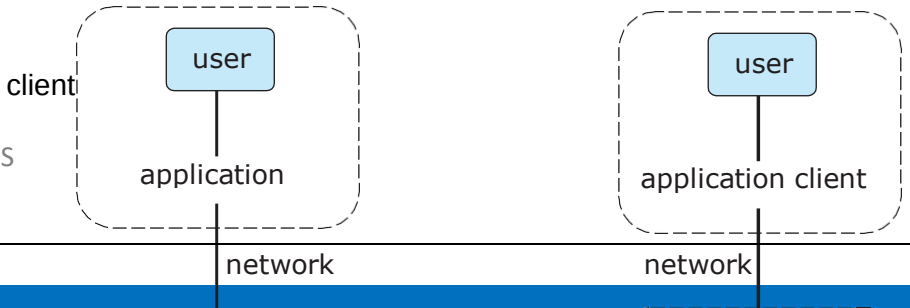
1.6. DatabaseArchitecture

We are now in a positionto provide a single picture (Figure 1.5) of the various components of a database system and the connections amongthem.

The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or client-server, where oneserver machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit par- allelcomputer architectures. Distributed databases span multiple geographically separatedmachines.

Most users of a database system today are not present at the site of the database system, but connect to it through a network. Wecan

therefore differen- tiatebetween client machines, on which remote database users work, and server machines, on which the database system runs.Databaseapplications are usually partitioned into two or three parts, as in Figure 1.6. In a two-tier architecture, the application resides at the client machine, where it invokes database system functionality at the server machine through



server

(a)Two-tierarchitecture (b) Three-tier architecture

Figure 1.6 Two-tier and three-tier architectures.

query language statements. Application program interface standards like
ODBC
and JDBC are used for interaction between the client and the server.
In contrast, in a three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls. Instead, theclient end communicates withan application server, usually through a forms interface. The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead ofbeing distributed across multiple clients. Three-tier applications aremore appropriate for large applications, and for applications that run on theWorld WideWeb.

Data Mining and InformationRetrieval

The term data mining refers loosely to the process ofsemiautomatically analyzing large databases to find useful patterns. Like knowledge discovery in artificial intelligence (also called machine learning) or statistical analysis, data mining attempts to discover rules and patterns from data. However, data mining differs from machine learning and statistics in that it deals with large volumes of data, storedprimarily

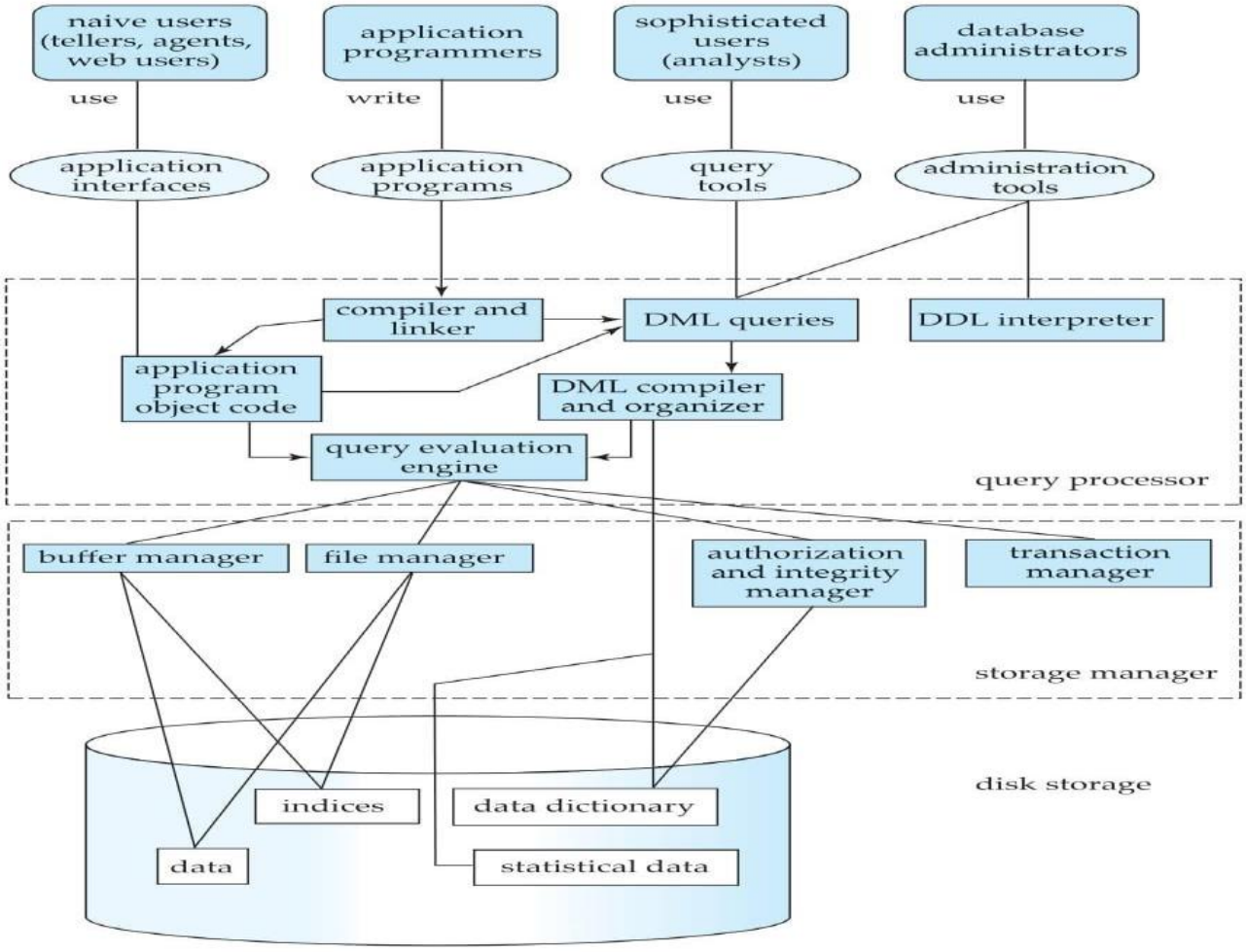


Figure 1.5 System structure.

ondisk.Thatis,dataminingdealswith“knowledgediscoveryin databases.”

Some types ofknowledge discovered from a database can be represented by a set of rules. The following is an example of a rule, statedinformally: “Youngwomenwithannualincomesgreaterthan $50,000arethemostlikelypeopletobuysmallsportscars.”Ofcoursesuch rules are not universally true, but rather have degrees of “support”and“confidence.”Othertypesofknowledgearerepresented by equations relating different variables to each other, or by other mechanisms for predicting outcomes when the values of some variables areknown.

There are a variety of possible typesof patterns that may be useful, and different techniques are used to find different types of patterns.

Usually there is a manual component to data mining, consisting of preprocess- ingdata to a form acceptable to the algorithms, and postprocessing of discovered patterns to find novel ones that could be useful. There may also be more than one type of pattern that can be discovered from a given database, and manual interaction may be needed to pick useful types of patterns. For this reason, data mining is really a semiautomatic process in real life. However, in our description we concentrate on the automatic aspect of mining.

Businesseshavebeguntoexploittheburgeoningdataonlinetomake better decisions about their activities, such as whatitemstostock and how best to target customers toincrease sales. Manyoftheir queries are rather complicated, however, and certaintypesofinformationcannotbeextractedevenbyusingSQL.Severaltechniquesand tools are available to help with decisionsupport.

Several tools for data analysis allow analysts to view data in different ways. Other analysis tools precompute summaries of very large amounts of data, in order to give fast responses to queries. The SQL standard contains additional constructs to support dataanalysis.

Large companies have diverse sources of data that they need to use for making business decisions. To execute queries efficiently on such diverse data, companies have built data warehouses. Data warehouses gather data from multiple sources under a unified schema, ata single site. Thus, they provide the user a single uniform interfaceto data.

Textual data, too, has grown explosively. Textual data is unstructured, unlike the rigidly structured data in relational databases. Querying of unstructured textual data is referred to as information retrieval. Information retrieval systems have much in common with database systems — in particular, the storage and retrieval of data on secondary storage. However, the emphasis in the field of information systems is different from that in database systems, concentrating on issuessuchasqueryingbasedonkeywords;therelevanceofdocuments to the query; and the analysis, classification, and indexing ofdocuments.

1.7. SpecialtyDatabases

Several application areas for database systems are limited by the restrictions of the relational data model. As a result, researchers have developed several data models to deal with these application domains, including object-based data models and semistructureddata models.

Object-Based Data Models

Object-oriented programming has become the dominant software- development methodology. This led to the development of an object- oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. Inheritance, object identity, andencapsulation (information hiding), with

methods to provide an interface to objects, are among the key concepts of object-oriented programming that have found applications in data modeling. The object-oriented data model also supports a rich type system, including structured and collection types. In the 1980s, severaldatabasesystemsbasedontheobject-orienteddatamodelwere developed.

The major database vendors presently support theobject-relational data model, a data model that combines features of the object-oriented data model and relational data model. It extends the traditional relational model with a variety of features such as structured and collection types, as well as objectorientation.

Semi structured Data Models

Semistructureddata models permit the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast with the data models mentioned earlier, where every data item of a particular type must have the same set of attributes.

The XML language was initially designed as a way of adding markupinfor- mationto text documents, but has become important because of its applications in data exchange. XML provides a way to represent data that have nested structure, and furthermore allows a great deal of flexibility in structuring of data, which is important for certain kinds of nontraditionaldata.

1.8. Database Users and Administrators

A primary goal of a database system is to retrieve informationfrom andstorenewinformationintothedatabase.Peoplewhoworkwitha
database can be categorized as database users or database administrators.

Database Users and UserInterfaces

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

Naıveusers are unsophisticated users who interact with the system by in- vokingone of the application programs that have been writtenpreviously. For example, a clerk inthe universitywho needs to add a new instructor to department A invokes a program called new hire. This program asks the clerk for the name of the new instructor, her new ID, the name of the department (that is, A), andthe salary.

The typical user interface for na¨ıve users is a forms interface, where the user can fill in appropriate fields of the form. Na¨ıve users may also simply read reports generated from thedatabase.

As another example, consider a student, who during class registration period, wishes to register for a class by using a Web interface. Such a userconnects to a Web application program that runs at a Web server. The appli- cation first verifies the identity of the user, and allows her to access a form where she enters the desired information. The form information is sent back to the Web application at the server, which then determines if there is room in the class (by retrieving information from the database) and if so adds the student information to the class roster in thedatabase.

Application programmers are computer professionals who write applicationprograms. Application programmers canchoose frommany tools to develop user interfaces. Rapid application development (RAD) tools are tools that en- ablean application programmer to construct forms and reports with minimal programmingeffort.

Sophisticated users interact with the system without writing programs. In-stead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submitqueries to explore data in the database fall in thiscategory.

Specialized users are sophisticated users who write specialized databaseapplications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, knowledge- base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

Database Administrator

One of the main reasons for usingDBMSs is to have central control of both the data and the programs that access those data. A person whohas such central control over the system is called a database administrator (DBA). The functions of a DBAinclude:

Schema definition. The DBA creates the original database schema by executingasetofdatadefinitionstatementsintheDDL.

Storage structureandaccess-method definition.

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

Granting of authorization for data access. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access.

The authorization information is kept in a special system structure that thedatabasesystemconsultswheneversomeoneattemptstoaccess the data in thesystem.

Routine maintenance. Examples of the database administrator's routine maintenance activitiesare:

Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such asflooding.

Ensuring that enough free disk space is available for normal operations, and upgrading disk space asrequired.

Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by someusers.

History of Database Systems

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

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

1950s and early 1960s: Magnetic tapes were developed for data storage. Data processing tasks such as payroll were automated, with data stored on tapes. Processing of data consisted of reading data from one or more tapes and writing data to a new tape. Data could also be input from punched card decks, and output to printers.

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

Late 1960s and 1970s: Widespread use of hard disks in the late1960s changed the scenario for data processing greatly, since hard disks allowed direct access to data. The position of data on disk was immaterial, since any location on disk could be accessed in just tens of milliseconds. Data were thus freedfrom the tyranny of sequentiality. With disks, network and hierarchical databases could be created that allowed data structures such as lists and trees to be stored on disk. Programmers could construct and manipulate these data structures.

A landmark paper by Codd [1970] defined the relational model and nonprocedural ways of querying data in the relational model, and relationaldatabaseswereborn.Thesimplicityoftherelationalmodeland the possibility of hiding

implementation details completely from the programmer were enticing indeed. Codd later won the prestigious Association of Computing Machinery Turing Award for hiswork.

1980s: Although academically interesting, the relational model wasnot used in practice initially, because of its perceived performance disadvantages; rela- tionaldatabases could not match the performance of existing network and hi- erarchical databases. That changed with System R, a groundbreaking project at IBM Research that developed techniques for the construction of an efficient relational database system. Excellent overviews of System R are provided by Astrahanet al. [1976] and Chamberlin et al. [1981]. The fully functional Sys- temR prototype led to IBM's first relational database product, SQL/DS. At the same time, theIngres system was being developed at the University of California at Berkeley. It led to a commercial product of the same name. Ini- tialcommercial relational database systems, such as IBM DB2, Oracle, Ingres, and DEC Rdb, played a major role in advancing techniques forefficient pro- cessingof declarative queries. By the early 1980s, relationaldatabases had become competitive with network and hierarchical database systems even in the area of performance. Relational databases were so easy to use that they eventually replaced network and hierarchical databases; programmers using such databases were forced to deal with many low- level implementation de- tails, and had to code their queries in aprocedural fashion. Most importantly, they had to keep efficiency in mind when designing their programs, which involved a lot of effort. In contrast, in a relational database, almost all these low-level tasks are carried out automatically by the database, leaving the programmer free to work at a logical level. Since attaining dominance in the 1980s, the relational model has reigned supreme among datamodels.

The 1980s alsosaw muchresearch on parallel and distributed databases, as well as initial work on object-oriented databases.

Early1990s:TheSQLlanguagewasdesignedprimarilyfordecisionsupportapplications, which are query-intensive, yet the mainstay of databases in the 1980s was transaction-processing applications, which are update- intensive. Decision support and querying re-emerged as a major application area for databases. Tools for analyzing large amounts of data saw large growths inusage.

Many database vendors introduced parallel database products in this period. Database vendors also began to add object-relational support to their databases.

1990s: The major event of the 1990s was the explosive growth of the World Wide Web. Databases were deployed muchmore extensivelythaneverbefore.D×atabasesystemsnowhadtosupport very high transaction-procesng rates, as well as veryhigh reliability and 247 availability (availability 24hours a day, 7 days a week, meaning no downtime for scheduled maintenance activities). Database systems also had to support Web interfaces to data.

2000s: The first half of the 2000s saw the emerging of XML and theassociated query language XQuery as a new database technology. Although XML is widely used for data exchange, aswell as for storing certain complex data types, relational databases still form the core of a vast majority of large-scale database applications. In this time period we have also witnessed the growthin “autonomic-computing/auto-admin” techniquesfor minimizing system administration effort. This period also saw a significant growth in use of open-source database systems, particularly PostgreSQL andMySQL.

1.9. Introduction to the Relational Model

The relational model is today the primary data model for commercial data- processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model.

Structure of Relational Databases

A relational database consists of a collection of tables, each of which is assigned a unique name. For example, consider the instructor table of Figure 2.1, which stores information about instructors. The table has four column headers: ID,name,deptname,andsalary.Eachrowofthistablerecordsinformationabout an instructor, consisting of the instructor’s ID, name, dept name, and salary. Similarly, the course table of Figure 2.2 stores information about courses, consisting of a course id, title, dept name, 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 2.3 shows a third table, prereq, which stores the prerequisite courses for each course. The table has two columns, course id and prereq id. Each row consists of a pair of course identifiers suchthatthesecondcourseisaprerequisiteforthefirstcourse.

Ingeneral,arowinatablerepresentsa relationshipamongasetofvalues.Sincea tableisacollectionofsuchrelationships,thereisaclosecorrespondencebetweenthe concept of table and the mathematical concept of relation, from which 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 repre- sented mathematically by an n-tupleof values, i.e., a tuple with n values, which corresponds to a row in a table.

Figure2.2Thecourserelation.

course id	prereq id

BIO-301	BIO-101
BIO-399 CS-190	BIO-101 CS-101
CS-315 CS-319	CS-101CS-101C
CS-347	S-101
EE-181	PHY-101

Thus, in the relational model the term relation is used to refer to a table, while the term tuple is used to refer to a row. Similarly, the term attribute refers to a column of a table.
Examining Figure 2.1, we can see that the relation instructor has four attributes:

ID, name, dept name, and salary.
Weusethetermrelationinstancetorefertoaspecificinstanceofarelation,i.e., containingaspecificsetofrows.TheinstanceofinstructorshowninFigure2.1has12 tuples, corresponding to 12instructors.
In this chapter, 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. They do not include all the data an actual university database would contain, in order to simplify ourpresentation.

The order in which tuples appear in a relation is irrelevant, since a relation is a setof tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure2.1,orareunsorted,asinFigure2.4,doesnotmatter;therelationsin
the two figures are the 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 attribute of the instructorrelationisthesetofallpossiblesalaryvalues,whilethedomainof thenameattributeisthesetofallpossibleinstructornames.
We require that, for all relations r, the domains of all attributes of r beatomic. Units. For example, suppose the table instructor had an attribute phone number, which can store a set of phone numbers corresponding to the instructor. Then the domain of phone number would not be atomic, since an
element of the domain is a set of phone numbers, and it has subparts, namely the individual phone numbers in theset.
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. If we treat each phone number as a single indivisible unit, then the attribute phone number would have an atomicdomain.

The null value is a special value that signifies that the value is unknown or does not exist. 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 that the telephone number is unlisted. We would then have to use the null value to signify 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 shouldbeeliminatedifatallpossible.Weshallassumenullvaluesareabsent initially, and in Section 3.6 we describe the effect of nulls on different operations.

Database Schema

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,whichisasnapshotofthedatainthedatabaseatagiveninstantin time.

The concept of a relation corresponds to the programming-language notionofavariable,whiletheconceptofarelationschemacorrespondstothe programming-language notion of typedefinition.

In general, a relation schema consists of a list of attributes and their correspondingdomains.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

Figure 2.5 The department relation.

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 often use the same name, such as instructor, to refer to both the schema and the instance. Where required, we explicitly refer to the schema or to the instance, for example “the instructor schema,” or “an instance of the instructor relation.” However, where it is clear whether we mean the schema or the instance, we simply use the relationname.

Consider the department relation of Figure 2.5. The schema for that relation is

department (dept name, 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. For example, suppose we wish to find the information about all the instructors who work in the Watson building. We look first at the department relation to find the dept nameofallthedepartmentshousedinWatson.Then,foreachsuchdepartment,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 be offered multiple times, across different semesters, or even within a semester. We need a relation to describe each individ-ualoffering,orsection,oftheclass.Theschemais

section (course id, sec id, semester, year, building, room number, time slot id) Figure 2.6 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	Semester	building	room number	time slot id
BIO-101	Summer	Painter	514	B A H
BIO-301 CS-101	Summer Fall	Painter	514	F E A
CS-101 CS-190	Spring	Packard	101	D B C
CS-190 CS-315	SpringSpringSp	Packard Taylor	101	A C B
CS-319 CS-319	ringSpringSprin	Taylor Watson	3128	C D
CS-347 EE-181	g Fall Spring	Watson Taylor	3128	A
FIN-201 HIS-351	SpringSpringSp	TaylorTaylor	120	
MU-199	ring	Packard	100	
PHY-101	Fall	Painter	3128	
		Packard	3128	
		Watson	3128	
			101	
			514	
			101	
			100	

Figure 2.6 The section relation.

As you can imagine, there are many more relations maintained in a real uni-versity database. In addition to those relations we have listed already, instructor, department,course,section,prereq,andteaches,weusethefollowingrelationsinthis text:

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

Figure 2.7 The teaches relation.

student (ID, name, dept name, totcred)
advisor (s id, iid)
takes (ID, course id, sec id, semester, year,grade)
classroom (building, room number,capacity)
time slot (time slot id, day, start time, endtime)

1.10. Keys

We must have a way to specify how tuples within a given relation are distin- guished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can uniquely identify the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes. Asuperkeyisasetofoneormoreattributesthat,takencollectively,allowustoidentify uniquely a tuple in the relation. For example, the ID attribute of the relation instructor is sufficient to distinguish one instructor tuple from another. Thus, ID is a superkey.Thenameattribute of

instructor, on the other hand, is not a superkey, because several instructors might have the same name.

student (ID, name, dept name, totcred)
advisor (s id, iid)
takes (ID, course id, sec id, semester, year,grade)
classroom (building, room number,capacity)
time slot (time slot id, day, start time, endtime)
Keys
We must have a way to specify how tuples within a given relation are distin- guished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can uniquely identify the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes. Asuperkeyisasetofoneormoreattributesthat,takencollectively,allowustoidentify uniquely a tuple in the relation. For example, the ID attribute of the relation instructor is sufficient to distinguish one instructor tuple from another. Thus, ID is a superkey.Thenameattribute of instructor, on the other hand, is not a superkey, because several instructors might have the same name.
The primary key should be chosen such that its attribute values are never, or very rarely, 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 theotherattributes;forexample,thedeptnameattributeofdepartmentislistedfirst,sinceit is the primary key. Primary key attributes are also underlined.
A relation, say r1, may include among its attributes the primary key of an- other relation, say r2. This attribute is called a foreign key from r1, referencing r2. The relation r1 is also called the referencing relation of the foreign key depen- dency, and r2 is called the referenced relation of the foreign key. For example, the attribute dept name in instructor is a foreign key from instructor, referencing depart- ment, since dept name is the primary key of department. In any database instance, given any tuple, say ta, from the instructor relation, there must be some tuple, say tb, in the department relation such that the value of the dept name attribute of tais the same as the value of the primary key, dept name, of tb.
Now consider the section and teaches relations. It would be reasonable to require that if a section exists for a course, it must be taught by at least one instructor; however, it could

possibly be taught by more than one instructor. To enforce thisconstraint,wewouldrequirethatifaparticular(courseid,secid,semester,year)combination appears in section, then the same combination must appear inteaches.

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 otherdirection,fromteachestosection).Theconstraintfromsectiontoteachesisanexampleofarefer entialintegrityconstraint;areferentialintegrityconstraintrequires 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 referencedrelation.

Schema Diagrams

A database schema, along with primary key and foreign key dependencies, can be depicted by schema diagrams. Figure 2.8 shows the schema diagram for our university 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 ofthereferencingrelationtotheprimarykeyofthereferencedrelation

Referential integrity constraints other than foreign key constraints are not shown explicitly in schema diagrams. Entity- relationship diagrams let us represent several kinds of constraints, including general referential integrityconstraints.

Many database systems provide design tools with a graphical user interface for creating schema diagrams. Figure 2.9 gives the relational schema that we use in our examples, with primary- key attributesunderlined.

1.11. Relational Query Languages

A query language is a language in which a user requests information from the database. These languages are usually on a level higher than that of a standard programming language. Query languages can be categorized as either procedural or nonprocedural. In a procedural language, the user instructs the system to perform a

sequence of operations on the database to compute the desired result. In a nonprocedural language, the user describes the desired information without giving a specific procedure for obtaining that information.

classroom(building, room number, capacity) department(dept name, building,

budget) course(course id, title, dept name, credits) instructor(ID, name, dept name, salary)

section(course id, sec id, semester, year, building, room number, time slot id) teaches(ID, course id, sec id, semester,

year) student(ID, name, dept name, tot cred)

takes(ID, course id, sec id, semester, year, grade)

advisor(s ID, i ID)

time slot(time slot id, day, start time, end time) prereq(course id, prereq id)

Query languages used in practice include elements of both the procedural and the nonprocedural approaches. .

There are a number of “pure” query languages: The relational algebra is pro- cedural, whereas the tuple relational calculus and domain relational calculus are nonprocedural. These query languages are terse and formal, lacking the “syntactic sugar” of commercial languages, but they illustrate the fundamental techniques for extracting data from the database. The relational algebra consists of a set of operations that take one or two relations as input and produce a new relation as their result. The relational calculus uses predicate logic to define the result desired without giving any specific algebraic procedure for obtaining thatresult.

Relational Operations

All procedural relational query languages provide a set of operations that can be applied to either a single relation or a pair of relations. These operations have the nice and desired property that their result is always a single relation. This property allows one to combine several of these operations in a modular way.

Specifically, since the result of a relational query is itself a relation, relational operations can be applied to the resultsofqueriesaswellastothegivensetofrelations.

The specific relational operations are expressed differently depending on the language, but fit the general framework we describe in this section.

The most frequent operation is the selection of specific tuples from a sin- gle relation (sayinstructor)thatsatisfiessomeparticularpredicate(saysalary>85,000).Theresultis a new relation that is a subset of the originalrelation

ID	name	dept name	salary
12121	Wu	Finance	90000
22222	Einstein	Physics	95000
33456	Gold	Physics Comp. Sci.	87000
83821	Brandt		92000

Figure 2.10 Result of query selecting instructor tuples with salary greater than $85000.

For example, if we select tuples from the instructor relation of Figure 2.1, satisfying the predicate “salary is greater than $85000”, we get the result shown in Figure 2.10.

Another frequent operation is to select certain attributes (columns) from a relation. The result is a new relation having only those selected attributes. For example, suppose we want a list of instructor IDs and salaries without listing the nameanddeptnamevaluesfromtheinstructorrelationofFigure2.1,thentheresult,

shown in Figure 2.11, has the two attributes ID and salary. Each tuple in the resultis

derived from a tuple of the instructor relation but with only selected attributes shown.

The join operation allows the combining of two relations by merging pairs of tuples, one from each relation, into a single tuple. There are a number of different ways to join relations . Figure 2.12 shows an example of joining the tuples from the instructor and department tables with the new tuples showing the information about each instructor and the department in which she is working. This result was

formed by combining each tuple in the instructor relation with the tuple in the department relation for the instructor’s department. In the form of join shown in Figure2.12,whichiscalledanaturaljoin,atuplefromtheinstructorrelationmatches a tuple in the department relation if the values.

Figure 2.11 Result of query selecting attributes ID and salary from the instructor relation.

of their dept name attributes are the same. All such matching pairs of tuples are present in the join result. In general, the natural join operation on two relations matches tuples whose values are the same on all attribute names that are common to both relations.

The Cartesian product operation combines tuples from two relations, but unlike the join operation, its result contains all pairs of tuples from the two relations, regardless of whether their attribute values match.

Because relations are sets, we can perform normal set operations on relations. The union operation performs a set union of two “similarly structured” tables (say a table of all graduate students and a table of all undergraduate students). For example, one can obtain the set of all students in a department. Other set operations, suchasintersectionandsetdifferencecanbeperformedaswell.

As we noted earlier, we can perform operations on the results of queries. For example, if we want to find the ID and salary for those instructors who have salary greater than $85,000, we would perform the first two operations in our example

above.Firstweselectthosetuplesfromtheinstructorrelationwherethesalaryvalue

is greater than $85,000 and then, from that result, select the two attributes IDand

salary, resulting in the relation shown in Figure 2.13 consisting of the ID.

ID	salary

12121	90000
22222	95000
33456	87000
83821	92000

Figure 2.13 Result of selecting attributes ID and salary of instructors with salary greater than$85,000

Symbol (Name)	Example of Use
o (Selection)	osalary>=85000(instructor)
	Return rows of the input relation that satisfy the predicate.
M (Projection)	MID,salary(instructor)
	Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
0 (Natural join)	instructor 0 department
	Outputpairsofrowsfromthetwoinputrela- tions that have the same value on all attributes that have the same name.
×(Cartesian product)	instructor × department
	Outputallpairsofrowsfromthetwoinput relations (regardless of whether or not they have the same values on commonattributes)
∪(Union)	Mname(instructor)∪Mname(student)
	Outputtheunionoftuplesfromthetwoinputrelations.

and salary. In this example, we could have performed the operations in either order, but that is not the case for all situations, as we shall see.

Sometimes, the result of a query contains duplicate tuples. For example, if we select the dept name¯ attribute from the instructor relation, there are several cases of duplication, including “Comp. Sci.”, which shows up three times. Certain rela- tional languages adhere strictly to the mathematical definition of a set and remove duplicates.

10. Assignments

11. Part A- Question & Answers

12. Part B- Questions

13. Supportive Online Certification Courses

https://nptel.ac.in/courses/106/105/106105175/
https://academy.oracle.com/en/resources-oracle-certifications.html
https://www.coursera.org/courses?query=database%20management

14. Real Time Applications

S.No	Application	CO
1	• Railway Reservation System • Library Management System • Banking • Universities and colleges	1
2	• Credit card transactions • Social Media Sites • Telecommunications • Finance	2
3	• Military • Online Shopping • Human Resource Management	3
4	• Manufacturing • Airline Reservation system	4

15. Contents Beyond the Syllabus

1. Database System Architecture: Centralized and client server systems Server system architecture

centralized: all calculation are done on one particular computer
distributed: the calculations is distributed to multiple computers when you have a large amount f data then you can divide it and send each part to a particular computers which will make the calculations for their part.

2. Parallel systems and Distributed systems

in parallel computing multiple processors performs multiple takes assigned to them simultaneously. memory in parallelism can either be shared or distributed. Parallel computing provides concurrency and saves time and memory.

in distributed computing we have multiple autonomous computers which seems to be the user as single system. in distributed systems there is mo shared memory and computers communicate either each other through message passing. in distributed computing a single task is divided among different computers.

16. Prescribed Text Books & Reference Books

Text Book

1. A.Silberschatz, H.F.Korth, S.Sudarshan, “Database System Concepts”, 6/e, TMH 2019

References:

1. Shamkant B. Navathe, “Database Management System” 6/e RamezElmasriPEA
2. “Database Principles Fundamentals of Design Implementation and Management”, Carlos Coronel, Steven Morris, Peter Robb, CengageLearning.
3. Raghurama Krishnan, Johannes Gehrke, “Database Management Systems”, 3/e,TMH

17. Mini Project Suggestion

1. Railway Reservation System

the case study is used to design and developed a database maintaining the records of different trains, train status and passengers. the record of train includes its number, name, source, destination and days on which it is available, where as the records of train status includes dates for which ticker takes can be booked ,total number of seats available, and number of seats already booked .

2. Library Management System

LMS gives us the complete information about the library and the daily transactions done in library.

3. Banking

Banking gives us the complete information about the banking and the daily transactions done in banking.

UNIT-II

2.1 <u>OVERVIEW OF THE SQL QUERY LANGUAGE</u>

SQL is a language to operate databases; it includes database creation, deletion, fetching rows, modifying rows, etc. SQL is an ANSI (American National Standards Institute) standard language, but there are many different versions of the SQL language.

What is SQL?

SQL is Structured Query Language, which is a computer language for storing, manipulating and retrieving data stored in a relational database. SQL is the standard language for Relational Database System. All the Relational Database Management Systems (RDMS) like MySQL, MS Access, Oracle,

Sybase, Informix, Postgres and SQL Server use SQL as their standard database language. Also, they are using different dialects, such as −

- MS SQL Server using T-SQL,
- Oracle using PL/SQL,
- MS Access version of SQL is called JET SQL (native format) etc.

Why SQL?

SQL is widely popular because it offers the following advantages −

- Allows users to access data in the relational database management systems.
- Allows users to describe the data.
- Allows users to define the data in a database and manipulate that data.
- Allows to embed within other languages using SQL modules, libraries & precompilers.
- Allows users to create and drop databases and tables.
- Allows users to create view, stored procedure, functions in a database.
- Allows users to set permissions on tables, procedures and views.

A Brief History of SQL

1. 1970 − Dr. Edgar F. "Ted" Codd of IBM is known as the father of relationaLdatabases. He described a relational model for databases.
2. 1974 − Structured Query Language appeared.
3. 1978 − IBM worked to develop Codd's ideas and released a product named System/R.
4. 1986 − IBM developed the first prototype of relational database and standardized by ANSI. The first relational database was released by Relational Software which later came to be known as Oracle.

SQL Process

When you are executing an SQL command for any RDBMS, the system determines the best way to carry out your request and SQL engine figures out how to interpret the task. There are various components included in this process.

These components are −

- Query Dispatcher
- Optimization Engines
- Classic Query Engine
- SQL Query Engine, etc.

A classic query engine handles all the non-SQL queries, but a SQL query engine won't handle logical files.

RDBMS: RDBMS stands for Relational Database Management System. RDBMS is the basis for SQL, and for all modern database systems like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

2.2 SQL DATA DEFINITION

A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.

table: The data in an RDBMS is stored in database objects which are called as tables. This table is basically a collection of related data entries and it consists of numerous columns and rows. Remember, a table is the most common and simplest form of data storage in a relational database. The following program is an example of a CUSTOMERS table −

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh  | 32 |   Ahmedabad | 2000.00 |
| 2 | Khilan    |  25 |   Delhi          | 1500.00 |
| 3 | kaushik  | 23 |   Kota           | 2000.00 |
| 4 | Chaitali  | 25 |   Mumbai      | 6500.00 |
| 5 | Hardik    |  27 |  Bhopal        | 8500.00 |
| 6 | Komal    |  22 |  MP             | 4500.00 |
| 7 | Muffy    |  24 |  Indore         | 10000.00 |
+----+----------+-----+-----------+----------+
```

field:

Every table is broken up into smaller entities called fields. The fields in the CUSTOMERS table consist of ID, NAME, AGE, ADDRESS and SALARY. A field is a column in a table that is designed to maintain specific information about every record in the table.

Record or a Row:

A record is also called as a row of data is each individual entry that exists in a table. For example, there are 7 records in the above CUSTOMERS table. Following is a single row of data or record in the CUSTOMERS table −

```
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
+----+----------+-----+-----------+----------+
```

A record is a horizontal entity in a table.

Column:

A column is a vertical entity in a table that contains all information associated with a specific field in a table. For example, a column in the CUSTOMERS table is ADDRESS, which represents location description and would be as shown below −

```
+-----------+
| ADDRESS |
+-----------+
| Ahmedabad |
| Delhi |
| Kota |
| Mumbai |
| Bhopal |
| MP |
| Indore |
+----+------+
```

NULL:

A NULL value in a table is a value in a field that appears to be blank, which means a field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces. A field with a NULL value is the one that has been left blank during a record creation.

SQL Constraints:

Constraints are the rules enforced on data columns on a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Constraints can either be column level or table level. Column level constraints are applied only to one column whereas, table level constraints are applied to the entire table.

Following are some of the most commonly used constraints available in SQL −

1. NOT NULL Constraint − Ensures that a column cannot have a NULL value.
2. DEFAULT Constraint − Provides a default value for a column when none is Specified.
3. UNIQUE Constraint − Ensures that all the values in a column are different.
4. PRIMARY Key − Uniquely identifies each row/record in a database table.
5. FOREIGN Key − Uniquely identifies a row/record in any another database table.
6. CHECK Constraint − The CHECK constraint ensures that all values in a column satisfy certain conditions.
7. INDEX − Used to create and retrieve data from the database very quickly.

Data Integrity

The following categories of data integrity exist with each RDBMS −

1. Entity Integrity − There are no duplicate rows in a table.
2. Domain Integrity − Enforces valid entries for a given column by restricting the type,
3. the format, or the range of values.
4. Referential integrity − Rows cannot be deleted, which are used by other records.
5. User-Defined Integrity − Enforces some specific business rules that do not fall into entity, domain or referential integrity.

SQL is followed by a unique set of rules and guidelines called Syntax. This tutorial gives you a quick start with SQL by listing all the basic SQL Syntax. All the SQL statements start with any of the keywords like SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW and the entire statements end with a semicolon (;).

The most important point to be noted here is that SQL is case insensitive, which means SELECT and select have same meaning in SQL statements. Where as, MySQL makes difference in table names. So, if you are working with MySQL, then you need to give table names as they exist in the database.

2.3. BASIC STRUCTURE OF SQL QUERIES

All the examples given in this tutorial have been tested with a MySQL server.

SQL SELECT Statement

```
SELECT column1, column2....columnN
FROM table_name;
```

SQL DISTINCT Clause

```
SELECT DISTINCT column1, column2....columnN
FROM table_name;
```

SQL WHERE Clause

```
SELECT column1, column2....columnN
FROM table_name
WHERE CONDITION;
```

SQL AND/OR Clause

```
SELECT column1, column2....columnN
FROM table_name
WHERE CONDITION-1 {AND|OR} CONDITION-2;
```

SQL IN Clause

```
SELECT column1, column2....columnN
FROM table_name
WHERE column_name IN (val-1, val-2,...val-N);
```

SQL BETWEEN Clause

```
SELECT column1, column2....columnN
FROM table_name
```

WHERE column_name BETWEEN val-1 AND val-2;

SQL LIKE Clause

```
SELECT column1, column2....columnN
FROM table_name
WHERE column_name LIKE { PATTERN };
```

SQL ORDER BY Clause

```
SELECT column1, column2....columnN
FROM table_name
WHERE CONDITION
ORDER BY column_name {ASC|DESC};
```

SQL GROUP BY Clause

```
SELECT SUM(column_name)
FROM table_name
WHERE CONDITION
GROUP BY column_name;
```

SQL COUNT Clause

```
SELECT COUNT(column_name)
FROM table_name
WHERE CONDITION;
```

SQL HAVING Clause

```
SELECT SUM(column_name)
FROM table_name
WHERE CONDITION
GROUP BY column name
HAVING (arithmetic function condition);
```

SQL CREATE TABLE Statement

```
CREATE TABLE table name(
column1 datatype,
column2 datatype,
column3 datatype,
.....
columnN datatype,
PRIMARY KEY( one or more columns )
);
```

SQL DROP TABLE Statement

```
DROP TABLE table_name;
```

SQL CREATE INDEX Statement

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

SQL DROP INDEX Statement

```
ALTER TABLE table_name
DROP INDEX index_name;
```

SQL DESC Statement

```
DESC table_name;
```

SQL TRUNCATE TABLE Statement

```
TRUNCATE TABLE table_name;
```

SQL ALTER TABLE Statement

```
ALTER TABLE table_name {ADD|DROP|MODIFY} column_name {data_ype};
```

SQL ALTER TABLE Statement (Rename)

```
ALTER TABLE table_name RENAME TO new_table_name;
```

SQL INSERT INTO Statement

INSERT INTO table_name(column1, column2....columnN)
VALUES (value1, value2....valueN);

SQL UPDATE Statement

UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[WHERE CONDITION];
SQL DELETE Statement
DELETE FROM table_name
WHERE {CONDITION};

SQL CREATE DATABASE Statement

CREATE DATABASE database_name;

SQL DROP DATABASE Statement

DROP DATABASE database_name;

SQL USE Statement

USE database_name;

SQL COMMIT Statement

COMMIT;

SQL ROLLBACK Statement

ROLLBACK;

2.4. ADDITIONAL BASIC OPERATIONS

The Rename Operation

Consider again the query that we used earlier:
select name, course id
from instructor, teaches
where instructor. ID = teaches. ID ;
The result of this query is a relation with the following attributes:
name, course id
The names of the attributes in the result are derived from the names of the attributes in the relations in the from clause. SQL provides a way of renaming the attributes of a result relation. It uses the as clause, taking the form:
old-name as new-name
The as clause can appear in both the select and from clauses. For example, if we want the attribute name name to be replaced with the name instructor name, we can rewrite the preceding query as:

select name as instructor name, course id
from instructor, teaches
where instructor. ID = teaches. ID ;

The as clause is particularly useful in renaming relations. One reason to rename a relation is to replace a long relation name with a shortened version that is more convenient to use elsewhere in the query. To illustrate, we rewrite the query “For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught.”

select T.name, S.course id
from instructor as T, teaches as S
where T. ID = S. ID ;

String Operations:

SQL specifies strings by enclosing them in single quotes, for example, 'Computer'. A single quote character that is part of a string can be specified by using two single quote characters;
for example, the string "It's right" can be specified by "It"s right".
The SQL standard specifies that the equality operation on strings is case sensitive; as a result the expression "'comp. sci.' = 'Comp. Sci.'" evaluates to false. However, some database systems, such as M y SQL and SQL S erver, do not distinguish uppercase from lowercase when matching strings; as a result "'comp. Sci.' = 'Comp. Sci.'" would evaluate to true on these databases. This default behavior can, however, be changed, either at the database level or at the level of specific attributes. SQL also permits a variety of functions on character strings, such as concate- nating (using " "), extracting substrings, finding the length of strings, converting strings to uppercase (using the function upper(s) where s is a string) and low- ercase (using the function lower(s)), removing spaces at the end of the string (using trim(s)) and so on. There are variations on the exact set of string functions supported by different database systems. See your database system's manual for more details on exactly what string functions it supports. Pattern matching can be performed on strings, using the operator like. We describe patterns by using two special characters:
• Percent (%): The % character matches any substring.
• Underscore ( ): The character matches any character.
Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

1. 'Intro%' matches any string beginning with "Intro".
2. '%Comp%' matches any string containing "Comp" as a substring, for example, 'Intro. To Computer Science', and 'Computational Biology'.
3. '_ _ _' matches any string of exactly three characters.
4. '_ _ _ %' matches any string of at least three characters.

SQL expresses patterns by using the like comparison operator. Consider the query "Find the names of all departments whose building name includes the substring 'Watson'." This query can be written as:

select dept name
from department
where building like '%Watson%';

For patterns to include the special pattern characters (that is, % and), SQL allows the specification of an escape character. The escape character is used immediately before a special pattern character to indicate that the special pattern character is to be treated like a normal character. We define the escape character for a like comparison using the escape keyword. To illustrate, consider the following patterns, which use a backslash (\) as the escape character:

1. like 'ab\%cd%' escape '\' matches all strings beginning with "ab%cd".
2. like 'ab\\cd%' escape '\' matches all strings beginning with "ab\cd".

Attribute Specification in Select Clause:
The asterisk symbol " * " can be used in the select clause to denote "all attributes." Thus, the use of instructor.* in the select clause of the query:

select instructor.*
from instructor, teaches
where instructor. ID = teaches. ID ;

indicates that all attributes of instructor are to be selected. A select clause of the form select * indicates that all attributes of the result relation of the from clause are selected.

Ordering the Display of Tuples:

SQL offers the user some control over the order in which tuples in a relation are displayed. The order by clause causes the tuples in the result of a query to appear in sorted order. To list in alphabetic order all instructors in the Physics department, we write:

select name
from instructor
where dept name = 'Physics'
order by name;

By default, the order by clause lists items in ascending order. To specify the sort order, we may specify desc for descending order or asc for ascending order. Furthermore, ordering can be performed on multiple attributes. Suppose that we wish to list the entire instructor relation in descending order of salary. If several instructors have the same salary, we order them in ascending order by name. We express this query in SQL as follows:

select * from instructor order by salary desc, name asc;

Where Clause Predicates:

SQL includes a between comparison operator to simplify where clauses that specify that a value be less than or equal to some value and greater than or equal to some other value. If we wish to find the names of instructors with salary amounts between $90,000 and $100,000, we can use the between comparison to write:

select name from instructor
where salary between 90000 and 100000;

instead of:

select name
from instructor
where salary <= 100000 and salary >= 90000;

Similarly, we can use the not between comparison operator. We can extend the preceding query that finds instructor names along with course identifiers, which we saw earlier, and consider a more complicated case in which we require also that the instructors be from the Biology department:

"Find the instructor names and the courses they taught for all instructors in the Biology department who have taught some course."

To write this query, we can modify either of the SQL queries we saw earlier, by adding an extra condition in the where clause. We show below the modified form of the SQL query that does not use natural join.

select name, course id
from instructor, teaches
where instructor. ID = teaches. ID and dept name = 'Biology';

SQL permits us to use the notation (v 1 , v 2 , . . . , v n) to denote a tuple of arity n containing values v 1 , v 2 , . . . , v n . The comparison operators can be used on tuples, and the ordering is defined lexicographically. For example, (a 1 , a 2) <= (b 1 , b 2)

COURSEID
CS -101

CS -347
PHY -101

The c1 relation, listing courses taught in Fall 2009. is true if a 1 <= b 1 and a 2 <= b 2 ; similarly, the two tuples are equal if all their attributes are equal. Thus, the preceding SQL query can be rewritten as follows:

select name, course id
from instructor, teaches
where (instructor. ID , dept name) = (teaches. ID , 'Biology');

2.5. SET OPERATIONS

The SQL operations union, intersect, and except operate on relations and correspond to the mathematical set-theory operations , ∩, and −. We shall now construct queries involving the ∪ union, intersect, and except operations over two sets.

• The set of all courses taught in the Fall 2009 semester:
select course id
from section
where semester = 'Fall' and year= 2009;

• The set of all courses taught in the Spring 2010 semester:
select course id
from section
where semester = 'Spring' and year= 2010;

In our discussion that follows, we shall refer to the relations obtained as the result of the preceding queries as c1 and c2, respectively, and show the results when these queries are run on the section relation of Figure 2.6 in Figures 3.9 and 3.10. Observe that c2 contains two tuples corresponding to course id CS -319, since two sections of the course have been offered in Spring 2010.

course id
CS -101
CS -315
CS -319
CS -319
FIN -201
HIS -351
MU -199

The Union Operation:To find the set of all courses taught either in Fall 2009 or in Spring 2010, or both,
(select course id from section where semester = 'Fall' and year= 2009)
union
(select course id from section where semester = 'Spring' and year= 2010);

The union operation automatically eliminates duplicates, unlike the select clause. Thus, using the section relation of Figure 2.6, where two sections of CS -319 are offered in Spring 2010, and a section of CS -101 is offered in the Fall 2009 as well as in the Fall 2010 semester, CS -101 and CS -319 appear only once in the result, shown in Figure 3.11. If we want to retain all duplicates, we must write union all in place of union:

(select course id from section where semester = 'Fall' and year= 2009)

union all
(select course id from section where semester = 'Spring' and year= 2010);

The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both c1 and c2. So, in the above query, each of CS -319 and CS -101 would be listed twice. As a further example, if it were the case that sections of ECE -101 were taught in the Fall 2009 semester and 2 sections of ECE -101

course id
CS -101
CS -315
CS -319
CS -347
FIN -201
HIS -351
MU -199
PHY -101

The Intersect Operation
To find the set of all courses taught in the Fall 2009 as well as in Spring 2010 we write:

(select course id from section where semester = 'Fall' and year= 2009)
intersect
(select course id from section where semester = 'Spring' and year= 2010);

The result relation, shown in Figure 3.12, contains only one tuple with CS -101. The intersect operation automatically eliminates duplicates. For example, if it were the case that 4 sections of ECE -101 were taught in the Fall 2009 semester and sections of ECE -101 were taught in the Spring 2010 semester, then there would be only 1 tuple with ECE -101 in the result.
If we want to retain all duplicates, we must write intersect all in place of intersect:

course id
CS -101

(select course id from section where semester = 'Fall' and year= 2009)
intersect all
(select course id from section where semester = 'Spring' and year= 2010);

The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both c1 and c2. For example, if 4 sections of ECE -101 were taught in the Fall 2009 semester and 2 sections of ECE -101 were taught in the Spring 2010 semester, then there would be 2 tuples with ECE -101 in the result.

The Except Operation
To find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we write:
(select course id from section where semester = 'Fall' and year= 2009)
except
(select course id from section where semester = 'Spring' and year= 2010);

The result of this query is shown. Note that this is exactly relation c1 except that the tuple for CS -101 does not appear. The except operation 7 outputs all tuples from its first input that do not occur in the second input; that is, it performs set difference. The operation automatically eliminates duplicates in the inputs before performing set difference. For example, if 4 sections of ECE -101 were taught in the

Fall 2009 semester and 2 sections of ECE -101 were taught in the Spring 2010 semester, the result of the except operation would not have any copy of ECE -101. If we want to retain duplicates, we must write except all in place of except:

(select course id from section where semester = 'Fall' and year= 2009)
except all
(select course id from section where semester = 'Spring' and year= 2010);

course id
CS -347
PHY -101

2.6. Null Values

Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations. The result of an arithmetic expression (involving, for example +, −, , or /) is null if any of the input values is null. For example, if a ∗ query has an expression r.A + 5, and r.A is null for a particular tuple, then the expression result must alsobe null for that tuple. Comparisons involving nulls are more of a problem. For example, consider the comparison "1 < null". It would be wrong to say this is true since we do not know what the null value represents. But it would likewise be wrong to claim this expression is false; if we did, "not (1 < null)" would evaluate to true, which does not make sense. SQL therefore treats as unknown the result of any comparison involving a null value (other than predicates is null and is not null, which are described later in this section). This creates a third logical value in addition to true and false.
Since the predicate in a where clause can involve Boolean operations such as and, or,and not on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value unknown.
• and: The result of true and unknown is unknown, false and unknown is false,
while unknown and unknown is unknown.
• or: The result of true or unknown is true, false or unknown is unknown, while
unknown or unknown is unknown.
• not: The result of not unknown is unknown.
SQL uses the special keyword null in a predicate to test for a null value. Thus, to find all instructors who appear in the instructor relation with null values for salary, we write:

select name
from instructor
where salary is null;

Aggregate Functions:

Aggregate Functions are all about

- Performing calculations on multiple rows
- Of a single column of a table
- And returning a single value.

The ISO standard defines five (5) aggregate functions namely;

1. COUNT
2. SUM
3. AVG
4. MIN
5. MAX

COUNT Function

The COUNT function returns the total number of values in the specified field. It works on both numeric and non-numeric data types.All aggregate functions by default exclude nulls values before working on the data.
COUNT (*) is a special implementation of the COUNT function that returns the count of all the rows in a specified table. COUNT (*) also considers Nulls and duplicates. The table shown below shows data in movie rentals table

reference_ number	transaction_ date	return_date	membership_ number	movie_id	movie_ returned
11	20-06-2012	NULL	1	1	0
12	22-06-2012	25-06-2012	1	2	0
13	22-06-2012	25-06-2012	3	2	0
14	21-06-2012	24-06-2012	2	2	0
15	23-06-2012	NULL	3	3	0

Let's suppose that we want to get the number of times that the movie with id 2 has been rented out
SELECT COUNT(`movie_id`) FROM `movierentals` WHERE `movie_id` = 2;
COUNT('movie_id')
3

DISTINCT Keyword
The DISTINCT keyword that allows us to omit duplicates from our results. This is achieved by grouping similar values together .
To appreciate the concept of Distinct, lets execute a simple query

```
SELECT `movie_id` FROM `movierentals`;
movie_id
1
2
2
2
3
```

Now let's execute the same query with the distinct keyword -
SELECT DISTINCT `movie_id` FROM `movierentals`;
As shown below , distinct omits duplicate records from the results.

```
movie_id
1
2
3
```

MIN function

The MIN function returns the smallest value in the specified table field. As an example, let's suppose we want to know the year in which the oldest movie in our library was released, we can use MySQL's MIN function to get the desired information.

The following query helps us achieve that

```
SELECT MIN(`year_released`) FROM `movies`;
```

Executing the above query in MySQL workbench against myflixdb gives us the following results.

```
MIN('year_released')
2005
```

MAX function:

It returns the largest value from the specified table field. Let's assume we want to get the year that the latest movie in our database was released. We can easily use the MAX function to achieve that.

The following example returns the latest movie year released.

```
SELECT MAX(`year_released`) FROM `movies`;
```

Executing the above query in MySQL workbench using myflixdb gives us the following results.

```
MAX('year_released')
2012
```

SUM function

Suppose we want a report that gives total amount of payments made so far. We can use the MySQL SUM function which returns the sum of all the values in the specified column. SUM works on numeric fields only. Null values are excluded from the result returned. The following table shows the data in payments table payment_id

payment_id	membership_number	payment_date	description	amount_paid	external_reference_number
1	1	23-07-2012	Movie rental payment	2500	11
2	1	25-07-2012	Movie rental payment	2000	12
3	3	30-07-2012	Movie rental payment	6000	NULL

The query shown below gets the all payments made and sums them up to return a single result.

```
SELECT SUM(`amount_paid`) FROM `payments`;
```

Executing the above query in MySQL workbench against the myflixdb gives the following results.

```
SUM('amount_paid')
10500
```

AVG function:

MySQL AVG function returns the average of the values in a specified column. Just like the SUM function, it works only on numeric data types. Suppose we want to find the average amount paid. We can use the following query –

SELECT AVG(`amount_paid`) FROM `payments`;
Executing the above query in MySQL workbench, gives us the following results.

AVG('amount_paid')
3500

2.7.Nested sub queries

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
There are a few rules that subqueries must follow −

1) Subqueries must be enclosed within parentheses.
2) A subquery can have only one column in the SELECT clause, unless multiple
3) columns are in the main query for the subquery to compare its selected columns.
4) An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
5) Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
6) The SELECT list cannot include any references to values that evaluate to a BLOB, ARRAY, CLOB, or NCLOB.
7) A subquery cannot be immediately enclosed in a set function.
8) The BETWEEN operator cannot be used with a subquery. However, the BETWEEN operator can be used within the subquery.

Subqueries with the SELECT Statement
Subqueries are most frequently used with the SELECT statement. The basic syntax is as follows −

```
SELECT column_name [, column_name ]
FROM table1 [, table2 ]
WHERE column_name OPERATOR
(SELECT column_name [, column_name ]
FROM table1 [, table2 ]
[WHERE])
```

Example
Consider the CUSTOMERS table having the following records –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Now, let us check the following subquery with a SELECT statement.

```
SQL> SELECT *
FROM CUSTOMERS
WHERE ID IN (SELECT ID
FROM CUSTOMERS
WHERE SALARY > 4500) ;
```

This would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
7	Muffy	24	Indore	10000.00

Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date or number functions.

The basic syntax is as follows.

```
INSERT INTO table_name [ (column1 [, column2 ]) ]
SELECT [ *|column1 [, column2 ]
FROM table1 [, table2 ]
[ WHERE VALUE OPERATOR ]
```

Example

Consider a table CUSTOMERS_BKP with similar structure as CUSTOMERS table. Now to copy the complete CUSTOMERS table into the CUSTOMERS_BKP table, you can use the following syntax.

```
SQL> INSERT INTO CUSTOMERS_BKP
SELECT * FROM CUSTOMERS
WHERE ID IN (SELECT ID
FROM CUSTOMERS) ;
```

Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

The basic syntax is as follows.

```
UPDATE table
SET column_name = new_value
[ WHERE OPERATOR [ VALUE ]
(SELECT COLUMN_NAME
FROM TABLE_NAME)
[ WHERE) ]
```

Example

Assuming, we have CUSTOMERS_BKP table available which is backup of CUSTOMERS table. The following example updates SALARY by 0.25 times in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> UPDATE CUSTOMERS
SET SALARY = SALARY * 0.25
WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
WHERE AGE >= 27 );
```

This would impact two rows and finally CUSTOMERS table would have the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  35 | Ahmedabad |   125.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  2125.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

The basic syntax is as follows.

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
(SELECT COLUMN_NAME
FROM TABLE_NAME)
[ WHERE) ]
```

Example

Assuming, we have a CUSTOMERS_BKP table available which is a backup of the CUSTOMERS table. The following example deletes the records from the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

```
SQL> DELETE FROM CUSTOMERS
WHERE AGE IN (SELECT AGE FROM CUSTOMERS_BKP
WHERE AGE >= 27 );
```

This would impact two rows and finally the CUSTOMERS table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Database modification

The SQL Modification Statements make changes to database data in tables and columns. There are 3 modification statements:

1) INSERT Statement -- add rows to tables
2) UPDATE Statement -- modify columns in table rows
3) DELETE Statement - remove rows from tables

INSERT Statement

The INSERT Statement adds one or more rows to a table. It has two formats:

INSERT INTO table-1 [(column-list)] VALUES (value-list)
and,
INSERT INTO table-1 [(column-list)] (query-specification)

The first form inserts a single row into table-1 and explicitly specifies the column values for the row. The second form uses the result of query-specification to insert one or more rows into table-1. The result rows from the query are the rows added to the insert table. Note: the query cannot reference table-1. Both forms have an optional column-list specification. Only the columns listed will be assigned values. Unlisted columns are set to null, so unlisted columns must allow nulls. The values from the VALUES Clause (first form) or the columns from the query-specification rows (second form) are assigned to the corresponding column in column-list in order. If the optional column-list is missing, the default column list is substituted. The default column list contains all columns in table-1 in the order they were declared in

CREATE TABLE, or
CREATE VIEW.
VALUES Clause

The VALUES Clause in the INSERT Statement provides a set of values to place in the columns of a new row. It has the following general format:

VALUES (value-1 [, value-2] ...)

value-1 and value-2 are Literal Values or Scalar Expressionsinvolving literals. They can also specify NULL. The values list in the VALUES clause must match the explicit or implicit column list for INSERT in degree (number of items). They must also match the data type of corresponding column or be convertible to that data type.

```
INSERT INTO sp
SELECT s.sno, p.pno, 500
FROM s, p
WHERE p.color='Green' AND s.city='London
```

UPDATE Statement

The UPDATE statement modifies columns in selected table rows. It has the following general format:

UPDATE table-1 SET set-list [WHERE predicate]

The optional WHERE Clause has the same format as in the SELECT Statement. See WHERE Clause. The WHERE clause chooses which table rows to update. If it is missing, all rows are in table-1 are updated. The set-list contains assignments of new values for selected columns. See SET Clause. The SET Clause expressions and WHERE Clause predicate can contain subqueries, but the subqueries cannot reference table-1. This prevents situations where results are dependent on the order of processing.

SET Clause

The SET Clause in the UPDATE Statement updates (assigns new value to) columns in the selected table rows. It has the following general format:

SET column-1 = value-1 [, column-2 = value-2] ...

column-1 and column-2 are columns in the Update table. value-1 and value2 are expressions that can reference columns from the update table. They also can be the keyword -- NULL, to set the column to null. Since the assignment expressions can reference columns from the current row, the expressions are evaluated first. After the values of all Set expressions have been computed, they are then assigned to the referenced columns. This avoids results dependent on the order of processing.

UPDATE Examples

UPDATE sp SET qty = qty + 20

Before

After

Sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

=>

sno	pno	qty
S1	P1	NULL
S2	P1	220
S3	P1	1020
S3	P2	220

UPDATE s
SET name = 'Tony', city = 'Milan'
WHERE sno = 'S3'

Before

sno	name	city
S1	Pierre	Paris
S2	John	London
S3	Mario	Rome

=> After

Sno	name	city
S1	Pierre	Paris
S2	John	London

S3 Tony Milan

DELETE Statement

The DELETE Statement removes selected rows from a table. It has the following general format:
DELETE FROM table-1 [WHERE predicate]
The optional WHERE Clause has the same format as in the SELECT Statement. See WHERE Clause. The WHERE clause chooses which table rows to delete. If it is missing, all rows are in table-1 are removed.
The WHERE Clause predicate can contain subqueries, but the subqueries cannot reference table-1. This prevents situations where results are dependent on the order of processing.
DELETE Examples
DELETE FROM sp WHERE pno = 'P1'
Before
After

sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

=>

sno	pno	qty
S3	P2	200

DELETE FROM p WHERE pno NOT IN (SELECT pno FROM sp)
Before

Pno	descr	color
P1	Widget	Blue
P2	Widget	Red
P3	Dongle	Green

=> After

pno	descr	color
P1	Widget	Blue
P2	Widget	Red

2.8. JOINS

The SQL Joins clause is used to combine records from two or more tables in a database.
JOIN is a means for combining fields from two tables by using values common to each.
Consider the following two tables −
Table 1 − CUSTOMERS Table

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00

4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

+----+----------+-----+-----------+----------+

Table 2 − ORDERS Table

+-----+---------------------+-------------+--------+

|OID | DATE | CUSTOMER_ID | AMOUNT |

+-----+---------------------+-------------+--------+

102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

+-----+---------------------+-------------+--------+

Now, let us join these two tables in our SELECT statement as shown below.

```
SQL> SELECT ID, NAME, AGE, AMOUNT
FROM CUSTOMERS, ORDERS
WHERE CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

+----+----------+-----+--------+-----------+

| ID | NAME | AGE | AMOUNT |

+----+----------+-----+--------+-----------+

3	kaushik	23	3000
3	kaushik	23	1500
2	Khilan	25	1560
4	Chaitali	25	2060

+----+----------+-----+--------+---------+

Here, it is noticeable that the join is performed in the WHERE clause. Several operators can be used to join tables, such as =, <, >, <>, <=, >=, !=, BETWEEN, LIKE, and NOT; they can all be used to join tables. However, the most common operator is the equal to symbol.

There are different types of joins available in SQL −

1) INNER JOIN − returns rows when there is a match in both tables.
2) LEFT JOIN − returns all rows from the left table, even if there are no matches in the right table.
3) RIGHT JOIN − returns all rows from the right table, even if there are no matches in the left table.
4) FULL JOIN − returns rows when there is a match in one of the tables.
   The most important and frequently used of the joins is the INNER JOIN. They are also referred to as an EQUIJOIN.

The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row.

Syntax:

The basic syntax of the INNER JOIN is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

Example

Consider the following two tables.

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
| OID | DATE | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using the INNER JOIN as follows −

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
INNER JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+--------+---------------------+---------+
| ID | NAME | AMOUNT |        DATE          |
+----+----------+--------+---------------------+----------+
| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

The SQL LEFT JOIN returns all rows from the left table, even if there are no matches in the right table. This means that if the ON clause matches 0 (zero) records in the right table; the join will still return a row in the result, but with NULL in each column from the right table. This means that a left join returns all the values from the left table, plus matched values from the right table or NULL in case of no matching join predicate.

Syntax

The basic syntax of a LEFT JOIN is as follows.

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

Example

Consider the following two tables,

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+-----------+
| ID | NAME | AGE | ADDRESS     | SALARY |
+----+----------+-----+-----------+----------+-----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan      | 25 | Delhi              | 1500.00 |
| 3 | kaushik | 23 | Kota              | 2000.00 |
| 4 | Chaitali | 25 | Mumbai          | 6500.00 |
| 5 | Hardik   | 27 | Bhopal             | 8500.00 |
| 6 | Komal   | 22 | MP                        | 4500.00 |
| 7 | Muffy    | 24 | Indore             | 10000.00 |
+----+----------+-----+-----------+----------+-----------------+
```

Table 2 − Orders Table is as follows.

```
+-----+---------------------+-------------+--------+
| OID | DATE | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using the LEFT JOIN as follows.

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result –

```
+----+----------+--------+---------------------+
| ID | NAME | AMOUNT | DATE |
+----+----------+--------+---------------------+
| 1 | Ramesh | NULL | NULL |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
| 5 | Hardik | NULL | NULL |
| 6 | Komal | NULL | NULL |
| 7 | Muffy | NULL | NULL |
+----+----------+--------+---------------------+
```

The SQL RIGHT JOIN returns all rows from the right table, even if there are no matches in the left table. This means that if the ON clause matches 0 (zero) records in the left table; the join will still return a row in the result, but with NULL in each column from the left table. This means that a right join returns all the values from the right table, plus matched values from the left table or NULL in case of no matching join predicate.

Syntax

The basic syntax of a RIGHT JOIN is as follow.

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
```

ON table1.common_field = table2.common_field;

Example

Consider the following two tables,

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID | DATE | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using the RIGHT JOIN as follows.

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result −

```
+------+----------+--------+---------------------+
| ID | NAME | AMOUNT | DATE |
+------+----------+--------+---------------------+
| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

The SQL FULL JOIN combines the results of both left and right outer joins. The joined table will contain all records from both the tables and fill in NULLs for missing matches on either side.

Syntax

The basic syntax of a FULL JOIN is as follows −

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

Here, the given condition could be any given expression based on your requirement.

Example

Consider the following two tables.

Table 1 − CUSTOMERS Table is as follows.

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

Table 2 − ORDERS Table is as follows.

```
+-----+---------------------+-------------+--------+
|OID | DATE | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 00:00:00 | 4 | 2060 |
+-----+---------------------+-------------+--------+
```

Now, let us join these two tables using FULL JOIN as follows.

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
FULL JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result −

```
+------+----------+--------+---------------------+
| ID | NAME | AMOUNT | DATE |
+------+----------+--------+---------------------+
| 1 | Ramesh | NULL | NULL |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
| 5 | Hardik | NULL | NULL |
| 6 | Komal | NULL | NULL |
| 7 | Muffy | NULL | NULL |
| 3 | kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | kaushik | 1500 | 2009-10-08 00:00:00 |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

If your Database does not support FULL JOIN (MySQL does not support FULL JOIN), then you can use UNION ALL clause to combine these two JOINS as shown below.

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION ALL
```

```
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
```

2.8. VIEWS

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query. A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view. Views, which are a type of virtual tables allow users to do the following −

1. Structure data in a way that users or classes of users find natural or intuitive.
2. Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
3. Summarize data from various tables which can be used to generate reports.

Creating Views:

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables or another view. To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows −

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following records –

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM CUSTOMERS;
```

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SQL > SELECT * FROM CUSTOMERS_VIEW;
```

This would produce the following result.

```
+----------+-----+
| name | age |
+----------+-----+
| Ramesh | 32 |
| Khilan | 25 |
| kaushik | 23 |
| Chaitali | 25 |
| Hardik | 27 |
| Komal | 22 |
| Muffy | 24 |
+----------+-----+
```

The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition. If they do not satisfy the condition(s), the UPDATE or INSERT returns an error. The following code block has an example of creating same view

```
CUSTOMERS_VIEW with the WITH CHECK OPTION.
CREATE VIEW CUSTOMERS_VIEW AS
SELECT name, age
FROM CUSTOMERS
WHERE age IS NOT NULL
WITH CHECK OPTION;
```

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below −

1. The SELECT clause may not contain the keyword DISTINCT.
2. The SELECT clause may not contain summary functions.
3. The SELECT clause may not contain set functions.
4. The SELECT clause may not contain set operators.
5. The SELECT clause may not contain an ORDER BY clause.
6. The FROM clause may not contain multiple tables.
7. The WHERE clause may not contain subqueries.
8. The query may not contain GROUP BY or HAVING.
9. Calculated columns may not be updated.

All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function. So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

```
SQL > UPDATE CUSTOMERS_VIEW
SET AGE = 35
WHERE name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
```

| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
1	Ramesh	35	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00
+----+----------+-----+-----------+----------+

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command. Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22.

SQL > DELETE FROM CUSTOMERS_VIEW
WHERE age = 22;

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
1	Ramesh	35	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
7	Muffy	24	Indore	10000.00
+----+----------+-----+-----------+----------+

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −

DROP VIEW view_name;

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

DROP VIEW CUSTOMERS_VIEW;

2.9. TRANSACTION

A transaction is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program. A transaction is the propagation of one or more changes to the database. For example, if you are creating a record or updating a record or deleting a record from the table, then you are performing a transaction on that table. It is important to control these transactions to ensure the data integrity and to handle database errors. Practically, you will club many SQL queries into a group and you will execute all of them together as a part of a transaction.

Properties of Transactions

Transactions have the following four standard properties, usually referred to by the acronym ACID.

1. Atomicity − ensures that all operations within the work unit are completed successfully. Otherwise, the transaction is aborted at the point of failure and all the previous operations are rolled back to their former state.
2. Consistency − ensures that the database properly changes states upon a Successfully committed transaction.
3. Isolation − enables transactions to operate independently of and transparent to each other.
4. Durability − ensures that the result or effect of a committed transaction persists in case of a system failure.

   Transaction Control

   The following commands are used to control transactions.
5. COMMIT − to save the changes.
6. ROLLBACK − to roll back the changes.
7. SAVEPOINT − creates points within the groups of transactions in which to
8. ROLLBACK.
9. SET TRANSACTION − Places a name on a transaction.

Transactional Control Commands

Transactional control commands are only used with the DML Commands such as - INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database.

The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by atransaction to the database.The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for the COMMIT command is as follows.

COMMIT;

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+-----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+-----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example which would delete those records from the table which have age = 25 and then COMMIT the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
WHERE AGE = 25;
SQL> COMMIT;
```

Thus, two rows from the table would be deleted and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

The ROLLBACK Command

The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for a ROLLBACK command is as follows −

```
ROLLBACK;
```

Example

Consider the CUSTOMERS table having the following records −

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

Following is an example, which would delete those records from the table which have the age = 25 and then ROLLBACK the changes in the database.

```
SQL> DELETE FROM CUSTOMERS
WHERE AGE = 25;
SQL> ROLLBACK;
```

Thus, the delete operation would not impact the table and the SELECT statement would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

The SAVEPOINT Command

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction. The syntax for a SAVEPOINT command is as shown below.

SAVEPOINT SAVEPOINT_NAME;

This command serves only in the creation of a SAVEPOINT among all the transactional statements. The ROLLBACK command is used to undo a group of transactions. The syntax for rolling back to a SAVEPOINT is as shown below.
ROLLBACK TO SAVEPOINT_NAME;
Following is an example where you plan to delete the three different records from the CUSTOMERS table. You want to create a SAVEPOINT before each delete, so that you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.
Example
Consider the CUSTOMERS table having the following records.

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

The following code block contains the series of operations.

```
SQL> SAVEPOINT SP1;
Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=1;
1 row deleted.
SQL> SAVEPOINT SP2;
Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=2;
1 row deleted.
SQL> SAVEPOINT SP3;
Savepoint created.

SQL> DELETE FROM CUSTOMERS WHERE ID=3;
1 row deleted.
```

Now that the three deletions have taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone –

```
SQL> ROLLBACK TO SP2;
Rollback complete.
```

Notice that only the first deletion took place since you rolled back to SP2.

```
SQL> SELECT * FROM CUSTOMERS;
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
| 7 | Muffy | 24 | Indore | 10000.00 |
+----+----------+-----+-----------+----------+
```

6 rows selected.

The RELEASE SAVEPOINT Command

The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created.

The syntax for a RELEASE SAVEPOINT command is as follows.

```
RELEASE SAVEPOINT SAVEPOINT_NAME;
```

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

The SET TRANSACTION Command

The SET TRANSACTION command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For example, you can specify a transaction to be read only or read write.

The syntax for a SET TRANSACTION command is as follows.

```
SET TRANSACTION [ READ WRITE | READ ONLY ];
```

SQL Constraints

SQL Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table. Constraints can be divided into the following two types,

1.Column level constraints: Limits only column data.

2.Table level constraints: Limits whole table data.

Constraints are used to make sure that the integrity of data is maintained in the database.

Following are the most used constraints that can be applied to a table.

1. NOT NULL
2. UNIQUE
3. PRIMARY KEY
4. FOREIGN KEY
5. CHECK
6. DEFAULT

NOT NULL CONSTRAINT

NOT NULL constraint restricts a column from having a NULL value. Once NOT NULL constraint is applied to a column, you cannot pass a null value to that column. It enforces a column to contain a proper value.

One important point to note about this constraint is that it cannot be defined at table level. Example using NOT NULL constraint

```
CREATE TABLE Student(s_id int NOT NULL, Name varchar(60), Age int);
```

The above query will declare that the s_id field of Student table will not take NULL value.

UNIQUE CONSTRAINT

UNIQUE constraint ensures that a field or column will only have unique values. A UNIQUE constraint field will not have duplicate data. This constraint can be applied atcolumn level or table level. Using

UNIQUE constraint when creating a Table (Table Level) Here we have a simple CREATE query to create a table, which will have a column s_id with unique values.

```
CREATE TABLE Student(s_id int NOT NULL UNIQUE, Name
varchar(60), Age int);
```

The above query will declare that the s_id field of Student table will only have unique values and wont take NULL value. Using UNIQUE constraint after Table is created (Column Level)

```
ALTER TABLE Student ADD UNIQUE(s_id);
```

The above query specifies that s_id field of Student table will only have unique value.

PRIMARY KEY CONSTRAINT

Primary key constraint uniquely identifies each record in a database. A Primary Key must contain unique value and it must not contain null value. Usually Primary Key is used to index the data inside the table.

Using PRIMARY KEY constraint at Table Level

```
CREATE table Student (s_id int PRIMARY KEY, Name varchar(60)
NOT NULL, Age int);
```

The above command will creates a PRIMARY KEY on the s_id.

Using PRIMARY KEY constraint at Column Level

```
ALTER table Student ADD PRIMARY KEY (s_id);
```

The above command will creates a PRIMARY KEY on the s_id.

FOREIGN KEY CONSTRAINT

FOREIGN KEY is used to relate two tables. FOREIGN KEY constraint is also used to restrict actions that would destroy links between tables.

In Customer_Detail table, c_id is the primary key which is set as foreign key in Order_Detail table. The value that is entered in c_id which is set as foreign key in Order_Detail table must be present in Customer_Detail table where it is set as primary key. This prevents invalid data to be inserted into c_id column of Order_Detail table.

If you try to insert any incorrect data, DBMS will return error and will not allow you to insert the data.

Using FOREIGN KEY constraint at Table Level.

```
CREATE table Order_Detail
(
order_id int PRIMARY KEY,
order_name varchar(60) NOT NULL,
c_id int FOREIGN KEY REFERENCES Customer_Detail(c_id)
);
```

In this query, c_id in table Order_Detail is made as foriegn key, which is a reference of c_id column in Customer_Detail table.

Using FOREIGN KEY constraint at Column Level

```
ALTER table Order_Detail ADD FOREIGN KEY (c_id) REFERENCES
Customer_Detail(c_id);
```

Behaviour of Foriegn Key Column on Delete

There are two ways to maintin the integrity of data in Child table, when a particular record is deleted in the main table. When two tables are connected with Foriegn key, and certain data in the main table is deleted, for which a record exits in the child table, then we must have some mechanism to save the integrity of data in the child table.

1.On Delete Cascade : This will remove the record from child table, if that value of foriegn key is deleted from the main table.

2.On Delete Null : This will set all the values in that record of child table as NULL, for which the value of foriegn key is deleted from the main table.

3.If we don't use any of the above, then we cannot delete data from the main table for which data in child table exists. We will get an error if we try to do so.
ERROR : Record in child table exist.

CHECK Constraint

CHECK constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database. Its like condition checking before saving data into a column.
Using CHECK constraint at Table Level

```
CREATE table Student(
s_id int NOT NULL CHECK(s_id > 0),
Name varchar(60) NOT NULL,
Age int
);
```

The above query will restrict the s_id value to be greater than zero.
Using CHECK constraint at Column Level

```
ALTER table Student ADD CHECK(s_id > 0);
```

2.10.SQL DATA TYPE

SQL Data Type is an attribute that specifies the type of data of any object. Each column, variable and expression has a related data type in SQL. You can use these data types while creating your tables. You can choose a data type for a table column based on your requirement.

AUTHORIZATION

We may assign a user several forms of authorizations on parts of the database.
Authorizations on data include:
1. Authorization to read data.
2. Authorization to insert new data.
3. Authorization to update data.
4. Authorization to delete data.

Each of these types of authorizations is called a privilege. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

Granting and Revoking of Privileges:

The SQL standard includes the privileges select, insert, update, and delete. The privilege all privileges can be used as a short form for all the allowable privi-leges. A user who creates a new relation is given all privileges on that relation automatically. The SQL data-definition language includes commands to grant and revoke privileges. The grant statement is used to confer authorization. The basic form of this statement is:

```
grant <privilege list>
on <relation name or view name>
to <user/role list>;
```

The privilege list allows the granting of several privileges in one command. The select authorization on a relation is required to read tuples in the relation. The following grant statement grants database users Amit and Satoshi select authorization on the department relation:

```
grant select on department to Amit, Satoshi;
```

This allows those users to run queries on the department relation. This grant statement gives users Amit and Satoshi update authorization on the budget attribute of the department relation:

```
grant update (budget) on department to Amit, Satoshi;
```

The insert authorization on a relation allows a user to insert tuples into the relation. The insert privilege may also specify a list of attributes; any inserts to the relation must specify only these attributes, and the system either gives each of the remaining attributes default values (if a default is defined for the

attribute) or sets them to null. The delete authorization on a relation allows a user to delete tuples from a relation.

To revoke an authorization, we use the revoke statement. It takes a form almost identical to that of grant:

revoke <privilege list>
on <relation name or view name>
from <user/role list>;

Thus, to revoke the privileges that we granted previously, we write revoke select on department from Amit, Satoshi;

revoke update (budget) on department from Amit, Satoshi;

Roles:

The notion of roles captures this concept. A set of roles is created in the database. Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users. Each database user is granted a set of roles (which may be empty) that she is authorized to perform. Roles can be created in SQL as follows:

create role instructor;

Roles can then be granted privileges just as the users can, as illustrated in this statement:

grant select on takes
to instructor;

Roles can be granted to users, as well as to other roles, as these statements show:

grant dean to Amit;
create role dean;
grant instructor to dean;
grant dean to Satoshi;

Thus the privileges of a user or a role consist of:

• All privileges directly granted to the user/role.
• All privileges granted to roles that have been granted to the user/role.

Authorization on Views in our university example, consider a staff member who needs to know the salaries of all faculty in a particular department, say the Geology department. This staff member is not authorized to see information regarding faculty in other departments. Thus, the staff member must be denied direct access to the instructor relation. But, if he is to have access to the information for the Geology department, he might be granted access to a view that we shall call geo instructor, consisting of only those instructor tuples pertaining to the Geology department. This view can be defined in SQL as follows:

create view geo instructor as
(select *
from instructor
where dept name = 'Geology');

Suppose that the staff member issues the following SQL query:

select *
from geo instructor;

Authorizations on Schema:

The SQL standard specifies a primitive authorization mechanism for the database schema:

Only the owner of the schema can carry out any modification to the schema, such as creating or deleting relations, adding or dropping attributes of relations, and adding or dropping indices

Transfer of Privileges:

A user who has been granted some form of authorization may be allowed to pass on this authorization to other users. By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. If we wish to grant a privilege and to allow the recipient to pass the privilege on to other users, we append the with grant option clause to the appropriate grant command. For example,

if we wish to allow Amit the select privilege on department and allow Amit to grant this privilege to others, we write:
grant select on department to Amit with grant option;
The creator of an object (relation/view/role) holds all privileges on the object, including the privilege to grant privileges to others.

Revoking of Privileges:

revocation of a privilege from a user/role may cause other users/roles also to lose that privilege. This behavior is called cascading revocation. In most database systems, cascading I the default behavior. However, the revoke statement may specify restrict in order to prevent cascading revocation:
revoke select on department from Amit, Satoshi restrict;

ADVANCED SQL

2.11. Accessing SQL From a Programming Language

SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general-purpose programming language. However, a database programmer must have access to a general-purpose programming language for at least two reasons:
1. Not all queries can be expressed in SQL , since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Cobol that cannot be expressed in SQL . To write such queries, we can embed SQL within a more powerful language.
2. Nondeclarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface —cannot be done from within SQL . Applications usually have several components, and querying or updating data is only one component; other components are written in general-purpose programming languages. For an integrated application, there must be a means to combine SQL with a general-purpose programming language. There are two approaches to accessing SQL from a general-purpose programming language:

• **Dynamic SQL :** A general-purpose program can connect to and communicate with a database server using a collection of functions (for procedural languages) or methods (for object-oriented languages). Dynamic SQL allows theprogram to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time. The dynamic SQL component of SQL allows programs to construct and submit SQL queries at runtime.

• **Embedded SQL** : Like dynamic SQL , embedded SQL provides a means by which a program can interact with a database server. However, under embedded SQL , the SQL statements are identified at compile time using a preprocessor. The preprocessor submits the SQL statements to the database system for precompilation and optimization; then it replaces the SQL statements in the application program with appropriate code and function calls before in voking the programming-language compiler.

JDBC

The JDBC standard defines an application program interface (API) that Java programs can use to connect to database servers.

```
public static void JDBCexample(String userid, String passwd)
{
try
{
Class.forName ("oracle.jdbc.driver.OracleDriver");
```

```
Connection conn =
DriverManager.getConnection("jdbc:oracle:thin:@db.yale.edu:1521:univdb",userid,
passwd);
Statement stmt = conn.createStatement();
try {
stmt.executeUpdate(
"insert into instructor values('77987', 'Kim', 'Physics', 98000)");
} catch ( SQLException sqle)
{
System.out.println("Could not insert tuple. " + sqle);
}
ResultSet rset = stmt.executeQuery(
"select dept name, avg (salary) "+" from instructor "+" group by dept name");
while (rset.next()) {
System.out.println(rset.getString("dept name") + " " +rset.getFloat(2));
}
stmt.close();
conn.close();
}
catch (Exception sqle)
{
System.out.println("Exception : " + sqle);
}
}
```

The above code shows an example Java program that uses the JDBC interface. It illustrates how connections are opened, how statements are executed and results processed, and how connections are closed. We discuss this example in detail in this section. The Java program must import java.sql.* , which contains the interface definitions for the functionality provided by JDBC .

1. Connecting to the Database
2. Shipping SQL Statements to the Database System
3. Retrieving the Result of a Query
4. Prepared Statements
5. Callable Statements
6. Metadata Features

ODBC:

The Open Database Connectivity (ODBC) standard defines an API that applications can use to open a connection with a database, send queries and updates, and get back results. Applications such as graphical user interfaces, statistics packages, and spreadsheets can make use of the same ODBC API to connect to any database server that supports ODBC .

Each database system supporting ODBC provides a library that must be linked with the client program. When the client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results. The above code shows an example of C code using the ODBC API.

Embedded SQL:

The SQL standard defines embeddings of SQL in a variety of programming languages, such as C, C++, Cobol, Pascal, Java, PL/I , and Fortran. A language in which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language constitute embedded SQL . Programs written in the host language can use the embedded SQL syntax to access and update data stored in a database. An embedded SQL program must be processed by a special preprocessor prior to compilation. The preprocessor replaces embedded SQL requests with

hostlanguage declarations and procedure calls that allow runtime execution of the database accesses. Then, the resulting program is compiled by the host-language compiler. This is the main distinction between embedded SQL and JDBC or ODBC . In JDBC , SQL statements are interpreted at runtime (even if they are prepared first using the prepared statement feature). When embedded SQL is used, some SQL-related errors (including data-type errors) may be caught at compile time. To identify embedded SQL requeststo the preprocessor, we use the EXEC SQL statement; it has the form:
EXEC SQL <embedded SQL statement >;
The exact syntax for embedded SQL requests depends on the language in which SQL is embedded. In some languages, such as Cobol, the semicolon is replaced with ENDEXEC. Before executing any SQL statements, the program must first connect to the database.
This is done using:
EXEC SQL connect to server user user-name using password;
The open statement for our sample query is as follows:
EXEC SQL open c;
We must use the close statement to tell the database system to delete the temporary relation that held the result of the query. For our example, this statement takes
the form
EXEC SQL close c;

2.12.PROCEDURES:

A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.
A subprogram can be created −

1. At the schema level
2. Inside a package
3. Inside a PL/SQL block

At the schema level, subprogram is a standalone subprogram. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.
A subprogram created inside a package is a packaged subprogram. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'. PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

1. Functions − These subprograms return a single value; mainly used to compute and return a value.
2. Procedures − These subprograms do not return a value directly; mainly used to perform an action.

This chapter is going to cover important aspects of a PL/SQL procedure. We will discuss PL/SQL function in the next chapter.

Parts of a PL/SQL Subprogram
Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts
−
1 Declarative Part
It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested

subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

2 Executable Part

This is a mandatory part and contains statements that perform the designated action.

3 Exception-handling

This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
< procedure_body >
END procedure_name;
```

Where,

1. procedure-name specifies the name of the procedure.
2. [OR REPLACE] option allows the modification of an existing procedure.
3. The optional parameter list contains name, mode and types of the parameters.

IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

1. Procedure-body contains the executable part.
2. The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Procedure created.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways −

- Using the EXECUTE keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named 'greetings' can be called with the EXECUTE keyword as −

```
EXECUTE greetings;
```

The above call will display −

```
Hello World
PL/SQL procedure successfully completed.
```

The procedure can also be called from another PL/SQL block −

```
BEGIN
```

```
greetings;
END;
/
```
The above call will display −
```
Hello World
PL/SQL procedure successfully completed.
```
Deleting a Standalone Procedure:

A standalone procedure is deleted with the DROP PROCEDURE statement. Syntax for deleting a procedure is −
```
DROP PROCEDURE procedure-name;
```
You can drop the greetings procedure by using the following statement −
```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms:

The following table lists out the parameter modes in PL/SQL subprograms −

S.No Parameter Mode & Description

1.IN

An IN parameter lets you pass a value to the subprogram. It is a read-only parameter. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. It is the default mode of parameter passing. Parameters are passed by reference.

2 OUT

An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. The actual parameter must be variable and it is passed by value.

3 IN OUT

An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. Actual parameter is passed by value.

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.
```
DECLARE
a number;
b number;
c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
IF x < y THEN
z:= x;
ELSE
z:= y;
END IF;
END;
BEGIN
a:= 23;
```

```
b:= 45;
findMin(a, b, c);
dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Minimum of (23, 45) : 23
PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
x := x * x;
END;
BEGIN
a:= 23;
squareNum(a);
dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Square of (23): 529
PL/SQL procedure successfully completed.
```

Methods for Passing Parameters

Actual parameters can be passed in three ways −

- Positional notation
- Named notation
- Mixed notation

Positional Notation

In positional notation, you can call the procedure as −

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, a is substituted for x, b is substituted for y, c is substituted for z and d is substituted for m.

Named Notation

In named notation, the actual parameter is associated with the formal parameter using the arrow symbol (=>). The procedure call will be like the following −

```
findMin(x => a, y => b, z => c, m => d);
```

Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal −

```
findMin(a, b, c, m => d);
```

However, this is not legal:

```
findMin(x => a, b, c, d);
```

FUNCTIONS:

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
< function_body >
END [function_name];
```

Where,

1. function-name specifies the name of the function.
2. [OR REPLACE] option allows the modification of an existing function.
3. The optional parameter list contains name, mode and types of the parameters.
4. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
5. The function must contain a return statement.
6. The RETURN clause specifies the data type you are going to return from the function.
7. Function-body contains the executable part.
8. The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table. We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter −

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+----+----------+-----+-----------+----------+
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
total number(2) := 0;
BEGIN
SELECT count(*) into total
FROM customers;

RETURN total;
```

```
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result − Function created.

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the last end statement is reached, it returns the program control back to the main program. To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function totalCustomers from an anonymous block −

```
DECLARE
c number(2);
BEGIN
c := totalCustomers();
dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total no. of Customers: 6
PL/SQL procedure successfully completed.
```

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
a number;
b number;
c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
z number;
BEGIN
IF x > y THEN
z:= x;
ELSE
Z:= y;
END IF;
RETURN z;
END;
BEGIN
a:= 23;
b:= 45;
c := findMax(a, b);
dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result − Maximum of (23,45): 45
PL/SQL procedure successfully completed.
PL/SQL Recursive Functions
We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.
To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −
n! = n*(n-1)!
= n*(n-1)*(n-2)!
...
= n*(n-1)*(n-2)*(n-3)... 1
The following program calculates the factorial of a given number by calling itself recursively –

```
DECLARE
num number;
factorial number;

FUNCTION fact(x number)
RETURN number
IS
f number;
BEGIN
IF x=0 THEN
f := 1;
ELSE
f := x * fact(x-1);
END IF;
RETURN f;
END;
BEGIN
num:= 6;
factorial := fact(num);
dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result − Factorial 6 is 720

PL/SQL procedure successfully completed.

TRIGGERS:

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

- A database manipulation (DML)s tatement (DELETE, INSERT, or

UPDATE)

- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation(SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

1. Generating some derived column values automatically
2. Enforcing referential integrity
3. Event logging and storing information on table access
4. Auditing
5. Synchronous replication of tables
6. Imposing security authorizations
7. Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is −

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
Declaration-statements
BEGIN
Executable-statements
EXCEPTION
Exception-handling-statements
END;
```

Where,

1. CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with thetrigger_name.
2. {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view. {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
3. [OF col_name] − This specifies the column name that will be updated.
4. [ON table_name] − This specifies the name of the table associated with the trigger.
5. [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
6. [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
7. WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

   Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

```
Select * from customers;
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
```

1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
+----+----------+-----+-----------+----------+

The following program creates a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
sal_diff number;
BEGIN
sal_diff := :NEW.salary - :OLD.salary;
dbms_output.put_line('Old salary: ' || :OLD.salary);
dbms_output.put_line('New salary: ' || :NEW.salary);
dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result − Trigger created.

The following points need to be considered here −

OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers. If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state. The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The

UPDATE statement will update an existing record in the table –

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result −
Old salary: 1500
New salary: 2000
Salary difference: 500

2.13. Relational Algebra

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are

1. select: σ
2. project: Π
3. union: ∪
4. set difference: –
5. Cartesian product: x
6. rename: ρ

The operators take one or two relations as inputs and produce anew relation as a result. The select, project, and rename operations are called unary operations, because they operate on one relation. The other three operations operate on pairs of relations and are, therefore, called binary operations. The Select Operation The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. The instructor relation

Thus, to select those tuples of the instructor relation where the instructor is in the “Physics” department, we write: σ dept name = “Physics” (instructor) If the instructor relation is as shown in table, then the relation that results from the preceding query is as shown in Figure 2. We can find all instructors with salary greater than $90,000 by writing: σ salary > 90000 (instructor).

In general, we allow comparisons using =, =, , and ≥ in the selection predicate. Furthermore,we can combine several predicates into a larger predicate by using the connectives and (∧), or (∨), and not (¬). Thus, to find the instructors in Physics with a salary greater than $90,000, we write:

σ dept name =“Physics”∧ salary > 90000 (instructor)

The Project Operation Suppose we want to list all instructors’ ID, name, and salary, but do not care about the dept name. The project operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter pi (Π). Π ID, name, salary (instructor).

The Union Operation:

Consider a query to find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both. The information is contained in the section relation. To find the set of all courses taught in the Fall 2009 semester, we write: Π course id (σ semester =“Fall”∧year=2009 (section)) To find the set of all courses taught in the Spring 2010 semester, we write: Π course id (σ semester =“Spring”∧year=2010 (section))

To answer the query, we need the union of these two sets; that is, we need all section IDs that appear in either or both of the two relations. We find these data by the binary operation union, denoted, as in