RELATIONAL ALGEBRA

11.1 Introduction:

Relational Algebra is a procedural language that can be used to tell the DBMS how to build a new relation from one or more relations in the database. While using the relational algebra user has to specify what is required and what are the procedure or steps to obtain the required output. Relational algebra is a formal and user friendly language. It is used as the basis for other high level Data Manipulation Languages (DMLs) for relational databases. It illustrates the basic operations required of any DML and serve as the standard of comparison for other relational databases.

11.2 Relational Algebra

The relational algebra is a theoretical language with operations that work on one or more relations to define another relation without changing the original relation(s). Thus, both the operands and the results are relations and so the output from one operation can become the input to another operation. This allows expressions to be nested in the relational algebra just as we nest arithmetic operations. This property is called closure: relations are closed under the algebra just as numbers are closed under arithmetic operations.

There are many variations of the operations that are included in relational algebra Codd originally proposed Eight operations, but several others have been developed.

The five fundamental operations in relational algebra are

1) Selection
2) Projection
3) Cartesian Product
4) Union
5) Difference

They perform most of the data retrieval operations, which can be expressed in terms of the five basic operations.

In relational algebra each operation takes one or more relations as its operands and produces another relation as its result. Consider an example of mathematical algebra as shown below

3+5=8

Here 3 and 5 are operands and + is an arithmetic operator which gives result as 8.

Similarly, in relational algebra R1+ R2 = R3. Here R1 and R2 are relations (operands) and + is the relational operator which gives R3 as a resultant relation.

A) BASIC RELATIONAL ALGEBRA OPERATIONS

Basic relational algebra operations are also called as traditional set operators , the various traditional set operators are :

1) UNION
2) INTERSECTION
3) DIFFERENCE
4) CARTESIAN PRODUCT

UNION

In mathematical set theory, the union of two sets is the set of all elements belonging to both sets. The set, which results from the union, must not of course contain duplicate elements. It is denoted by U. Thus the union of sets:

S1 = { 1 , 2 , 3 , 4, 5 } and
S2 = { 4 , 5 , 6, 7 , 8 }

would be the set { 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 } .

A union operation on two relational tables follows the same basic principle but is more complex in practice. In order to perform the Union operation, both operand relations must be union compatible i.e. they must have same number of columns drawn from the same domain (means must be of same data type}

Suppose two tables, R and S have the following tuples at some instant in time and that their header parts are as shown below:

R

Cust_name	**Cust_status**
Sham	Good
Rahul	Excellent
Mohan	Bad
Sachin	Excellent
Dinesh	Bad

S

Cust_name	**Cust_status**
Karan	Bad
Sham	Good
Sachin	Excellent
Rohan	Average

These can certainly be combined into one table containing a valid relation by the relational union operator (R U S) as follows :

R U S

Cust_name	**Cust_status**
Sham	Good
Rahul	Excellent
Mohan	Bad
Sachin	Excellent
Dinesh	Bad
Karan	Bad
Rohan	Average

INTERSECTION

In mathematics an intersection of two sets produces a set, which contains all the elements that are common to both sets. Thus the intersection of two sets:

S1 = { 1 , 2 , 3 , 4 , 5 } and
S2 = { 4 , 5 , 6 , 7 , 8 }
would be { 4 , 5 } .

In above example both the tables are union compatible and can be intersected together. The intersection operation on the R and S tables defined above would be

Cust_name	**Cust_status**
Sham	Good
Sachin	Excellent

The intersection operator is used in the similar fashion to the union operator, but provides an ‘and ‘ function.

DIFFERENCE

In mathematics, the difference between two sets S1 and S2 produces a set, which contains all the members of one set, which are not in the other. It is denoted by “ – “ sign.

The order in which the difference is taken is obviously significant. Thus the difference between two sets:

S1 = { 1 , 2 , 3 , 4 , 5 }
Minus
S2 = { 4 , 5 , 6 , 7 , 8 }
Would be { 1 , 2 , 3 } and between
S2 = { 4 , 5 , 6 ,7 , 8 }
Minus
S1 = { 1, 2 , 3 , 4 , 5 }
would be { 6 , 7 , 8 }

As for the other set operations discussed so far, the difference operation can also be performed on tables that are union compatible. The difference operation on the R and S (R – S) defined above would return.

R – S

Cust_name	Cust_status
Rahul	Excellent
Mohan	Bad
Dinesh	Bad

And for **S – R**

Cust_name	Cust_status
Karan	Bad
Rohan	Average

It is used in a similar fashion to the union and intersection operators , but provides a qualifying “not” function .

Minus is not associative

In order to prove this mathematically consider three sets A, B, C with following members

A = { 1 , 2 , 3 , 4 , 5 }

B = { 2 , 3 }

C = { 1 , 4 }

(A MINUS B) MINUS C = { 1 , 4 , 5 } MINUS { 1 , 4 } = { 5 }

A MINUS { B MINUS C) = { 1 , 2 , 3 , 4 , 5 } MINUS { { 2 , 3 }MINUS { 1 , 4 }} = { 1 , 2 , 3 , 4 , 5 } MINUS { 2 , 3 } = { 1 , 4 , 5 }

Both the cases give different result. So minus is not an associative operator.

Minus is not commutative

It means that A MINUS B is different from B MINUS A . In order to prove it we again take the above values of A and B .

A MINUS B = { 1 , 4 , 5 }

B MINUS A is empty or null because there is not any value, which is in B but not in A.

CARTESIAN PRODUCT

In mathematics, the Cartesian product of two sets is the set of all ordered pairs of elements such that the first element in each pair belongs to the first set and the second element in each pair belongs to the second set. It is denoted by cross (x). It is for example, given two sets:

S1 = { 1 , 2 , 3 } and

S2 = { 4 , 5 , 6 }

The Cartesian product S1 x S2 is the set :

{ (1, 4), (1, 5), (1 , 6), (2, 4), (2, 5), (2 , 6), (3, 4), (3, 5), (3 , 6) }

Consider the two tables with sample population as below

Female

Name	**Job**
Komal	Clerk
Amita	Sales
Sonia	Production
Nidhi	Clerk

Male

Name	**Job**
Rohit	Clerk
Amit	Sales
Sohan	Production
Nitin	Clerk

Assume that the tables refer to male and female staff respectively. Now, in order to obtain all possible inter-staff marriages, the Cartesian product can be taken giving the Table MALE_FEMALE.

Male-Female

Female_Name	**Female_Job**	**Male_Name**	**Male_Job**
Komal	Clerk	Rohit	Clerk
Komal	Clerk	Amit	Sales
Komal	Clerk	Sohan	Production
Komal	Clerk	Nitin	Clerk
Amita	Sales	Rohit	Clerk
Amita	Sales	Amit	Sales
Amita	Sales	Sohan	Production
Amita	Sales	Nitin	Clerk
Sonia	Sales	Rohit	Clerk
Sonia	Sales	Amit	Sales
Sonia	Sales	Sohan	Production
Sonia	Sales	Nitin	Clerk
Nidhi	Clerk	Rohit	Clerk

Nidhi	Clerk	Amit	Sales
Nidhi	Clerk	Sohan	Production
Nidhi	Clerk	Nitin	Clerk

In order to preserve unique names for attributes, the original attribute names have had to be concantenated with the original tablenames. The new table has also been given an identity.

B) SPECIAL RELATIONAL OPERATIONS

There are four special relational algebra operations which are as under

1) SELECTION
2) PROJECTION
3) JOIN
4) DIVISION

Selection

The selection operator yields a horizontal subset of a given relation that is that subset of tuples or rows of a table should be selected within the given relation for which a particular condition is satisfied.

In mathematics a set can have any number of subsets. A set is said to be a subset of another if all its members are also members of the other set. Thus, in the following example:

S1 = { 1 , 2 , 3 , 4 , 5 }

S2 = { 2 , 3 , 4 }

S2 is a subset of S1. Since the body part of a table is a set, it is possible for it to have subsets, that is a selection from its tuples can be used to form another relation.

However, this would be a meaningless operation of no new information were to be gained from the new relation. On the other hand a subset if say an EMPLOYEE relation , which contained all tuples where the employee represent those employees who earn more than some given values of salary, would be useful. What is required is that some explicit restriction be placed on the sub-setting operation.

Restriction as originally defined was defined on relations only and is achieved using the comparison operators such as equal to (=), not equal to (!=), greater than (>), less than (<), greater than or equal to (>=) and less than or equal to (<=).

Example : Consider the database having following tables :

The **Supplier** table

SNo	**Sname**	**Status**	**City**
S1	Suneet	20	Qadian
S2	Ankit	10	Amritsar
S3	Amit	30	Amritsar
S4	Raj	20	Amritsar

The **Parts** table

Pno	Pname	Color	Weight	City
P1	Nut	Red	12	Qadian
P2	Bolt	Red	17	Amritsar
P3	Screw	Blue	17	Jalandhar
P4	Screw	Red	14	Qadian

The **Shipment** table

SNo	Pno	Qty
S1	P1	250
S2	P2	300
S3	P3	500
S4	P1	250
S5	P2	500
S6	P2	300

Here in Supplier table

Sno	-	Supplier number of supplier that is unique
Sname	-	Supplier name
City	-	City of the supplier
Status	-	Status of the city e.g A grade cities may have status 10 , B grade cities may have status 20 and so on .

Examples :

S WHERE CITY = ‘ Qadian ‘

Sno	Sname	Status	City
S1	Suneet	20	Qadian

P WHERE WEIGHT < 15

Pno	Pname	Color	Weight	City
P1	Nut	Red	12	Qadian
P4	Screw	Red	14	Qadian

SP where Sno = ‘ S1’ and Pno = ‘P1’

Sno	Pno	Qty
S1	P1	300

PROJECTION

The projection operation on a table simply forms another table by copying specified columns (both header and body parts) from original table eliminating any duplicated rows. The projection operator yields a vertical subset of a given relation – that is, the subset obtained by selecting specified attributes, in a specified left to right order, and then eliminating duplicate tuples within the attributes selected. It is denoted by pi (⑥). For example consider the table EMPLOYEE as shown :

Table **Employee**

Personnel_number	**Name**	**Age**	**Salary**
123	Sham	23	7500
124	Karan	43	10000
125	Rahul	23	10000

The projections of the ' age ', the ' age and salary ' and the ' personnel _number and name ' columns would return the three tables , say , A , B and C respectively :

A

⑥ age **(employee)**

Age
23
43

B

⑥ age,salary **(employee)**

Age	**Salary**
23	7500
43	10000
23	10000

C

⑥ personnel_number,name **(employee)**

Personnel_number	**Name**
123	Sham
124	Karan
125	Rahul

JOIN

The most general form of join operation is called a theta join, where theta has the same meaning as 'compares with' as it was used in the context of the restriction operation. That is, it stands for any of the comparative operators equals, not equals, greater than and so forth. A theta join is performed on two tables, which have one or more columns in common which are domain compatible.

It forms a new table which contains all the columns from both the joined tables whose tuples are those defined by the restriction applied.

For example consider the tables:
EMPLOYEE_PRODUCT

Name	**Product**
Raja	Pen
Sparsh	Pen
Raja	Pencil
Sparsh	Rubber

PRODUCT_CUSTOMER

C_Product	**Customer**
Pen	Karan
Pen	Suneet
Pencil	Suneet

The tables list employees who make products and customers who buy those products and can be joined over the columns 'product' and 'c_product' in both tables since the values in both columns are domain compatible. The result of a theta join, where the restriction is that the product attribute values in EMPLOYEE_PRODUCT should be equal to the product attribute values in PRODUCT_CUSTOMER would be:

Table EMPLOYEE_PRODUCT_CUSTOMER

Name	**Product**	**C_Product**	**Customer**
Raja	Pen	Pen	Karan
Raja	Pen	Pen	Suneet
Raja	Pencil	Pencil	Suneet
Sparsh	Pen	Pen	Karan
Sparsh	Pen	Pen	Suneet

Note: If both tables have same common column then one of the common column has to be renamed in the resultant table to preserve the uniqueness of the names in its header part.

In the above example the theta operator was 'equals' and this , the most common form of theta join is referred to as an equi-join. Note that an equi-join must always result in a table which has pairs of columns like 'product; and 'c_product' in the above example, which contain identical lists of attribute values.

By far the most common form of join is a variation of the equi-join where this duplication of column values is eliminated by taking a projection of the table which includes only one of the duplicated columns. This is reffered to as a natural join.

The natural join of the tables in the last example would give the table :

Name	**Product**	**Customer**
Raja	Pen	Karan
Raja	Pen	Suneet
Raja	Pencil	Suneet
Sparsh	Pen	Karan
Sparsh	Pen	Suneet

It may help in understanding the different types of join if the operation is looked at from a different point of view. The join is actually a composite operator. The theta join is a Cartesian product operation on the two tables followed by a restriction operation on the resultant table.

The tuples of the Cartesian product of the two tables in the earlier example would be :

Name	**Product**	**C_Product**	**C_Customer**
Raja	Pen	Pen	Karan
Raja	Pen	Pen	Suneet
Raja	Pen	Pencil	Suneet
Sparsh	Pen	Pen	Karan
Sparsh	Pen	Pen	Suneet
Sparsh	Pen	Pencil	Suneet
…….	……	……	…..
Raja	Pencil	Pencil	Suneet

The restriction operation on this product selects only those tuples from this relation, which confirm to the restriction . In the example, the restriction was that the 'product' attributes should have equal values in each tuple and the result of this as shown below:

Name	**Product**	**C_Product**	**Customer**
Raja	Pen	Pen	Karan
Raja	Pen	Pen	Suneet
Raja	Pencil	Pencil	Suneet
Sparsh	Pen	Pen	Karan
Sparsh	Pen	Pen	Suneet

Since theta equated to 'equals' this was an equi-join. By carrying out a further projection operation which eliminates one of the duplicated 'product' column resulting from the equi-join, the natural join isobtained.

Thus, Join operator is combination of Cartesian product, Selection and Projection operator.

The examples given so far have all been of so-called inner joins. The fact that Jones makes Rubbers is not recorded in any of the resultant tables from the joins, because the joining values must exist in both tables. If it suffices that the value exist in only one table, then a so-called outer join is produced.

An outer join of the EMPLOYEE_PRODUCT and PRODUCT_ CUSTOMER tables exemplified above would return :

Employee_name	Product_name	Customer_name
Raja	Pen	Karan
Raja	Pen	Suneet
Sparsh	Pen	Karan
Sparsh	Pen	Suneet
Raja	Pencil	Suneet
Sparsh	Rubber	-

The expression A JOIN B is defined if and only if, for every unqualified attribute-name that is common to A and B, the underlying domain is the same for both relations. Assume that this condition is satisfied. Let the qualified attribute –names for A and B, in their left-to-right order, be **A.A**1,**............A.Am** AND **B.B** (m+1)......................**, B.B** (m+n) respectively;

Let **C**i **.......,C**j be the unqualified attribute name that are common to A and B and let **B**r**..........B**s be the unqualified attribute- names remaining for b (with their relative order undisturbed) after removal of **C**i, ... **C**j.

Then **A JOIN B** defined to be equivalent to (**A TIMES B) [A.A**1 **..........A.A**m **, B.B**r ....**B.B**s **]**

where **A.C**i = **B.C**i

and ....

and **A.C**j = **B.C**j**.........**

Apply this definition to JOIN operation on Emp and Dept tables with following attributes:

EMP(empno,ename,job,sal,deptno)

DEPT(deptno,dname,loc)

EMP JOIN DEPT = EMP TIMES DEPT

[emp.empno,emp.ename,emp.job,emp.sal,emp.deptno,dept.dname, dept.loc] where EMP.deptno = DEPT. deptno

So, w can say that JOIN is a combination of Product, Selection and Projection operators. JOIN is an associative operator, which means:

(A JOIN B) JOIN C = A JOIN (B JOIN C) .

JOIN is also commutative .

A JOIN B = B JOIN A

DIVISION

The division operator divides a dividend relation A of degree (means number of columns in a relation) m+n by a divisor relation B of degree n and produces a resultant relation of degree m .

Relation A

Sno	Pno
S1	P1
S1	P2
S1	P3
S1	P4
S1	P5
S1	P6
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

Relation B

CASE 1

Pno
P1

CASE 2

Pno
P2
P4

CASE 3

Pno
P1
P2
P3
P4
P5
P6

A DIVIDED BY B

CASE 1

Sno
S1
S2

CASE 2

Sno
S1
S4

Case 3

Sno
S1

In this example dividend relation A has two attributes of Sno,Pno (of degree 2) and division relation B has only one attribute Pno (of degree 1). Then A divided by B gives a resultant relation of degree 1. It means it has only one attribute of Sno.

$$\frac{A}{B} = \frac{SNO * PNO}{PNO} = SNO$$

The resultant relation has those tuples that are common values of those attributes, which appears in the resultant attribute sno .

For example ,in CASE 2,

P2 has Snos → S1,S2,S3,S4

P4 has Snos → S1,S4

S1, S4 are the common supplier who supply both P2 and P4. So the resultant relation has tuples S1 and S4.

In CASE 3

There is only one supplier S1 who supply all the parts from P1 to P6.

11.3 Summary

Relational Algebra is a procedural language which specifies the operations to be performed on the existing relations to derive result relations. Relational Algebric operations can divided into basic and special relational operators. Relational Calculus is a non procedural language which is an alternate way of formulating queries. It is based on Predicate Calculus which means to formulate set of predicates to which the answer to a query must conform instead of specifying a series of subsequent singular operations together with objects involved in these operations.

11.4 Questionnaires

Q1. What is relational Algebra and what are its uses?

Q2. Explain the following operations with examples:

1. Union 2. Intersection 3. Differenc
4. Cartesian Product 5. Division

PAPER : DBMS

LESSON NO. : 12

DATA BASE DESIGN

12.1 Introduction

The concept of functional dependency is the basis for Normalization. The functional dependencies are the consequence of the interrelationships among attributes of a relation (table) represented by some link or association. It must be taken care that the database design must be very good and that needs careful decomposition of the relations into further relations. In the following sections we will study how to decompose the relations so that it leads to good database design. And if we do not do decomposition with care it will result in bad database design which includes repetition of data like problems.

12.2 Functional Dependencies

Functional dependencies play a key role in differentiating good database designs from bad database design. A functional dependency is a type of constraint that is a generalization of the notion of key.

12.2.1 *Basic Concepts*

Functional dependencies are constraints on the set of legal relations. They allow us to express facts about the enterprise that we are modeling with our database.

Functional Dependency is a many-to-one relationship from one set of attributes to another within a given relation.

We define the notion of a super-key as follows. Let R be a relation schema. A subset K of R is a super-key of R if, in any legal relation r(R), for all pairs t_1 and t_2 of tuples in r such that $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$. That is no two tuples in any legal relation in r (R) may have the same value on attribute set K.

The notion of functional dependency generalizes the notion of super-key.
Consider a relation schema R, and let A ¯R and þ ¯R. The functional dependency

A → þ

holds on schema R if, in any legal relation r(R) for all pairs of tuples t_1 and t_2 in r such that t_1[A] = t_2 [A], it is also the case that t_1[þ] = t_2[þ].

Using the functional-dependency notation, we say that K is a super-key of R if K→ R. That is K is a super-key if, whenever $t_1[K] = t_2$ [K] it is also the case that $t_1[R] =$ t_2 [R] (that is $t_1 = t_2$).

Functional dependencies allow us to express constraints that we cannot express with super-keys. Consider the schema

Loan-info-schema = (*loan-number*, *branch-name*, *customer-name*, *amount*) which is simplification of the lending-schema that we saw earlier. The set of functional dependencies that we expect to hold on this relation schema is

loan-number → *amount*
loan-number → *branch-name*

We would not, however, expect the functional dependency

loan-number → *customer-name*

to hold, since in general a given loan can be made to more than one customer (for example, to both members of a husband – wife pair)

We shall use functional dependencies in two ways:

1 To test relations to see whether they are legal under a given set of functional dependencies. If a relation r is legal under a set F of functional dependencies, we say that r satisfies F.

2 To specify constraint on the set of legal relations. We shall thus concern ourselves with only those relations that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on schema R that satisfy a set F of functional dependencies, we say that F holds on R. Let us consider the relation r of figure below:

A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2
a2	b2	c2	d2
a2	b3	c2	d3
a3	b3	c2	d4

Sample relation r

to see which functional dependencies are satisfied. Observe that A→C is satisfied. There are two tuples that have an A value of a_1. These have the same C value – namely c_1. Similarly, the two tuples with an A value of a_2 have the same C value, c_2. There are not other pairs of distinct tuples that have the same a value. The functional dependency C—A is not satisfied however. To see that it is not, consider the tuples t_1 = (a_2, b_3,c_2, d_3) and t_2 = (a_3, b_3, c_2, d_4) these two tuples have the same C values c_2, but they have different A values a_2 and a_3, respectively. Thus we have found a pair of tuples t_1 and t_2 such that t_1 [C] = t_2 [C] but $t_1[A] \neq t_2[A]$.

Many other functional dependencies are satisfied by r, including, for example, the functional dependency AB→D. Note that we use AB as shorthand for {A, B}, to conform with standard practice. Observe that there is no pair of distinct tuples t_1 and t_2 such that $t_1[AB] = t_2[AB]$. Therefore, if $t_1[AB] = t_2[AB]$, it must be that $t_1 = t_2$ and thus $t_1[D] = t_2[D]$. So satisfies AB→D.

Some functional dependencies are said to be trivial because they are satisfied by all relations. For example, A→A is satisfied by all relations involving attribute A. Reading the definition of functional dependency literally, we see that, for all tuples t_1 and t_2 such that $t_1[A] = t_2[A]$ it is the case that $t_1[A] = t_2[A]$. Similarly, AB → A is satisfied by all relations involving attribute A. In general a functional dependency of the form A → þ is trivial if þ ≤ A.

To distinguish between the concepts of a relation satisfying a dependency and a dependency holding on a schema, we return to the banking example. If we consider the *customer* relation (on *customer-schema*) in Figure below, we see that *customer-street* → *customer-city* is satisfied. However, we believe that in the real world, two cities can have streets with the same name.

Customer-name	**Customer-street**	**Customer-city**
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

The *customer* relation

Thus, it is possible, at some time to have an instance of the *customer* relation in which *customer-street*→ *customer-city* is not satisfied. So we would not include *customer-street*→ *customer-city* in the set of functional dependencies that hold on *Customer-schema.*

In the *loan* relation (on *loan-schema*) of figure below, we see that the dependency *loan-number* → *amount* is satisfied. In contrast to the case of *customer-city* and *customer-street* in *customer-schema*, we do believe that the real world enterprise that we are modeling requires each loan to have only one amount. Therefore we want to require that *loan-number*→ *amount* be satisfied by the *loan* relation at all times. In other words, we require that the constraint *loan number*→ *amount* hold on *loan-schema.*

The *loan* relation:

Loan-number	**Branch-name**	**Amount**
L-17	Downtown	1000
L-23	Redwood	2000
L-15	Perryridge	1500
L-14	Downtown	1500
L-93	Mianus	500
L-11	Round Hill	900
L-29	Pownal	1200
L-16	North Town	1300
L-18	Downtown	2000
L-25	Perryridge	2500
L-10	Brighton	2200

In the branch relation of Figure below, we see that *branch-name*→ *assets* is satisfied, as is *assets*→ *branch-name*. We want to require that *branch-name*→ *assets* hold on *branch-schema*. However we do not wish to require that *assets*→ *branch-name* hold since it is possible to have several branches that have the same asset value.

Branch-name	**Branch-city**	**Assets**
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
Pownal	Bennington	300000
North Town	Rye	3700000
Brighton	Brooklyn	7100000

The *branch* relation

In what follows, we assume that, when we design a relational database, we first list those functional dependencies that must always hold. In the banking example our list of dependencies includes the following:

- On *branch-schema*:
  Branch-name→ branch-city
  Branch-name→ assets
- On *customer-schema*:
  customer-name→ customer-city
  customer-name→ customer-street
- On *Loan-schema*:
  Loan-number→ amount
  Loan-number→ branch-name
- On *Borrower-schema*:
  No functional dependencies
- On *Account-schema*:
  Account-number→ branch-name
  Account-number→ balance
- On *depositor-schema*:
  No functional dependencies

12.2.2 *Closure of a set of Functional dependencies*

It is not sufficient to consider the given set of functional dependencies. Rather, we need to consider all functional dependencies that hold. We shall see that given a set F of functional dependencies, we can prove that certain other functional dependencies hold. We say that such functional dependencies are “logically implied” by F.

More formally given a relational schema R, a functional dependency f on R is logically implied by a set of functional dependencies F of R if every relation instance r(R) that satisfied F also satisfies f.

Suppose we are given a relation schema R = (A, B, C, G, H, I,) and the set of functional dependencies

A→ B
A → C
CG→ H
CG→ I
B→ H

The functional dependency

A→ H

is logically implied. That is, we can show that, whenever our given set of functional dependencies holds on a relation, A → H must also hold on the relation. Suppose that t_1 and t_2 are tuples such that $t_1[A] = t_2 [A]$

since we are given that A → B, it follows from the definition of functional dependency that

$$t_1[B] = t_2 [B]$$

then, since we are given that B → H, it follows from the definition of functional dependency that

$$t_1[H] = t_2 [H]$$

Therefore it shows that whenever t_1 and t_2 are tuples such that $t_1 [A] = t_2[A]$ it must be that $t_1[H] = t_2[H]$. But that is exactly the definition of A → H.

Let f be a set of functional dependencies logically. The closure of F, denoted by F^+, is the set of all functional dependencies implied by F. Given F, we can compute f directly from the formal definition of functional dependency. If F were large, this process would be lengthy and difficult. Such a computation of F^+ requires arguments of the type just used to show that A → H is in the closure of our example set ofdependencies.

Axioms or rules of inference provide a simpler technique for reasoning about functional dependencies. In the rules that follow, we use Greek letters for sets of attributes, and uppercase Roman letters from the beginning of the alphabet for individual attributes. We use AÞ to denote A U þ.

We can use the following three rules to find implied functional dependencies. By applying these rules repeatedly, we can find all of F^+, given F. This collection of rules is called Armstrong's axioms in honor of the person who first proposed it.

- **Reflexivity rule**. If A is a set of attributes and þ A, then A → þ holds.
- **Augmentation rule**. If A → þ holds and y is a set of attributes, then yA → yþ holds.
- **Transitivity rule**. If A → þ holds and þ→ y holds, then A → y holds.

Armstrong's axioms are sound, because they do not generate any incorrect functional dependencies. They are complete, because for a given set F of functional dependencies, they allow us to generate all F^+.

Although Armstrong's axioms are complete, it is tiresome to use them directly for the computation of F^+. To simplify matters further, we list additional rules. It is possible to use Armstrong's axioms to prove that these rules are correct.

- **Union rule**. If A → þ holds and A → y holds, then A → þy holds.
- **Decomposition rule**. If A → þy holds, then A → þ holds and A → y holds.
- **Pseudotransitivity rule**. If A → þ holds and yþ→ 6 holds, then Ay→ 6 holds.

Let us apply our rules to the example of schema R = (A, B, C, G, H, I) and the set F of functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. We list several members of F^+ here.

- $A \rightarrow H$. Since $A \rightarrow B$ and $B \rightarrow H$ hold, we apply the transitivity rule. Observe that it was much easier to use Armstrong's axioms to show that $A \rightarrow H$ holds than it was to argue directly from the definitions, as we did earlier in this section.
- $CG \rightarrow HI$. Since $CG \rightarrow H$ and $CG \rightarrow I$, the union rule implies that $CG \rightarrow HI$.
- $AG \rightarrow I$. Since $A \rightarrow C$ and $CG \rightarrow I$, the pseudotransitivity rule implies that $AG \rightarrow I$ holds.

Another way of finding that $AG \rightarrow I$ holds is as follows. We use the augmentation rule on $A \rightarrow C$ to infer $AG \rightarrow CG$. Applying the transitivity rule to this dependency and $CG \rightarrow I$, we infer $AG \rightarrow I$.

Figure below shows a procedure that demonstrates formally how to use Armstrong's axioms to compute F^+. In this procedure, when a functional dependency is added to F^+, it may be already present, and in that case there is no change to F^+. We will also see an alternative way of computing F in next section.

```
$F^+ = F$
repeat
        for each functional dependency f in $F^+$
                apply reflexivity and augmentation rules on f
                add the resulting functional dependencies to $F^+$
        for each pair of functional dependencies $f_1$and $f_2$ in $F^+$
                if $f_1$ and $f_2$ can be combined using transitivity
                add the resulting functional dependency to $F^+$
        until $F^+$ does not change any further
```

The left-hand and right-hand sides of a functional dependency are both subsets of R. Since a set of size n has 2^n subsets, there are a total of $2 \times 2^{n} = 2^{n+1}$ possible functional dependencies, where n is the number of attributes in R. Each iteration of the repeat loop of the procedure, except the last iteration, adds at least one functional dependency to F^+. Thus, the procedure is guaranteed to terminate.

12.2.3 Closure of Attribute Sets

To test whether a set A is a super-key, we must devise an algorithm for computing the set of attributes functionally determined by A. One way of doing this is to compute F^+, take all functional dependencies with A as the left-hand side, and take the union of the right-hand sides of all such dependencies. However doing so can be expensive, since F^+ can be large·

An efficient algorithm for computing the set of attributes functionally determined by A is useful not only for testing whether A is a super-key, but also for several other tasks, as we will see later in thus section.

Let A be a set of attributes. We call the set of all attributes functionally determined by A under a set F of functional dependencies the closure of A under F; we denote it by A^+. Figure below shows an algorithm written in pseudocode to compute A^+. The input is a set F of functional dependencies and the set A of attributes. The output is stored in the variable *result.*

```
result := A
while (changes to result) do
        for each functional dependency þ→ y in F do
        begin
                if þ ≤ result then result := result U y
        end
```

To illustrate how the algorithm works, we shall use it to compute $(AG)^+$ with the functional dependencies defined in preceding section. We start with result = AG. The first time that we execute the while loop to test functional dependency, we find that

- A→ B cause us to include B in *result.* To see fact, we observe that A→ B is in F, A ≤ *result* (which is AG), so *result* := *result* U B.
- A→ C causes *result* to become ABCG.
- CG→ H causes *result* to become ABCGH.
- CG→ I causes *result* to become ABCGHI.

The second time that we execute the while loop, no new attributes are added to result, and the algorithm terminates.

Let us see why the algorithm of Figure above is correct. The step is correct, since A→ A always holds (by the reflexivity rule). We claim that for any subset þ of result, A→þ. Since we start the while loop with A→ *result* being true, we can add y to result only if þ ≤ *result* and þ→ y. But then *result* → þ by the reflexivity rule, so A→ þ by transitivity. Another application of transitivity shows that A→ y (using A→ þ and þ→ y). The union rule implies that A→ *result* U y, so functionally determines any new result generated in the while loop. Thus any attribute returned by the algorithm is in A^+.

It is easy to see that the algorithm finds all A^+. If there is an attribute in A^+ that is not yet in *result,* then there must be a functional dependency þ→ y for which þ ≤ *result,* and at least one attribute in y is not in *result.*

It turns out that, in the worst case, this algorithm may take an amount of time quadratic in the size of F.

There are several uses of the attribute closure algorithm:

- To test if A is a super-key, we compute A^+, and check if A^+ contains all attributes of R.

- We can check if a functional dependency A→ þ holds (or, in other words, is in F^+) by checking if þ ¯ A^+. That is we compute A^+ by using attribute closure and then check if it contains þ. This test is particularly useful, as we will see later in this chapter.
- It gives us an alternative way to compute F^+: for each y ¯R, we find the closure y^+, and for each S ¯ y^+, we output a functional dependency y → S.

12.3 Decomposition

The bad design of database suggests that we should decompose a relation schema thathas many attributes into several schemas with fewer attributes. Careless decomposition, however may lead to another form of bad design.

Consider an alternative design in which we decompose Lending-schema into the following two-schemas:

Branch-customer-schema= (*branch-name, branch-city, assets, customer-name*)

Customer-loan-schema = (*customer-name, loan-number, amount*)

Using the lending relation described in "Problems arising out of bad database design" topic (Figure *W*) that we will discuss next, we construct our new relations *branch-customer* (*Branch-customer*) and *customer-loan* (*customer-loan-schema*):

branch–customer = $\Pi_{\textit{branch-name}, \textit{branch-city}, \textit{assets}, \textit{customer-name}}$ (lending)

customer-loan = $\Pi_{\textit{customer-name}, \textit{loan-number}, \textit{amount}}$ (lending)

Figure *X* and *Y* respectively show the resulting branch-customer and customer-name relations.

Branch-name	**Branch-city**	**Assets**	**Customer-name**
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks

Figure *X*: The relation *branch-customer*

Customer-name	**Loan-number**	**Amount**
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-18	2000
Glenn	L-25	2500
Brooks	L-10	2200

Figure Y: The relation *customer-loan*

Of course, there are cases in which we need to reconstruct the *loan* relation. For example, suppose that we wish to find all branches that have loans with amounts less than $1000. No relation in our alternative database contains these data. We need to reconstruct the lending relation. It appears that we can do so by writing

branch-customer | *customer-loan*

Figure *Z* below shows the result of computing *branch-customer* | *customer-loan*.

Branch-name	**Branch-city**	**Assets**	**Customer-name**	**Loan-number**	**Amount**
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

When we compare this relation and the lending relation with which we started (Figure *W*), we notice a difference: Although every tuple that appears in the lending relation appears in *branch-customer* | *customer-loan*, there are tuples in *branch-customer* | *customer-loan* that are not in *lending*. In our example, *branch-customer* | *customer-loan* has the following additional tuples:

(Downtown, Brooklyn, 9000000, Jones, L-93, 5000
(Perryridge, Horseneck, 1700000, Hayes L-16, 1300)
(Mianus, Horseneck, 400000 Jones, L-17, 1000)
(North Town, Rye 3700000, Hayes L-15, 1500)

Consider the query "Find all bank branches that have made a loan in an amount less than $1000. If we look back at Figure *W*, we see that the only branches with loan amounts less than $1000 are Mianus and Round Hill. However, when we apply the expression

$\Pi_{\textit{branch-name}}$ ($\sigma_{\textit{amount} <1000}$ (*branch-customer* | *customer-loan*)

we obtain three branch names Mianus, Round Hill and Downtown.

A closer examination of this example shows why. If a customer happens to have several loans from different branches, we cannot tell which loan belongs to which branch. Thus when we join *branch-customer* and *customer-loan*, we obtain not only the tuples we had originally in *lending*, but also several additional tuples. Although we have more tuples in *branch-customer* | *customer-loan*, we actually have less information. We are no longer able, in general, to represent in the database information about which customers are borrowers from which branch. Because of this loss of information, we call the decomposition of *lending-schema* into *Branch-customer-schema* and *customer-loam-schema* a **lossy decomposition**, or a **lossy-join decomposition**. A decomposition that is not a lossy join decomposition is a **lossless join decomposition**. It should be clear from our example that a lossy-join decomposition is, in general a bad database design.

Why is the decomposition lossy? There is one attribute in common between *branch customer-schema* and *customer-loan-schema*:

Branch-customer-schema ∩ *customer-loan-schema* = {*customer-name*}

The only way that we can represent a relationship between, for example, *loan number* and *branch-name* is through *customer-name*. This representation is not adequate because a customer may have several loans, yet these loans are not necessarily obtained from the same branch.

Let us consider another alternative design, in which we decompose *Lending-schema* into the following two schemas:

Branch-schema = (*branch-name*, *branch-city*, *assets*)

Loan-info-schema = (*branch-name*, *customer-name*, *loan-number*, *amount*)

There is one attribute in common between these two schemas:

Branch-loan-schema ∩ *customer-loan-schema* = {*branch-name*}

Thus the only way that we can represent a relationship between for example *customer-name* and *asset* is through *branch-name*. The difference between this example and the preceding one is that the assets of a branch are the same, regardless of the customer to which we are referring, whereas the lending branch associated with a certain loan amount does depend on the customer to which we are referring. For a given *branch-name*, there is exactly one *assets* value and exactly one *branch-city*; whereas a similar statement cannot be made for *customer-name*. That is the functional dependency

Branch-name→ {*assets, branch-city*}

holds, but customer-name does not functionally determine loan-number

The notion of lossless joins is central to much of relational database design. Therefore, we restate the preceding examples more concisely and more formally. Let r be a relational schema. A set of relational schema $\{R_1, R_2, \ldots, R_n\}$ is a decomposition of R if

$$R = R_1 \cup R_2 \cup \ldots \cup R_n$$

That is $\{R_1, R_2, \ldots, R_n\}$ is a decomposition of R if, for i = 1,2,......n, each R_i is a subset of R, and every attribute in R appears in at least one R_i.

Let r be a relation on schema r, and let $r_i = \Pi_{Ri}(r)$ for i = 1,2....n . That is $\{r_1, r_2, \ldots, r_n\}$ is the database that results from decomposition of r into $\{R_1, R_2, \ldots R_n\}$ it is always the case that

$$r \leq r_1 \infty r_2 \infty \ldots \infty r_n$$

To see that this assertion is true consider a tuple t in relation r,. When we compute the relations $r_1, r_2, \ldots r_n$ the tuple t gives rise to one tuple t_i in each r_i, i = 1,2...n .These n tuples combine to regenerate t when we compute $r_1 \infty r_2 \infty \ldots \infty r_n$. The details are left for you to complete as an exercise. Therefore every tuple in r appears in $r_1 \infty r_2 \infty \ldots \infty r_n$.

In general $r \neq r_1 \infty r_2 \infty \ldots \infty r_n$. As an illustration, consider our earlier example in which

- n = 2
- R = *Lending –schema*
- R_1 = *Branch-customer-schema*
- R_2 = *customer-loan-schema*
- r = the relation shown in Figure *W*.
- r_1 = the relation shown in Figure *X*.
- r_2 = the relation shown in Figure *Y*.

- $r_1 \infty r_2$ = the relation shown in Figure *Z*.

Note that the relations in Figure *W* and *Z* are not the same.

To have a lossless-join decomposition, we need to impose constraints on the set of possible relations. We found that the decomposition of *Lending-schema* into *Branch-schema* and *Loan-info-schema* is lossless because the functional dependency.

branch-name → branch-city assets

holds on *branch-schema* We say that a relation is legal if it satisfies all rules, or constraints that we impose on our database.

Let C represent a set of constraints on the database and let R be a relation schema. A decomposition $\{R_1, R_2.......R_n\}$ of R is a lossless join decomposition if for all relations r on schema R that are legal under C,

$$r = \Pi_{R1}(r) \infty \Pi_{R2}(r) \infty \ldots \infty \Pi_{Rn}(r)$$

12.3.1 Desirable Properties of Decomposition

We can use a given set of functional dependencies in designing a relational database in which most of the undesirable properties discussed above do not occur. When we design such systems, it may become necessary to decompose a relation into several relations.

Lending-schema = (*branch-name*, *branch-city*, *assets*, *customer-name*, *loan-number*, *amount*)

The set F of functional dependencies that we require to hold Lending schema are

branch-name → {branch-city, assets}

loan-number → {amount, branch-name}

Lending-schema is an example of a bad database design. Assume that we decompose it to the following three relations:

Branch-schema = (*branch-name*, *branch-city*, *assets*)

Loan-schema = (*loan-number*, *branch-name*, *amount*)

Borrower-schema = (*customer-name*, *loan-number*)

We claim that this decomposition has several desirable properties, which we discuss next.

Lossless-join decomposition

When we decompose a relation into a number of smaller relations, it is crucial that the decomposition be lossless. We must first present a criterion for determining whether decomposition is lossy.

Let R be a relation schema, and let F be a set of functional dependencies on R. Let R_1 and R_2 form a decomposition of R. This decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies is in F^+.

$R_1 \cap R_2 \rightarrow R_1$

- $R_1 \cap R_2 \rightarrow R_2$

In other words if $R_1 \cap R_2$ forms a super-key of either R_1 or R_2 the decomposition of r is a lossless-join decomposition. We can use attribute closure to efficiently test for super-keys as we have seen earlier.

We now demonstrate that our decomposition of *Lending-schema* is a lossless-join decomposing *Lending-schema* into two schemas:

Branch-schemas = (*branch-name*, *branch-city*, *assets*)

Loan-info-schema = (*branch-name*, *customer-name*, *loan-number*, *amount*)

Since *branch-name → {branch-name, assets}* the augmentation rule for functional dependencies implies that

Branch-name→ {branch-name, branch-city, assets}

Since *Branch-schema* ∩ *Loan-info-schema* = {*branch-name*}, it follows that our initial decomposition is a lossless-join decomposition

Next we decompose *loan-info-schema* into

Loan-schema = (*loan-number*, *branch-name*, *amount*)

Borrower-schema = (*customer-name*, *loan-number*)

This step results in a lossless-join decomposition since *loan-number* is a common attribute and *loan-number→ amount branch-name*.

For the general case of decomposition of a relation into multiple parts at once the test for lossless join decomposition is more complicated.

While the test for binary decomposition is clearly a sufficient condition for lossless join, it is a necessary condition only if all constraints are functional dependencies.

12.3.2 Dependency Preservation

There is another goal in relational database design: *dependency preservation.* When an update is made to the database, the system should be able to check that the update will not create an illegal relation-that is, one that does not satisfy all the given functional dependencies. If we are to check updates efficiently, we should design relational-database schemas that allow update validation without the computation of joins.

To decide whether joins must be computed to check an update, we need to determine what functional dependencies can be tested by checking each relation individually. Let F be a set of functional dependencies on a schema R and let R_1, R_2.....R_n be a decomposition of R. The restriction of F to R_i is the set F_i of all functional dependencies in F^+ that include only attributes of R_i. Since all functional dependencies in a restriction involve satisfaction of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.

Note that the definition of restriction uses all dependencies in F^+, not just those in F. For instance, suppose F= {A→B, B→C} and we have a decomposition into AC and AB. The restriction of F to AC is then A→C, since A→C is in F^+ even though it is not in F.

The set of restrictions F_1, F_2....F_n is the set of dependencies that can be checked efficiently. We now must ask whether testing only the restrictions is sufficient. Let F' = F_1 U F_2 U .....U F_n. F' is a set of functional dependencies on schema R but in general F ≠ F. However even if F' ≠ F may be that $F'^+ = F^+$. If the latter is true, then every dependency in F is logically implied by F' and if we verify that F' is satisfied we have verified that F is satisfied. We say that a decomposition having the property $F'^+ = F^+$ is a dependency preserving decomposition.

Figure *V* shows an algorithm for testing dependency preservation. The input is a set D = {R_1, R_2....R_n} of decomposed relation schemas and a set F of functional dependencies. This algorithm is expensive since it requires computation of F; we will describe another algorithm that is more efficient after giving an example testing for dependency preservation.

```
compute $F^+$;
for each schema $R_i$ in D do
begin
        $F_i$ = the restriction of $F^+$ to $R_i$
end
$F' := \Phi$
for each restriction $F_i$ do
        begin
                $F' = F' \cup F_i$
        end
compute $F'^+$;
if ($F'^+ = F^+$) then return (true)
        else return (false);
```

Figure *V*: Testing for dependency preservation

We can now show that our decomposition of *Lending-schema* is dependency preserving. Instead of applying the algorithm of Figure *V*, we consider an easier alternative; We consider each member of the set F of functional dependencies that we require to hold on *Lending-schema* and show that each one can be tested in at least one relation in the decomposition.

- We can test the functional dependency: *branch-name* → *{branch-city assets}* using *Branch-schema* = (*branch-name*, *branch-city*, *assets*).
- We can test the functional dependency : *loan-number* → *{amount branch-name}* using *Loan-schema* = (*branch-name*, *loan-number*, *amount*)

If each member of F can be tested on one of the relations of the decomposition, then the decomposition is dependency preserving. However there are cases where even though the decomposition is dependency preserving, there is a dependency in F that cannot be tested in any one relation in the decomposition. The alternative test can therefore be used as a sufficient condition that is checked. If it fails we cannot conclude that the decomposition is not dependency preserving instead we will have to apply the general test.

We now give a more efficient test for dependency preservation, which avoids computing F^+. The idea is to each functional dependency A → þ in F by using a modified form of attribute closure to see if it is preserved by the decomposition. We apply the following procedure to each → in F.

```
result = A
while (changes to result) do
        for each $R_i$ in the decomposition

        $t = (result \cap R_i)^+ \cap R_i$
        $result = result \cup t$
```

The attribute closure is with respect to the functional dependencies in F. If *result* contains all attribute in þ then the functional dependency A → þ is preserved. The decompositions is dependency preserving if and only if all the dependencies in F are preserved.

Note that instead of precomputing the restriction of F on R_i and using it for computing the attribute closure of result, we use attribute closure on (result $\cap$ R_i) with respect to F, and then intersect it with R_i, to get an equivalent result. This procedure takes polynomial time, instead of the exponential time required to compute F^+.

12.3.3 Repetition of Information

The decomposition of *Lending-schema* does not suffer from the problem of repetition of information that we will discuss in section about Bad Database Design. In *Lending-schema*, it was necessary to repeat the city and assets of a branch for each loan. The decomposition separates branch and loan data into distinct relations, thereby eliminating this redundancy. Similarly observe that, if a single loan is made to several customers, we must repeat the amount of the loan once for each customer (as well as the city and assets of the branch) in *Lending-schema*. In the decomposition, the relation on schema *Borrower-schema* contains the loan-number, customer-name relationship and not other schema does. Therefore we have one tuple for each customer for a loan in only the relation on *Borrower-schema*. In the other relational involving loan-number (those on schemas *Loan-schema* and *Borrower-schema*) only one tuple per loan needs to appear.

12.4 Problems arising out of bad database design (Pitfalls in Relational-Database design)

Let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may have are:

- Repetition of information
- Inability to represent certain information.

We shall discuss these problems with the help of a modified database design for our banking example: Suppose the information concerning loans is kept in one single relation, *lending* which is defined over the relation schema

Lending-schema = (*branch-name, branch-city, assets, customer-name, loan-number, amount*)

Figure below shows an instance of the relation *lending* (*Lending-schema*). A tuple t in the *lending* relation has the following intuitive meaning:

- t[*assets*] is the asset figure for the branch named t[*branch-name*]
- t[*branch-city*] is the city which the branch named t[*branch-name*] is located
- t[*loan-number*] is the number assigned to a loan made by the branch named t[*branch-name*] to the customer named t[*customer-name*]
- t[*amount*] is the amount of the loan whose number is t[*loan-number*]

Suppose that we wish to add a new loan to our database. Say that the loan is made by the Perryridge branch to Adams in the amount of $1500. Let the loan-number be L-31. In our design, we need a tuple with values on all the attributes of *Lending schema*. Thus we must repeat the asset and city data for the Perryridge branch, and must add the tuple

(Perryridge, Horseneck, 1700000, Adams, L-31, 1500)

to the *lending* relation. In general, the asset and city data for a branch must appear once for each loan made by that branch.

Branch-name	**Branch-city**	**Assets**	**Customer-name**	**Loan-number**	**Amount**
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Figure *W*: Sample *lending* relation

The repetition of information in our alternative design is undesirable. Repeating information wastes space. Furthermore it complicates updating the database. Suppose for example, that the assets of the Perryridge branch change from 1700000 to 1900000. Under our original design, one tuple of the branch relation need to be changed. Under our alternative design many tuples of the lending relation need to be changed. Thus updates are more costly under the alternative design than under the original design. When we perform the update in the alternative design database, we must ensure that every tuple pertaining to the Perryridge branch is updated, or else our database will show two different asset values for the Perryridge branch.

That observation is central to understanding why the alternative design is bad. We know that a bank branch has a unique value of assets, so given a branch name we can uniquely identify the assets value. On the other hand, we know that a branch may make many loans, so given a branch name, we cannot uniquely determine a loan number. In other words, we say that the functional dependency.

branch-name → assets

Holds on *Lending-schema*, but we do not expect the functional dependency *branch-name → loan- number* to hold. The fact that a branch has particular value of assets and the fact that a branch makes a loan are independent, and, as we have seen, these facts are best represented in separate relations. We shall see that we can use functional dependencies to specify formally when a database design is good.

Another problem with the Lending-schema design is that we cannot represent directly the information concerning a branch 9branch-name, branch-city, assets) unless there exists at least one loan at the branch. This is because tuples in the lending relation require value for loan-number, amount and customer-name.

One solution to this problem is to introduce null values, as we did to handle updates through views. However, Null values are difficult to handle. If we are not willing to deal with Null values, then we can create the branch information only when the first loan application at that branch is made. Worse, we would have to delete this information when all the loans have been paid. Clearly, this situation is undesirable, since, under our original database design, the branch information would be available regardless of whether or not loans are currently maintained in the branch, and without restoring to null values.

12.5 Summary

Functional dependencies play a key role in differentiating good database designs from bad database design. An attribute Y of a relation R is said to be functionally dependent upon attribute X of relation R if and only if for each value of X in R has associated with it only one of Y in R at any given time. It is represented by as X-> Y, where X attributes is known as determinant and Y is known as determined. Using the concept of Functional Dependencies we decompose the relations. The bad design of database suggests that we should decompose a relation schema that has many attributes into several schemas with fewer attributes. Careless decomposition, however may lead to another form of bad design. When we decompose a relation into a number of smaller relations, it is crucial that the decomposition be lossless.

12.6 Questionnaires:

1. What do you mean by Functional Dependency? What is its importance in Database design? Explain with example.
2. Why we need decomposition? What is its need? What are the steps involved in decomposing the relations.
3. What are the various problems that arise due to bad database design?

NORMALIZATION

13.1 Introduction

In this lesson, we will discuss the normalization process and define the first three normal forms for relation schemas. The definitions of second and third normal form presented here are based on the functional dependencies and primary keys of a relation schema. More general definitions of these normal forms, which take into account all candidate keys of a relation rather than just the primary key, are also presented. We also define Boyce-Codd Normal Form (BCNF), and further normal forms that are based on other types of data dependencies. We first informally discuss what normal forms are and what the motivation behind their development was. We then present first normal form (1NF). Then we present definitions of second normal form (2NF) and third normal form (3NF) respectively that are based on primary keys. Then we will proceed for multivalued dependency and further the fourth and fifth Normal Forms that are based on MVDs. In the last we will discuss about the database design process.

13.2 Normalization

The normalization process as first proposed by Codd (1972) takes a relation schema through a series of tests to “certify” whether or not, it belongs to a certain normal form. Initially Codd proposed three normal forms, which he called first, second and third normal form. A stronger definition of 3NF was proposed later by Boyce and Codd and is known as Boyce-Codd normal form. All these normal forms are based on the functional dependencies among the attributes of a relation. Later fourth normal form (4NF) and a fifth normal forms (5NF) were proposed, based on the concepts of multi-valued dependencies and join dependencies, respectively. Normalization of data can be looked on as a process during which unsatisfactory relation schemas are decomposed by breaking up their attributes into smaller relation schemas that possess desirable properties. One objective of the original normalization process forms provides database designers with:

- A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes.
- A series of tests that can be carried out on individual relation schemas so

that the relational database can be normalized to any degree. When a test fails, the relation violating that test must be decomposed into relations that individually meet the normalization tests.

- To free relations from undesirable insertion, deletion and update anomalies.

Normal forms, when considered in isolation from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each relation schema in the database is, say, in BCNF or 3NF. Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relation schemas, taken together, should possess. Two of these properties are:

- The loss less join or no additive join property, which guarantees that the spurious tuple problem does not occur
- The dependency preservation property, which ensures that all functional dependencies are represented in some of the individual resulting relations.

In this section we concentrate on an intuitive discussion of the normalization process. Notice that the normal forms mentioned in this section are not only the possible ones. Additional normal forms may be defined to meet other desirable criteria, based on additional types of constraints. The normal forms up to BCNF are defined by considering only the functional dependency and key constraints, whereas 4NF considers an additional constraint called a multi-valued dependency and 5NF considers an additional constraint called a join dependency. The practical utility of normal forms becomes questionable when the constraints on which they are based are hard to understand or to detect by the database designers and users who must discover these constraints.

Another point worth noting is that the database designers need not normalize to the highest possible normal form. Relations may be left in lower normal forms for performance reasons.

Before proceedings further, we recall the definitions of keys of a relation schema. A super key of a relation schema R = {A1, A2,..........., An} is a set of attributes S ¯(sub set of) R with the property that no two tuples t1 and t2 in any legal relation state r of R will have t_1[S] = t_2[S]. A key K is a super-key with the additional property that removal of any attribute from K will cause K not to be a super-key any more. The difference between a key and super key is that a key has to be "minimal" that is, if we have a key K = {A_1, A_2......., A_k} then K – A is not a key for 1<=i<=k. In figure given below {SSN} is a key for EMPLOYEE, whereas {SSN}, {SSN, ENAME}, {SSN, ENAME, BDATE} etc. are all super keys.

EMPLOYEE

f.k.

ENAME	SSN	BDATE	ADDRESS	DNUMBER

p.k.

If relation schema has more than one "minimal" key, each is called a candidate key. One of the candidates keys is arbitrarily designated to be the primary key, In figure above {SSN} is the only candidates key for EMPLOYEE, so it is also the primary key.

An attribute of relation schema R is called a prime attribute of R if it is a member of any key of R. An attribute is called nonprime if it is not a prime attribute-that is, if it is not a member of any candidate key.

We now present the first three normal forms: 1NF, 2NF and 3NF. These were proposed by Codd (1972) as a sequence to ultimately achieve the desirable state of 3NF relations by progressing through the intermediate states of 1NF and 2NF if needed.

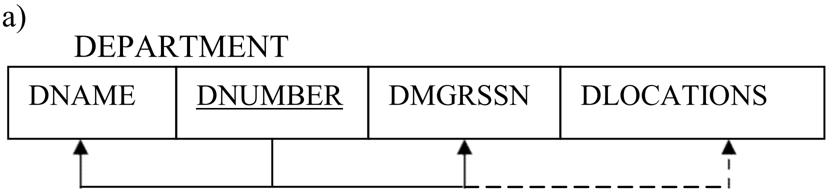
13.3 First Normal Form (1 NF)

First normal form is now considered to be part of the formal definition of a relation; historically, it was defined to disallow multi-valued attributes, composite attributes, and their combinations. **It states that the domains of attributes must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute**. Hence, 1NF disallows having a set of values, a tuple of values or a combination of both as an attribute value for a single tuple. In other words, 1NF disallows "relations within relations" or "relations as attributes of tuples". The only attribute values permitted by 1NF are single atomic (or indivisible) values.

Consider the DEPARTMENT relation schema shown in following figure, whose primary key is DNUMBER, and suppose that we extend it by including the DLOCATIONS attribute shown within dotted lines. We assume that each department can have a number of locations. The DEPARTMENT schema and example extension are shown in Figures that follow. As we can see, this is not in 1NF because DLOCATIONS is not an atomic attribute, as illustrated by the first tuple in Figure (b). There are two ways we can look at the DLOCATIONS attribute:

- The domain of DLOCATIONS contains atomic values, but some tuple can have a set of these values. In this case, DNUMBER*→ DLOCATIONS.
- The domain of DLOCATIONS contains sets of values and hence in monatomic. In this case, DNUMBER→DLOCATIONS, because each set is considered a single member of the attribute domain (In this case we can consider the domain of DLOCATIONS to e the power set of single locations; that is, the domain is made up of all possible subsets of the set of single locations).

a)



b)

DEPARATMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

c) DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

Figure showing Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Example relation instance. (c) 1NF relation with redundancy.

In either case, the DEPARTMENT relation of figures above is not in 1NF; in fact, it does not even qualify as a relation, we break up its attributes into the two relations DEPARTMENT and DEPT_LOCATIONS shown in Figure here:

DEPARTMENT

DNAME	DNUMBER	DMGRSSN

DEPT_LOCATIONS

DNUMBER	DLOCATIONS

The idea is to remove the attribute DLOCATIONS that violates 1NF and place it in a separate relation DEPT_LOCATIONS along with the primary key DNUMBER of DEPARTMENT. The primary key of this relation is the combination {DNUMBER, DLOCATION}, as shown in Figure above. A distinct tuple in DEPT_LOCATIONS exists for each location of a department. The DLOCATIONS attribute is removed from the DEPARTMENT relation of Figure showing the normalization into 1NF, decomposing the non-1NF relation into two 1NF relations DEPARTMENT and DEPT_DLOCATIONS of Figure above.

Notice that a second way to normalize into 1NF is to have a tuple in the original DEPARTMENT relation for each location of a DEPARTMENT, as shown in Figure (c). In this case, the primary key becomes the combination {DNUMBER, DLOCATION}, and redundancy exists in the tuples. The first solution is superior because it does not suffer from this redundancy problem. In fact, if we choose the second solution, it will be decomposed further during subsequent normalization steps into the first solution.

The first normal form also disallows composite attribute that are themselves multi-valued. These are called nested relations because each tuple can have a relation within it. Figure *A* below shows how an EMP_PROJ relation can be shown if nesting is allowed. Each tuple represents an employee entity, and a relation PROJS (PNUMBER, HOURS} within each tuple represents the employee's projects and the hours per week that the employee works on each project. The schema of the EMP_PROJ relation can be represented as follows:

EMP_PROJ (SSN, ENAME, {PROJS (PNUMBER, HOURS)})

The set braces {} identify the attribute PROJS as multi-valued, and we list the component attribute that form PROJS between parentheses (). Interestingly, recent research into the relational model is attempting to allow and formalize nested relations, which were disallowed early on by 1NF.

Notice that SSN is the primary key of the EMP_PROJ relation in Figure *A*(a) and (b), while PNUMBER is the partial primary key of each nested relation; that is, within each tuple, the nested relation attributes into a new relation and propagate the primary key into; the primary key of the new relation will combine the partial key with the primary key of the original relation. Decomposition and primary key propagation yield the schemas shown in Figure *A*(c).

Here is the figure *A*:

a)

EMP_PROJ

SSN	ENAME	PROJS	
		PNUMBER	HOURS

b) **EMP_PROJ**

SSN	ENAME	PNUMBER	HOURS
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.5
333445555	Wong, Franklin T.	2	10.5
		3	10.5
		10	10.5
		20	10.5

c) **EMP_PROJ1**

SSN	ENAME

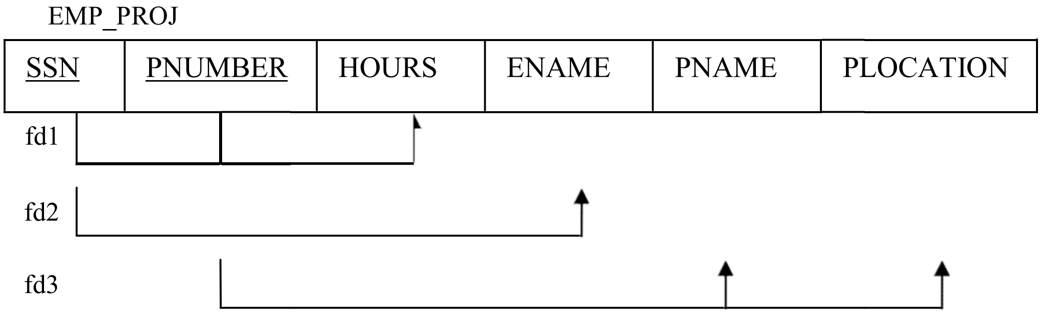
EMP_PROJ2

SSN	PNUMBER	HOURS

This procedure can be applied recursively to a relation with multi-valued level nesting to unnest the relation into a set of 1NF relations. This is useful in converting hierarchical schemas into 1NF relations. As we shall see in the coming topics, restricting relations to 1NF leads to the problems associated with multi-valued dependencies and 4NF.

13.4 Second Normal Form (2NF)

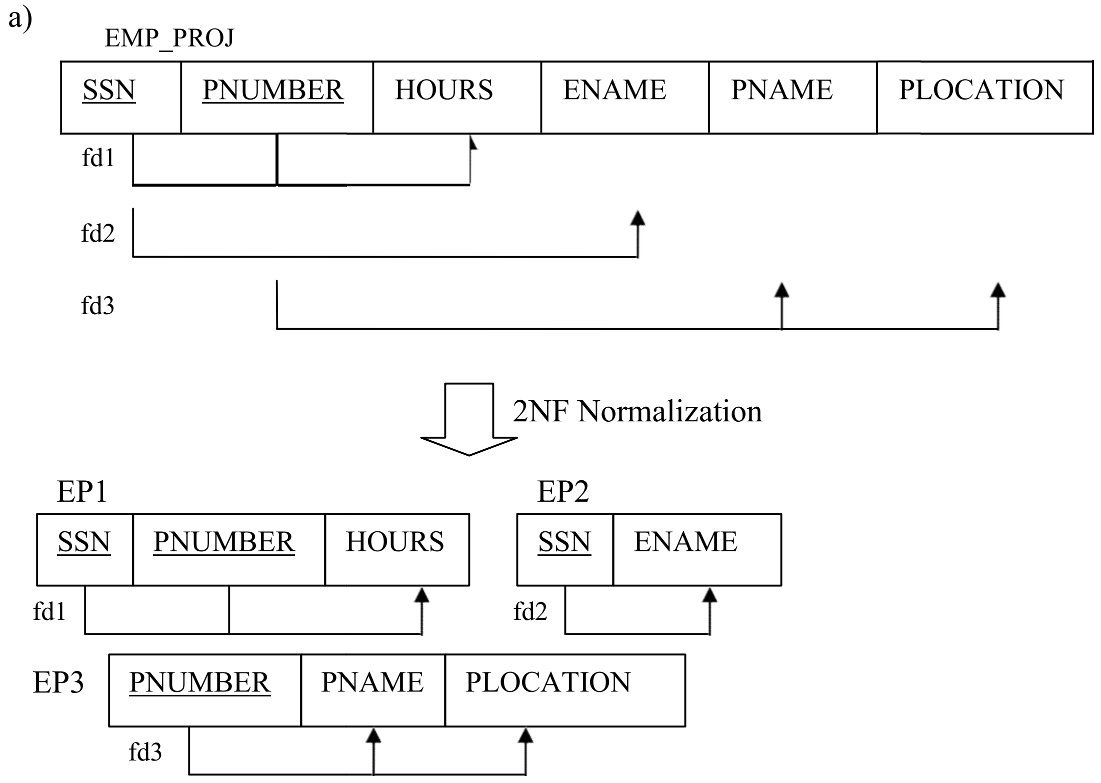
Second Normal form is based on the concept of full functional dependency. A functional dependency X→Y is a full functional dependency if removal of any attribute a from X means that the dependency does not hold any more; that is, for any attribute A ε X,
(X-{A}) *→ Y. A functional dependency X→Y is a partial dependency if some attribute A ε X can be removed from X and the dependency still holds; that is for some A ε X, (X – {A})→Y. In figure below, {SSN, PNUMBER} → HOURS is a full dependency (neither SSN →HOURS nor PNUMBER→HOURS holds). However, the dependency {SSN PNUMBER}→ENAME is partial because SSN→ENAME holds.



A relation schema R is in 2NF if every nonprime attribute A in R is fully functionally dependent on the primary key of R. The EMP_PROJ relation in figure above is in 1NF but is not in 2NF. The nonprime attribute ENAME violates 2NF because of fd2, as do the nonprime attribute PNAME and PLOCATION because of fd3. The functional dependencies fd2 and fd3 make ENAME, PNAME and PLOCATION partially dependent on the primary key {SSN, PNUMBER} of EMP_PROJ thus violating 2NF.

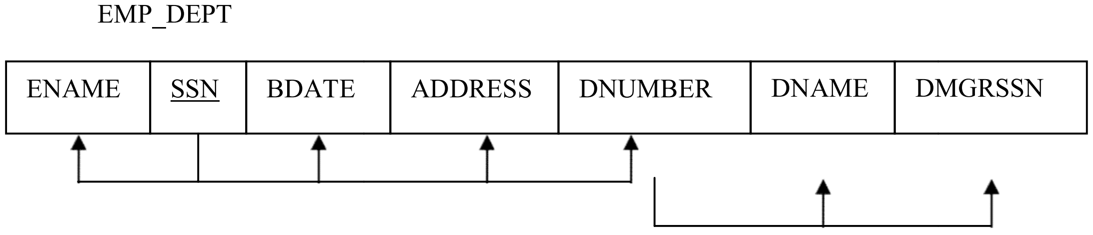
If a relation schema is not in 2NF it can be further normalized into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. The functional dependencies fd1, fd2 and fd3 in Figure above hence lead to the decomposition of EMP_PROJ into the three relation schemas EP1, EP2 and EP3 shown in Figure *B* each of which is in 2NF. We can see that the relations Ep1, Ep2 and EP3 are devoid of the update anomalies from which EMP_PROJ of Figure above suffers.

Figure *B*:



13.5 Third Normal Form (3NF)

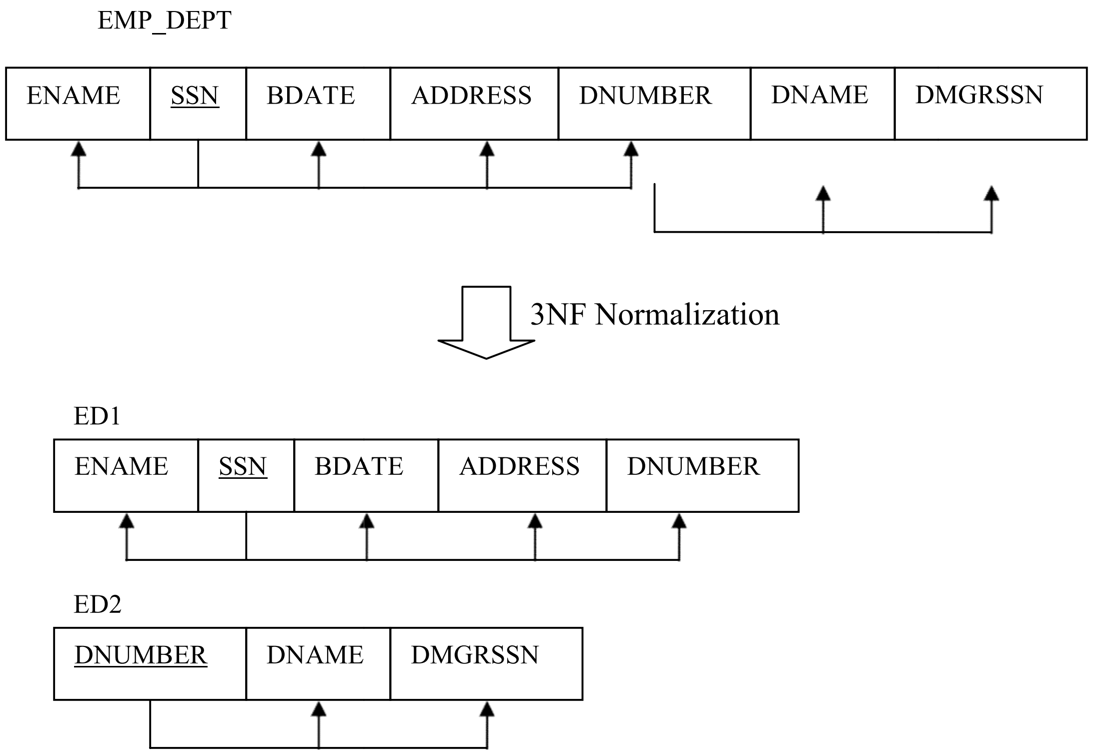
Third Normal form is based on the concept of transitive dependency. A functional dependency X →Y in a relation schema R is a transitive dependency if there is a set of attributes Z that is not a subset of any key of R, and both X→Z and Z→Y hold. The dependency SSN→DMGRSSN is transitive through DNUMBER in EMP_DEPT of Figure here:



DNUMBER is not a subset of the key of EMP_DEPT. Intuitively; we can see that dependency of DMGRSSN on DNUMBER is undesirable in EMP_DEPT since DNUMBER is not a key of EMP_DEPT.

According to Codd's original definition, a relation schema R is in 3NF if it is in 2NF and no nonprime attribute of R is transitively dependent on the

primary key. The relation schema EMP_DEPT in Figure above is in 2NF since no partial dependencies on a key exist. However EMP_DEPT is not in 3NF because of the transitive dependency of DMGRSSN (and also DNAME) on SSN via DNUMBER. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and Ed2 shown in Figure below:



EMP_DEPT

ENAME | SSN | BDATE | ADDRESS | DNUMBER | DNAME | DMGRSSN

3NF Normalization

ED1

ENAME | SSN | BDATE | ADDRESS | DNUMBER

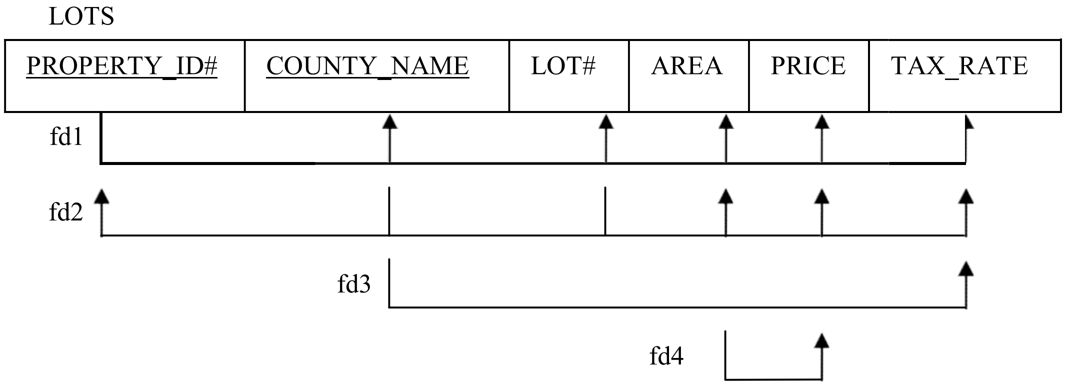
ED2

DNUMBER | DNAME | DMGRSSN

Intuitively, we see that Ed1 and Ed2 represent independent entity facts about employees and departments. A NATURAL JOIN operation on ED1 and ED2 will recover the original relation EMP_DEPT without generating spurious tuples.

13.6 Boyce-Codd Normal Form (BCNF)

Boyce-Codd normal form is stricter than 3NF, meaning that every relation in BCNF is also in 3NF; however, a relation in 3NF is not necessarily in BCNF, intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema of Figure below with its four functional dependencies fd1 through fd4.



LOTS

PROPERTY_ID# | COUNTY_NAME | LOT# | AREA | PRICE | TAX_RATE

Suppose that we have thousands of lots in the relation but the lots are from only two counties; Marion County and Liberty County. Suppose also that lot size in

Marion County are only 0.5, 0.6, 0.7, 0.8, l 0.9, and 2.0 acres. In such a situation we should have the additional functional dependency fd5; AREA → COUNTY_NAME. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because COUNTY_NAME is a prime attribute.

The area versus county relationship represented by fd5 can be represented by 16 tuples in a separate R (AREA, COUNTY_NAME) since there are only 16 possible AREA values. This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a stranger normal form that would disallow LOTS1A and suggest the need for decomposing it.

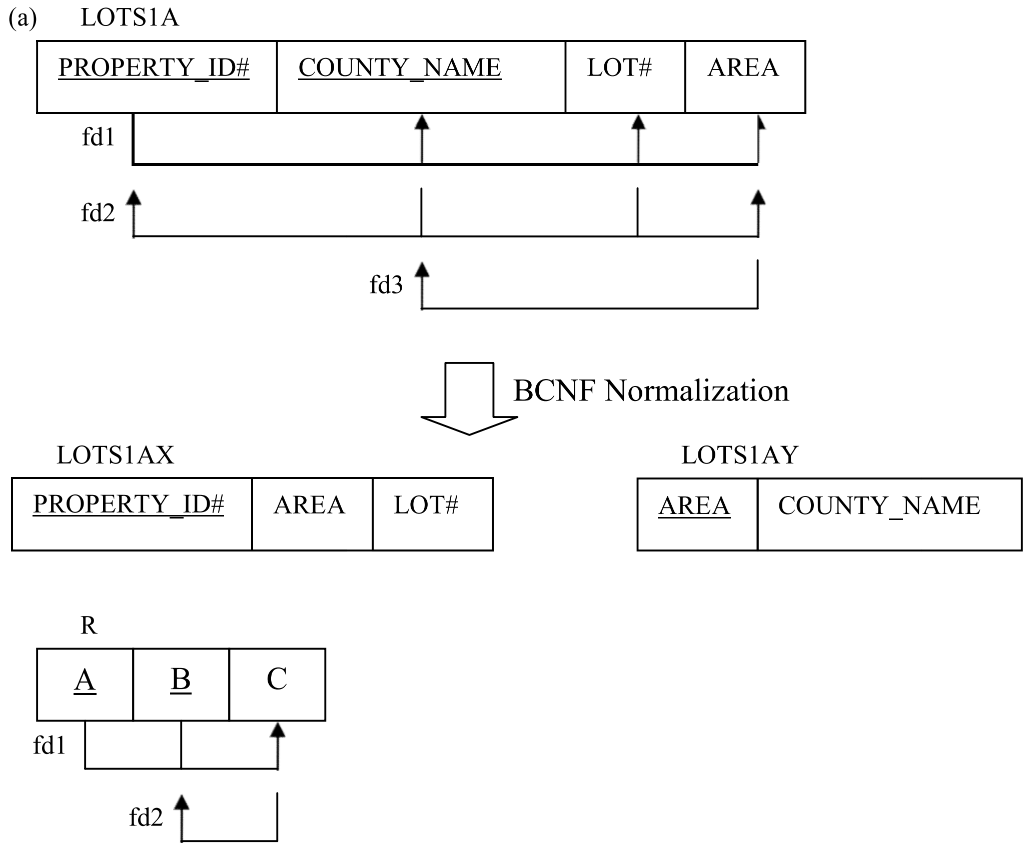
This definition of Boyce-Codd differs slightly from the definition of 3NF. **A relation schema R is in BCNF if whenever a functional dependency X → A holds in R, then X is a super-key of R.** The only difference between BCNF and 3NF is that condition (b) of 3NF, which allows A to be prime if X is not a super-key, is absent from BCNF.

In our example, fd5 violates BCNF in LOTS1A because AREA is not a super-key of LOTS1A. Note that fd5 satisfies 3NF LOTS1A because COUNTY_NAME is a prime attribute (Condition (b)), but this condition does not exist in the definition of BCNF. We can decompose LOTS1A into two BCNF relations LOTS1AX and LOTS1AY, shown in Figure *C*(a).

In practice most relation schema that are in 3NF are also in BCNF. Only if a dependency X τ A exists in a relation schema R with X not a super-key and A a prime attribute will R be in 3NF but not in BCNF. The relation schema R shown in Figure *C*(b) illustrates the general case of such a relation.

It is best to have relation schemas in BCNF, if that is not possible, 3NF will do. However, 2NF and 1 NF are not considered good relation schema designs. These normal forms were developed historically as stepping stones to 3NF and BCNF.

Here is Figure *C*:



13.7 Multi-valued Dependencies

Multi-valued dependencies are a consequence of first normal form, which disallowed an attribute in a tuple to have a set of values. If we have two or more multi-valued independent attributes in the same relation schema, we get into a problem of having to repeat every value of one the attribute with every value of the other attribute to keep the relation instance consistent. This constraint is specified by a multi-valued dependency.

For example, consider the relation EMP shown in Figure *D* (a). A tuple in this EMP relation represents the fact that an employee whose name is ENAME works on the project whose name is PNAME and has a dependent whose name is DNAME. An employee may work on several projects and may have several dependents, and the employees project and dependents are not directly related to one another. To keep the tuples in the relation consistent, we must keep a tuple to represent every combination of an employee's dependent and an employee's project. This constraint is specified as a multi-valued dependency on the EMP relation. Informally, whenever two independent 1: N relationships A: B and A: C are mixed on the same relation, an MVD may arise.

Figure *D*:

(a) **EMP**

ENAME	PNAME	DNAME
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

(b) **EMP_PROJECTS**

ENAME	PNAME
Smith	X
Smith	Y

EMP_DEPENDENTS

ENAME	DNAME
Smith	John
Smith	Anna

(c) **SUPPLY**

SNAME	PARTNAME	PROJNAME
Smith	Bolt	ProjX
Smith	Nut	ProjY
Adamsky	Bolt	ProjY
Walton	Nut	ProjZ
Adamsky	Nail	ProjX
Adamsky	Bolt	ProjX
Smith	Bolt	ProjY

(d) R1

SNAME	PARTNAME
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R2

SNAME	PROJNAME
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R3

PARTNAME	PROJNAME
Bolt	ProjX
Nut	ProjY
Bolt	ProjY
Nut	ProjZ
Nail	ProjX

Formal Definition of Multi-valued Dependency

Formally a multi-valued dependency (MVD) X→→Y specified on relation schema R where X and Y are both subsets of R, specifies the following constraint on any relation r of R. If two tuples t_1 and t_2 exist in r such that $t_1[X] = t_2[X]$ then two tuples t_3 and t_4 should also exist in r with the following properties:

- $t_3 [X] = t_4[X]=t_1[X]=t_2[X]$
- $t_3[4]=t_1 [Y]$ and $t_4[Y]=t_2[Y]$
- $t_3[R-(XY)] = t_2[R-(XY)]$ and $t_4[R-(XY)] = t_1[R-(XY)]$.

Whenever X→→Y holds, we say that X multi-determines Y. Because of the symmetry in the definition, whenever X→→Y holds in R, so does X→→ (R-(XY)). Recall that (R-XY) is the same as R-(X U Y) = Z. Hence X→→Y implies X→→Z and therefore it is sometimes written as X→→Y/Z.

The formal definition specifies that given a particular value of X, the set of values of Y determined by this value of X is completely determined by X alone and does not depend on the values of the remaining attributes Z of the relation schema R. Hence whenever two tuples exist that have distinct values of Y but the same value of X these values of Y must be related with every distinct value of Z that occurs with that same value of X. This informally corresponds to Y being a multi-valued attribute of the entities represented by tuple in R.

In Figure *D* (a) the MVDs ENAME→→PNAME and ENAME→→DNAME or ENAME→→PNAME/DNAME hold in the EMP relation. The employee with ENAME Smith works on project with PNAME 'X' and 'Y' and has two dependents with DNAME John and 'Anna'. If we stored only the first two tuples in EMP (< Smith', 'X', 'John'> and <Smith', 'Y' 'Anna'> and <Smith', 'Y', 'John'>) to show that {'X', 'Y'} and {John', 'Anna} are associated only 'Smith' that is there is no association between PNAME and DNAME.

An MVD X→→Y in R is called a trivial MVD if (a) Y is a subset of X or (b) X U Y=R, for example the relation EMP_PROJECTS in Figure *D* (b) has the trivial MVD ENAME→→PNAME. An MVD that satisfies neither (a) nor (b) is called a nontrivial

MVD. A trivial MVD will hold in any relation instance r of R, it is called trivial because it does not specify any constraint on R.

If we have a nontrivial MVD in relation, we may have to repeat values redundantly in the tuples. In the EMP relation of Figure the values 'X' and 'Y' of PNAME are repeated with each value of DNAME (or by symmetry, the values 'John' and 'Anna' of DNAME are repeated with each value of PNAME). This redundancy is clearly undesirable However; the EMP schema is in BCNF because no functional dependencies hold in EMP. Therefore, we need to define a fourth normal form that is stronger than BCNF and disallows relation sch4emas such as EMP. We first discuss some of the properties of MVDs and consider how they are related to functional dependencies.

Inference Rules for Functional and Multi-valued Dependencies

As with functional dependencies (FDs), we can develop inference rules for MVDs. It is better through, to develop a unified framework that includes both FDs and MVDs so that both types of constraints can be considered together. The following inference rules IR1 through IR8 form a sound and complete set for inferring functional and multi-valued dependencies from a given set of dependencies. Assume that all attributes are included in a "universal" relation schema R= {A_1, A_2 ....A_n} and that X, Y, Z and W are subsets of R.

(IR1) (Reflexive rule for FDs0: if X ≥ Y, then X → Y.

(IR2) (Augmentation rule for FDs): {X →Y) ╞ XZ→YZ

(IR3) (Transitive rule for FDs): {X→Y, Y→Z} ╞ X→Z.

(IR4) (Complementation rule for MVDs): {X→→Y) ╞ {X→→ (R – (X U Y))}.

(IR5) (Augmentation rule for MVDs): If X →→Y and W≥Z then WX→→YZ

(IR6) (Transitive rule for MVDs): {X→→Y, Y→→Z} ╞ X →→ (Z-Y)

(IR7) (Replication rule FD to MVD)): {X→Y} ╞ X→→Y

(IR8) (Coalescence rule for FDs and MVDs): If X→→Y and there exists W with the properties that (a) W ∩ Y is empty, (b) W→Z and (c) Y ≥ Z then X→Z.

IR1 through IR3 are Armstrong's inference rules for FDs alone. IR4 through Ir6 are inference rules pertaining to MVDs only. IR7 and IR8 relate FDs and MVDs. In particular IR7 says that a functional dependency is a special case of a multi-valued dependency; that is, every FD is also an MVD. An FD X→Y is an MVD X→→Y with the additional restriction that at most one value of Y is associated with each value of X. Given a set F of functional and multi-valued dependencies specified on R = {A_1, A_2,.......A_n), we can use IR1 through IR8 to infer the (complete) set of all dependencies (functional or multi-valued) F^+ that will hold in every relation instance r of R that satisfies F. We again call F^+ closure of F.

13.8 Fourth Normal Form

We now present the definition of 4NF which is violated when a relation has undesirable multi-valued dependencies, and hence can be used to identify and decompose such relations. **A relation schema R is in 4NF respect to a set of dependencies F if for every nontrivial multi-valued dependency X→→Y in F^+, X is a super-key for R.**

The EMP relation of Figure *D* (a) is not 4NF because in the nontrivial MVDs ENAME→→PNAME and ENAME →→ DNAME, ENAME is not a super-key of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS shown in Figure *D*(b). Both EMP PROJECTS and EMP_DEPENDENTS are in 4NF, because ENAME→→PNAME is a trivial MVD in EMP PROJECTS and ENAME→→DNAME is a trivial MVD in EMP_DEPENDENTS. In fact no nontrivial MVDS hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

To illustrate why it is important to keep relations in 4NF, Figure *E*(a) shows the EMP relation with an additional employee Brown who has three dependents (‘Jim’ ‘Joan’, and ‘Bob) and works on four different projects (‘W’, ‘X’, ‘Y’) and ‘Z’). There are 16 tuples in EMP in figure *E*(a). I few decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS as shown in Figure *E*(b) we need only store a total of 11 tuples in both relations. More ever these tuples are much smaller than the tuples in EMP. In addition the update anomalies associated with multi-valued dependencies are avoided. For example, if Brown starts working on another project, we must insert three tuples in EMP – one for each dependent. If we forgot to insert any one of those, the relation becomes inconsistent in that is incorrectly implies a relationship between project and dependent. However only a single tuple need be inserted in the 4NF relation EMP_PROJECTS. Similar problems occur with deletion and modification anomalies if a relation is not in 4NF.

Figure *E*:

(a) **EMP**

ENAME	PNAME	DNAME
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John
Brown	W	Jim
Brown	X	Jim
Brown	Y	Jim
Brown	Z	Jim
Brown	W	Joan
Brown	X	Joan
Brown	Y	Joan
Brown	Z	Joan
Brown	W	Bob
Brown	X	Bob
Brown	Y	Bob
Brown	Z	Bob

(b) **EMP_PROJECTS**

ENAME	PNAME
Smith	X
Smith	Y
Brown	W
Brown	X
Brown	Y
Brown	Z

EMP_DEPENDENTS

ENAME	DNAME
Smith	John
Smith	Anna
Brown	Jim
Brown	Joan
Brown	Bob

The EMP relation in Figure *D* (a) is not in 4NF because it represents two independent 1: N relationships—one between employees and the projects they work on and the other between employees and their dependents. We sometimes have a relationship between three entities that depends on all three participating entities, such as the SUPPLY relation shown in Figure *D* (c) (Consider only the tuple in Figure *D*(c) above the dotted line for now). In this case a tuple represents a supplier supplying a specific part to a particular project, so there are no nontrivial MVDs. The SUPPLY relation is already in 4NF and should not be decomposed. Notice that relations containing nontrivial MVDs tend to be all key relations;

that is, their key is all their attributes taken together.

Lossless Join Decomposition into 4NF Relations

Whenever we decomposed a relation schema R into R_1 = (X U Y) and R_2 = (R – Y) based on an MVD X →→ Y that holds in R, the decomposition has the lossless join property. It can be shown that this is a necessary and sufficient condition for decomposing a schema into two schemas that have the lossless join property as given by property.

PROPERTY Lj1'

The relation schemas R_1 and R_2 form a lossless join decomposition of R if and only if (R_1 ∩ R_2) →→ (R_1 - R_2) (or by symmetry, if and only if (R_1 ∩ R_2) →→ (R_2 – R_1).

This is similar to property Lj1 of Section 13.1.3 except that Lj1 dealt with FDs only, whereas Lj1' deals with both FDs and MVDs. We can use algorithm below which creates lossless join decomposition into relation schemas that are in 4NF(rather than in BCNF). Algorithm below does not necessarily produce a decomposition that preserves FDs.

```
ALGORITHM Lossless join decomposition into 4NF relations
    Set D :={ R};
    While there is a relation schemas Q in D that is not in 4NF do
        begin
        Choose a relation schema Q in D that is not in 4Nf;
        Find a nontrivial MVD X→→Y in Q that violates 4NF;
        Replace Q in D by two schemas (Q-Y) and (X U Y)
    End;
```

13.9 Join Dependencies and Fifth Normal Form

We saw that Lj1 and Lj1' give the condition for a relation schemas R to be decomposed into two schemas R_1 and R_2 where the decompositions has the lossless join property. However in some cases there may be no lossless join decomposition into two relation schemas but there may be a lossless join decomposition into more than two relation schemas. These cases are handled by join dependency and fifth normal form. It is important to note that these cases occur very rarely and are difficult to detect in practice.

A **join dependency (JD)** denoted by JD (R_1, R_2.........R_n) specified on relation schema R, specifies a constraint on instances r of R. The constraint states that every legal instance r of R should have a lossless join decomposition into R_1, R_2 ..... R_n that is,

$* (\Pi_{<R1>}(r), \Pi_{<R2>}(r), ..., \Pi_{<Rn>}(r)) = r$

Notice that a MVD is a special case of a JD where n=2. A join dependency JD(R_1 R_2,........R_n) specified on relation schema R, is a trivial if one of the relation schemas R_i in JD(R_1, R_2.......R_n) is equal to R. Such dependency is called trivial because it has the lossless join property for any relation instance r of R and hence does not specify any constraint on R. We can now specify fifth normal form, which is also called project join normal form. A relation schema R is in fifth normal form (5NF) (or project join normal form (PJNF)) with respect to a set functional multi-valued and join dependencies if for every nontrivial join dependency JD (R_1, R_2..........R_n) in F^+ (that is implied by F) every R, is a super-key of R.

For an example of a consider once again the SUPPLY relation of Figure *D* (c). If it does not have a lossless decomposition into any number of smaller tables. Suppose that the following additional constraint always holds: Whenever a supplier supplies part p and a project j uses part p and the supplies at least one part to project j, then supplier will also be supplying part p to project j. This constraint can be restated in other ways and specifies a join dependency JD (R1, R2, R3) among the three projections R (SNAME, PARTNAME), R2 (SNAME, PROJNAME) and R3 (PARTNAME, PROJNAME) of supply. If this constraint holds the tuples below the dotted line in Figure *D* (c) must exist in any legal instance of the SUPPLY relation with the join dependency is decomposed into three relations R1, R2 and R3 that are each in 5NF. Notice that applying NATURAL JOIN to any two of these relations produces spurious tuples, but applying NATURAL JOIN to all three together does not. The reader should verify this on the example relation of Figure *D*(c) and its projections in Figure *D*(d). This is because only the JD exists but no MVDS are specified. Notice too that the JD (R1, R2, R3) is specified on all legal relation instance not just on the one shown in Figure *D*(c).

Discovering JDs in practical data based with hundreds of attributes is difficult; hence current practice of data base design pays scant attention to them.

13.10 Overall Database design process

In Normalization we have assumed that we have a schema R, and proceeded to normalize it. There are several ways in which we could have come up with the schema R:

1. R could have been generated when converting an E-R Diagram to a set of tables.
2. R could have been a single relation containing all the attributes that are of interest. The normalization process breaks up R into smaller relations.
3. R could have been the result of some ad hoc design of relations, which we then test to verify that it satisfies a desired normal form.

No we examine the implications of these approaches and also the practical issues in database design, including de-normalization for performance and example of bad database design not detected by normalization.

E-R model and Normalization

We carefully define an E-R Diagram, identifying all entities correctly; the tables generated from the E-R diagram should not need further normalization. However, there can be functional dependencies between the attributes of an entity. For instance, suppose an *employee* entity had attributes *department-number* and *department-address*, and there is a functional dependency *department-number* → *department-address*. We would then need to normalize the relation generated from employee.

Most examples of such dependencies arise out of poor E-R diagram design. In the above example, if we did the E-R diagram correctly, we would have created a *department* entity with attribute *department-address* and a relationship between *employee* and *department*. Similarly, a relationship involving two or more than two entities many not be in a desirable normal form, since most relationships are binary, such cases are relatively rare. (In fact, some E-R diagram variants actually make it difficult or impossible to specify non-binary relations.).

Functional dependencies can help us detect poor E-R design. If the generated relations are not in desired normal form, the problem can be fixed in the E-R diagram. That is normalization can be done formally as part of data modeling. Alternatively, normalization can be left to the designer's intuition during E-R modeling and can be done formally on the relations generated from the E-R model.

The Universal Relation Approach

The second approach to database design is to start with a single relation schema containing all attributes of interest and decompose it. One of our goals in choosing a decomposition was that it be a lossless-join decomposition. To consider losslessness, we assumed that it is valid to talk about the join of all relations of the decomposed database.

Consider the database of Figure *F*, showing a decomposition of the loan-info relation. The figure depicts a situation in which we have not yet determined the amount of loan L-58, but wish to record the remainder of the data on the loan. If we compute the natural join of these relations, we discover that all tuples referring to loan L-58 disappear. In other words there is no loan-info relation corresponding to the relations of Figure *F*. Tuples that disappear when we compute the join are dangling tuples formally let $r_1(R_1)$, $r_2(R_2)$ ……, $r_n(R_n)$ be a set of relations. A tuple t of relation r is a dangling tuple if t is not in the relation.

$\Pi_{R_i} (r_1 | r_2 | \dots | r_n)$

Dangling tuples may occur in practical database applications. They represent incomplete information, as they do in our example, where we wish to store data about a loan that is still in the process of being negotiated. The relation $r_1 | r_2 | \dots | r_n$ is called a universal relation, since it involves all the attributes in the universe defined by $R_1 \cup R_2 \cup \dots \cup R_n$.

Banch-name	**Loan-number**
Round Hill	L-58

Loan-number	**Amount**

Loan-number	**Customer-name**
L-58	Johnson

Figure *F* : Decomposition of *loan-info*.

The only way that we can write a universal relation for the example of Figure *F* is to include null values in the universal relation. We know that null values present several difficulties. Because of them, it may be better to view the relations of the decomposed design as representing the database, rather than as the universal relation whose we decomposed during the normalization process.

Note that we cannot enter all incomplete information into the database of Figure *F* without resorting to null values. For example, we cannot enter a loan number unless we know at least one of the followings:

- The customer name
- The branch name
- The amount of the loan

Thus, a particular decomposition defines a restricted form of incomplete information that is acceptable in our database.

The normal forms that we defined generate good database design from the point of view of representation of incomplete information. Returning again to the example of Figure *F* we should not want to allow storage of the following fact. "There is a loan (whose number is unknown) to Jones in the amount of $100." This is because

Loan-number → customer-name amount

And therefore the only way that we can relate *customer-name* and *amount* is through loan-number. If we do not know the loan number, we cannot distinguish this loan from other loans with unknown numbers.

In other words, we do not want to store data for which the key attributes are unknown. Observe that the normal forms that we have defined do not allow us to store that type of information unless we use null values. Thus our normal forms allow representation of acceptable incomplete information via dangling tuples, while prohibiting the storage of undesirable incomplete information.

Another consequence of the universal relation approach to database design is that attribute names must be unique in the universal relation. We cannot use *name* to refer to both *customer-name* and to *branch-name*. It is generally preferable to use unique names, as we have done. Nevertheless, if we defined our relation schemas directly rather than in terms of a universal relation, we could relations on schemas such as the following for our banking example:

branch-loan (name, number)
loan-customer (number, name)
amt (number, amount)

Observe that, with the preceding relations expressions such as *branch-loan* ∞ *loan- customer* are meaningless. Indeed the expression *branch-loan* ∞ *loan-customer* finds loans made by branches to customers who have the same name as the name of the branch.

In a language such as SQL, however a query involving branch-loan and loan-customer must remove ambiguity in references to *name* by prefixing the relation name. In such environments, the multiple roles for *name* (as branch name and as customer name) are less troublesome and may be simpler to use.

We believe that using the unique-role assumption-that each attribute name has a unique meaning in the database- is generally preferable to reusing of the same name in multiple roles. When the unique role assumption is not made, the database designer must be especially careful when constructing a normalized relational-database design.

De-normalization for Performance

Occasionally database designers choose a schema that has redundant information; that is, it is not normalized. They use the redundancy to improve performance for specific applications. The penalty paid for not using a normalized schema is the extra work in terms of coding time and execution time) to keep redundant data consistent.

For instance, suppose that the name of an account holder has to be displayed along with the account number and balance every time the account is accessed. In our normalized schema, this requires a join of *account* with *depositor.*

One alternative to computing the join on the fly is to store a relation containing all the attribute of *account* and *depositor.* This makes displaying the account information faster. However the balance information for an account is repeated for every person who owns the account and all copies must be updated by the application, when ever the account balance is updated. The process of taking a normalized schema and making it non0normalized is called de-normalization, and designers use it to tune performance of systems to support time-critical operations.

A better alternative, supported by many database systems today, is to use the normalized schema, and additionally store the join or account and depositor as a materialized view. (Recall that a materialized view is a view whose result is stored in the database, and brought up to date when the relations used in the view are updated.) Like de-normalization, using materialized view does have space and time overheads; however, it has the advantage that keeping the view up to date is the job of the database system, not the application programmer.

Other Design Issues

There are some aspects of database design that are not addressed by normalization and can thus lead to bad database design. We give examples here obviously, such designs should be avoided.

Consider a company database, where we want to store earnings of companies in different years. A relation *earnings (company-id, year, amount)* could be used to store the earnings information. The only functional dependency on this relation is *company-*

id, year→ amount, and the relation is in BCNF.

An alternative design is to use multiple relations, each storing the earnings for a different year. Let us say the years of interest are 2000, 2001 and 2002; we would then have relations of the form *earnings*-2000, *earnings*-2001, and *earning*s-2002, all of which are on the schema (*company-id, earnings*). The only functional dependency here on each relation would be *company-id→ earnings* so these relations are also in BCNF.

However this alternative design is clearly a bad idea—we would have to create a new relation every year, and would also have to write new queries every year, to take each new relation into account. Queries would also be more complicated since they may have to refer to many relations.

Yet another way of representing the same data is to have a single relation *company-year (company-id, earnings-2000, earnings-2001, earnings-2002).* Here the only functional dependencies are from company-id to the other attributes, and again the relation is in BCNF. This design is also a bad idea since it has problems similar to the previous every year. Queries would also be more complicated, since they may have to refer many attributes.

Representations such as those in the company-year relation with one column for each value on an attribute, are called crosstab; they are widely used in spreadsheets and reports and in data analysis tolls. While such representations are useful for display to users, for the reasons just given, they are not desirable in a database design SQL extensions have been proposed to convert data from a normal relational representation to a crosstab, for display.

13.11 Summary

Normalization is a design technique that is widely used as a guide in designing relational databases. It is a process of decomposing a relation into relation(s) with fewer attributes by minimizing the redundancy of data and minimizing insertion, deletion and updation anomalies. It may be defined as step by step reversible process of transforming an unnormalized relation into relations with progressively simpler structures. The relation is in first normal form if all the attribute values are atomic and non decomposable. A relation is in 2NF if it is in 1 NF and non key attributes should be fully functionally dependent on the primary key. A relation is in 3 NF if it is 2 NF and non key attributes should not be transitively functionally dependent on Primary key. A relation is in BCNF if and only if every determinant is a candidate key. A relation is 4 NF if it is in BCNF and it contains no multivalued dependencies. And finaly a relation is in 5NF or Project Join Normal form if it cannot have a lossless decomposition into any number of smaller tables.

13.12 Questionnaires:

4. What do you mean by Normalization? Why there is a need for normalization?
5. Explain First, Second and third Normal Forms with the help of examples.
6. Explain Boyce-Codd Normal Form with example. How it is different from 3rd Normal Norm?
7. Does every relation having two attributes satisfy Boye Codd Normal form? If Yes, justify your answer giving suitable example.
8. Define Multi-valued dependency giving example.
9. Explain Fourth Normal form with example.

10. Define Join Dependency with example.
11. Explain fifth Normal form using Join Dependency using suitable example.
12. Explain the various insert, update and delete anomalies in various normal forms.

DATABASE INEGRITY AND RECOVERY

14.1 Introduction:

After completing the database we need to take measures for protecting the database. For protecting the database we have to take care of database integrity and in the coming section we will study the various methods for maintaining the integrity of the database. Database Protection also includes data recovery that means if database get corrupted due to some reasons like Hard Disk failure or other reasons – how to recover the database.

14.2 Database Integrity

The term integrity refers to the correctness or accuracy of data in database. Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Thus integrity constraints guard against accidental damage to the database.

We have already seen two forms of integrity constraints:

- Key declarations – the stipulation that certain attributes form a candidate key for a given entity set.
- Form of a relationship- many to many, one to many, one to one.

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However arbitrary predicates may be costly to test. Thus we concentrate on integrity constraints that can be tested with minimal overhead. In addition to protecting against accidental introduction of inconsistency, the data stored in the database needs to be protected from unauthorized access and malicious destruction or alteration.

14.2.1 Domain Constraints

Domain Constraints state that a range of possible values must be associated with every attribute. There are a number of standard domain types, such as integer types, character types and date/time types defined in SQL. Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

It is possible for several attributes to have the same domain. For example the attribute *customer-name* and *employee-name* might have the same domain: the set of all person names. However, the domains of balance and branch-name certainly ought to be distinct. It is perhaps less clear whether *customer-name* and *branch-name* should have the same domain. At the implementation level both customer names and branch names are character strings. However we would normally not consider the query “Find all customers who have the same name as a branch" to be a meaningful query. Thus if we view the database at the conceptual, rather than the physical level, *customer-name* and *branch-name* should have distinct domains.

From the above discussion, we can see that a proper definition of domain constraint not only allows us to test values inserted in the database, but also permits us to test queries to ensure that the comparisons made make sense. The principle behind attribute domains is similar to that typing of variables in programming languages. Strongly typed programming languages allow the compiler to check the program in greater detail.

The **create domain** clause can be used to define new domains. For example the statements:

create domain *Dollars* **numeric** (12,2)
create domain *Pounds* **numeric** (12,2)

define the domains *Dollars* and *Pounds* to be decimal numbers with a total of 12 digits, two of which are placed after the decimal point. An attempt to assign a value of type *Dollars* to a variable of type *Ponds* would result in a syntax error, although both are of the same numeric type. Such an assignment is likely to be due to programmer error, where the programmer forgot about the differences in currency. Declaring different domains for different currencies helps catch such errors.

Values of one domain can be *cast* (that is, converted) to another domain. If the attribute A in relation r is of type *Dollars*, we can convert it to *Pounds* by writing

c**ast** r.A as *Pounds*

In a real application we would of course multiply r.A by a currency conversion facts before casting it to pounds. SQL also provides drop domain and after domain clauses to drop or modify or modify domains that have been created earlier.

The **check** clause in SQL permits domains to be restricted in powerful ways that most programming language type systems do not permit. Specifically the check clause permits the schema designer to specify a predicate that must be satisfied by any value assigned to a variable whose type is the domain. For instance a check clause can ensure that an hourly wage domain allows only values greater than a specified value (such as the minimum wage):

create domain *HourlyWage* **numeric**(5,2)
    constraint *wage-value-test* **check** (**value** >=4.00)

The domain *HourlyWage* has constraint that ensures that the hourly wage is greater than 4.00. The clause constraint *wage-value-test* is optional, and is used to give the name *wage-value-test* to the constraint. The name is used to indicate which constraint an update violated.

The **check** can also be used to restrict a domain to not contain any null values:

create domain *AccountNumber* **char**(10)
    constraint *account-number-test* **check** (**value not null**)

Another example, the domain can be restricted to contain only a specified set of values by using the in clause:

create domain *Account type* **char**(10)
constraint *account-type-test*
    check (**value in** ('Checking', 'Saving'))

The preceding check conditions can be tested quite easily when a tuple is inserted or modified. However in general the check conditions can be more complex (and harder to check), since sub queries that refer to other relations are permitted in the check condition. For example this constraint could be specified on the relation deposit.

check (*branch-name* **in** (**select** *branch-name* **from** *branch*))

The check condition verifies that the branch-name in each tuple in the deposit relation is actually the name of a branch in the branch relation. Thus the condition has to be checked not only when a tuple is inserted or modified in deposit but also when the relation branch changes (in this case, when a tuple is deleted or modified in relation branch).

The preceding constraint is actually an example of a class of constraints called referential-integrity constraints.

Complex check conditions can be useful when we want to ensure integrity of data but we should use them with care, since they may be costly to test.

14.2.2 Referential Integrity

Often, we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called referential integrity

Basic Concepts

Consider a pair of relations r(R) and s(S) and the natural join r | s. There may be a tuple t_r in r that does not join with any tuple in s. That is, there is no t_s in s such that t_r [R ∩ S] = t_s [R ∩ S]. Such tuples are called *dangling* tuples. Depending on the entity set or relationship set being modeled dangling tuples may or may not be acceptable.

Suppose there is a tuple t_1 in the account relation with t_1[branch-name] = "Lunartown" but there is no tuple in the branch relation for the “Lunartown” branch. This situation would be undersirable. We expect the branch relation to list all bank branches. Therefore tuple t_1 would refer to an account at a branch that does not exist. Clearly we would like to have an integrity constraint that prohibits dangling tuples of this sort.

Not all instances of dangling tuples are undersirable however. Assume that there is a tuple t2 in the branch relation with t_2 [branch-name] = “Mokan” but there is no tuple in the account relation for the Mokan branch. In this case a branch exists that has no accounts. Although this situation is not common it may arise when a branch is opened or its about to close. Thus we do not want to prohibit this situation.

The distinction between these two examples arises from two facts.

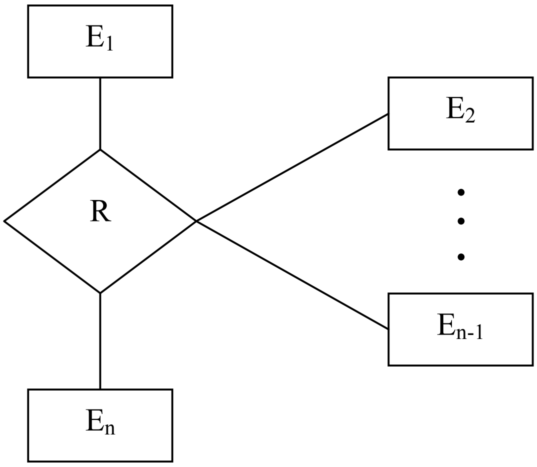
- The attribute branch-name in Account schema is a foreign key referencing the primary key of Branch schema.
- The attribute branch name in Branch schema is not a foreign key. (Recall that a foreign key is a set attribute in a relation schema that forms a primary key for another schema.)

In the Lunartown example, tuple t_1 in account has a value on the foreign key branch-name that does not appear in branch. In the Mokan-branch example tuple t_2 in branch has a value on branch-name that does not appear in account, but branch-name is not a foreign key. Thus the distinction between our two examples of dangling tuples is the presence of a foreign key.

Let r_1 (R_1) and r_2 (R_2) be relations with primary keys K_1 and K_2 respectively. We say that a subset α of R2 is a **foreign key** referencing K_1 in relation r_1 if it is required that for every t_2 in r_2 there must be a tuple t_1 in r_1 such that t_1 [K_1] = t_2 [α]. Requirements of this form are called referential integrity constraints or subset dependencies. The latter term arises because the preceding referential-integrity constraint can be written as Π_α (r_2) ≤ Π_{K1}(r1). Note that for a referential-integrity constraint to make sense either must be equal to K_1 or α and K_1 must be compatible sets of attributes.

Referential Integrity and the E-R Model

Referential-integrity constraints arise frequently. If we derive our relational-database schema by constructing tables from E-R diagrams, then every relation arising from a relationship set has referential-integrity constraints. Figure below shows an n-ary relationship set R, relating entity sets E_1, E_2, ... ,E_n. Let K_i denote the primary key of E_i. The attributes of the relation schema for relationship set R include K_1 U K_2 U ... U K_n. The following referential integrity constraints are then present: For each i, K_i in the schema for R is a foreign key referential K_i in the relation schema generated from entity set E_i.



An n-ary relationship set

Another source of referential-integrity constraints is weak entity sets. Recall that the relation schema for a weak entity set must include the primary key of the entity set on which the weak entity set depends. Thus the relation schema for each weak entity set includes a foreign key that leads to a referential integrity constraint.

14.2.4 Database Modification

Database modifications can cause violations of referential integrity. We list here the test that we must make each type of database modification to preserve the following referential integrity constraint:

Π_α (r2) ≤ Π_k (r1)

- **Insert**. If a tuple t_2 is inserted into r2, the system must ensure that there is a tuple t_1 in r1 such that t_1 [K] = t_2[α]

  t_2[α] ε Π_k (r1)

- **Delete**. If a tuple t_1 is deleted from r1 the system must compute the set of tuples in r2 that reference t_1:

  $\sigma_{\alpha = t1[k]}$ (r2)

If this set is not empty, either the delete command is rejected as an error, or the tuples that reference t_1 must themselves be deleted. The latter solution may lead to cascading deletions, since tuples may reference tuples that reference t_1 and so on.

Update : We must consider two cases for update: updates to the referencing relation and updates to the referenced relation (r1).

- ❑ If a tuple t_2 is updated in relation r2 and the update modifies values for the foreign key then a test similar to the insert case is made. Let t_2' denote the new value of tuple t_2. The system must ensure that

  $$t_2'[a] \ \varepsilon \ \Pi_k \, (r1)$$

- ❑ If a tuple t_1 is updated in r1 and the update modifies values for the primary key (K), then a test similar to the delete case is made. The system must compute.

  $$\sigma_{\alpha = t1[K]} \, (r2)$$

using the old value of t1 (the value before the update is applied). If this set is not empty, the update is rejected as an error or the update is cascaded in a manner similar to delete.

14.2.5 Referential Integrity in SQL

Foreign keys can be specified as part of the SQL **create table** statement by using the foreign key clause. We illustrate foreign-key declarations by using the SQL DDL definition of part of our bank database shown in Figure *K*.

By default a foreign key references the primary key attributes of the referenced table. SQL also supports a version of the references clause where a list of attributes of the referenced relation can be specified explicitly. The specified list of attributes must be declared as a candidate key of the referenced relation

We can use the following short form as part of an attribute definition to declare that the attribute forms a foreign key:

branch-name **char**(15) **references** *branch*

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation. However a foreign key clause can specify that if a delete or update action on the referenced relation violates the constraint, then instead of rejecting the action, the system must take steps to change the tuple in the constraint on the relation account:

```
create table account
        ( …
        Foreign key (branch-name) references branch
                                  on delete cascade
                                  on update cascade,
        … )
```

Figure *K*: SQL data definition for part of the bank database

```
create table customer
    (customer-name char(20)
    customer-street   char(30)
    customer-city     char(30)
    primary key (customer-name))
create table branch
    (branch-name char(15)
    branch-street   char(30)
    assets     integer
    primary key (branch-name)
    check (assets >=0))
create table account
    (account-number char(10)
    branch-name   char(15)
    balance     integer
    primary key (account-number)
    foreign key (branch-name) references branch,
check (balance >=0))
create table depositor
    (customer-name char(20)
    account-number   char(10)
    primary key (customer-name, account-number)
    foreign key (customer-name) references customer,
    foreign key (account-number) references account)
```

Because of the clause on delete cascade associated with the foreign key declaration if a delete of a tuple in branch results in this referential integrity constraint being violated the system does not reject the delete. Instead the delete "cascades" to the accout relation, deleting the tuple that refer to the branch tuple that was deleted. Similarly, the system does not reject an update to a field referenced by the constraint even if it violates the constraint; instead the system updates the field branch-name of the referencing tuples in account to the new value as well. SQL also allows the foreign key clause to specify actions other than cascade, if the constraint is violated. The referencing field (here, branch-name) can be set to null (by using set null in place of cascade), or to the default value for the domain (by using set default).

If there is a chain of foreign key dependencies across multiple relations, a deletion or update at one end of the chain can propagate across the entire chain. An interesting case where the foreign key constraint on a relation references the same relation appears in Exercise. If a cascading update or delete cause a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.

Null values complicate the semantics of referential integrity constraint in SQL. Attributes of foreign keys are allowed to be null, provided that they have not other wise been declared to be non-null. If all the columns of a foreign key are non-null in a given tuple, the usual definition of foreign key constraint is used for that tuple. If any of the foreign key columns is null, the tuple is defined automatically to satisfy the constraint.

This definition may not always be the right choice, so SQL also provides constructs that allow you to change the behavior with null values, we do not discuss the constructs here. To avoid such complexity, it is best to ensure that all columns of a foreign key specification are declared to be non-null.

Transactions may consist of several steps, and integrity may be violated temporarily after one step but a later step may remove the violation. For instance, suppose we have a relation-married person with primary key-name, and an attribute spouse, and suppose that spouse is a foreign key on married person. That is the constraint says that the spouse attribute must contain a name that is present in the person table. Suppose we wish to note the fact that John and Mary are married to each other by inserting two tuples one for John and one for Mary in the above relation. The insertion of the first tuple violate the foreign key constraint, regardless of which of the two tuples is inserted first. After the second is inserted the foreign key constraint would hold again.

To handle such situations integrity constraints are checked at the end of a transaction and not at intermediate steps.

14.2.6 Assertions

An Assertion is a predicate expressing a conditions that we wish the database always to satisfy. Domain constraint and referential integrity constraints are special forms of assertions. We have paid substantial attention to these forms of assertions because they are easily tested and apply to a wide range of database applications. However there are many constraints that we cannot express by using only these special forms. Two examples of such constraints are:

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.
- Every has at least one customer who maintains an account with a minimum balance of $1000.00.

An assertion in SQL takes the form

create assertion <assertion-name> **check** <predicate>

Here is how the two examples of constraints can be written. Since SQL does not provide a "for all X P (X)" construct (where P is a predicate) we are forced to implement the construct by an equivalent "not exists X" such that not P(X) construct which can be written in SQL. We write

```
create assertion sum-constraint check
    (not exists (select * from branch
    where (select sum (amount) from loan
        where loan. branch-name = branch.branch-name)
            >=(select sum (balance) from account
        where account.branch-name = branch,branch-name)))

        create assertion balance-constraint check
        (not exists (select * from loan
        where not exists(select *
        from borrower, depositor, account
        where loan.loan-number = borrower.loan-number
        and borrower.costomer-name = depositor.customer-name
        and depositor.account-number = account.account-number
```

and account.balance >= 1000)))

When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated. This testing may introduce a significant amount of overhead if complex assertions have been made. Hence, assertions should be used with great care. The high overhead of testing and maintaining assertions has led some system developers to omit support for general assertions, or to provide specialized forms of assertions that are easier to test.

14.3 Database Recovery

Recovery in database system means, primarily, recovering the database itself: that is, restoring the database to a state that is known to be correct (or rather, consistent) after some failure has rendered the current state inconsistent.

14.3.1 Transactions

We begin our discussions by examining the fundamental notion of a transaction. A transaction is a logical unit of work. Consider the following example. Suppose the parts relation P includes an additional attribute TOTQTY, representing the total shipment quantity for the part in question; in other words, the value of TOTQTY for any given part is supposed to be equal to the sum of all QTY values, taken over all shipments for that part. Now consider the pseudocode procedure shown in Figure below, the intent of which is to add a new shipment for supplier S5 and part P1, with quantity 1000, to the database (the INSERT inserts the new shipment, the UPDATE updates the TOTQTY value for part P1 accordingly).

```
BEGIN TRANSACTION;

INSERT INTO SP
        RELATION { TUPLE {S# S# ('S5'),
                          P# P# ('P1'),
                          QTY QYT (1000) } } ;
IF any error occurred THEN GO TO UNDO; END IF;

UPDATE P WHERE P# = P# ('P1')
        TOTQTY := TOTQTY + QTY (1000);
IF any error occurred THEN GO TO UNDO; END IF;

COMMIT;
GO TO FINISH;

UNDO:
        ROLLBACK;

FINISH:
        RETURN;
```

The point of the example is that what is presumably intended to be a single atomic operation- "add a new shipment"- in fact involves two updates to the database, one INSERT operation and one UPDATE operation. What is more, the database is not even consistent between those two updates; it temporarily violates the constraint that the value of TOTQTY for part P1 is supposed to be equal to the sum of all QTY values for part P1. Thus a logical unit of work (i.e., a transaction) is not necessarily just a single database operation; rather, it is a sequence of several such operations, in general that transforms a consistent state of the database into another consistent state, without necessarily preserving consistency at all intermediate points.

Now, it is clear that what must not be allowed to happen in the example is for one of the updates to be executed and the other not, because that would leave the database in an inconsistent state. Ideally of course we would like a cast iron guarantee that both updates will be executed. Unfortunately, it is impossible to provide such a guarantee-there is always a chance that things will go wrong, and go wrong moreover at the worst possible moment. For example, a system crash occur between the INSERT and the UPDATE, or an arithmetic overflow might occur on the UPDATE, etc. But a system that support transaction management does provide the next best thing to such a guarantee. Specifically, it guarantees that if the transaction reaches some updates and then a failure occurs (for whatever reason) before the transaction reaches its planned termination, then those updates will be undone. Thus the transaction either executes in its entirety or is totally canceled i.e. made as if it never executed at all. In this way, a sequence of operations that is fundamentally not atomic can be made to look as if it were atomic from an external point of view.

The system component that provides this atomicity- or resemblance of atomicity- is known as the transaction manager (also known as the transaction processing monitor or TP monitor) and the COMMIT and ROLLBACK operations are the keep to the way it works;

- The COMMIT operation signals successful end of transaction; it tells the transaction manager that a logical unit of work has been successfully completed and the database is (or should be) in a consistent state again and all of the updates made by that unit of work can now be committed or made permanent.
- By contrast the ROLLBACK operation signals unsuccessful end of transaction; it tells the transaction manager that something has gone wrong. The database might be in an inconsistent state and all of the updates made by the logical unit of work so far must be rolled back or undone.

In the example therefore we issue a COMMIT if we get through the two updates successfully which will commit the changes in the data base and make them permanent. If any thing goes wrong however- i.e., if either of the updates raises an error condition- then we issue a ROLLBACK instead to undo any changes made so far.
Note: Even if we issue a commit instead the system should in principal check the database integrity constraint. It detects the fact that the database is inconsistent and force a ROLLBACK any way. However we don't assume that the system is aware of all pertinent constraint and so the users issued ROLLBACK is necessary. Commercial DBMSs do not do very much COMMIT time integrity checking at the time of writing.

Incidentally we should point out that a realistic application will not only update the database (or attempt to) but will also send some kind of message back to the end user indicating what has happened. In the example we might send the message shipment added if the COMMIT is reached or the message error shipment not added otherwise. Message handling in turn has additional implications for recovery.

Note: At this juncture you might be wondering how it is possible to undo and update. The answer of course is that the system maintains a log or journal on tape or (more commonly) disk on which details of all updates- in particular before and images of the updated objects- are recorded. Thus, if it becomes necessary to undo some particular update the system can use the corresponding log entry to restore the updated object to its previous value.

(Actually the fore going paragraph is somewhat over simplified . In practice the log will consist of two portions an active or online portion and an archive or offline portion. The online portion is used during normal system operation to record details of updates as they are performed and is normally held on disk. When the online portion becomes full its contents are transferred to the offline portion which- because it is always processed sequentially- can be held on the tape.

One further point; the system must guarantee that individual statements are themselves atomic (all or nothing). This consideration becomes particularly significant in relational system, where statements are set-level and typically operate on many tuples at a time; it must not be possible for such a statement to fail in the middle and leave the database in an inconsistent state (e.g. with some tuples update and some not). In other words if an error does occur in the middle of such a statement, then the database must remain totally unchanged.

14.3.2 Transaction Recovery

A transaction begins with successful execution of a BEGIN TRANSACTION statement and it ends with successful execution of either COMMIT or a ROLLBACK statement. COMMIT establishes what is called, among many other things, a commit point (also especially in commercial products-known as a synch point). A commit point thus corresponds to the end of a logical unit of work, and hence to a point at which the database is or should be in a consistent state. ROLLBACK, by constraint rolls the database back to the state it was in at BEGIN TRANSACTION which effectively means back to the previous commit point. (The phrase "the previous commit point" is still accurate, even in the case of the first transaction in the program, if we agree to think of he first BEGIN TRANSACTION in the program as tacitly establishing an initial " commit point".

Note: Throughput this section the term "database" really means just that portion of the database being accessed by the transaction under consideration. Other transactions might be executing in parallel with that transaction and making changes to their own portions, and so "the total database" might not be in a fully consistent state at a commit point. However we are ignoring the possibility does not materially affect the issue at hand, of course.

When a commit point is established:

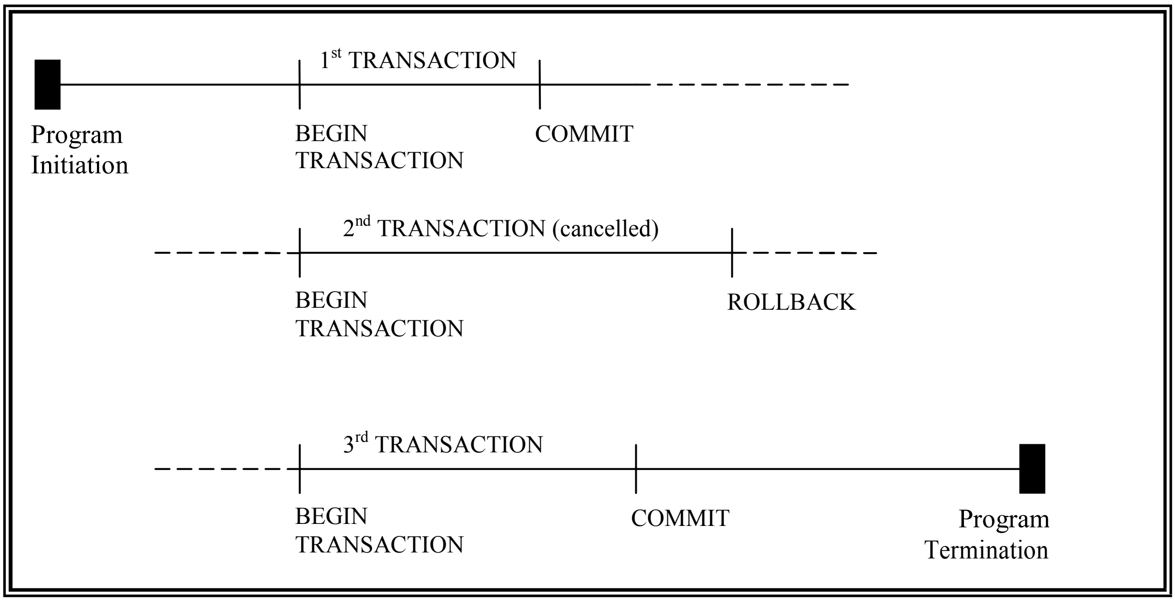
1 All updates made by the executing program since the previous commit points are committed; that is, they are made permanent. Prior to the commit point, all such updates should be regarded as tentative only—tentative in the sense that they might subsequently be undone (i.e. rolled back). Once committed an update is guaranteed never to be undone (this is the definition of " committed").

2 All database positioning is lost and all tuple locks are released. "Database poisoning" here refers to the idea that at any given time an executing program will typically have address ability to certain tuples (e.g., via certain cursors in the case of SQL, this address ability is lost at a commit point. "Tuple locks" are explained in the next chapter. Note some systems do provide an option by which the program in fact might

be able to retain address ability to certain tuples (and therefore retain certain tuple locks) from one transaction to the next.

Paragraph 2 here – excluding the remark about possibly retaining some address ability and hence possibly retaining certain tuple locks—also applies if a transaction terminates with ROLLBACK instead of COMMIT. Paragraph 1 of course does not.

Note carefully that COMMIT and ROLLBACK terminate the transaction, not the program. In general a single program execution will consist of a sequence of several transactions running one after another, as illustrated in Figure below:



Program execution is a sequence of transactions

Now let us return to the example of the previous section. In that example we include explicit tests for errors, and issued an explicit ROLLBACK if any error was detected. But of course the system cannot assume that application programs will always include explicit tests for all possible errors. Therefore the system will issue an implicit ROLLBACK for any transaction that fails for any reason to reach its planned termination (where "planned termination" means either an explicit COMMIT or an explicit ROLLBACK).

We can now see therefore, that transactions are not only the unit of works but also the unit of recovery. For if a transaction successfully commits, then the system will guarantee that its updates will be permanently installed in the database, even if the system crashed the very next moment. It is quite possible, for instance, that the system might crash after the COMMIT has been honored but before the updates have been physically written to the database- they might still be waiting in a main memory buffer and so be lost at the time of crash. Even if that happens the system's restart procedure will still install those updates in the database; it is able to discover the values to be written by examine the relevant entries in the log. (it follows that the log must be physically written before COMMIT processing can complete- the write ahead log rule.) Thus the restart procedure will recover any transactions that completed successfully but did not manage to get their updates physically written prior to the crash; hence as stated earlier transaction are in deed the unit of recovery.

Note: In the next chapter we will see there is a unit on concurrency also. Further since they are supposed to transform a consistent state of the database in to another consistent state they can also be regarded as a unit of integrity.

14.3.3 The ACID Properties

Transactions have four important properties- *atomicity*, *consistency*, *isolation* and *durability* (referred to colloquially as "the ACID properties")

- **Atomicity**: Transaction are atomic (all or nothing)
- **Consistency**: Transaction preserves database consistency. That is a transaction transforms a consistent state of the database in to another consistent state without necessarily preserving consistency at all intermediate points.
- **Isolation**: Transactions are isolated from one another. That is even though in general there will be many transactions running concurrently at any given transaction updates are concealed from all the rest until that transaction commits. Another way of seeing the same thing of that for any two distinct transactions T1 and T2, T1 might see T2's updates (after T2 has committed) or T2 might see T1's updates (after T1 has committed) but certainly not both.
- **Durability**: Once a transaction commits it updates survive in a database even if there is subsequent system crash.

14.3.4 System Recovery

The system must be prepared to recover not only from purely local failures such as occurrence of an over flow condition with in an individual transaction but also from "Global" failures such as power outage. A local failure by definition effects only the transaction in which the failure has actually occurred. A global failure, by contrast, affects all of the transactions in progress at the time of failure and hence has significant system wide implications. In this section and the next we briefly consider what is involved in recovering from a global failure. Such failures fall in to two categories :

- **System failures**: (e.g., power outage), which effect all transactions, currently in progress but do not physically damage the database. A system failure is some times called a soft crash.

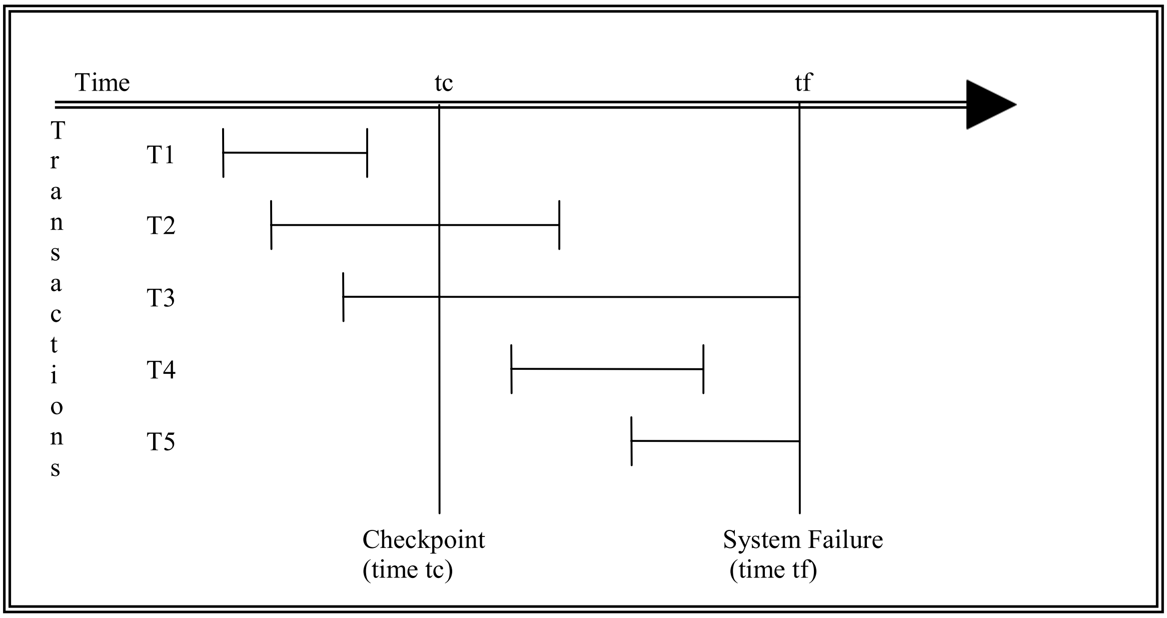
- **Media failures**: (e.g. head crash on disk), which do cause damage to the database or to some portion of it and effect at least those transactions currently using that portion. A media failure is sometimes called a *hard crash*.

The key point regarding system failure is that the contents of main memory are lost (in particular the database buffers are lost). The precise state of any transaction can therefore never been successfully completed and so must be undone- i.e. rolled back- when the system restarts.

Further more it might also be necessary to re do certain transactions at restart time that did successfully complete prior to the crash but did not manage to get their updates transferred from the database buffers to the physical database.

The obvious question therefore arises; how does the system know at restart time which transactions to undo and which to redo? The answer is as follows. At certain prescribed intervals typically whenever some prescribed numbers of entries have been written to the log- the system automatically take a check point. Taking a check point involves (a.) Physically writing "(force writing") the content of the database buffers out to the physical database and (b) physically writing a special check point record out to the physical log. The check point record gives a list of all transactions that were in progress at the time the check point was taken. To see how this information is used consider the following Figure which is read as follows(note that time in the fig. Flows from left to right)

- A system failure has occurred at time tf.
- The most recent check point prior to time tf was taken at a time tc.
- Transaction of type t1 completed prior to time tc.
- Transaction of type T2 started prior to time tc and completed after time tc and before time tf.
- Transaction of type T3 also started prior to time tc but did not complete by time tf.
- Transaction of type T4 started after time tc and completed before time tf.
- Finally transaction of type T5 also started after time tc but did not complete by time tf.



Five transaction categories

It should be clear that when the system is restarted transaction of type T3 and T5 must be undone, and transaction of types T2 and T4 must be redone. Note however that transactions of type T1 do not enter in to the restart process at all because its updates were forced to the database at time t_c as part of the check point process. Note two that transaction that completed unsuccessfully (i.e. with the rollback) before time tf also do not enter into the restart process at all(why not?).

At restart time therefore the system first goes through the following procedure in ordered to identify all transaction of types T2 to T5;

1 Start with two lists of transactions the undo list and the redo list. Set the undo list equal to the list of all transactions given in the most recent check point record; set the redo list is empty.

2 Search forward through the log starting from the check point record.

3 If a BEGIN TRANSACTION log entry is found for transaction T add T to the undo list.

4 If COMMIT log entries found for transaction T move T from the UNDO list to the REDO list.

5 When the end of log is reached the UNDO and REDO list, identify respectively transactions of types T3 and T5 and transaction of types T2 and T4.

The system now works backward through the log undoing the transactions in the UNDO list; then it works forward again redoing in the transaction in the REDO list. Note: Restoring the database to consistent state by undoing work is some times called backward recovery. Similarly restoring it to a consistent state by redoing work is some times called forward recovery.

Finally when all such recovery activities are complete, then (and only then) the system is ready to accept new work.

14.3.5 Media Recovery

A media failure is a failure such as a disk head crash or a disk controller failure in which some portion of a database has been physically destroyed. A recovery from such a failure basically involves reloading (or restoring) the database from a backup copy (or dump) and then using the log; both active and achieve portions in general – to redo all transactions that completed since that backup copy was taken. There is no need to undo transactions that were still in progress at the time of the failure since by definition all updates of such transactions have been undone (actually lost) any way.

The need to be able to perform media recovery implies the need for a dump/restore (or unload/reload) utility. The dump portion of that utility is used to make backup copies of the database on demand. (such copy can be kept on tape or other archival storage; it is not necessary that they be on direct access media). After a media failure the restore portion of the utility is used to recreate the database from a specified backup copy.

14.4 Summary

The term integrity refers to the correctness or accuracy of data in database. Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. In general an integrity constraint can be an arbitrary predicate pertaining to the database. Domain constraints are the most elementary form of integrity constraint. Often, we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called referential integrity. Recovery in database system means, primarily, recovering the database itself: that is, restoring the database to a state that is known to be correct (or rather, consistent) after some failure has rendered the current state inconsistent. Transactions have four important properties- *atomicity*, *consistency*, *isolation* and *durability.* The system must be prepared to recover not only from purely local failures such as occurrence of and over flow condition with in an individual transaction but also from "Global" failures such as power outage. A media failure is a failure such as a disk head crash or a disk controller failure in which some portion of a database has been physically destroyed.

14.5 Questionnaires:

1. What do you understand by data integrity? Explain various types of integrity constraints along with suitable example.
2. What do you understand by database recovery? Explain various types of recovery techniques.
3. What do you understand by a transaction? Explain the ACID properties of transactions.

DATA BASE SECURITY

Introduction
Objectives
Database Security
Authorization
Encryption and Authentication
Methods of implementing Security
Questionnaires

15.0 Introduction

Database Security is a crucial issue in the database management system as it contain important information which is very valuable and sensitive for an organization's database. Security in a database involves both policies and mechanisms to protect the data from unauthorized users to access and update. Authorization is a process of permitting users to perform certain operations on certain data objects in a shared database. Authorization is a process of granting a right or a privilege that enables user to have some rights to access a system or a system object. The various provisions that a database system may make for authorization may still not provide sufficient protection for highly sensitive data. In such cases data may be stored in encrypted form. It is not possible for encrypted data to be read unless the reader knows how to decipher (decrypt) them. Encryption also forms the basis of good schemes for authenticating users to a database. Authentication refers to the task of verifying the identity of a person/software connecting to a database. In the following sections we will study in details how database can be made secure.

15.1 Objective

After reading the lesson, we will be able to

- Learn about the database security
- Understand authorization
- Understand Authentication
- Understand various Encryption techniques
- Understand various methods of implementing database security

15.2 Database Security

The data stored in the database need protection from unauthorized access and malicious destruction or alteration, in addition to the protection against accidental introduction of inconsistency that integrity constraints provide. In this section, we examine the ways in which data may be misused or intentionally made inconsistent. We then present mechanisms to guard against such occurrences.

Security Violations among the forms of malicious access are :

- Unauthorized reading of data (theft of information)
- Unauthorized modification of data
- Unauthorized destruction of data

Database security refers to protection from malicious access. Absolute protection of the database from malicious abuse is not possible, but the cost to the perpetrator can be made high enough to deter most if not all attempts to access the database without proper authority.

To protect the database, we must take security measures at several levels.

- **Database system**. Some database users may be authorized to access only a limited portion of the database. Other users may be allowed to issue queries, but may be forbidden to modify the data. It is the responsibility of the database system to ensure that these authorization restrictions are not violated.
- **Operation system**. No matter how secure the database system is, weakness in operating-system security may serve as a means of unauthorized access to the database.
- **Network**. Since almost all database systems allow remote access through terminals or networks, software-level security within the network software is as important as physical security, both on the Internet and in private networks.
- **Physical**. Sites with computer systems must be physically secured against armed or surreptitious entry by intruders.
- **Human**. Users must be authorized carefully to reduce the chance of any user giving access to an intruder in exchange for a bride or other favors.

Security at all these levels must be maintained if database security is to be ensured. A weakness at a low level of security (physical or human) allows circumvention of strict high-level (database) security measures.

In the remainder of this section, we shall address security at the database-system level. Security at the physical and human levels, although important, is beyond the scope of this text.

Security within the operating system is implemented at several levels, ranging from passwords for access to the system to the isolation of concurrent processes running within the system. The file system also provides some degree of protection. The bibliographical notes reference coverage of these topics in operating-system texts. Finally, network-level security has gained widespread recognition as the Internet has evolved from an academic research platform to the basis of international electronic commerce. The bibliographic notes list textbook coverage of the basic principles of network security. We shall present our discussion of security in terms of the relational-data model, although the concepts of this chapter are equally applicable to all data models.

15.3 Authorization

We may assign a user several forms of authorization on parts of the database. For example,

- **Read authorization** allows reading, but not modification of data.
- **Insert authorization** allows inserting of new data, but not modification of existing data.
- **Update authorization** allows modification, but not deleting of data.
- **Delete authorization** allows deleting of data.

We may assign the user all, none or a combination of these types of authorization.

In addition to these forms of authorization for access of data, we may grant a user authorization to modify the database schema.

- **Index authorization** allows the creation and deleting of indices.
- **Resource authorization** allows the creation of new relations.
- **Alteration authorization** allows the addition or deleting of attributes in a relation.

- **Drop authorization** allows the deletion of relations.

The drop and delete authorization differ in that delete authorization allows deletion of tuples only. If a user deletes all tuples of a relation, the relation still exists, but it is empty. If a relation is dropped in to longer exists.

We regulate the ability to create new relations through resource authorization. A user with resource authorization who creates a new relation is given all privileges on that relation automatically.

Index authorization may appear unnecessary, since the creation or deleting of an index does not alter data in relations. Rather indices are a structure for performance enhancements. However, indices also consume space, and all database modifications are required to update indices. If index authorization were granted to all users, those who performed updates would be tempted to delete indices, whereas those who issued queries would be tempted to create numerous indices. To allow the *database administrator* to regulate the use of system resources, it is necessary to treat index creation as a privilege.

The ultimate form of authority is that given to the database administrator. The database administrator may authorize new users, restructure the database, and so on. This form of authorization is analogous to that of a superuser or operator for an operating system.

Authorization and Views

Views are a means of providing a user with a personalized model of the database. A view can hide data that a user does not need to see. The ability of views to hide data serves both to simplify usage of the system and to enhance security. Views simplify system usage because they restrict the user's attention to the data of interest. Although a user may be denied direct access to a relation, that user may be allowed to access part of that relation through a view. Thus a combination of relational-level security and view-level security limits a user's access to precisely the data that the user needs.

In our banking example consider a clerk who needs to know the names of all customers who have a loan at each branch. This is not authorized to see information regarding specific loans that the customer may have. Thus the clerk must be denied direct access to the loan relation. But, if she is to have access to the information needed, the clerk must be granted access to the view *cust-loan*, which consists of only the names of customers and the branches at which they have a loan. This view can be defines in SQL as follows:

```
create view cust-loan as
        (select branch-name, customer-name
        from borrower, loan
        where borrower.loan-number = loan.loan-number)
```

Suppose that the clerk issues the following SQL query:

```
select *
from cust-loan
```

Clearly, the clerk is authorized to see the result of this query. However, when the query processor translates it into a query on the actual relations in the database, it produces a query on *borrower* and *loan*. Thus the system must check authorization on the clerk's query before it begins query processing.

Creation of a view does not require resource authorization. A user who creates a view does not necessarily receive all privileges on that view. She receives only those privileges that provide no additional authorization beyond those that she already had. For example, a user cannot be given update authorization on a view without having update authorization on the relations used to define the view. If user creates a view on which no authorization can be granted, the system will deny the view creation request. In our *cust-loan* view example, the creator of the view must have read authorization on both *borrower* and *loan* relations.

Granting of Privileges

A user who has granted some form of authorization may be allowed to pass on this authorization to other users. However, we must be careful how authorization may be passed among users, to ensure that such authorization can be revoked at some future time.

Consider as an example, the granting of update authorization on the loan relation of the bank database. Assume that initially the database administrator grants update authorization on loan to users U$_1$, U$_2$ and U$_3$ who may in turn pass on this authorization to other users. The passing of authorization from one user to another can be represented by an **authorization graph**. The nodes of this graph are the users. The graph includes an edge U$_i$ → U$_j$ if user U$_i$ grants update authorization on loan to U$_j$. The root of the graph is the database administrator. In the sample graph in Figure *L*, observe that user U$_5$ is granted authorization by both U$_1$ and U$_2$; U$_4$ is granted authorization by only U$_1$.

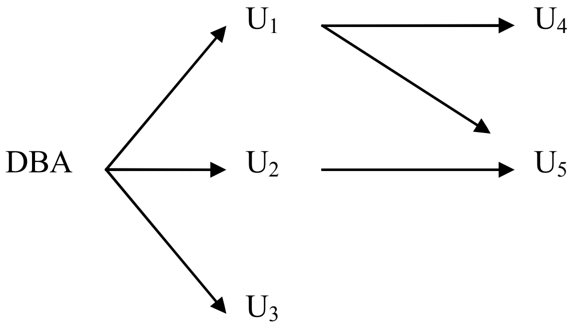


Figure *L*: Authorization-grant chart

A user has an authorization if and only if there is a path from the root of the authorization to that user. Suppose the DBA revokes the authorization of U$_1$. Since U$_4$ has authorization from U$_1$ that authorization should be revoked as well. However, U$_5$ was granted authorization by both U$_1$ and U$_2$. Since the database administrator did not revoke update authorization on loan from U$_2$, U$_5$ retains update authorization on loan. If U$_2$ eventually revokes authorization from U$_5$, then U5 loses the authorization.

A pair of devious users might attempt to defeat the rules for revocation of authorization by granting authorization to each other, as shown in Figure *M* (a). If the database administrator revokes authorization from U$_2$, U$_2$ retains authorization through U$_3$ as in Figure *M* (b). If authorization is revoked subsequently from U$_3$, U$_3$ appears to retain authorization through U$_2$, as in Figure *M* (c). However when the database administrator revokes authorization from U$_3$, the edges from U$_3$ to U$_2$ and from U$_2$ to U$_3$ are no longer part of a path starting with the database administrator.

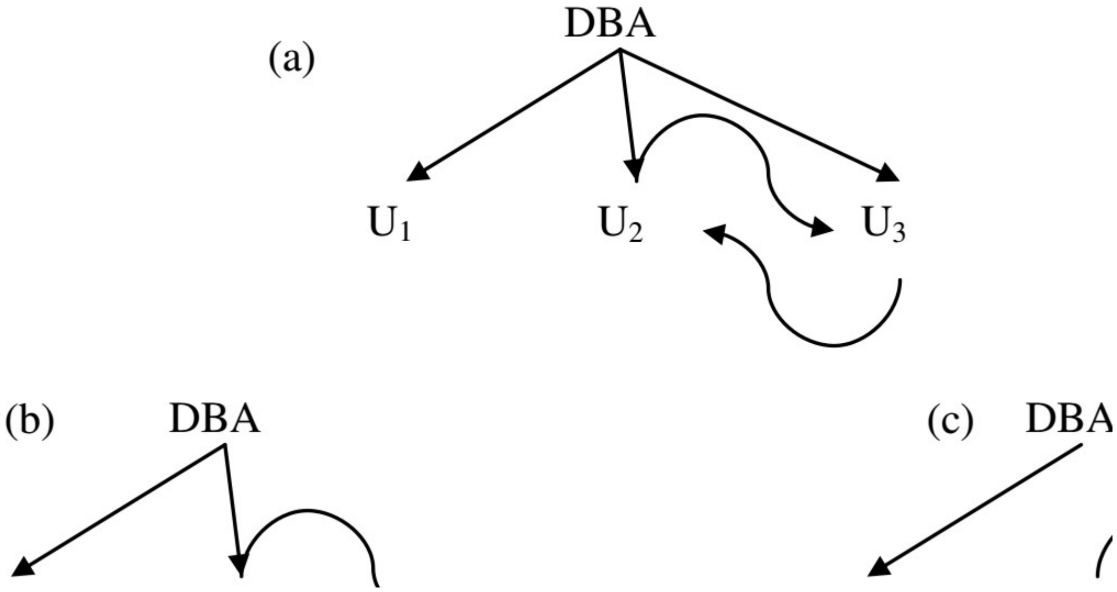
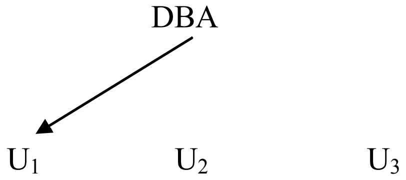


Figure *M*: Attempt to defeat authorization revocation

We require that all edges in an authorization graph be part of some path originating with the database administrator. The edges between U$_2$ and U$_3$ are deleted, and the resulting authorization graph is an in Figure below:



Authorization graph

Notion of Roles

Consider a bank where there are many tellers. Each teller must have the same types of authorization to the same set of relations. Whenever a new teller is appointed, she will have to be given all these authorizations individually.

A better scheme would be to specify the authorization that every teller is to be given, and to separately identify which database users are tellers. The system can use these two pieces of information to determine the authorizations of each who is a teller. When a new person is hired as a teller, a user identifier must be allocated to him, and he must be identified as a teller. Individual permissions given to tellers need not be specified again.

The notion of *roles* captures this scheme. A set of roles is created in the database. Authorization 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 he or she is authorized to perform.

In our bank database examples of roles could include *teller*, *branch-manager*, *auditor* and *system-administrator*.

A less preferable alternative would be to create a *teller* userid and permit each teller to connect to the database using the *teller* userid. The problem with this scheme is that it would not be possible to identity exactly which teller carried out a transaction, leading to security risks. The use of roles has the benefit of requiring users to connect to the database with their own userid.

Any authorization that can be granted to a user can be granted to a role. Roles are granted to users just as authorizations are. And like other authorization a user may also be granted authorization to grant a particular role to others. Thus branch managers may be granted authorization to grant the *teller* role.

Audit Trails

Many secure database applications require an audit trail be maintained. An audit trail is a log of all changes (inserts/ deletes/ updates) to the database, along with information such as which user performed the changes and when the change was performed.

The audit trails aids security in several ways. For instance if the balance on an account is found to be incorrect the bank may wish to trace all the updates performed on the account, to find out incorrect (or fraudulent) updates as well as the persons who carried out the updates. The bank could then also use the audit trail to trace all the tuples performed by these persons, in order to find other incorrect or fraudulent updates.

It is possible to create an audit trail by defining appropriate triggers on relation updates (using system-defined variables that identify the user name and time). However, many database systems provide built in mechanisms to create audit trails, which are much more convenient to use. Details of how to create audit trails vary across database systems, and you should refer the database system manuals for details.

Authorization in SQL

The SQL language offers a fairly powerful mechanism for defining authorizations. We describe these mechanisms, as well as their limitations, in this section

Privileges in SQL

The SQL standard includes the privileges **delete**, **insert**, **select** and **update**. The **select** privilege corresponds to the read privilege. SQL also includes a **references** privilege that permits a user/role to declare foreign keys when creating relations. If the relation to be created includes a foreign key that references attributes of another relation, the user/role must have been granted **references** privilege on those attributes. The reason that the **references** privilege is a useful feature is somewhat subtle; we explain the reason later in this section.

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 following **grant** statement grants users U_1, U_2 and U_3 select authorization on the account relation.

grant select on *account* **to** U_1, U_2, U_3

The **update** authorization may be given either on all attributes of the relation or on only some. If **update** authorization is included in a **grant** statement, the list of attributes on which update authorization is to be granted optionally appears in parentheses immediately after the **update** keyword. If the list of attributes is omitted, the update privileges will be granted on all attributes of the relation.

This **grant** statement gives users U_1, U_2 and U_3 update authorization on the amount attribute of the loan relation:

grant update (amount) **on** loan **to** U_1, U_2, U_3

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 SQL **references** privilege is granted on specific attributes in a manner like that for the **update** privilege. The following **grant** statement allows user U_1 to create relations that reference the key branch-name of the branch relation as a foreign key:

grant references (branch-name) **on** branch **to** U_1

Initially it may appear that there is no reason ever to prevent users from creating foreign keys referencing another relation. However, foreign key constraints restrict deleting and update operations on the referenced relation. In the preceding example, if U_1 creates a foreign key in a relation r referencing the *branch-name* attribute of the *branch* relation, and then inserts a tuple into r pertaining to the Perryridge branch, it is no longer possible to delete the Perryridge branch from the branch relation without also modifying relation r. Thus the definition of a foreign key by U_1 restricts future activity by other users; therefore there is a need for the **references** privilege.

The privilege **all privileges** can be used a short form for granting all the allowable privileges. Similarly the user name **public** refers to all current and future users of the system. SQL also includes a **usage** privilege that authorizes a user to use a specified domain (recall that a domain corresponds to the programming-language notion of a type, and may be user defined).

Roles

Roles can be created in SQL:1999 as follows

create role *teller*

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

grant select on *account*
to *teller*

Roles can be assigned to the users, as well as some other roles, as there statements show

grant *teller* **to** john
create role *manager*
grant *teller* **to** *manager*
grant *manager* **to** mary

Thus the privileges of 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

Note that there can be a chain of roles; for example the role *employee* may be granted to all *tellers*. In turn the role teller is granted to all *managers*. Thus the *manager* role inherits all privileges granted to the roles *employee* and to *teller* in addition to privileges granted directly to *manager*.

The Privilege to Grant Privileges

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 U1 the select privilege on branch and allow U1 to grant this privilege to others, we write

grant select on *branch* **to** U_1 **with grant option**

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> [**restrict | cascade**]

Thus, to revoke the privileges that we granted previously, we write

revoke select on *branch* **from** U_1, U_2, U_3
revoke update (*amount*) **on** *loan* **from** U_1, U_2, U_3
revoke references (*branch-name*) **on** *branch* **from** U_1

The revocation of a privilege from user/role may cause other users/role also to lose that privilege. This behavior is called cascading of the revoke. In most database system, cascading is the default behavior; the keyword **cascade** can thus be omitted, as we have done in the preceding examples. The **revoke** statement may alternatively specify **restrict**:
revoke select on *branch* **from** U_1, U_2, U_3 **restrict**

In this case the system returns an error if there are any cascading revokes, and does not carry out the revoke action. The following revoke statement revokes only the grant option rather than the actual select privilege:

revoke grant option for select on *branch* **from** U_1

Other Features

The creator of an object (relation/view/role) gets all privileges on the object, including the privilege to grant privileges to others.

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. Thus, schema modifications such as creating or deleting relations adding or dropping attributes or relations and adding or dropping indices—may be executed by only the owner of the schema. Several database implementations have more powerful authorization mechanisms for database schemas similar to those discussed earlier but these mechanisms are nonstandard.

Limitations of SQL Authorization

The current SQL standards for authorization have some shortcomings. For instance, suppose you want all students to be able to see their own grades, but not the grades of anyone else. Authorization must then be at the level of individual tuples, which is not possible in the SQL standard for authorization.

Furthermore with the growth in the Web, database accesses come primarily from Web application server. The end users may not have individual user identifiers on the database as indeed there may only be a single user identifier in the database corresponding to all users of an application server.

The task of authorization then falls on the application server; the entire authorization scheme of SQL is bypassed. The benefit is that fine-grained authorizations such as those to individual tuples can be implemented by the application. The problems are these:

- The code for checking authorization becomes intermixed with the rest of the application code.
- Implementing authorization through application code, rather than specifying it declaratively in SQL, makes it hard to ensure the absence of loopholes. Because of an oversight, one of the application programs may not heck for authorization, allowing unauthorized users access to confidential data. Verifying that all application programs make all required authorization checks involves reading through all the application server code a formidable task in a larger system.

15.4 Encryption and Authentication

The various provisions that a database system may make for authorization may still not provide sufficient protection for highly sensitive data. In such cases data may be stored in encrypted form. It is not possible for encryption data to be read unless the reader knows how to decipher (decrypt) them. Encryption also forms the basis of good schemes for authenticating users to a database.

Encryption Techniques

There are a vast number of techniques for the encryption of data. Simple encryption techniques may not provide adequate security, since it may be easy for an unauthorized user to break the code. As an example of a weak encryption technique, consider the substitution of each character with the next character in the alphabet. Thus,

Perryridge

becomes

Qfsszsjehf

If an unauthorized user sees on “Qfsszsjehf” she probably has insufficient information to break the code. However, if the intruder sees a large number of encrypted branch names, she could use statistical data regarding the relative frequency of characters to guess what substitution is being made (for example, E is the most common letter in English text, followed by T, A, O, N, I and so on).

A good encryption technique has the following properties

- It is relatively simple for authorized users to encrypt and decrypt data
- It depends not on the secrecy of the algorithm, but rather on a parameter of the algorithm called the encryption key.
- Its encryption key is extremely difficult for an intruder to determine.

One approach the *Data Encryption Standard* (DES), issued in 1977 does both a substitution of characters and a rearrangement of their order on the basis of an encryption key. For this scheme to work the authorized users must be provided with the encryption key via a secure mechanism. This requirement is a major weakness since the scheme is no more than the security of the mechanism by which the encryption key is transmitted. The DES standard was reaffirmed in 1983, 1987 and again in 1993. However weakness in DES was recognized in 1993 as reaching a point where a new standard to be called the Advanced Encryption Standard (AES), needed to be selected. In 2000, the Rijndael algorithm (named for the inventors V. Tijmen and J. Daemen) was selected to be the AES. The Rijndael algorithm was chosen for its significantly stronger level of security and its relative ease of implementation on current computer systems as well as such devices as smart cards. Like the DES standard, the Rijndael algorithm is a shared key (or symmetric key) algorithm in which the authorized users share a key.

Public-key encryption is an alterative scheme that avoids some of the problems that we face with the DES. It is based on two keys; a public key and a private key. Each user U_i has a public key E_i and a private Key D_i. All public keys are published. They can be seen by anyone. Each private key is known to only the one user to whom the key belongs. If user U_1 wants to store encrypted data, U_1 encrypts them using public key E_1. Decryption requires the private key D_1.

Because the encryption key for each user is public, it is possible to exchange information securely by this scheme. If user U_1 wants to share data with U_2, U_1 encrypts the data using E_2 the public key of U_2. Since only user U_2 know how to decrypt the data, information is transferred securely.

For public key encryption to work there must be a scheme for encryption that can be made public without making it easy for people to figure out the scheme for decryption. In other words it must be hard to deduce the private key given the public key. Such a scheme does not exist and is based on these conditions:

- There is an efficient algorithm for testing whether or not a number is prime.
- No efficient algorithm is known for finding the prime factors of a number.

For purposes of this scheme data are treated as a collection of integers. We create a public key by computing the product of two large prime numbers: P_1 and P_2. The private key consists of the pair (P_1, P_2). The decryption algorithm cannot be used successfully if only the product P_1P_2 is known it needs the individual values P_1 and P_2. Since all that is published is the product P_1P_2, an unauthorized user would need to be able to factor P_1P_2 to steal data. By choosing P_1 and P_2 to be sufficiently large (over 100 digits) we can make the cost of factoring P_1P_2 prohibitively high (on the order of years of computation time on even the fastest computers).

The details of Public-key encryption by this scheme is secure, it is also computationally expensive. A hybrid scheme used for secure communication is as follows: DES keys are exchanged via a public-key-encryption scheme and DES encryption is used on the data transmitted subsequently.

Authentication

Authentication refers to the task of verifying the identity of a person/software connecting to a database. The simplest form of authentication consists of a secret password which must be presented when a connection is opened to a database.

Password based authentication is used widely by operating systems an well as databases. However the use of passwords has some drawbacks especially over a network. If an eavesdropper is able to "sniff" the data being sent over the networks, she may be able to find the password as it is being sent across the networks. Once the eavesdropper has a user and password, she can connect to the database pretending to be the legitimate user.

A more secure scheme involves a **challenge-response** system. The database system sends a challenge string to the user. The user encrypts the challenge string using a secret password as encryption key and then returns the result. The database system can verify the authenticity of the user by decrypting the string with the same secret password and checking the result with the original challenge string. This scheme ensures that no passwords travel cross the network.

Public key systems can be used for encryption in challenge-response systems. The database encrypts a challenge string using the user's public key and sends it to the user. The user decrypts the string using her private key, and returns the result to the database system. The database system then checks the response. This scheme has the added benefit of not storing the secret password in the database where it could potentially be seen by system administrators.

Another interesting application of public-key encryption is in **digital signature.** To verify authenticity of data, digital signatures play the electronic role of physical signatures on documents. The private key is used to sign data and the signed data can be made public. Anyone can verify them by the public key but no one could have generated the signed data without having the private key. Thus we can **authenticate** the data; that is we can verify that the data were indeed created by the person who claims to have created them.

Furthermore digital signatures also serve to ensure **non-repudiation**. That is in case the person who created the data later claims she did not create it (the electronic equivalent of claiming not to have signed the check) we can prove that, that person must have created the data (unless her private key was leaked to others).

15.5 Summary

Database security refers to protection from malicious access. For making database secure, we may assign a user several forms of authorization on parts of the database. For example, **Read authorization** allows reading, but not modification, of data. **Insert authorization** allows inserting of new data, but not modification of existing data. **Update authorization** allows modification, but not deleting, of data. **Delete authorization** allows deleting of data. In addition to these forms of authorization for access of data, we may grant a user authorization to modify the database schema - Index authorization, Resource authorization, Alteration authorization, Drop authorization. Many secure database applications require an audit trail be maintained. An audit trail is a log of all changes (inserts/ deletes/ updates) to the database, along with information such as which user performed the changes and when the change was performed. The various provisions that a database system may make for authorization may still not provide sufficient protection for highly sensitive data. In such cases data may be stored in encrypted form. It is not possible for encryption data to be read unless the reader knows how to decipher (decrypt) them. Encryption also forms the basis of good schemes for authenticating users to a database. Authentication refers to the task of verifying the identity of a person/software connecting to a database. The simplest form of authentication consists of a secret password which must be presented when a connection is opened to a database.

15.6 Questionnaires:

1. What to you understand by Database security?
2. What are the various ways in which database can be made secure?
3. What you mean by Encryption? Explain various data encryption techniques.
4. What do you mean by Authentication?
5. What do you mean by authorization?

PGDCA **PAPER : DBMS**
LESSON NO. : 16

SQL

Introduction
SQL DML and DDL
Basic Data Types
SQL Commands
DATA CONSTRINTS
Arithmetic Operators
Logical Operators
Range Searching (BETWEEN)
Pattern Matching (LIKE, IN, NOT IN)
SQL String Functions
Summary

16.1 Introduction

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL is an ANSI (American National Standards Institute) standard

SQL is a declarative (non-procedural)language. SQL is (usually) not case-sensitive, but we'll write SQL keywords in upper case for emphasis.

Some database systems require a semicolon at the end of each SQL statement.

A table is database object that holds user data. Each column of the table will have specified data type bound to it. Oracle ensures that only data, which is identical to the datatype of the column, will be stored within the column.

16.2 SQL DML and DDL

SQL can be divided into two parts:

The Data Definition Language (DDL) and the Data Manipulation Language (DML).

Data Definition Language (DDL)

It is a set of SQL commands used to create, modify and delete database structure but not data. It also define indexes (keys), specify links between tables, and impose constraints between tables. DDL commands are auto COMMIT.

The most important DDL statements in SQL are:

- CREATE TABLE - creates a new table
- ALTER TABLE - modifies a table
- TRUNCATE TABLE- deletes all records from a table •

DROP TABLE - deletes a table

Data Manipulation Language (DML)

It is the area of SQL that allows changing data within the database. The query and update commands form the DML part of SQL:

- INSERT - inserts new data into a database
- SELECT - extracts data from a database
- UPDATE - updates data in a database
- DELETE - deletes data from a database

Data Control Language (DCL)

It is the component of SQL statement that control access to data and to the database. Occasionally DCL statements are grouped with DML Statements.

- COMMIT –Save work done.
- SAVEPOINT – Identify a point in a transaction to which you can later rollback. •

ROLLBACK – Restore database to original since the last COMMIT.

- GRANT – gives user's access privileges to database.
- REVOKE – withdraw access privileges given with GRANT command.

16.3 Basic Data Types

Data Type	Description
CHAR(size)	This data type is used to store character strings values of fixed length. The size in brackets determines the number of characters the cell can hold. The maximum number of character(ie the size) this data type can hold is 255 characters. The data held is right padded with spaces to whatever length specified.
VARCHAR(size) / VARCHAR2(size)	This data type is used to store variable length alphanumeric data. It is more flexible form of CHAR data type. VARCHAR can hold 1 to 255 characters. VARCHAR is usually a wiser choice than CHAR, due to its variable length format characteristic. But, keep in mind, that CHAR is much faster than VARCHAR, sometimes up to 50%.
DATE	This data type is used to represent data and time. The standard format is DD-MMM-YY. Date Time stores date in the 24-hour format. By default, the time in a date field is 12:00:00am.
NUMBER(P,S)	The NUMBER data type is used to store numbers(fixed or floating point). Number of virtually any magnitude maybe stored up to 38 digits of precision. The Precision(P), determines the maximum length of the data, whereas the scale(S), determine the number of places to the right of the decimal. Example: Number(5,2) is a number that has 3 digits before the decimal and 2 digits after the decimal.
LONG	This data type is used to store variable length character strings containing up to 2GB. LONG data can be used to store arrays of binary data in ASCII format. Only one LONG value can be defined per table.
RAW / LONG RAW	The RAW / LONG RAW data types are used to store binary data, such as digitized picture or image. RAW data type can have maximum length of 255 bytes. LONG RAW data type can contain up to 2GB.

16.4 SQL Commands

The Create Table command

The CREATE TABLE command defines each column of the table uniquely. Each column has a minimum of three attributes, name, datatype and size(i.e column width).each table column definition is a single clause in the create table syntax. Each table column definition is separated from the other by a comma. Finally, the SQL statement is terminated with a semi colon.

Rules for Creating Tables

- A name can have maximum upto 30 characters.
- Alphabets from A-Z, a-z and numbers from 0-9 are allowed.
- A name should begin with an alphabet.
- The use of the special character like _(underscore) is allowed.
- SQL reserved words not allowed. For example: create, select, alter.

Syntax:

```
CREATE TABLE <tablename>
(<columnName1>  <Datatype>(<size>),
 <columnName2> <Datatype>(<size>), ....... );
```

Example:

```
CREATE TABLE gktab
(Regno NUMBER(3), Name
VARCHAR(20),
 Gender CHAR,
 Dob DATE,
 Course CHAR(5));
```

Inserting Data into Tables

Once a table is created, the most natural thing to do is load this table with data to be manipulated later.

When inserting a single row of data into the table, the insert operation:

- ✓ Creates a new row(empty) in the database table.
- ✓ Loads the values passed(by the SQL insert) into the columns specified.

Syntax:

INSERT INTO <tablename>(<columnname1>, <columnname2>, ..)

Values(<expression1>,<expression2>...);

Example:

```
INSERT INTO gktab(regno,name,gender,dob,course) VALUES(101,'Varsh G
        Kalyan','F','20-Sep-1985','BCA');
```

Or you can use the below method to insert the data into table.

```
INSERT INTO gktab VALUES(102,'Mohith G Kalyan','M','20-Aug-1980','BBM'); INSERT INTO
gktab VALUES(106,'Nisarga','F','15-Jul-1983','BCom');
INSERT INTO gktab VALUES(105,'Eenchara','F','04-Dec-1985','BCA'); INSERT
INTO gktab VALUES(103,'Ravi K','M','29-Mar-1989','BCom'); INSERT INTO
gktab VALUES(104,'Roopa','F','17-Jan-1984','BBM');
```

Whenever you work on the data which has data types like CHAR,VARCHAR/VARCHAR2, DATE should be used between single quote(')

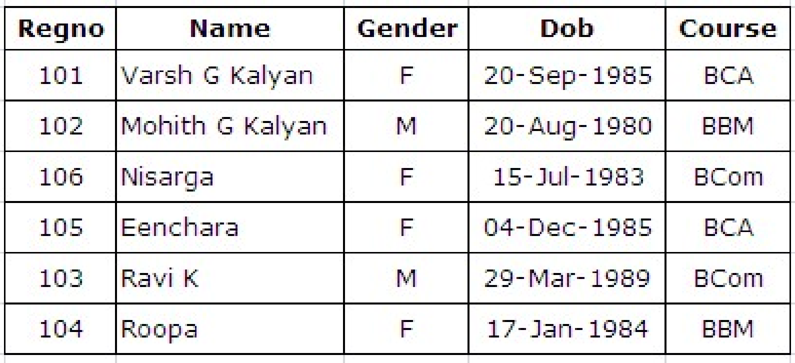
Viewing Data in the Tables

Once data has been inserted into a table, the next most logical operation would be to view what has been inserted. The **SELECT** SQL verb is used to achieve this. The SELECT command is used to retrieve rows selected from one or more tables.

All Rows and All Columns

```
SELECT * FROM <tablename>
  SELECT * FROM gktab;
```

It shows all rows and column data in the table



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
106	Nisarga	F	15-Jul-1983	BCom
105	Eenchara	F	04-Dec-1985	BCA
103	Ravi K	M	29-Mar-1989	BCom
104	Roopa	F	17-Jan-1984	BBM

Filtering Table Data

While viewing data from a table it is rare that all the data from the table will be required each time. Hence, SQL provides a method of filtering table data that is not required.

The ways of filtering table data are:

- Selected columns and all rows
- Selected rows and all columns
- Selected columns and selected rows

Selected Columns and All Rows

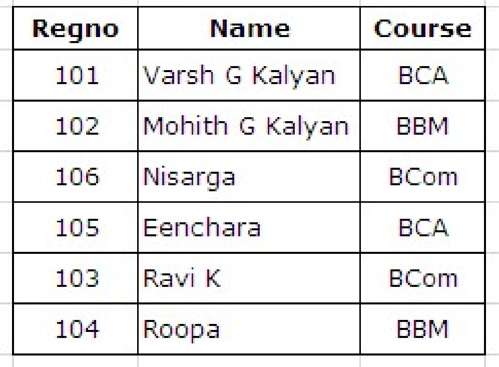
The retrieval of specific columns from a table can be done as shown below. Syntax

SELECT <columnname1>, <Columnname2> FROM <tablename>

Example

Show only Regno, Name and Course from gktab.

SELECT Regno, Name, Course FROM gktab;



Regno	Name	Course
101	Varsh G Kalyan	BCA
102	Mohith G Kalyan	BBM
106	Nisarga	BCom
105	Eenchara	BCA
103	Ravi K	BCom
104	Roopa	BBM

Selected Rows and All Columns

The WHERE clause is used to extract only those records that fulfill a specified criterion.

When a WHERE clause is added to the SQL query, the Oracle engine compares each record in the table with condition specified in the WHERE clause. The Oracle engine displays only those records that satisfy the specified condition.

Syntax

SELECT * FROM <tablename> WHERE <condition>;

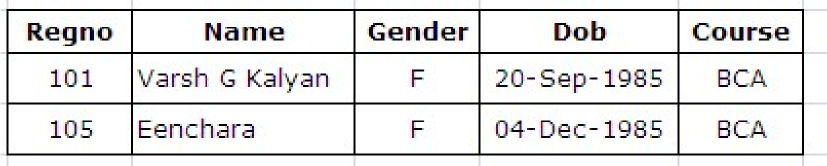
Here, ***<condition>*** is always quantified as ***<columnname=value>***

When specifying a condition in the **WHERE** clause all standard operators such as logical, arithmetic and so on, can be used.

Example-1:

Display all the students from BCA.

SELECT * FROM gktab WHERE Course='BCA';

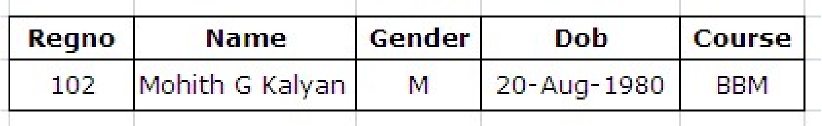


Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
105	Eenchara	F	04-Dec-1985	BCA

Example-2:

Display the student whose regno is 102. SELECT

* FROM gktab WHERE Regno=102;



Regno	Name	Gender	Dob	Course
102	Mohith G Kalyan	M	20-Aug-1980	BBM

Selected Columns and Selected Rows

To view a specific set of rows and columns from a table

When a WHERE clause is added to the SQL query, the Oracle engine compares each record in the table with condition specified in the WHERE clause. The Oracle engine displays only those records that satisfy the specified condition.

Syntax

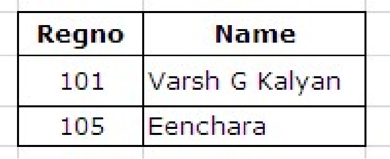
SELECT <columnname1>, <Columnname2> FROM <tablename> WHERE

<condition>;

Example-1:

List the student's Regno, Name for the Course BCA. SELECT

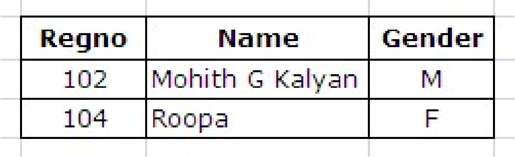
Regno, Name FROM gktab WHERE Course='BCA';



Regno	Name
101	Varsh G Kalyan
105	Eenchara

Example-2:

List the student's Regno, Name, Gender for the Course BBM. SELECT Regno, Name, Gender FROM gktab WHERE Course='BBM';



Regno	Name	Gender
102	Mohith G Kalyan	M
104	Roopa	F

Eliminating Duplicate Rows when using a SELECT statement

A table could hold duplicate rows. In such a case, to view only unique rows the **DISTINCT** clause can be used.

The **DISTINCT** clause allows removing duplicates from the result set. The **DISTINCT** clause can only be used with **SELECT** statements.

The **DISTINCT** clause scans through the values of the column/s specified and displays only unique values from amongst them.

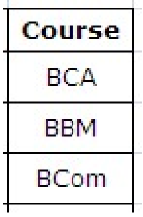
Syntax

SELECT DISTINCT <columnname1>, <Columnname2> FROM <Tablename>;

Example:

Show different courses from gktab

SELECT DISTINCT Course from gktab;



Course
BCA
BBM
BCom

Sorting Data in a Table

Oracle allows data from a table to be viewed in a sorted order. The rows retrieved from the table will be sorted in either ***ascending*** or ***descending*** order depending on the condition specified in the ***SELECT*** sentence.

Syntax

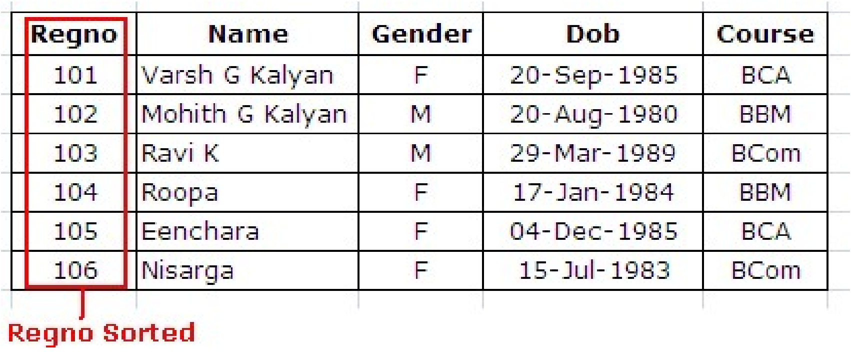
SELECT * FROM <tablename>

ORDER BY <Columnname1>,<Columnname2> <[Sort Order]>;

The ***ORDER BY*** clause sorts the result set based on the column specified. The ***ORDER BY*** clause can only be used in ***SELECT*** statements. The Oracle engine sorts in ***ascending order by default***

Example-1:

Show details of students according to Regno. SELECT *
FROM gktab ORDER BY Regno;



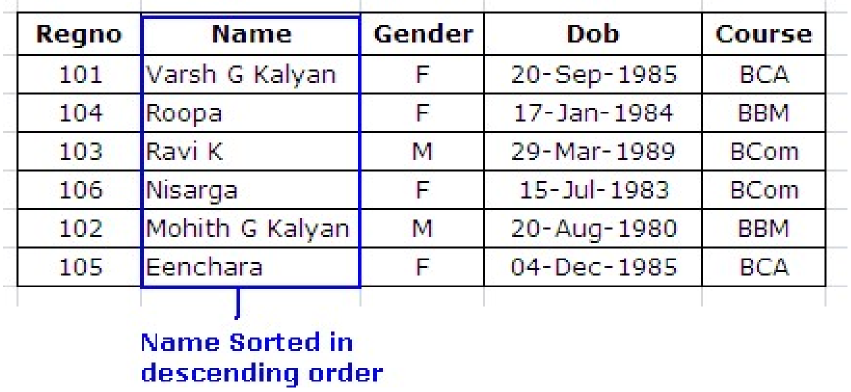
Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
103	Ravi K	M	29-Mar-1989	BCom
104	Roopa	F	17-Jan-1984	BBM
105	Eenchara	F	04-Dec-1985	BCA
106	Nisarga	F	15-Jul-1983	BCom

Regno Sorted

Example-2:

Show the details of students names in descending order.

SELECT * FROM gktab ORDER BY Name DESC;



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
104	Roopa	F	17-Jan-1984	BBM
103	Ravi K	M	29-Mar-1989	BCom
106	Nisarga	F	15-Jul-1983	BCom
102	Mohith G Kalyan	M	20-Aug-1980	BBM
105	Eenchara	F	04-Dec-1985	BCA

Name Sorted in
descending order

DELETE Operations

The ***DELETE*** command deletes rows from the table that satisfies the condition provided by its ***WHERE*** clause, and returns the number of records deleted.

The verb ***DELETE*** in SQL is used to remove either

- Specific row(s) from a table

  OR
- All the rows from a table

Removal of Specific Row(s)

Syntax:

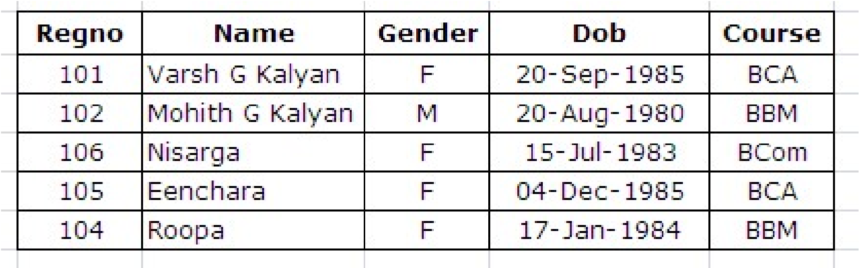
DELETE FROM tablename WHERE Condition;

Example:

DELETE FROM gktab WHERE Regno=103; 1

rows deleted

SELECT * FROM gktab;



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
106	Nisarga	F	15-Jul-1983	BCom
105	Eenchara	F	04-Dec-1985	BCA
104	Roopa	F	17-Jan-1984	BBM

In the above table, the Regno 103 is deleted from the table

Remove of ALL Rows

Syntax

DELETE FROM tablename;

Example

DELETE FROM gktab;

6 rows deleted

SELECT * FROM gktab;

no rows selected

Once the table is deleted, use Rollback to undo the above operations.

UPDATING THE CONTENTS OF A TABLE

The **UPDATE** Command is used to change or modify data values in a table. The verb update in SQL is used to either updates:

- ALL the rows from a table.
  OR
- A select set of rows from a table.

Updating all rows

The **UPDATE** statement updates columns in the existing table's rows with a new values. The **SET** clause indicates which column data should be modified and the new values that they should hold. The WHERE clause, if given, specifies which rows should be updated. Otherwise, all table rows are updated.

Syntax:

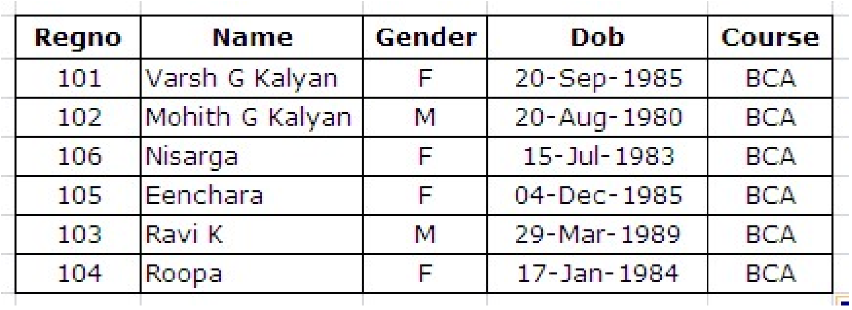
```
UPDATE tablename
 SET columnname1=expression1, columnname2=expression2;
```

Example: update the gktab table by changing its course to BCA.

```
UPDATE gktab SET  course='BCA';
```

6 rows updated

```
SELECT * FROM gktab;
```



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BCA
106	Nisarga	F	15-Jul-1983	BCA
105	Eenchara	F	04-Dec-1985	BCA
103	Ravi K	M	29-Mar-1989	BCA
104	Roopa	F	17-Jan-1984	BCA

In the above table, the course is changed to BCA for all the rows in the table.

Updating Records Conditionally

If you want to update a specific set of rows in table, then ***WHERE*** clause is used.

Syntax:

```
        UPDATE tablename
          SET Columnname1=Expression1, Columnname2=Expression2
          WHERE Condition;
```

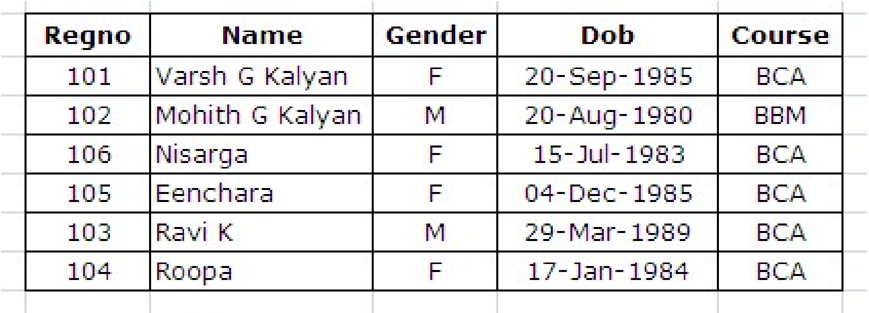
Example:

Update gktab table by changing the course BCA to BBM for Regno 102.

UPDATE gktab SET Course='BBM' WHERE Regno=102;

1 rows updated

SELECT * FROM gktab;



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
106	Nisarga	F	15-Jul-1983	BCA
105	Eenchara	F	04-Dec-1985	BCA
103	Ravi K	M	29-Mar-1989	BCA
104	Roopa	F	17-Jan-1984	BCA

MODIFYING THE STRUCTURE OF TABLES

The structure of a table can be modified by using the **ALTER TABLE** command. ***ALTER TABLE*** allows changing the structure of an existing table. With ***ALTER TABLE*** if is possible to ***add*** or ***delete*** columns, ***create*** or ***destroy*** indexes, ***change the data type*** of existing columns, or ***rename columns*** or the ***table*** itself.

ALTER TABLE works by making a temporary copy of the original table. The alteration is performed on the copy, then the original table is deleted and the new one is renamed. While ***ALTER TABLE*** is executing, the original table is still readable by the users of ORACLE.

Restrictions on the ALTER TABLE

The following task **cannot** be performed when using the ***ALTER TABLE*** Clause:

- Change the name of the table.
- Change the name of the Column.
- Decrease the size of a column if table data exists.

ALTER TABLE Command can perform

- ***Adding New Columns.***
- ***Dropping A Column from a Table.***
- ***Modifying Existing Columns.***

Adding New Columns

Syntax:

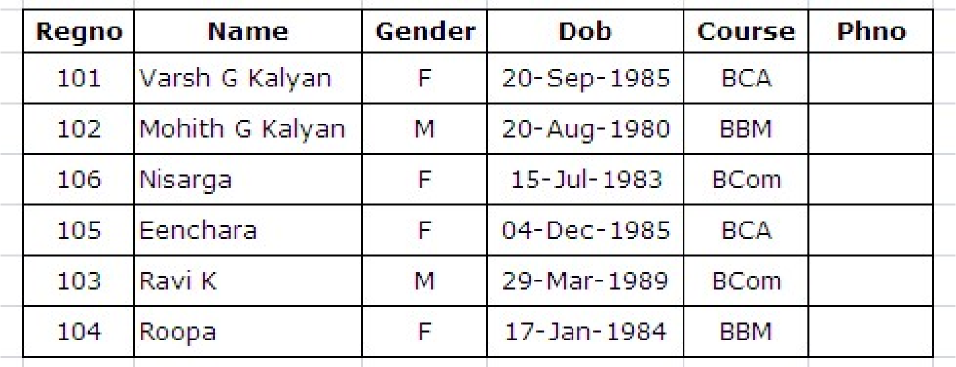
ALTER TABLE tablename
ADD(NewColumnname1   Datatype(size),
NewColumnname2   Datatype(size).....);

Example: Enter a new filed Phno to gktab.

ALTER TABLE gktab ***ADD(***Phno number(10)***);*** The

table is altered with new column Phno

Select * from gktab;



Regno	Name	Gender	Dob	Course	Phno
101	Varsh G Kalyan	F	20-Sep-1985	BCA	
102	Mohith G Kalyan	M	20-Aug-1980	BBM	
106	Nisarga	F	15-Jul-1983	BCom	
105	Eenchara	F	04-Dec-1985	BCA	
103	Ravi K	M	29-Mar-1989	BCom	
104	Roopa	F	17-Jan-1984	BBM	

You can also use ***DESC*** gktab, to see the new column added to table.

Dropping A Column from a Table.

Syntax:

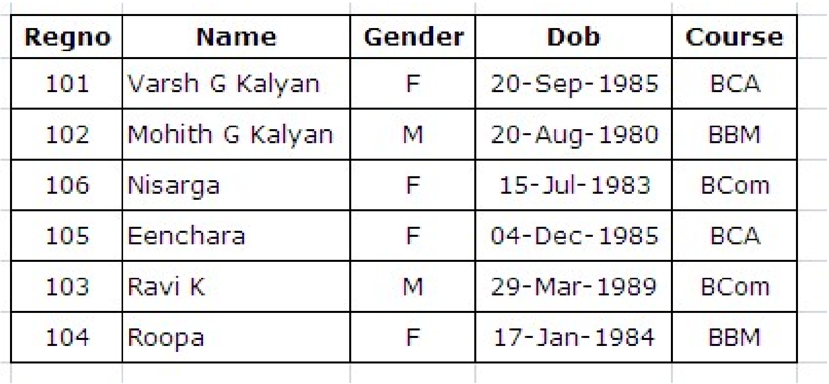
ALTER TABLE tablename DROP COLUMN Columnname;

Example: Drop the column Phno from gktab.

ALTER TABLE gktab ***DROP COLUMN*** Phno***;***

The table is altered, the column Phno is removed from the table.

Select * from gktab;



Regno	Name	Gender	Dob	Course
101	Varsh G Kalyan	F	20-Sep-1985	BCA
102	Mohith G Kalyan	M	20-Aug-1980	BBM
106	Nisarga	F	15-Jul-1983	BCom
105	Eenchara	F	04-Dec-1985	BCA
103	Ravi K	M	29-Mar-1989	BCom
104	Roopa	F	17-Jan-1984	BBM

You can also use ***DESC*** gktab, to see the column removed from the table.

Modifying Existing Columns.

Syntax:

```
ALTER TABLE tablename
        MODIFY(Columnname  Newdatatype(Newsize));
```

Example:

```
ALTER TABLE gktab MODIFY(Name VARCHAR(25));
```

The table altered with new size value 25.

DESC gktab;

RENAMING TABLES

Oracle allows renaming of tables. The rename operation is done atomically, which means that no other thread can access any of the tables while the rename process is running.

Syntax

```
RENAME tablename TO  newtablename;
```

TRUNCATING TABLES

TRUNCATE command deletes the rows in the table permanently.

Syntax:

```
TRUNCATE TABLE tablename;
```

The number of deleted rows are not returned. Truncate operations drop and re- create the table, which is much faster than deleting rows one by one.

Example:

TRUNCATE TABLE gktab***;***

Table truncated i.e., all the rows are deleted permanently.

DESTROYING TABLES

Sometimes tables within a particular database become obsolete and need to be discarded. In such situation using the **DROP TABLE** statement with table name can destroy a specific table.

Syntax:

DROP TABLE tablename***;***

Example:

DROP TABLE gktab***;***

If a table is dropped all the records held within and the structure of the table is lost and cannot be recovered.

COMMIT and ROLLBACK

Commit

Commit command is used to permanently save any transaction into database.

```
SQL> commit;
```

Rollback

Rollback is used to undo the changes made by any command but only before a commit is done. We can't Rollback data which has been committed in the database with the help of the commit keyword or DDL Commands, because DDL commands are auto commit commands.

```
SQL> Rollback;
```

Difference between DELETE and DROP.

The **DELETE** command is used to remove rows from a table. After performing a DELETE operation you need to COMMIT or ROLLBACK the transaction to make the change permanent or to undo it.

The **DROP** command removes a table from the database. All the tables' rows, indexes and privileges will also be removed. The operation cannot be rolled back.

Difference between DELETE and TRUNCATE.

The **DELETE** command is used to remove rows from a table. After performing a DELETE operation you need to COMMIT or ROLLBACK the transaction to make the change permanent or to undo it.

TRUNCATE removes all rows from a table. The operation cannot be rolled back.

Difference between CHAR and VARCHAR. CHAR

1. Used to store fixed length data.
2. The maximum characters the data type can hold is 255 characters.
3. It's 50% faster than VARCHAR.
4. Uses static memory allocation.

VARCHAR

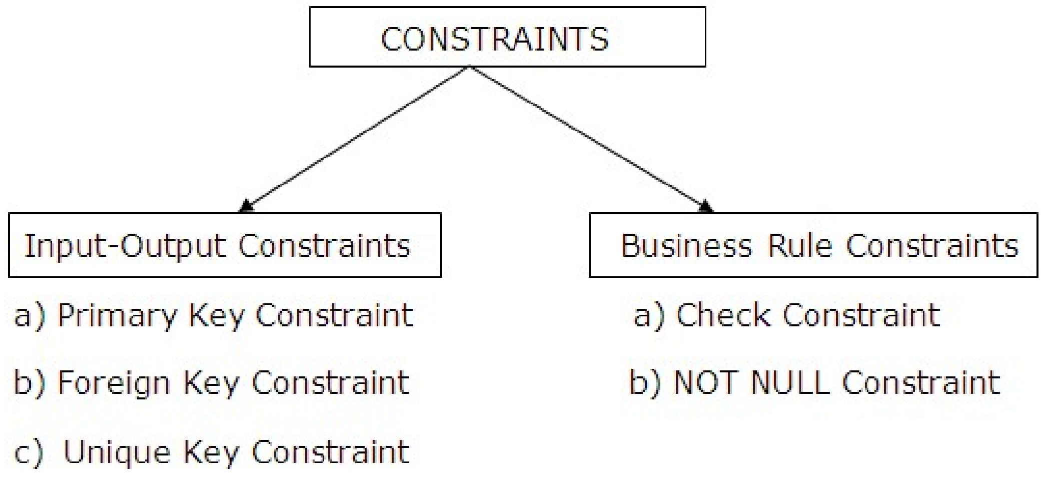
1. Used to store variable length data.
2. The maximum characters the data type can hold is up to 4000 characters.
3. It's slower than CHAR.
4. Uses dynamic memory allocation.

16.5 DATA CONSTRINTS

Oracle permits data constraints to be attached to table column via SQL syntax that checks data for integrity prior storage. Once data constraints are part of a table column construct, the oracle database engine checks the data being entered into a table column against the data constraints. If the data passes this check, it is stored in the table column, else the data is rejected. Even if a single column of the record being entered into the table fails a constraint, the entire record is rejected and not stored in the table.

Both **CREATE TABLE** and **ALTER TABLE** SQL verbs can be used to write SQL sentences that attach constraints to a table column.
The constraints are a keyword. The constraint is rules that restrict the values for one or more columns in a table. The Oracle Server uses constraints to prevent invalid data entry into tables. The constraints store the validate data and without constraints we can just store invalid data. The constraints are an important part of the table.

Types of DATA CONSTRAINTS



CONSTRAINTS

Input-Output Constraints

a) Primary Key Constraint

b) Foreign Key Constraint

c) Unique Key Constraint

Business Rule Constraints

a) Check Constraint

b) NOT NULL Constraint

Primary Key Constraint

A primary key can consist of one or more columns on a table. Primary key constraints define a column or series of columns that uniquely identify a given row in a table. Defining a primary key on a table is optional and you can only define a single primary key on a table. A primary key constraint can consist of one or many columns (up to 32). When multiple columns are used as a primary key, they are called a composite key. Any column that is defined as a primary key column is automatically set with a NOT NULL status. The Primary key constraint can be applied at column level and table level.

Foreign Key Constraint

A foreign key constraint is used to enforce a relationship between two tables. A foreign key is a column (or a group of columns) whose values are derived from the Primary key or unique key of some other table.
The table in which the foreign key is defined is called a Foreign table or Detail table. The table that defines primary key or unique key and is referenced by the foreign key is called Primary table or Master table.
The master table can be referenced in the foreign key definition by using the clause **REFERENCES Tablename.ColumnName** when defining the foreign key, column attributes, in the detail table. The foreign key constraint can be applied at column level and table level.

Unique Key Constraint

Unique key will not allow duplicate values. A table can have more than one Unique key. A unique constraint defines a column, or series of columns, that must be unique in value. The UNIQUE constraint can be applied at column level and table level.

CHECK Constraint

Business Rule validation can be applied to a table column by using **CHECk** constraint. **CHECK** constraints must be specified as a logical expression that evaluates either to **TRUE** or **FALSE**.

The **CHECK** constraint ensures that all values in a column satisfy certain conditions. Once defined, the database will only insert a new row or update an existing row if the new value satisfies the **CHECK** constraint. The **CHECK** constraint is used to ensure data quality.

A CHECK constraint takes substantially longer to execute as compared to NOT NULL, PRIMARY KEY, FOREIGN KEY or UNIQUE. The CHECK constraint can be applied at column level and table level.

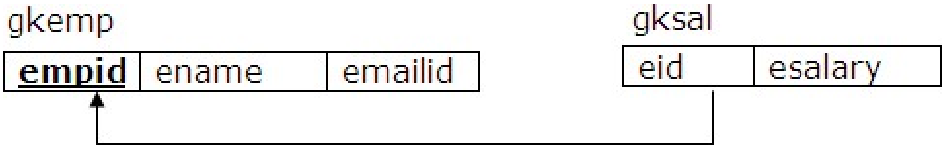
NOT NULL Constraint

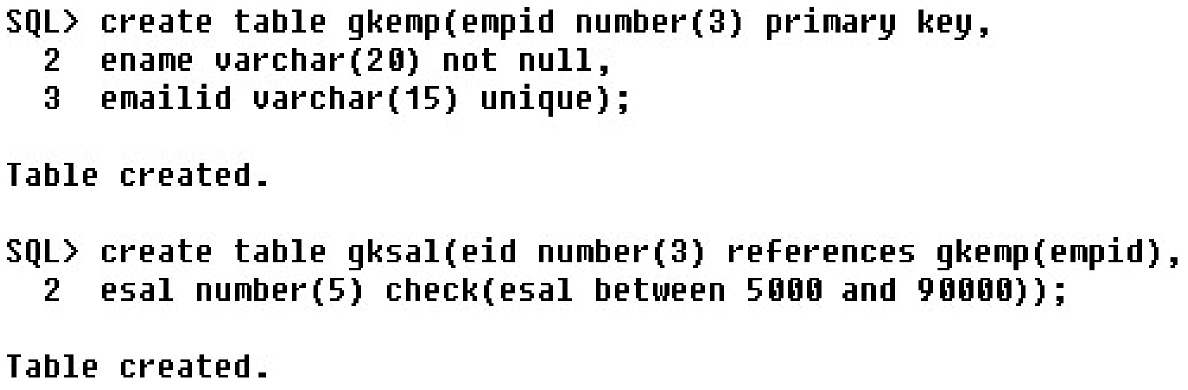
The NOT NULL column constraint ensures that a table column cannot be left empty.

When a column is defined as not null, then that column becomes a mandatory column. The NOT NULL constraint can only be applied at column level.

Example on Constraints

Consider the Table shown below





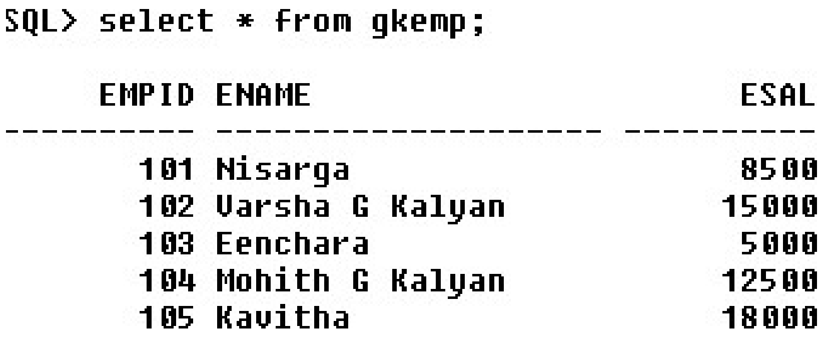
```
SQL> create table gkemp(empid number(3) primary key,
  2  ename varchar(20) not null,
  3  emailid varchar(15) unique);

Table created.

SQL> create table gksal(eid number(3) references gkemp(empid),
  2  esal number(5) check(esal between 5000 and 90000));

Table created.
```

16.6 Arithmetic Operators

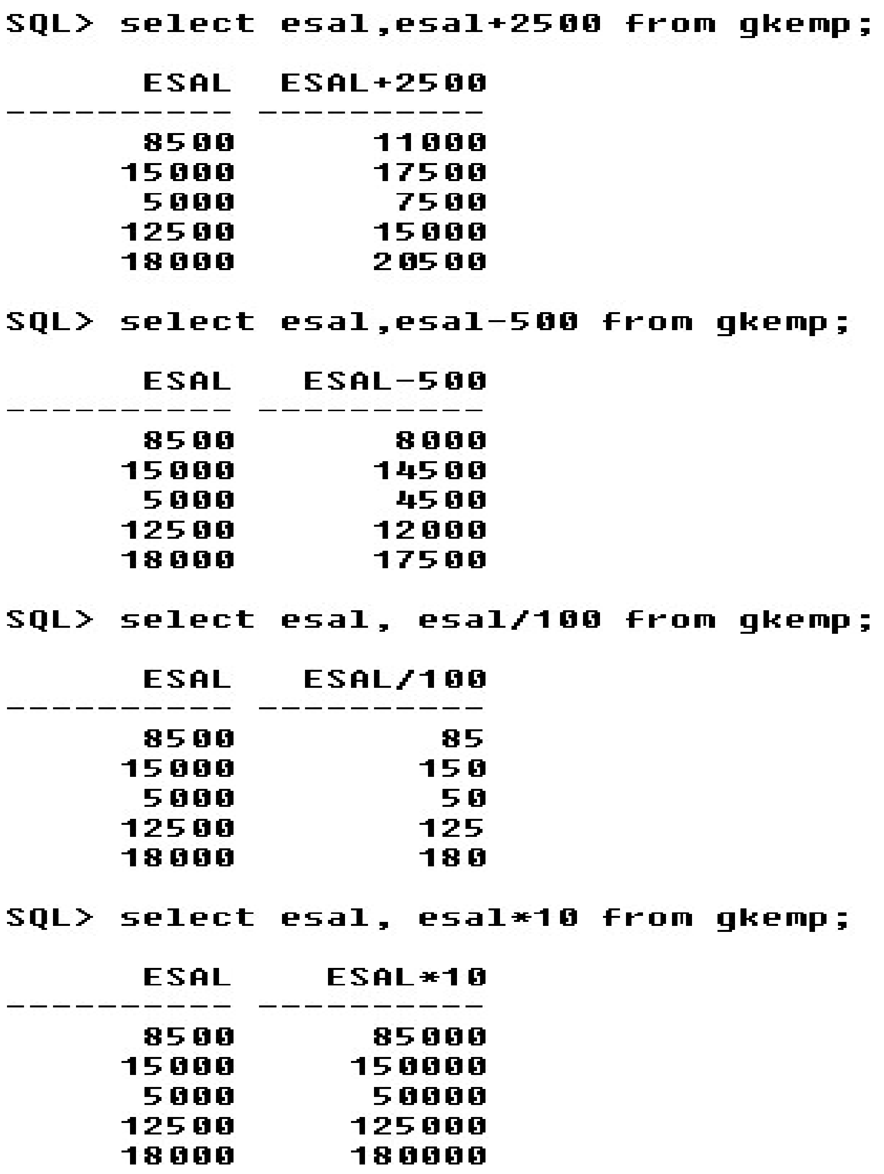
Oracle allows arithmetic operators to be used while viewing records from a table or while performing data manipulation operations such as insert, updated and delete. These are:

+	Addition
-	Subtraction
/	Division
*	Multiplication
()	Enclosed Operations

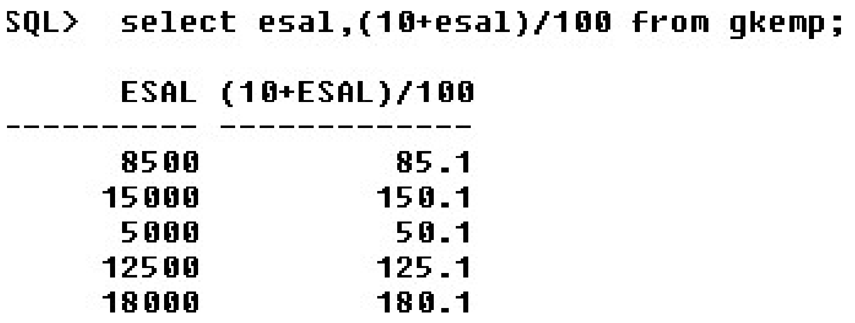
Consider the below employee table(gkemp)



```
SQL> select * from gkemp;

     EMPID ENAME                      ESAL
---------- -------------------- ----------
       101 Nisarga                    8500
       102 Varsha G Kalyan           15000
       103 Eenchara                   5000
       104 Mohith G Kalyan           12500
       105 Kavitha                   18000
```



```
SQL> select esal,esal+2500 from gkemp;

      ESAL  ESAL+2500
---------- ----------
      8500      11000
     15000      17500
      5000       7500
     12500      15000
     18000      20500

SQL> select esal,esal-500 from gkemp;

      ESAL   ESAL-500
---------- ----------
      8500       8000
     15000      14500
      5000       4500
     12500      12000
     18000      17500

SQL> select esal, esal/100 from gkemp;

      ESAL   ESAL/100
---------- ----------
      8500         85
     15000        150
      5000         50
     12500        125
     18000        180

SQL> select esal, esal*10 from gkemp;

      ESAL    ESAL*10
---------- ----------
      8500      85000
     15000     150000
      5000      50000
     12500     125000
     18000     180000
```



```
SQL>  select esal,(10+esal)/100 from gkemp;

      ESAL (10+ESAL)/100
---------- -------------
      8500          85.1
     15000         150.1
      5000          50.1
     12500         125.1
     18000         180.1
```

16.7 Logical Operators

Logical operators that can be used in SQL sentence are:

AND Operators
OR Operators
NOT Operators

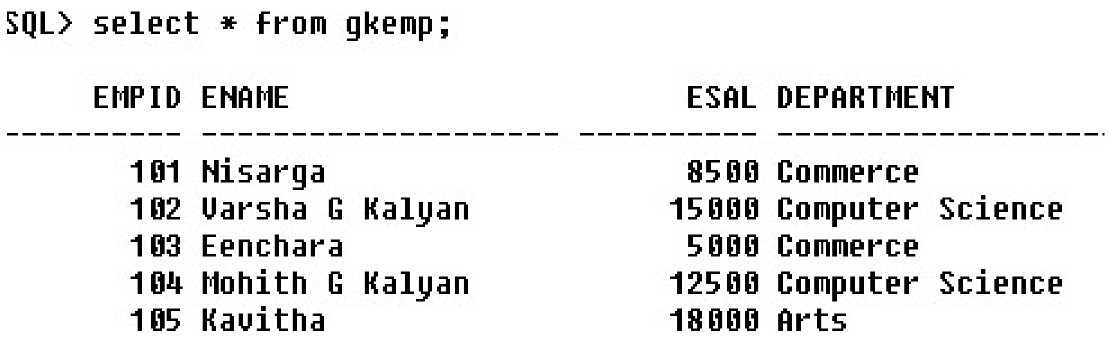
Operators ***Description***

OR :-For the row to be selected at least one of the conditions must be true.

AND :-For a row to be selected all the specified conditions must be true.

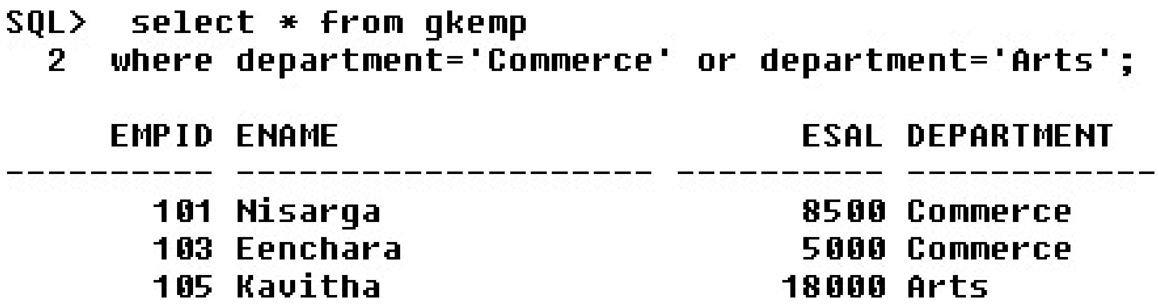
NOT :-For a row to be selected the specified condition must be false.

Consider the below employee table(gkemp)



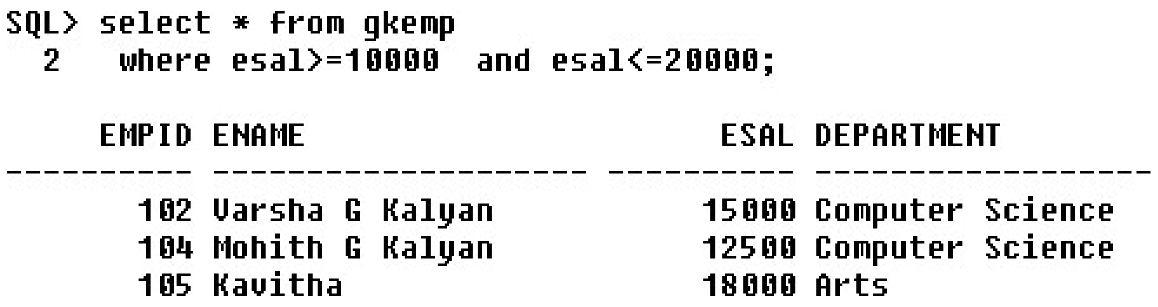
```
SQL> select * from gkemp;

     EMPID ENAME                      ESAL DEPARTMENT
---------- -------------------- ---------- ------------------
       101 Nisarga                    8500 Commerce
       102 Varsha G Kalyan           15000 Computer Science
       103 Eenchara                   5000 Commerce
       104 Mohith G Kalyan           12500 Computer Science
       105 Kavitha                   18000 Arts
```

For example: if you want to find the names of employees who are working either in Commerce or Arts department, the query would be like,



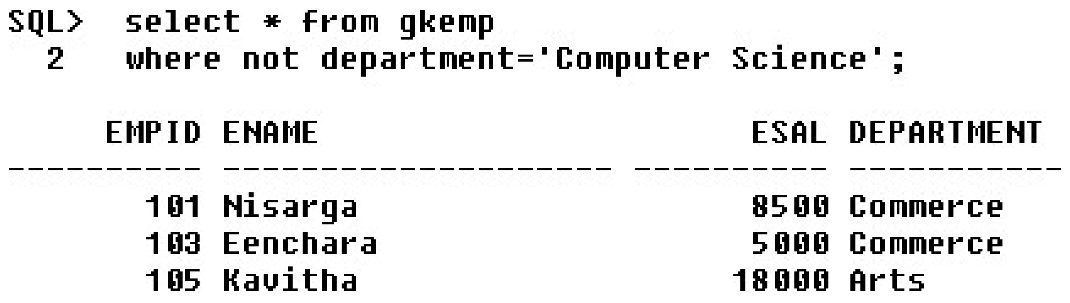
```
SQL>  select * from gkemp
  2  where department='Commerce' or department='Arts';

     EMPID ENAME                      ESAL DEPARTMENT
---------- -------------------- ---------- ------------
       101 Nisarga                    8500 Commerce
       103 Eenchara                   5000 Commerce
       105 Kavitha                   18000 Arts
```

For example: To find the names of the employee whose salary between10000 to 20000, the query would be like,



```
SQL> select * from gkemp
  2   where esal>=10000  and esal<=20000;

     EMPID ENAME                                ESAL DEPARTMENT
---------- -------------------- ---------- ------------------
       102 Varsha G Kalyan              15000 Computer Science
       104 Mohith G Kalyan              12500 Computer Science
       105 Kavitha                      18000 Arts
```

For example: If you want to find out the names of the employee who do not belong to computer science department, the query would be like,



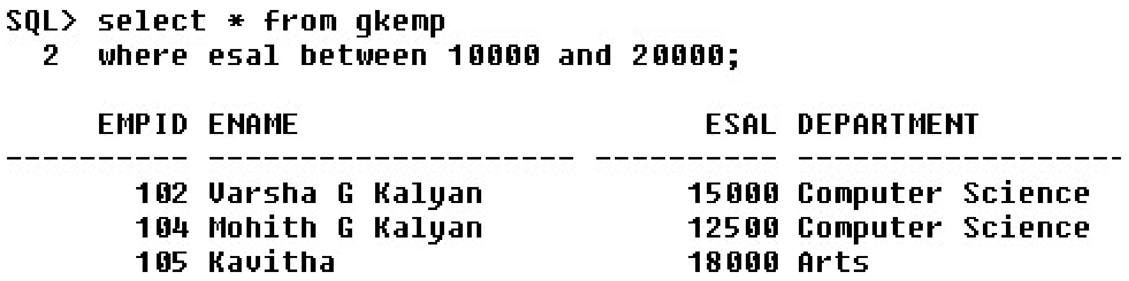
```
SQL>  select * from gkemp
  2   where not department='Computer Science';

     EMPID ENAME                                ESAL DEPARTMENT
---------- -------------------- ---------- -----------
       101 Nisarga                       8500 Commerce
       103 Eenchara                      5000 Commerce
       105 Kavitha                      18000 Arts
```

16.8 Range Searching (BETWEEN)

In order to select the data that is within a range of values, the **BETWEEN** operator is used. The BETWEEN operator allows the selection of rows that contain values within a specified lower and upper limit. The range coded after the word BETWEEN is inclusive.

The lower value must be coded first. The two values in between the range must be linked with the keyword **AND**. The BETWEEN operator can be used with both character and numeric data types. However, the data types cannot be mixed.

For example: Find the names of the employee whose salary between10000 and 20000, the query would be like,



```
SQL> select * from gkemp
  2  where esal between 10000 and 20000;

     EMPID ENAME                                ESAL DEPARTMENT
---------- -------------------- ---------- ------------------
       102 Varsha G Kalyan              15000 Computer Science
       104 Mohith G Kalyan              12500 Computer Science
       105 Kavitha                      18000 Arts
```

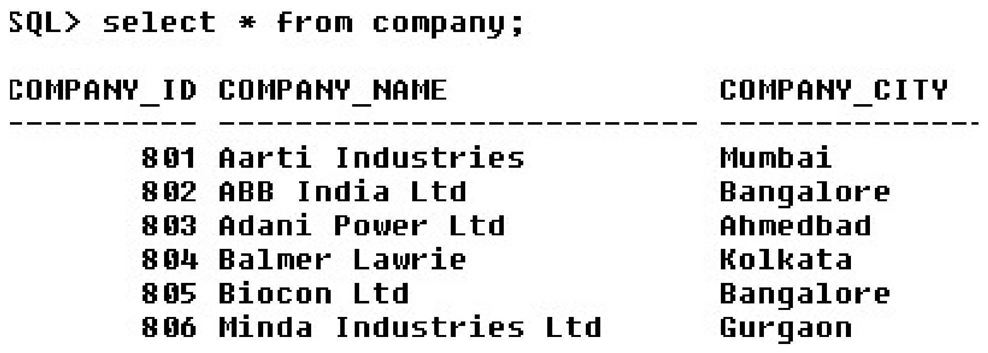
16.9 Pattern Matching (LIKE, IN, NOT IN)

The **LIKE** predicate allows comparison of one string value with another string value, which is not identical. this is achieved by using wild characters. Two wild characters that are available are:

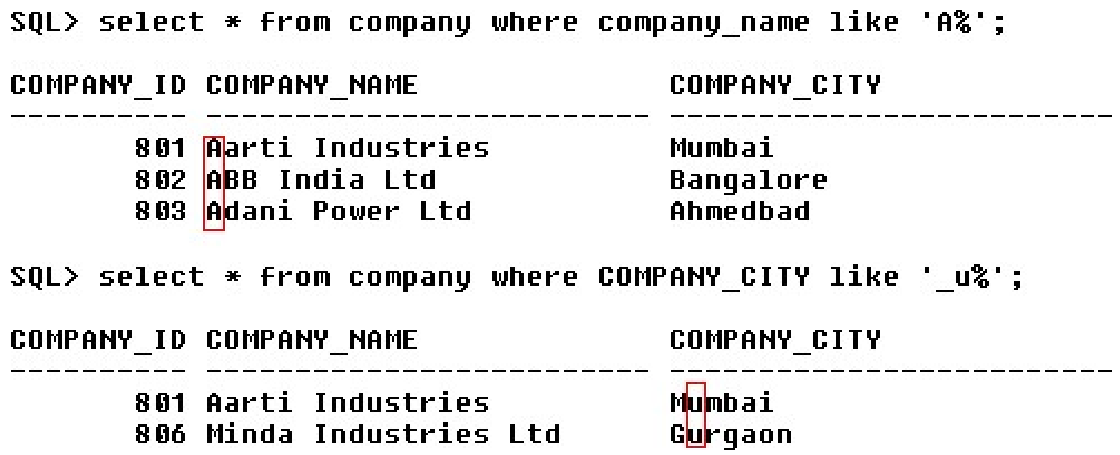
For character data types:

% allows to match any string of any length (including zero length).

_ allows to match on a single character.



```
SQL> select * from company;

COMPANY_ID COMPANY_NAME                   COMPANY_CITY
---------- ------------------------------ --------------
       801 Aarti Industries               Mumbai
       802 ABB India Ltd                  Bangalore
       803 Adani Power Ltd                Ahmedbad
       804 Balmer Lawrie                  Kolkata
       805 Biocon Ltd                     Bangalore
       806 Minda Industries Ltd           Gurgaon
```

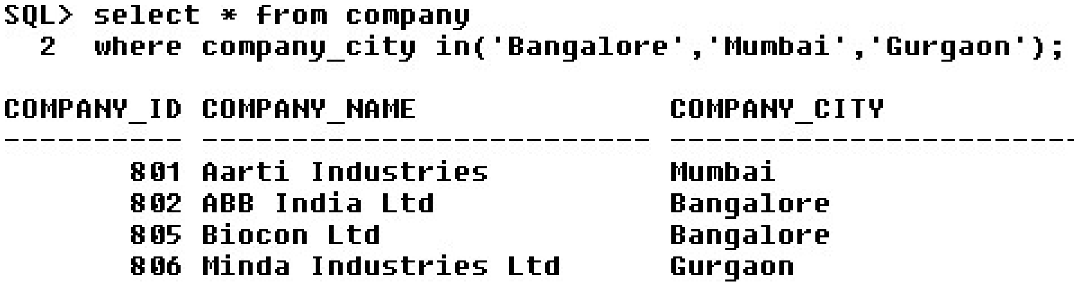


```
SQL> select * from company where company_name like 'A%';

COMPANY_ID COMPANY_NAME                   COMPANY_CITY
---------- ------------------------------ ------------------------------
       801 Aarti Industries               Mumbai
       802 ABB India Ltd                  Bangalore
       803 Adani Power Ltd                Ahmedbad

SQL> select * from company where COMPANY_CITY like '_u%';

COMPANY_ID COMPANY_NAME                   COMPANY_CITY
---------- ------------------------------ ------------------------------
       801 Aarti Industries               Mumbai
       806 Minda Industries Ltd           Gurgaon
```

IN

The IN operator is used when you want to compare a column with more than one value. It is similar to an OR condition.

For example: If you want to find the names of company located in the city Bangalore, Mumbai, Gurgaon, the query would be like,

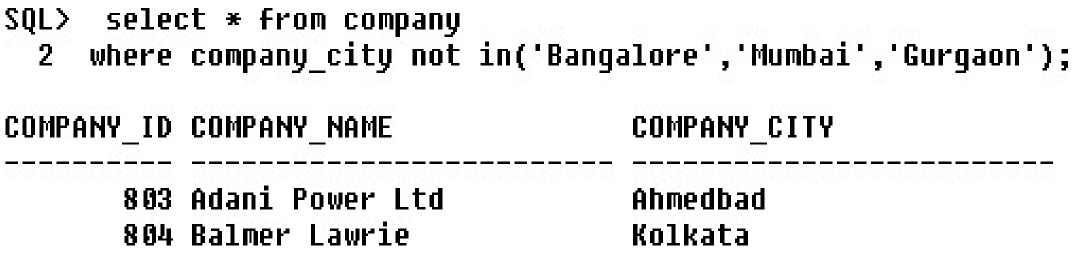


```
SQL> select * from company
  2  where company_city in('Bangalore','Mumbai','Gurgaon');

COMPANY_ID COMPANY_NAME              COMPANY_CITY
---------- ------------------------- -------------------------
       801 Aarti Industries          Mumbai
       802 ABB India Ltd             Bangalore
       805 Biocon Ltd                Bangalore
       806 Minda Industries Ltd      Gurgaon
```

NOT IN

The NOT IN operator is opposite to IN.

For example: If you want to find the names of company located in the other city of Bangalore, Mumbai, Gurgaon, the query would be like,

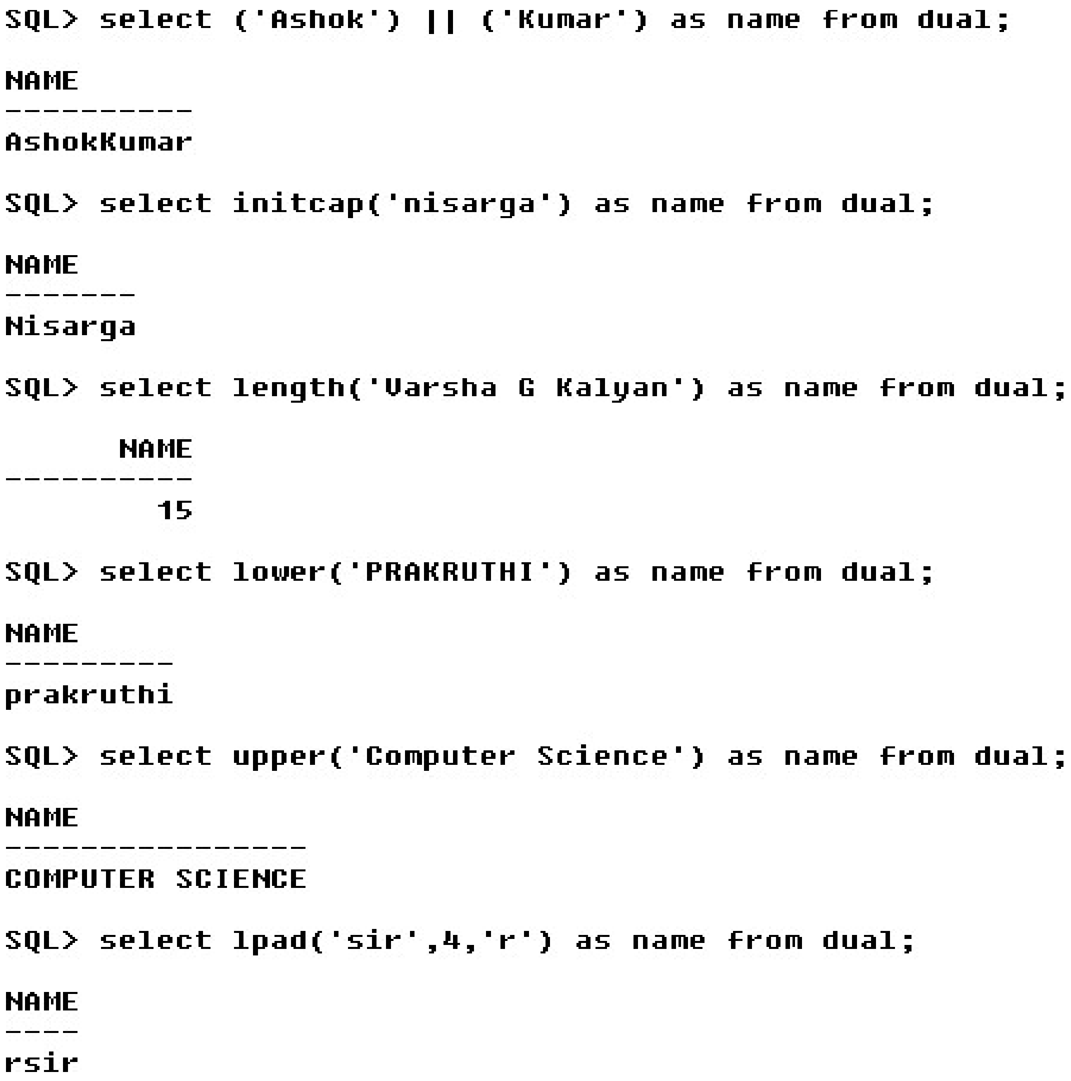


```
SQL>  select * from company
  2  where company_city not in('Bangalore','Mumbai','Gurgaon');

COMPANY_ID COMPANY_NAME              COMPANY_CITY
---------- ------------------------- -------------------------
       803 Adani Power Ltd           Ahmedbad
       804 Balmer Lawrie             Kolkata
```

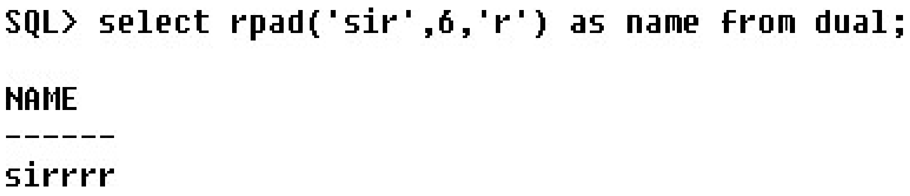
16.10 SQL String Functions

SQL string functions are used primarily for string manipulation. The following table details the important string functions:

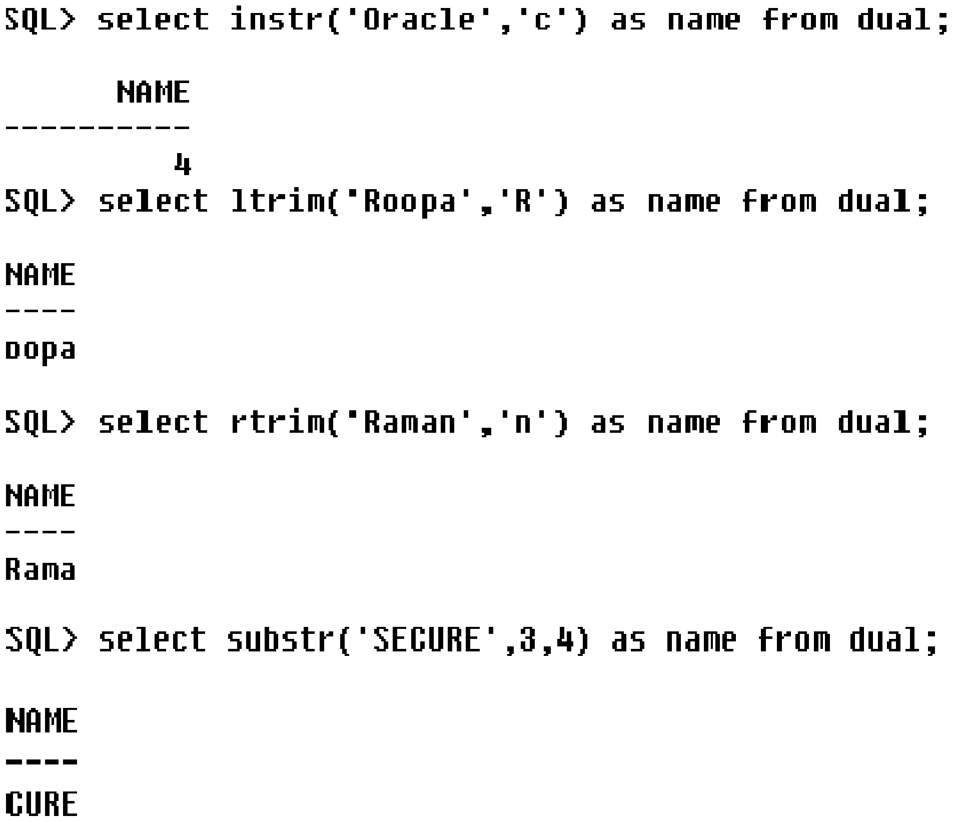
SQL Command	Meaning
||	It used for concatenation.
INITCAP	Return a string with first letter of each word in upper case.
LENGTH	Return the length of a word.
LOWER	Returns character, with all letters forced to lowercase.
UPPER	Returns character, with all letters forced to uppercase.
LPAD	Returns character, left-padded to length n with sequence of character specified.
RPAD	Returns character, right-padded to length n with sequence of character specified.
LTRIM	Removes characters from the left of char with initial characters removed upto the first character not in set.
RTRIM	Returns characters, with final characters removed after the last character not in the set.
SUBSTR	Returns a portion of characters, beginning at character **m**, and going upto character **n**. if **n** is omitted, it returns upto the last character in the string. The first position of char is 1.
INSTR	Returns the location of substring in a string.



```
SQL> select ('Ashok') || ('Kumar') as name from dual;

NAME
----------
AshokKumar

SQL> select initcap('nisarga') as name from dual;

NAME
-------
Nisarga

SQL> select length('Varsha G Kalyan') as name from dual;

      NAME
----------
        15

SQL> select lower('PRAKRUTHI') as name from dual;

NAME
---------
prakruthi

SQL> select upper('Computer Science') as name from dual;

NAME
----------------
COMPUTER SCIENCE

SQL> select lpad('sir',4,'r') as name from dual;

NAME
----
rsir
```



```
SQL> select rpad('sir',6,'r') as name from dual;

NAME
------
sirrrr
```



```
SQL> select instr('Oracle','c') as name from dual;

      NAME
----------
         4
SQL> select ltrim('Roopa','R') as name from dual;

NAME
----
oopa

SQL> select rtrim('Raman','n') as name from dual;

NAME
----
Rama

SQL> select substr('SECURE',3,4) as name from dual;

NAME
----
CURE
```

16.11 Summary

SQL stands for "Structured Query Language" and is a language to communicate with relational and object oriented databases. With SQL new tables (relations, schemes) can be created, altered, and deleted using the commands CREATE TABLE, ALTERTABLEand DROP TABLE. This part of SQL is also known as the Data Definition Language (DDL). More important in the daily use of SQL are the data query and manipulation (DML) commands. These commands allow you to INSERT, DELETE, and UPDATE values in the database. SQL is also used to control access restrictions to the database or to parts of it.