Department of Distance Education

Punjabi University, Patiala

Class : B.A. II (Computer Applications) **Semester : 4**

Paper : (BAP 203) Data Base Management System (DBMS)

Medium : English **Unit : II**

Lesson No.

2.1 : Relational Algebra

2.2 : Database Design

2.3 : Normalization

2.4 : Database Integrity And Recovery

2.5 : Database Security

2.6 : Database Concurrency

2.7 : MS-Access

2.8 : Queries in MS-Access

2.9 : Introduction to Forms

2.10 : Reports and Macro

Department website : www.pbidde.org

LESSON NO. 2.1 **AUTHOR : VISHAL GOYAL**

RELATIONAL ALGEBRA

Structure:

2.1.0 Introduction
2.1.1 Objectives
2.1.2 Relational Algebra
2.1.3 Summary
2.1.4 Self Understanding
2.1.5 Further Readings

2.1.0 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.

2.1.1 Objectives

After reading this lesson you will be able to learn the concepts of Relational Algebra.

2.1.2 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 (π). 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

π age **(employee)**

Age
23
43

B

π age,salary **(employee)**

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

C

π 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 is obtained.

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

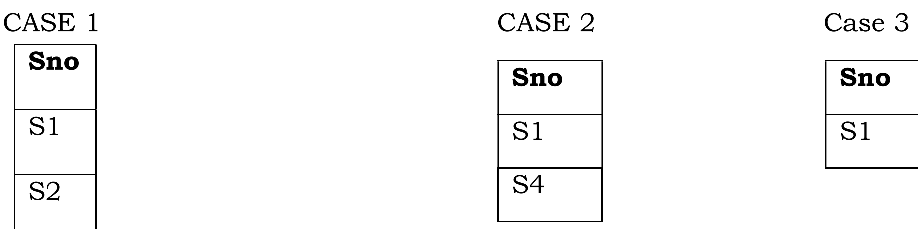
CASE 2

Pno
P2
P4

CASE 3

Pno
P1
P2
P3
P4
P5
P6

A DIVIDED BY B



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.

2.1.3 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.

2.1.4 Self Understanding

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

2.1.5 Further Readings

Bipin C. Desai, *An introduction to Database System,* Galgotia Publication, New Delhi.

C. J. D ate, An *introduction to database Systems*, Sixth Edition, Addison Wesley.

Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison Wesley.

LESSON NO. 2.2 **AUTHOR : VISHAL GOYAL**

DATA BASE DESIGN

Structure:

2.2.0 Introduction
2.2.1 Objectives
2.2.2 Functional Dependency
2.2.3 Decomposition
2.2.4 Problems arising out of bad database Design
2.2.5 Summary
2.2.6 Self Understanding
2.2.7 Further Readings

2.2.0 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.

2.2.1 Objectives

After completing this lesson, you will be able to:

- Define Functional Dependency and its importance in database design
- Understand decomposition of relation
- Understand the problems that arise due to bad database design

2.2.2 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.

2.2.2.1 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 $\alpha \subseteq R$ and $\beta \subseteq R$. The functional dependency

$$\alpha \rightarrow \beta$$

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 $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Using the functional-dependency notation, we say that K is a super-key of R if $K \rightarrow R$. 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 $\alpha \rightarrow \beta$ is trivial if $\beta \leq \alpha$.

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

2.2.2.2 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

$$A \rightarrow B$$
$$A \rightarrow C$$
$$CG \rightarrow H$$
$$CG \rightarrow I$$
$$B \rightarrow H$$

The functional dependency

$$A \rightarrow H$$

is logically implied. That is, we can show that, whenever our given set of functional dependencies holds on a relation, $A \rightarrow H$ 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 $A \rightarrow B$, it follows from the definition of functional dependency that

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

then, since we are given that $B \rightarrow H$, 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 $A \rightarrow H$.

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 $A \rightarrow H$ is in the closure of our example set of dependencies.

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 $\alpha\beta$ to denote $\alpha \cup \beta$.

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 α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- **Augmentation rule**. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule**. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ 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 $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition rule**. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- **Pseudotransitivity rule**. If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ 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 f1 and f2 in F+
        if f1 and f2 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.

2.2.2.3 Closure of Attribute Sets

To test whether a set α is a super-key, we must devise an algorithm for computing the set of attributes functionally determined by α. One way of doing this is to compute F^+, take all functional dependencies with α 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 α is useful not only for testing whether α is a super-key, but also for several other tasks, as we will see later in thus section.

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

```
result := α
while (changes to result) do
        for each functional dependency β→ γ in F do
        begin
                if β ≤ result then result := result U γ
        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

- $A \rightarrow B$ cause us to include B in *result*. To see fact, we observe that $A \rightarrow B$ is in F, $A \leq$ *result* (which is AG), so *result* := *result* U B.
- $A \rightarrow C$ causes *result* to become ABCG.
- $CG \rightarrow H$ causes *result* to become ABCGH.
- $CG \rightarrow I$ 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 $\alpha \rightarrow \alpha$ always holds (by the reflexivity rule). We claim that for any subset β of result, $\alpha \rightarrow \beta$. Since we start the while loop with $\alpha \rightarrow$ *result* being true, we can add γ to result only if $\beta \leq$ *result* and $\beta \rightarrow \gamma$. But then *result* $\rightarrow \beta$ by the reflexivity rule, so $\alpha \rightarrow \beta$ by transitivity. Another application of transitivity shows that $\alpha \rightarrow \gamma$ (using $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$). The union rule implies that $\alpha \rightarrow$ *result* U γ, so functionally determines any new result generated in the while loop. Thus any attribute returned by the algorithm is in α^+.

It is easy to see that the algorithm finds all α^+. If there is an attribute in α^+ that is not yet in *result*, then there must be a functional dependency $\beta \rightarrow \gamma$ for which $\beta \leq$ *result*, and at least one attribute in γ 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 α is a super-key, we compute α^+, and check if α^+ contains all attributes of R.
- We can check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+) by checking if $\beta \subseteq \alpha^+$. That is we compute α^+ by using attribute

closure and then check if it contains β. This test is particularly useful, as we will see later in this chapter.

- It gives us an alternative way to compute F^+: for each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

2.2.3 Decomposition

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.

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

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

$\Pi_{branch\text{-}name} (\sigma_{amount < 1000} (branch\text{-}customer \bowtie customer\text{-}loan)$

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 $\{R_1, R_2, \dots, R_n\}$ is a decomposition of R if

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

That is $\{R_1, R_2, \dots, R_n\}$ 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 $\{r_1, r_2, \dots, r_n\}$ is the database that results from decomposition of r into $\{R_1, R_2, \dots R_n\}$ it is always the case that

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

To see that this assertion is true consider a tuple t in relation r,. When we compute the relations $r_1, r_2, \dots r_n$ 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 $r_1 \infty r_2 \infty \dots \infty r_n$. The details are left for you to complete as an exercise. Therefore every tuple in r appears in $r_1 \infty r_2 \infty \dots \infty r_n$.

In general $r \neq r_1 \infty r_2 \infty \dots \infty r_n$. 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 $\{R_1, R_2 \dots R_n\}$ of R is a lossless join decomposition if for all relations r on schema R that are legal under C,

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

2.2.3.1 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.

2.2.3.2 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 Ri in D do
begin
        Fi = the restriction of F+ to Ri
end
F':=Φ
for each restriction Fi do
        begin
                F' = F' U Fi
        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 $\alpha \rightarrow \beta$ 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 = α
while (changes to result) do
        for each Ri in the decomposition
```

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

result = *result* U t

The attribute closure is with respect to the functional dependencies in F. If *result* contains all attribute in β then the functional dependency $\alpha \rightarrow \beta$ 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^+.

2.2.3.3 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.

2.2.4 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.

1.5 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.

2.2.6 Self Understanding:

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?

2.2.7 Further Readings:

1. Bipin C. Desai, *An introduction to Database System*, Galgotia Publication, New Delhi.
2. C. J. Date, An *introduction to database Systems*, Sixth Edition, Addison Wesley.
3. Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison Wesley.

LESSON NO. 2.3 **AUTHOR : VISHAL GOYAL**

NORMALIZATION

Structure:

2.3.0 Introduction
2.3.1 Objectives
2.3.2 Normalization
2.3.3 First Normal Form
2.3.4 Second Normal Form
2.3.5 Third Normal Form
2.3.6 Boyce-Codd Normal Form
2.3.7 Multi-valued Dependency
2.3.8 Fourth Normal Form
2.3.9 Join Dependencies and Fifth Normal Form
2.3.10 Database Design Process
2.3.11 Summary
2.3.12 Self Understanding
2.3.13 Further Readings

2.3.0 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.

2.3.1 Objectives

After completing this lesson, you will be able to:

- Define Normalization, its need and importance
- Various types of Normal Forms
- Define Multivalued Functional Dependency

- Understand database design process

2.3.2 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.

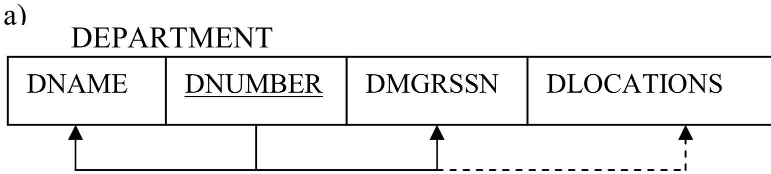
We now present the first three normal forms: 1NF, 2NF and 3NF. These were proposed by Codd (1972) as a sequence to ultimately achieve the desirable state of 3NF relations by progressing through the intermediate states of 1NF and 2NF if needed.

2.3.3 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).



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*:

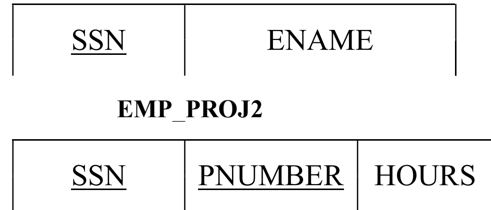
a) **EMP_PROJ**

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

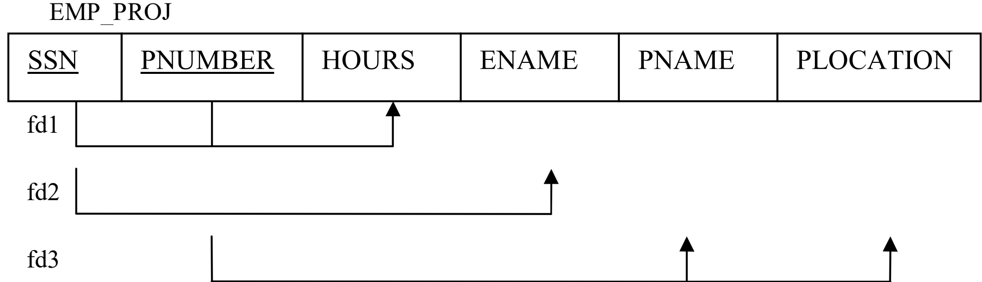
This procedure can be applied recursively to a relation with multi-valued level nesting to unnest the relation into a set of 1NF relations. This is useful in converting hierarchical schemas into 1NF relations. As we shall see in the coming topics, restricting relations to 1NF leads to the problems associated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

2.3.4 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,

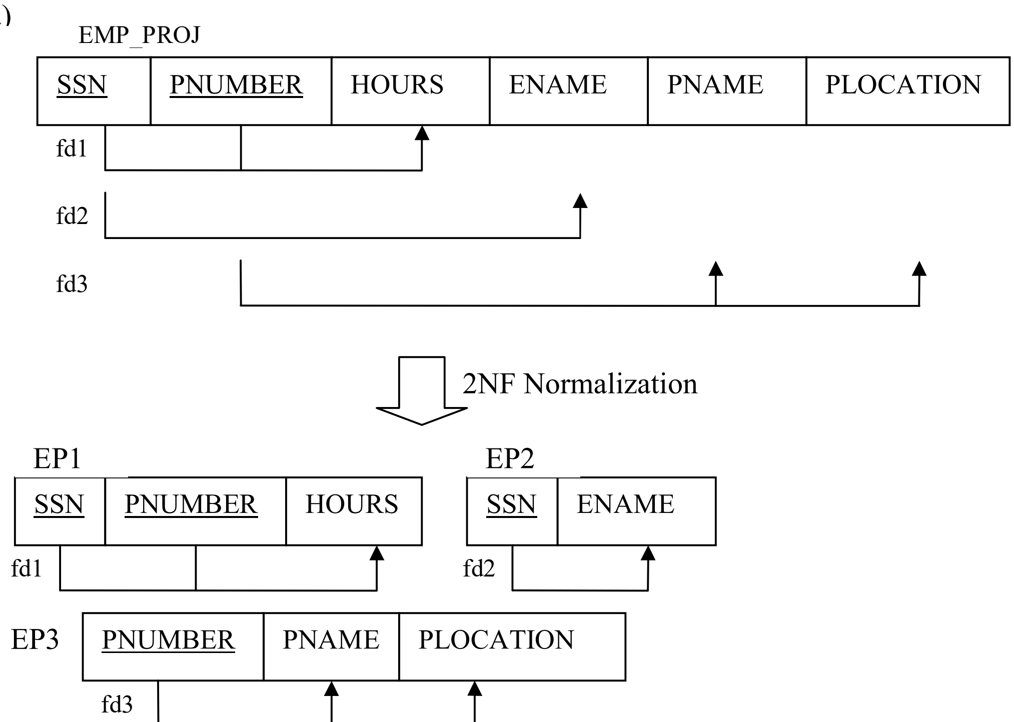
(X-{A}) *→ Y. A functional dependency X→Y is a partial dependency if some attribute A ε X can be removed from X and the dependency still holds; that is for some A ε X, (X – {A})→Y. In figure below, {SSN, PNUMBER} → HOURS is a full dependency (neither SSN →HOURS nor PNUMBER→HOURS holds). However, the 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.

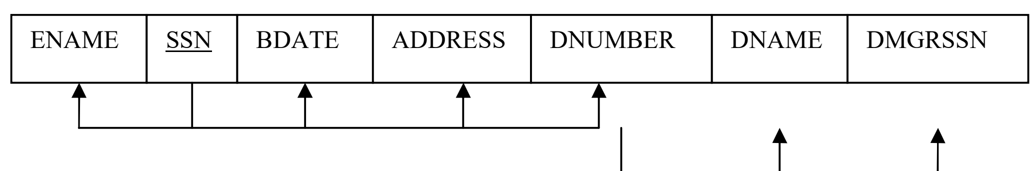
If a relation schema is not in 2NF it can be further normalized into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent. The functional dependencies fd1, fd2 and fd3 in Figure above hence lead to the 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*:



2.3.5 Third Normal Form (3NF)

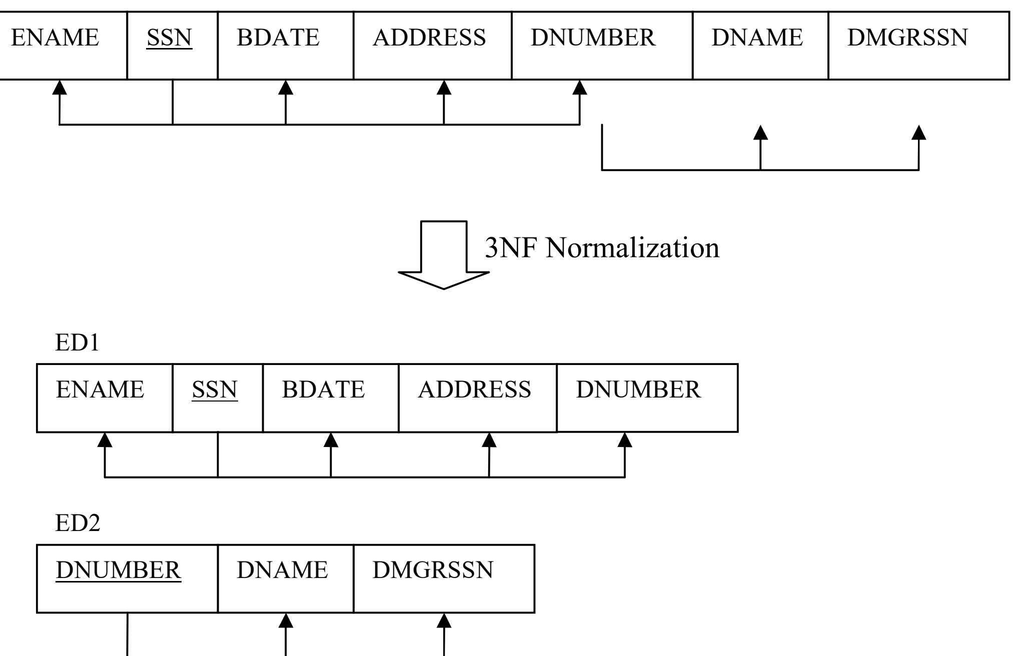
Third Normal form is based on the concept of transitive dependency. A functional dependency X →Y in a relation schema R is a transitive dependency if there is a set of attributes Z that is not a subset of any key of R, and both X→Z and Z→Y hold. The dependency SSN→DMGRSSN is transitive through DNUMBER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



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.

2.3.6 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

fd1

fd2

fd3

fd4

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.

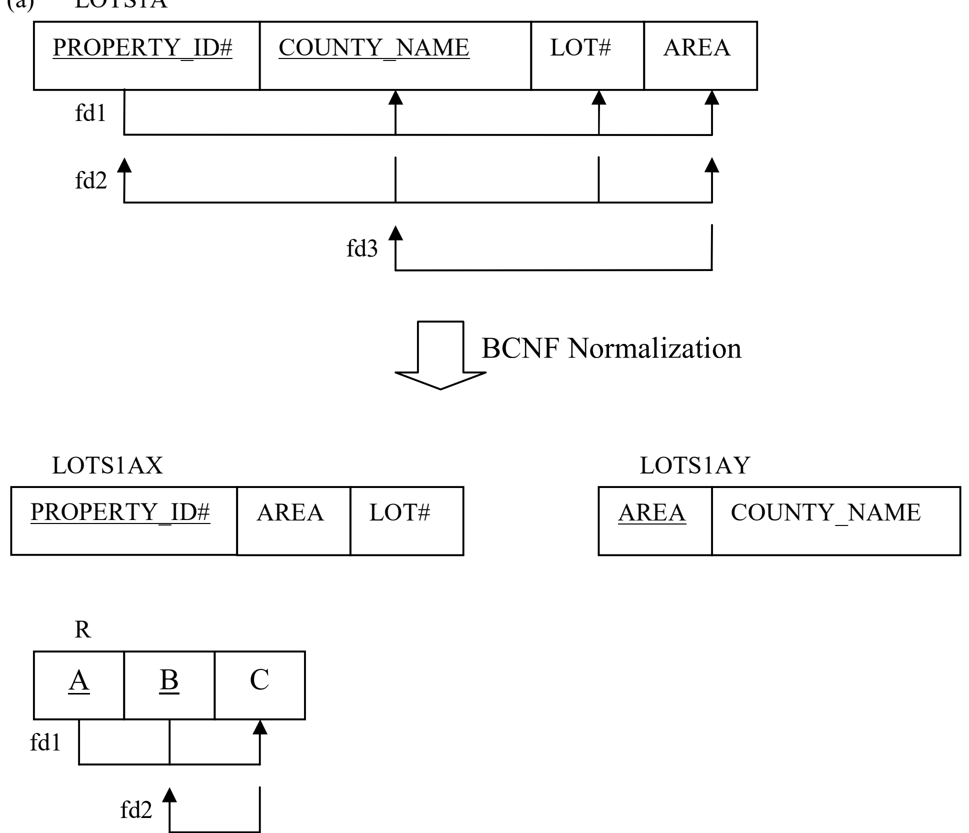
This definition of Boyce-Codd differs slightly from the definition of 3NF. **A relation schema R is in BCNF if whenever a functional dependency X → A holds in R, then X is a super-key of R.** The only difference between BCNF and 3NF is that condition (b) of 3NF, which allows A to be prime if X is not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



2.3.7 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**

<u>SNAME</u>	<u>PARTNAME</u>
Smith	Bolt
Smith	Nut
Adamsky	Bolt
Walton	Nut
Adamsky	Nail

R2

<u>SNAME</u>	<u>PROJNAME</u>
Smith	ProjX
Smith	ProjY
Adamsky	ProjY
Walton	ProjZ
Adamsky	ProjX

R3

<u>PARTNAME</u>	<u>PROJNAME</u>
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.

- ($_{IR}$1) (Reflexive rule for FDs0: if $X \geq Y$, then $X \rightarrow Y$.
- ($_{IR}$2) (Augmentation rule for FDs): $\{X \rightarrow Y\} \models XZ \rightarrow YZ$
- ($_{IR}$3) (Transitive rule for FDs): $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.
- ($_{IR}$4) (Complementation rule for MVDs): $\{X \rightarrow\rightarrow Y\} \models \{X \rightarrow\rightarrow (R - (X \cup Y))\}$.
- ($_{IR}$5) (Augmentation rule for MVDs): If $X \rightarrow\rightarrow Y$ and $W \geq Z$ then $WX \rightarrow\rightarrow YZ$
- ($_{IR}$6) (Transitive rule for MVDs): $\{X \rightarrow\rightarrow Y, Y \rightarrow\rightarrow Z\} \models X \rightarrow\rightarrow (Z-Y)$
- ($_{IR}$7) (Replication rule FD to MVD)): $\{X \rightarrow Y\} \models X \rightarrow\rightarrow Y$
- ($_{IR}$8) (Coalescence rule for FDs and MVDs): If $X \rightarrow\rightarrow Y$ and there exists W with the properties that (a) $W \cap Y$ is empty, (b) $W \rightarrow Z$ and (c) $Y \geq Z$ then $X \rightarrow Z$.

$_{IR}$1 through $_{IR}$3 are Armstrong's inference rules for FDs alone. $_{IR}$4 through $_{Ir}$6 are inference rules pertaining to MVDs only. $_{IR}$7 and $_{IR}$8 relate FDs and MVDs. In particular $_{IR}$7 says that a functional dependency is a special case of a multi-valued dependency; that is, every FD is also an MVD. An FD $X \rightarrow Y$ is an MVD $X \rightarrow\rightarrow Y$ 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 $_{IR}$1 through $_{IR}$8 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.

2.3.8 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 $(R_1 \cap R_2) \rightarrow\rightarrow (R_1 - R_2)$ (or by symmetry, if and only if $(R_1 \cap R_2) \rightarrow\rightarrow (R_2 - R_1)$.

This is similar to property Lj1 of Section 2.3.1.3 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;
```

2.3.9 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,

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

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.

2.3.10 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.

$\Pi_{R_i} (r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n)$

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 $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$ is called a universal relation, since it involves all the attributes in the universe defined by $R_1 \cup R_2 \cup \ldots \cup R_n$.

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.

2.3.11 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.

2.3.12 Self Understanding:

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

7. Define Join Dependency with example.
8. Explain fifth Normal form using Join Dependency using suitable example.
9. Explain the various insert, update and delete anomalies in various normal forms.

2.3.13 Further Readings:

1. Bipin C. Desai, *An introduction to Database System,* Galgotia Publication, New Delhi.
2. C. J. Date, An *introduction to database Systems*, Sixth Edition, Addison Wesley.
3. Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison Wesley.

LESSON NO. 2.4 **AUTHOR : VISHAL GOYAL**

DATABASE INEGRITY AND RECOVERY

Structure:

2.4.0 Introduction
2.4.1 Objectives
2.4.2 Database Integrity
2.4.3 Database Recovery
2.4.4 Summary
2.4.5 Self Understanding
2.4.6 Further Readings

2.4.0 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.

2.4.1 Objectives

After completing this lesson, you will be able to:

- Understand database Integrity
- Understand database recovery

2.4.2 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.

2.4.2.1 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.

2.4.2.2 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 $r \bowtie s$. 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.)

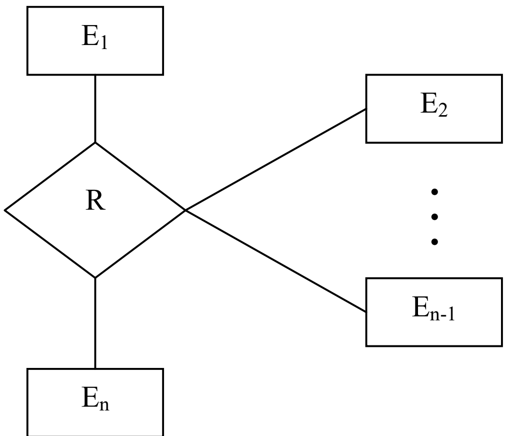
In the Lunartown example, tuple t_1 in account has a value on the foreign key branch-name that does not appear in branch. In the Mokan-branch example tuple t_2 in branch has a value on branch-name that does not appear in account, but branch-name is not a foreign key. Thus the distinction between our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

2.4.2.3 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.

2.4.2.4 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:

$$\Pi_{\alpha} (r2) \leq \Pi_{k} (r1)$$

- **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] = $t_2[\alpha]$

  $$t_2[\alpha] \ \varepsilon \ \Pi_{k} (r1)$$

- **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.

2.4.2.5 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.

2.4.2.6 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.

2.4.3 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.

2.4.3.1 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.

2.4.3.2 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:

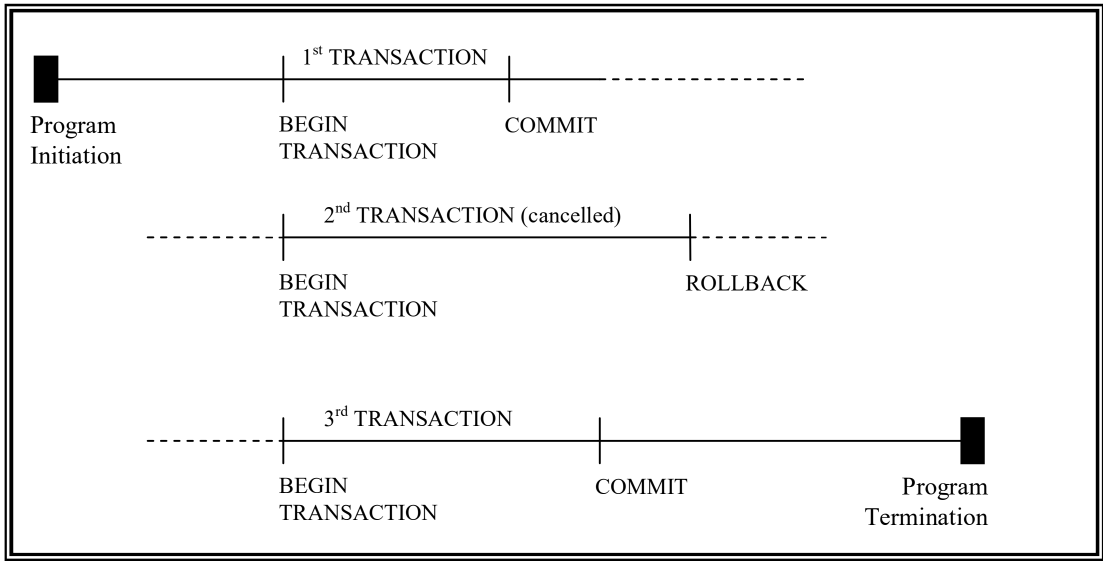
1 All updates made by the executing program since the previous commit points are committed; that is, they are made permanent. Prior to the commit point, all such updates should be regarded as tentative only—tentative in the sense that they might subsequently be undone (i.e. rolled back). Once committed 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.

2.4.3.3 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.

2.4.3.4 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

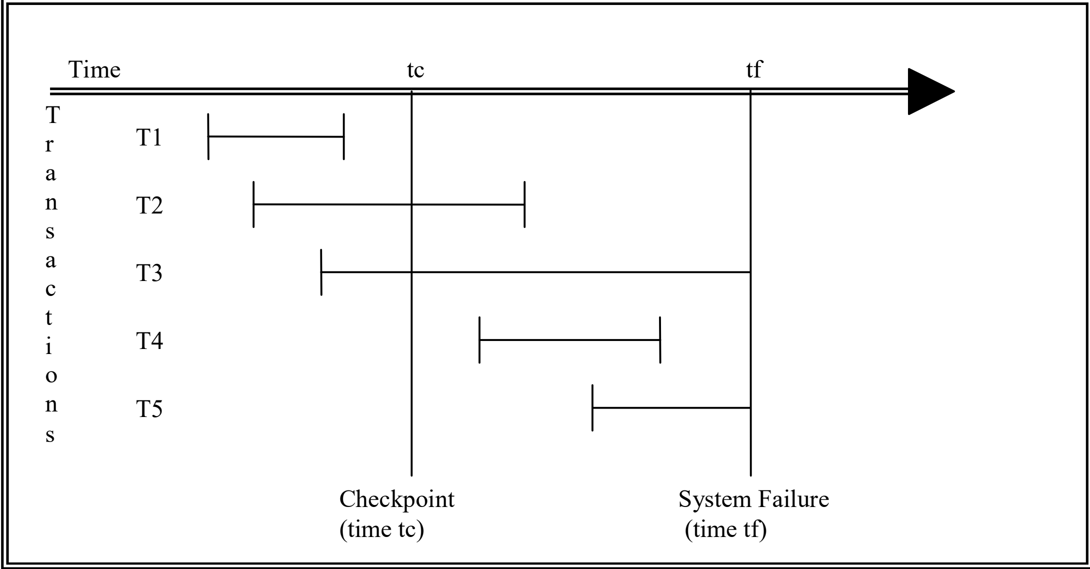
currently using that portion. A media failure is sometimes called a *hard crash*.

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.

2.4.3.5 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.

2.4.4 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.

2.4.5 Self Understanding:

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.

2.4.6 Further Readings:

1. Bipin C. Desai, *An introduction to Database System,* Galgotia Publication, New Delhi.
2. C. J. Date, An *introduction to database Systems*, Sixth Edition, Addison Wesley.
3. Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison Wesley.

LESSON NO. 2.5 **AUTHOR : VISHAL GOYAL**

DATA BASE SECURITY

Structure:

2.5.0 Introduction
2.5.1 Objectives
2.5.2 Database Security
2.5.3 Authorization
2.5.4 Encryption and Authentication
2.5.5 Methods of implementing Security
2.5.6 Self Understanding
2.5.7 Further Readings

2.5.0 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.

2.5.1 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

2.5.2 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.

2.5.3 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 $U_i \rightarrow U_j$ 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.

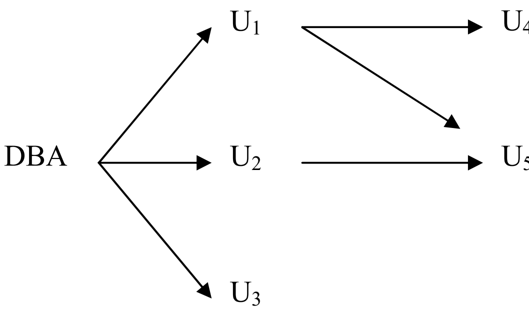


Figure *L*: Authorization-grant chart

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.

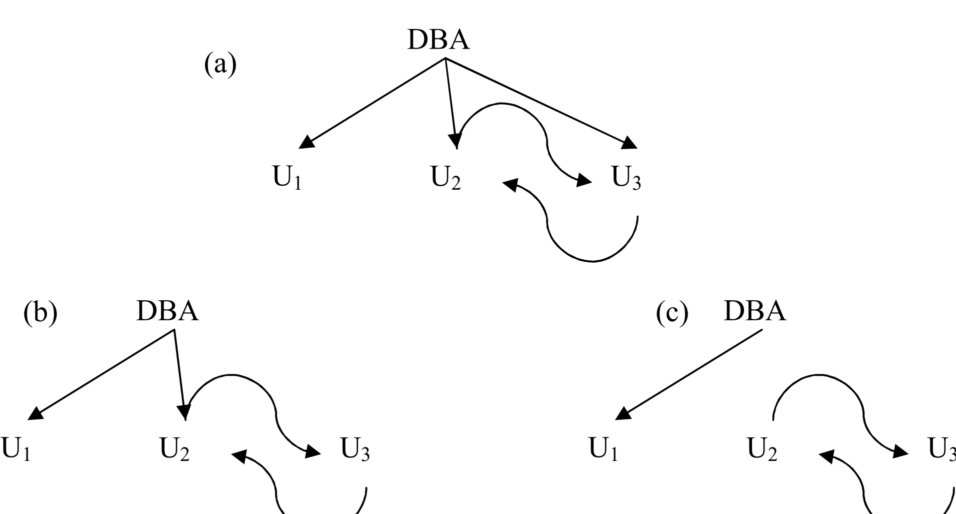
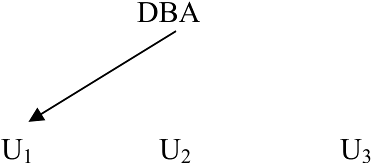


Figure *M*: Attempt to defeat authorization revocation

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.

2.5.4 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).

2.5.5 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.

2.5.6 Self Understanding:

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?

2.5.7 Further Readings:

1. Bipin C. Desai, *An introduction to Database System,* Galgotia Publication, New Delhi.
2. C. J. Date, An *introduction to database Systems*, Sixth Edition, Addison Wesley.
3. Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison Wesley.

LESSON NO. 2.6 **AUTHOR : VISHAL GOYAL**

DATA BASE CONCURRENCY

Structure:

2.6.0 Introduction
2.6.1 Objectives
2.6.2 Database Concurrency
2.6.3 Problems arising out of concurrency
2.6.4 Methods of handling concurrency
2.6.5 Summary
2.6.6 Self Understanding
2.6.7 Further Readings

2.6.0 Introduction

Large computer systems are typically used by many users. These systems usually allow multiple transactions to run concurrently i.e. at the sane time. There are three different problems that arise due to concurreny viz. Lost Update,Dirty Read and Incorrect Analysis Problem. In order to prevent these problems, we need to make transactions schedule serializable. One way of ensuring serializibility is to use the locking mechanism. In the following section we will study how we can we prevent deadlocks during achieving the concurrency.

2.6.1 Objectives

After reading this lesson, you will be able to understand

- About database concurrency
- Problems arising out of concurrency
- Methods of handling concurrency
- Locking Technique
- Two Phase Protocol
- Time Stamping
- Serializability

2.6.2 Database Concurrency

The term *concurrency* refers to the fact that DBMSs typically allow many transactions to access the same database at the same time – and in such a system, as is well known, some kind of concurrency mechanism is needed to ensure that concurrent transactions do not interfere with each other.

2.6.3 Problems arising out of concurrency

Three Concurrency Problems

We begin by considering some of the problems that any concurrency control mechanism must address. There are essentially three ways in which things can go wrong, that is, in which a transaction, though correct in itself, can nevertheless produce the wrong answer if some other transaction interferes with it in some way. The three problems are:

- The lost update problem
- The uncommitted dependency problem and
- The inconsistent analysis problem.

We consider each in turn.

The Lost Update Problem

Consider the situation illustrated in Figure below:

Transaction A	time	Transaction B
–		–
–		–
		–
RETRIEVE t	t1	–
–		
–	t2	RETRIEVE t
		–
UPDATE t	t3	–
–		
–	t4	UPDATE t
–	↓	–

That figure is meant to be read as follows; Transaction A retrieves some tuple t at time t1; transaction B retrieves that same tuple t at time t2; transaction A updates the tuple on the basis of the values seen at time (t1) at time t3; and transaction B updates the same tuple t on the basis of the values seen at time t2, which are the same as those seen at time t1) at time t4. Transaction A's updates is lost at time t4, because transaction B overwrites it without even looking at it.

The Uncommitted Dependency Problem

The uncommitted dependency problem arises if one transaction is allowed to retrieve-or worse update-a tuple that has been updated by some other transaction but not yet committed by that other transaction. For if it has not yet been committed, there is always a possibility that it never will be committed but will be rolled back instead-in which case the first transaction will have seen some data that now no longer exists (and in a sense never did exist). Consider the following diagrams:

Transaction A	time	Transaction B
–		–
–		–
–		
–	t1	UPDATE t
		–
RETRIEVE t	t2	–
–		–
–	t3	ROLLBACK
–	↓	

Transaction A becomes dependent on an uncommitted change at time t2

Transaction A	time	Transaction B
–		–
–		–
–		
–	t1	UPDATE t
		–
UPDATE t	t2	–
–		–
–	t3	ROLLBACK
–	↓	

Transaction A updates an uncommitted change at time t2, and loses that update at time t3

In the first example transaction A sees an uncommitted update (also called an uncommitted change) at time t2. That update is then undone at time t3. Transaction A is therefore operating on a false assumption-namely the assumption that tuple t has the value seen at time t2 whereas in fact it has whatever value it had prior to time t1. As a result transaction A might well produce incorrect output. Note by the way that the rollback of transaction B might be due to no fault of B's-it might, for example be the result of a system crash. (And transaction A might already have terminated by at that time, in which case the crash would not cause a rollback to be issued for A also.)

The second example is even worse. Not only does transaction A become dependent on an uncommitted change at time t2 but actually loses an update at time t3-because the rollback at time t3 cause tuple t to be restored to its value prior to time t1. This is another version of the lost update problem.

The Inconsistent Analysis Problem

Consider Figure below which shows two transactions A and B operating on account (ACC) tuples:

ACC 1	ACC 2	ACC 3
40	50	30

Transaction A	time	Transaction B
–		–
–		–
RETRIEVE ACC 1: sum = 40	t1	–
RETREIVE ACC 2: sum = 90	t2	–
–	t3	RETREIVE ACC 3
–	t4	UPDATE ACC 3: 30 → 20
–	t5	RETREIVE ACC 1
–	t6	UPDATE ACC 1: 40 → 50
–	t7	COMMIT
RETREIVE ACC 3: sum = 110, *not* 120	t8	

Transaction A is summing account balances, transaction B is transferring an amount 10 from account 3 to account 1. The result produced by A, 110, is obviously incorrect; if A were to go on to write that back into the database. It would actually leave the database in an inconsistent state. We say that A has seen an inconsistent state of the database and has therefore performed an inconsistent analysis. Note the different between this example and the previous one. There is no question here A being dependent on an uncommitted change, since B commits all its updates before A sees ACC 3.

2.6.4 Methods of handling concurrency

Locking Techniques for concurrency Control

One of the main techniques used to control concurrent execution of transactions is based on the concept of locking data items. A lock is a variable associated with a data item in the data base and describes the status of that item with respect to possible operations that can be applied to the item. Generally there is one lock for each data item in the data base. We use locks as a means of synchronizing the access by concurrent transactions to the data base items.

Types of Locks

Several types of locks can be used in concurrency control. We first present binary locks, which are simple but somewhat restrictive in their use. Then we discuss shared and exclusive which provide more general locking capabilities.

Binary Locks : A binary lock can have two states or values-locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. if the value of lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0 the item can be accessed when requested. We refer to the value of the lock associated with item X as LOCK(X).

Two operations lock item and unlock item must be included in the transactions when binary locking is used. A transaction requests access to an item X by issuing a lock_item (X) operation. If LOCK(X) = 1 the transaction is forced to wait; otherwise the transaction sets LOCK(X): =1 (locks the item) and is allowed access. When it sets LOCK(X):=0 (unlocks the item) so that X may be accessed by other transactions. Hence a binary lock enforces mutual exclusion on the data item. A description of the lock_item(X) and unlock item(X) operations is shown in Figure below.

```
lock_item (X):
        B: if LOCK (X) =0 (*item is unlocked*)
        then LOCK (X) ← 1 (*lock the item *)
        else begin
                wait (until LOCK (X) =0 and
                the lock manager wakes up the transaction);
                go to B
        end;

unlock_item (X):
        LOCK (X) ← 0 (*unlock the item *)
        If any transactions are waiting
        Then wakeup one of the waiting transactions;
```

Notice that the lock item and unlock item operations must be implemented as indivisible units (known as critical sections in operations systems) that is no interleaving should be allowed once a lock or unlock operation is started until the operation terminated or the transaction waits. In figure above the wait command within the lock item(X) operation is usually implemented by putting the transaction on a waiting queue for the item X until X is unlocked and the transaction is granted access to it. Other transactions that also want to access X are placed on the same queue. Hence the wait command is considered to be outside the lock_item operation. The DBMS has a lock manager subsystem to keep track of and control access to locks.

When the binary locking scheme is used, every transaction must obey the following rules:

1. A transaction T must issue the operation lock_item(X)_ before any read_item(X) or write(X) operations are performed in T.
2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.
4. A transaction T will not issue a lock(X) operation unless it already holds the lock on item X.

These rules can be enforced by a module of the DBMS. Between the lock_item(X) and unlock_item(X) operations in transaction T, T is said to hold the lock on item X. At most one transaction can hold the lock on a particular item. No two transactions can access the same item concurrently. Notice that it is quite simple to implement a binary lock all that is needed is a binary valued variable LOCK associated with each data item X in the data base. In its simplest form each lock can be a record with two fields<data item name LOCK> plus a queue for waiting transactions. The system only needs to maintain these records for locked items in a lock table.

Shared and Exclusive Locks : The preceding binary locking scheme is too restrictive in general because at most one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for reading purpose only. However, if a transaction is to write an item X, it must have exclusive access to X. For this purpose we can use a different type of lock called a multiple-mode lock. In this scheme there are three locking operations read lock(X), write lock(X) and unlock(X). A lock associated with an item X, LOCK(X) now has three possible states 'read locked' 'write-locked' or 'unlocked'. A read locked item is also called share locked because other transactions are allowed to read the item whereas a write locked item is called exclusive locked because a single transaction exclusively holds the lock on the item.

One simple though not completely general method for implementing the preceding three operations on a multiple-mode lock is to keep track of the number of transactions that hold a shared lock on an item. Each lock can be a record with three fields:

<data item name, LOCK, no_of_reads>. The value of LOCK is one-read locked, write locked or unlocked suitable coded. Again to save space, the system need only maintain lock records for locked items in the lock table. The three operations read_lock(X), write_lock(X) and unlock(X) are described in Figure below. As before each of the three operations should be considered indivisible, no interleaving should be allowed once one of the operations is started until either the operation terminates or the transaction ids placed on awaiting queue for the item.

```
read_lock(X)
B: if LOCK(X)=unlocked
Then begin LOCK(X) ← read locked
          No-of_reads(X) ← 1
          End
Else if LOCK(X)=read locked
          Then no_of_reads(X) ← no_of_reads(X) +1
          else begin wit(until LOCK(X)=unlocked and
          the lock manager wakes up the transaction);
          go to B
end;

write-lock(X)
B:if LOCK(X)=unlocked
          then LOCK(X)← write locked
else begin
          wait(until LOCK(X)=unlocked and
          the lock manager wakes up the transaction);
                    go to B
end;

unlock_item(X)
if LOCK(X)=write-locked
          then begin LOCK(X) ← unlocked
          wakeup one of the waiting transactions if any
          end
else if LOCK(X)=read-locked
          then begin
                    no_of_reads(X) ← no_of_reads(X) – 1
                    if no_of_reads(X)=0
                    then begin LOCK (X)=unlocked
                    wakeup one of the transactions if any
                    end
          end;
```

When we use the multiple mode licking scheme, the system must enforce the following:

1 A Transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.

2 A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.

3 A Transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.

4 A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed as we discuss shortly.

5 A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed as we discuss shortly.

6 A transaction T will not issue an unlock (X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Sometimes it is desirable to relax conditions 4 and 5 in the preceding list. For example, it is possible for a transaction T to issue a read_lock(X) and then later on toi upgrade the lock by issuing a write_lock(X) operation. If T is the only transaction with a read lock X at the time it issues the write_lock operation, we can upgrade the lock. It is also possible for a transaction T to issue a write_lock(X) and then later on to downgrade the lock by issuing a read_lock(X) operation. If we allow upgrading and downgrading of locks, we must include transaction identifiers in the record structure for each lock to store the information on which transactions hold locks on the items and we must change the descriptions of the read_lock(X) and write_lock(X) operations in Figure above appropriately. We have this as an exercise for the reader.

Using binary locks or multiple-mode locks in transactions as described earlier does not guarantee serializability of schedule in which the transactions participate. Figure *G* shows an example where the preceding locking rules are followed but a non-serializable schedule may still result. This is because in Figure *G* (a) the items Y in T1 and X in T2 were unlocked too early. This allows a schedule such as the one shown in Figure *G* (c) to occur; this is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow n additional protocol concerning the positioning of locking and unlocking operations in every transaction. The best known protocol, two phase locking is described next.

Here is figure *G*:

(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
Write_lock(X);	Write_lock(Y);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
Write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

(b)

Initial values: X=20, Y=30
Result of serial schedule T_1 followed by T_2:
X=50, Y=80
Result of serial schedule T_2 followed by T_1:
X=70, Y=50

(c)

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
X:=X+Y;	
write_item(X);	
unlock(X);	

Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction. Such a transaction can be divided into two phases; an expanding (or growing) phase, during which new locks on items can be acquired but none can be released and a shrinking phase during which existing locks can be released but no new locks can be acquired. If upgrading of locks is allowed this definition is unchanged. However if downgrading of locks is also allowed this definition must be changed slightly because all

downgrading must be done in the shrinking phase. Hence a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase of the transaction.

Transactions T1 and T2 of Figure *G*(a) do not follow the two- phase locking protocol. This is because the write-Lock(X) operation follows the unlock(Y) operation in T1 and similarly the write_lock(Y) operation follow the unlock(X) operation in T2. If we enforce two phase locking the transaction can be rewritten as T1 and T2 as shown in Figure *H*. Now, the schedule shown in Figure *G*(c) is not permitted for T1 and T2 under the rules of locking. This is because T1 will issue its write_lock (X) before is unlocks item Y; consequently when T2 issues its read_lock(X) it is forced to wait until T1 issues its unlock (X) in the schedule.

Figure *H*:

T_1’	T_2’
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y/);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

It can be proved that, if every transaction in a schedule follows the two phase locking protocol the schedule is guaranteed to be serializable obviating the need to test for serializability of schedules any more. The locking mechanism by enforcing two phase locking rules also enforces serializability.

Two-phase locking may limit the amount of concurrency that can occur in a schedule. This is because a transaction T may not be able to release an item X after it is through using it if Y before it needs it so that it can release X. Hence, X must remain locked by T until all items that the transaction needs have been locked; only then can X be released by T. Meanwhile another transaction seeking to access X may be forced to wait, even though T is using X; conversely, if Y is locked earlier than it is needed, another transaction seeking to access Y is forced to wait even though T is not using Y yet. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

Basic, Conservative, and Strict Two-Phase Locking : These are a number of variations of two-phase locking (2PL). The technique just described is known as basic 2PL. A variation known as conservative 2PL (or static 2PL) requires a transaction to lock all the items it accesses before the transaction begins execution by predeclaring its read set and write set. Recall that the read set of transaction is the set of all items

that the transaction reads and the write set is the set of all items that the transaction writes. If any of the predeclared items needed cannot be locked the transaction does not lock any item instead it waits until all the items are available for locking. Conservative 2PL is a deadlock free protocol, as we shall see when we discuss deadlock problem.

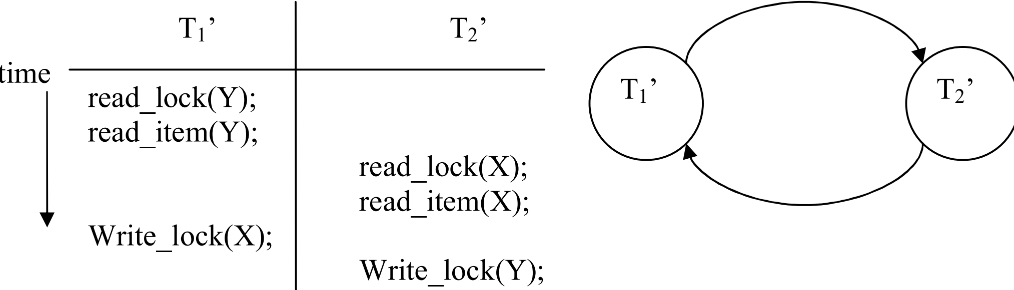
In practice the most popular variation of 2PL is strict 2PL, which guarantee strict schedules. In this variation a transaction T does not release any of its locks until after it commits or aborts. Hence no other transaction can read or write an item that is written by T unless T has committed leading to a strict schedule for recoverability. Notice the difference between conservative and strict 2PL; the former must lock all its items before it starts whereas the latter does not unlock any of its items until after it terminates(by committing or aborting). Strict 2PL is not deadlock free unless it is combined with conservative 2PL.

Although two-phase locking guarantees serializability, the use of locks can cause two additional problems deadlock and live lock. We discuss these problems and their solutions in the next section.

Dealing with Deadlock and Live lock

Deadlock occurs when each of two transactions is waiting for the other to release the lock on an item. A simple example is shown in Figure *I*(a) where the two transactions T1 and T2 are deadlocked in a partial schedule; T1 is waiting for T2 to release item W while T2 is waiting for T1 to release item Y. Meanwhile neither can proceed to unlock the item that the other is waiting for, and other transactions can access neither item X nor item Y. Deadlock is also possible when more than two transactions are involved as we shall see.

Figure *I*:



One way to prevent deadlock is to use a deadlock prevention protocol. One dead lock prevention protocol which is used in conservative two phase locking requires tht every transaction locks all the items it needs in advance. If any of the items cannot be obtained, none of the items are locked. Rather the transaction waits and then tries again to lock all the items it needs. This solution obviously further limits concurrency. A second protocol which also limits concurrency involves ordering all the item in the

data base and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer be aware of the chosen order of the items which is not very practical in the database context.

A number of other deadlock prevention schemes have been proposed that make a decision on whether a transaction involved in a possible deadlock situation should be blocked and made to wait, should be aborted or should preempt and abort another transaction. These techniques use the concept of transaction timestamp TS(T) which is a unique identifier assigned to each transaction. The timestamps are ordered based on the order in which transaction are started; hence if transaction T_1 starts before transaction T_2 then $TS(T_1)<TS(T_2)$. Notice that the older transaction has the smaller timestamp value. Two schemes that prevent deadlock are called wait die and wound wait. Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are as follow:

wait die: if $TS(T_i) < TS(T_j)$ (T_i is older than T_j)
then Ti is allowed to wait
otherwise abort T_i(T_i dies and restart it later with the same timestamp).
wound-wait: if $TS(T_i)<TS(T_j)$ (T_i is older then T_j)
then abort T_j (T_i wounds T_j) and restart it later with the same timestamp
otherwise T_i is allowed to wait.

In wait-die an older transaction is allowed to wait on a younger transaction whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound wait approach does the opposite a younger transaction is allowed to wait on an older one whereas older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it. Both schemes end up aborting the younger of the two transactions that may be involved in a deadlock, and it can be shown that these two techniques are deadlock free. However both techniques cause some transactions to be aborted and restarted even though those transactions may never actually cause a deadlock. Another problem can occur with the wait die where the transaction T_i may be aborted and restarts several times in a row because an older transaction T_j continues to hold the data that Ti need.

Another group of protocols that prevent deadlock do not requires timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms. In the no waiting algorithm, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will occur. The cautious waiting approach was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a confliction lock. The cautions waiting rules are as follows:

cautious waiting: if T_j is not blocked (not waiting for some other locked item)

then Ti is blocked and allowed to wait

otherwise abort T_i.

It can be shown that cautious waiting is deadlock free by considering the times at which a transaction T gets blocked b(T). If the two transactions T_i and T_j above both become blocked and T_i is waiting on T_j then $b(T_i)<b(T_j)$ since a transaction can only wait on a transaction when it is not blocked. Hence the blocking times from s total ordering on all blocked transactions so no cycle that causes deadlock can occur.

Another deadlock prevention scheme involves using time outs. If a transaction waits longer than a system defined timeout the system assumes that the transaction is dead locked and aborts it regardless of whether a deadlock situation actually exists.

A second approach of dealing with deadlock is deadlock detection where we periodically check to see if the system is in a state of deadlock. This solution is attractive if we know there will be little interference among the transactions that is if different transactions will rarely access the same item. This can happen if the transactions are short and each transaction locks only a few items or if the transaction load is light. On the other hand, if transactions are long and each transaction uses many items or if the transaction load is quite heavy, it is advantageous to use a deadlock prevention scheme.

A simple way to detect a state of deadlock is to construct a **wait-for graph**. One node is created in the wait for graph for each transaction that is currently executing in the schedule. Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j create a directed edge ($T_i \rightarrow T_j$). When T_j release the lock(s) on the items that T_i was waiting for directing edge is dropped from the wait for graph. We have a state of deadlock if and only if the wait for graph has a cycle. One problem with this approach is the matter of determining when the system should check for deadlock. Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be aborted. Choosing which transaction to abort is known as victim selection. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates and should try instead to select transaction that have not made many changes or that are involved in more than one deadlock cycle in the wait for graph. A problem known as cyclic restart may occur, where a transaction is aborted and restarted only to be involved in another deadlock. The victim selection algorithm can use higher priorities for transaction that have been aborted multiple times so that they are not selected as victims repeatedly.

Another problem that may occur when we use locking is livelock. A transaction is in a state of livelock if it cannot proceed for an indefinite period of time while other transaction in the system continue in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. The standard solution for livelock is to have a fair waiting scheme. One such

scheme uses a first come first serve queue. Transaction are enabled to lock an item in the order in which they originally requested to lock the item. Another scheme allows some transactions to have priority over others but increase the priority and proceeds. A similar problem to livelock called starvation can occur in the algorithms for dealing with deadlock. It occurs if the algorithms select the same transaction as victims repeatedly thus causing it to abort and never finish execution. The wait die and wound wait schemes discussed above avoid starvation.

Concurrency Control Based on Timestamp Ordering

The use of locks combined with two-phase locking protocol, allows us to guarantee serializability of schedules. The order of transactions in the equivalent serial schedule is based on the order in which executing transaction lock the items they require. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equivalent serial schedule.

Timestamps

A timestamp is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transaction are submitted to the system, so a timestamp can be thought of as the transaction start time. We will refer to the timestamps do not use locks; hence, deadlocks cannot occur.

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3,... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current value of the system clock and ensure that no two timestamp values are generated during the same tick of the lock.

The Timestamp Ordering Algorithm

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the equivalent serial schedule has the transactions in order of their timestamp values. This is called timestamp ordering (TO). Notice how this differs from two-phase locking. In two phase locking a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocol; in timestamp ordering, however the schedule is equivalent to the particular serial order that corresponds to the order of the transaction timestamps. The algorithm must ensure that for each item accessed by more than one transaction in the schedule, the order in which the item is accessed does not violate the serializability of the schedule. To do this the basic TO algorithm associates with each database item X two timestamp (TS) values:

1. read_TS(X): The read timestamp of item X; this is the largest timestamp among all the timestamps of transaction that have successfully read item X.
2. write_TS(X): The write timestamp of item X; this is the largest of all the timestamps of transaction that have successfully written item X.

Whenever some transaction T tries to issue a read_item(X) or a write_item (X) operation the basic TO algorithm compares the timestamp of T with the read timestamp and the write timestamp of X to ensure that the timestamp order or execution of the transactions is not violated. If the timestamp order is violated by the operation, then transaction T will violate the equivalent serial schedule so T is aborted. Then T is resubmitted to the system as a new transaction with a new timestamp. If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back. Similarly any transaction T2 that may have used a value written by T1 must also be rolled back, and so on. This effect is known as cascading rollback and is one of problems associated with basic TO, since the schedule produced are not recoverable. The concurrency control algorithm must check whether the timestamp ordering of transaction is violated in the following two cases:

1. **Transaction T issue a write_item(X) operation;**
   a. If read TS(X)> TS(T) or if write_TS(X)>TS(T) then abort and roll back T and reject the operation. This should be done because some transaction with a timestamp greater than TS(T)—and hence after T in the timestamp ordering has already read or written the value of item X before T had a chance to write X thus violating the timestamp ordering.
   b. If the condition in part a does not occur then execute the write_item(X) operation of T and set writeTS(X) to TS(T)
2. **Transaction T issue a read_item(X) operation:**
   a. If write_TS(X)> TS(T) then abort and roll back T and reject the operation. This should be done because some transaction with timestamp greater than TS(T)- and hence after T in the timestamp ordering has already written the value of item X before T had a chance to read X.
   b. If read TS(X),TS(T) then execute the read item(X) operation of T and set read TS(X) to the larger of TS(T) and the current read TS(X).

Hence the basic TO algorithm checks whenever two conflicting operation occur in the incorrect order and rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be conflict serializable. A modification of the algorithm known as Thomas's write rule

does not enforce conflict serializability but it rejects fewer write operations, by modifying the checks for the write_item(X) operation as follows:

a. if read_TS(X) >TS(T) then abort and roll back T and reject the operation.

b. If read_TS(X) >TS(T) then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T) and hence after T in the timestamp ordering has already written the value of X. hence we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case a.

c. If neither the condition in part a nor the condition in part b occurs then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

The timestamp ordering protocol, like the two phase locking protocol, guarantees serializability of schedules. However some schedules are possible under each protocol that are not allowed under the other. Hence neither protocol allows all possible serializable schedules. As mentioned earlier, dead back does not occur with timestamp ordering However cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

As was mentioned earlier the basic TO algorithm enforces conflict serializability but it does not ensure recoverable schedules; and hence it does not ensure cascade less or strict schedule either a variation of basic TO called strict TO ensure that the schedules are both strict and (conflict) serializable. In this variation, transaction T that issues a read_item(X) or write_item(X) such that TS(T) >write_TS(X) has its read or write operation delayed until the transaction T that wrote the value of X (hence TS(T) = write TS(X)) has committed or aborted. To implement this algorithm it is necessary to simulate the locking of an item X that has been written by transaction T until T is either committed or aborted. This algorithm does not cause deadlock since T waits for T only if TS(T) >TS(T).

Multi-version concurrency Control Techniques

Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as multi version concurrency control techniques, because several versions (values) of an item are maintained. When a transaction requires access to an item, an appropriate version is chosen to maintain the serilizability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an older version of the item to maintain serializability. When a transaction writes an item, it writes a new version and the old version of the item is retained. In general, multi version concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multi version techniques is that more storage is needed to maintain multiple versions of the data base items. However older versions may have to be maintained anyway for example for recovery purposes. In addition some data

base applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a temporal data base, which keeps track of all changes and the times at which they occurred. In such cases, there is no additional penalty for Multi-version techniques since older versions are already maintained.

Several Multi-version techniques concurrency control schemes have been proposed. We discuss two schemes here, one based on timestamp ordering and the other based on two phase locking.

Multi-version Technique based on Timestamp Ordering

In this Multi-version technique, several versions X_1, X_2, ……….. X_k of each data item X are kept by the system. For each version the value of version X1 and the following two timestamps are kept:

1. **read_TS(X):** The read timestamp of X_i this is the largest of all the timestamps of transactions that have successfully read version X_i.
2. **Write_TS(X):** The write timestamp of X_i this is the timestamp of the transaction that wrote the value of version X_i.

Whenever a transaction T is allowed to execute a write_item(X) operation a new version X_{k+1} is created with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(X) the value of read_TS(X) is set to the larger of read_TS(X) and TS(T).

To ensure serializability we use the following two rules to control the reading and writing of data items.

1. If transaction T issue a write_item (X) operation, and version I of X has the highest write_TS(X) of all versions of X that is also less than or equal to TS(T) and TS(T)<read_TS(X) then abort and roll back transaction T; otherwise create a new version X_j of X with read TS(X) = write_TS(X) =TS(T).
2. If transaction T issues a read_item(X) operation find the version I of X that has the highest write_TS(X) of all versions of X that is also les than or equal to TS(T) then return the value of Xi to transaction T and set the value of read_TS(X) to the larger of TS(T) and the current read_TS(C)

In case , transaction t may be aborted and rolled back. This happens if T is attempting to write a version of X that should have been read by another transaction T whose timestamp us read_TS(X) however T has already read_version X which was written by the transaction with timestamp equal to write_TS(X). If this conflict occurs T is rolled back otherwise a new version of X, written by transaction T is created Notice that if T is rolled back cascading rollback may occur. Hence to ensure recoverability a transaction T is not allowed to commit until after all the transaction that have written versions that T has real have committed.

Multi-version Two-Phase Locking

In this scheme there are three locking modes for an item read write and certify. Hence the state of an item X can be one of “read locked” “write locked” and “unlocked” In the standard locking scheme with only read and write locks a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme by means of the lock compatibility table shown in Figure *J* below.

(a)

	Read	Write
Read	yes	no
Write	no	no

(b)

	Read	Write	Certify
Read	yes	no	no
Write	yes	no	no
Certify	no	no	no

An entry of yes means that if a transaction T holds the type of lock specified in the column header on item X and if transaction T requests the type of lock specified in the row header on the same item X then T can obtain the lock because the locking modes are compatible. On the other hand an every of no in the table indicates that the locks are not compatible so T must until t release the lock.

In the standard locking scheme once a transaction obtains a write lock on an item no other transactions can access that item. That idea behind Multi-version two phase locking is to allow other transactions T to read an item X while a single transaction T holds a write lock on X. This is accomplished by allowing two versions for each item X one version must always have been written by some committed transaction. The second version X is created when a transaction T acquires a write lock on a item. Other transaction can continue to read the committed version X whole T holds write lock. Now transaction T can change the value of X as needed without affecting the value of committed version X. However once T is ready to commit it must obtain a certify lock on all items that it currently holds write lock on before it can commit. The certify lock is not compatible with the read locks so the transactions may have to delay its commit until all its write locks items are released by reading transactions. At this point the committed version X of the date item is set to the value of version X, version X is discarded and the certify locks are then released. The lock compatibility table for this scheme is shown in Figure *J* (b)

In this Multi-version to phase locking scheme read can proceed concurrently with a write operation-an arrangement not permitted under the standard to phase locking scheme. The cost is that a transaction may have to delay it commit until it obtains exclusive certify locks on all the items it has updated. It can be shown that this

scheme avoids cascading aborts since transaction are only allow to read the version X that was written by a committed transaction.

2.6.5 Summary

The term *concurrency* refers to the fact that DBMSs typically allow many transactions to access the same database at the same time – and in such a system, due to concurrency, three problems are - The lost update problem, the uncommitted dependency problem and, the inconsistent analysis problem. One of the main techniques used to control concurrent execution of transactions is based on the concept of locking data items. A lock is a variable associated with a data item in the data base and describes the status of that item with respect to possible operations that can be applied to the item. Generally there is one lock for each data item in the data base. We use locks as a means of synchronizing the access by concurrent transactions to the data base items. To control concurrency two type of locks are used – Binary and Shared/ Exclusive Locks. A transaction is said to follow the two phase locking protocol if all locking operations proceed the first unlock operation in the transaction. A time stamp is a unique identifier assigned by the database system to identify the order in which the transaction is submitted to the system. So, time stamping can be thought of a transaction's start time. A deadlock occurs when each transaction in a set of two or more transaction, is waiting for data item which is locked by other transactions in a set i.e. two transactions are waiting for a condition that will never occur. The deadlock prevention protocols are the set of rules that are used to ensure that the schedule will never enter the deadlock state.

2.6.6 Self Understanding

1. What is concurrency?Why we need to control concurrency?
2. Discuss the problems arising due to concurrency.
3. What is a schedule? Explain with example.
4. What is a serial schedule and serializable schedule?
5. What is a lock? Explain different types of locks.
6. Explain the two Phase Locking Protocol with example.
7. What do you understand by deadlock? Explain the various deadlock handling techniques.
8. What is timestamp? Discuss the timestamp ordering protocol.
9. What are the various necessary conditions for deadlock to occur?

2.6.7 Further Readings

1. Bipin C. Desai, *An introduction to Database System,* Galgotia Publication, New Delhi.
2. C. J. Date, An *introduction to database Systems*, Sixth Edition, Addison Wesley.
3. Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems*, Addison Wesley.

MS-ACCESS

Structure:

17.0 Introduction
2.7.1 Objectives
2.7.2 MS-Access Basics
2.7.3 Brief overview of Relational Databases and Database Applications
2.7.4 A Business Example
2.7.5 Starting MS-Access
2.7.6 Creating and Viewing Tables
2.7.7 Viewing and Adding Data to a Table
2.7.8 Summary
2.7.9 Self Understanding

2.7.0 Introduction

MS-Access is a relational database management system. It is the part of MS-Office developed by the Microsoft Corporation. Using the Ms-Access we can create database of our data and can manage the data well. In this lesson we learn the basics of MS-Access, Opening the MS-Access on our system, create databases and tables and last viewing, editing, deleting data from tables of the database.

2.7.1 Objectives

After reading this lesson you will be able to learn

- Basics of the MS-Access
- Starting MS-Access
- Creating and Modifying tables
- Entering data into tables
- Viewing, modifying and deleting data from tables.

2.7.2 MS-Access Basics

Microsoft Access is a Relational Database Management System (RDBMS). At the most basic level, a DBMS is a program that facilitates the storage and retrieval of structured information on a computer's hard drive. Examples of well-know industrial-strength relational DBMSs include

- ➢ Oracle
- ➢ Microsoft SQL Server
- ➢ IBM DB2
- ➢ Informix

Well-know PC-based ("desktop") relational DBMSs include

- Microsoft Access
- Microsoft FoxPro
- Borland dBase

Different Faces of Access

Microsoft generally likes to incorporate as many features as possible into its products. For example, the Access package contains the following elements:

- A Relational database system that supports two industry standard query languages: Structured Query Language (SQL) and Query By Example (QBE);
- A full-featured **procedural programming language** essentially a subset of Visual Basic, a simplified procedural **macro language** unique to Access;
- A **rapid application development environment** complete with visual form and report development tools;
- A sprinkling of **objected-oriented extensions;**
- Various **wizards and builders** to make development easier.

For new users, these "multiple personalities" can be a source of enormous frustration. The problem is that each personality is based on a different set of assumptions and a different view of computing. For instance

- The relational database personality expects you to view your application as sets of data;
- The procedural programming personality expects you to view your application as commands to be executed sequentially;
- The object-oriented personality expects you to view your application as objects which encapsulate state and behavior information.

Microsoft makes no effort to provide an overall logical integration of these personalities (indeed, it is unlikely that such an integration is possible). Instead, it is up to you as a developer to pick and choose the best approach for implementing your application.

Since there are often several vastly different ways to implement a particular feature in Access, recognizing the different personalities and exploiting the best features (and avoiding the pitfalls) of each are important skills for Access developers.

The advantage of these multiple personalities is that it is possible to use Access to learn about an enormous range of information systems concepts without having to interact with a large number of "single-personality" tools, for example:

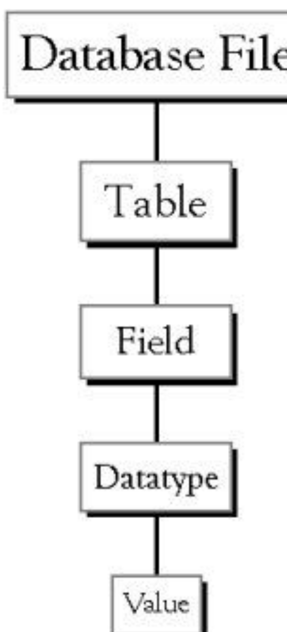
- Oracle for relational databases
- PowerBuilder for rapid applications development,
- SmallTalk for object-oriented programming.

Keep this advantage in mind as we switch back and forth between personalities and different computing paradigms.

Important Definitions related to database in context to Microsoft Access:

Microsoft Access is a powerful program to create and manage your databases. It has many built in features to assist you in constructing and viewing your information. Access is much more involved and is a more genuine database application than other programs such as Microsoft Works.

This lesson will help you get started with Microsoft Access and may solve some of your problems, but it is a very good idea to use the Help Files that come with Microsoft Access. First of all you need to understand how Microsoft Access breaks down a database. Some keywords involved in this process are: *Database File, Table, Record, Field, Data-type.* Here is the Hierarchy that Microsoft Access uses in breaking down a database.



Database File: This is your main file that encompasses the entire database and that is saved to your hard-drive or floppy disk. Example: StudentDatabase.mdb

Table:A table is a collection of data about a specific topic. There can be multiple tables in a database.
Example: Students,Teachers

Field: Fields are the different categories within a Table. Tables usually contain multiple fields.
Example: Student Last Name , Student First Name

Datatypes: Data types are the properties of each field.
A field only has one datatype.
FieldName: Student Last Name
Datatype: Text

This lesson will help you get started with Microsoft Access and may solve some of your problems, but it is a very good idea to use the Help Files that come with Microsoft Access (or any program you use for that matter), for further assistance.

2.7.3 Brief overview of Relational Databases and Database Applications

The first databases implemented during the 1960s and 1970s were based upon either flat data files or the hierarchical or networked data models. These methods of storing data were relatively inflexible due to their rigid structure and heavy reliance on applications programs to perform even the most routine processing.

In the late 1970s, the *relational database model* which originated in the academic research community became available in commercial implementations such as IBM DB2 and Oracle. The relational data model specifies data stored in *relations* that have some *relationships* among them (hence the name *relational*).

In relational databases such as Sybase, Oracle, IBM DB2, MS SQL Server and MS Access, data is stored in *tables* made up of one or more *columns* (Access calls a column a *field*). The data stored in each column must be of a single *data type* such as Character, Number or Date. A collection of values from each column of a table is called a *record* or a *row* in the table.

Different tables can have the same column in common. This feature is used to explicitly specify a relationship between two tables. Values appearing in column A in one table are shared with another table.

Below are two examples of tables in a relational database for a local bank:

Customer Table

CustomerID	Name	Address	City	State	Zip
Number	***Character***	***Character***	***Character***	***Character***	***Character***
1001	Mr. Smith	123 Lexington	Smithville	KY	91232
1002	Mrs. Jones	12 Davis Ave.	Smithville	KY	91232
1003	Mr. Axe	443 Grinder Ln.	Broadville	GA	81992
1004	Mr. & Mrs. Builder	661 Parker Rd.	Streetville	GA	81990

Accounts Table

CustomerID	AccountNumber	AccountType	DateOpened	Balance
Number	***Number***	***Character***	***Date***	***Number***
1001	9987	Checking	10/12/1989	4000.00
1001	9980	Savings	10/12/1989	2000.00
1002	8811	Savings	01/05/1992	1000.00
1003	4422	Checking	12/01/1994	6000.00
1003	4433	Savings	12/01/1994	9000.00
1004	3322	Savings	08/22/1994	500.00
1004	1122	Checking	11/13/1988	800.00

The Customer table has 6 columns (CustomerID, Name, Address, City, State and Zip) and 4 rows (or records) of data. The Accounts table has 5 columns (CustomerID, AccountNumber, AccountType, DateOpened and Balance) with 7 rows of data.

Each of the columns conforms to one of three basic *data types*: Character, Number or Date. The data type for a column indicates the type of data values that may be stored in that column.

- Number - may only store numbers, possibly with a decimal point.
- Character - may store numbers, letters and punctuation. Access calls this data type **Text**.
- Date - may only store date and time data.

In some database implementations other data types exist such as Images (for pictures or other data). However, the above three data types are most commonly used.

Notice that the two tables share the column CustomerID and that the values of the CustomerID column in the Customer table are the same as the values in the CustomerID column in the Accounts table. This *relationship* allows us to specify that the Customer **Mr. Axe** has both a Checking and a Savings account that were both opened on the same day: December 1, 1994.

Another name given to such a relationship is *Master/Detail*. In a master/detail relationship, a single master record (such as Customer 1003, Mr. Axe) can have many details records (the two accounts) associated with it.

In a Master/Detail relationship, it is possible for a Master record to exist without any Details. However, it is impossible to have a Detail record without a matching Master record. For example, a Customer may not necessarily have any account information at all. However, any account information *must* be associated with a single Customer.

Each table also must have a special column called the **Key** that is used to uniquely identify rows or records in the table. Values in a key column (or columns) may never be duplicated. In the above tables, the CustomerID is the key for the Customer table while the AccountNumber is the key for the Accounts table.

2.7.4 A Business Example

In this section, we will outline a business example that will be used as a basis for the examples throughout the lesson. In organizations, the job of analyzing the business and determining the appropriate database structure (tables and columns) is typically carried out by *Systems Analysts*. A Systems Analyst will gather information about how the business operates and will form a *model* of the data storage requirements. From this model, a database programmer will create the database tables and then work with the application developers to develop the rest of the database application.

For this lesson, we will consider a simple banking business. The bank has many customers who open and maintain one or more accounts. For each Customer, we keep a record of their name and address. We also assign them a unique CustomerID. We assign this unique identifier both for convenience and for accuracy. It is much easier to identify a single customer using their CustomerID rather than by looking up their full name and address. In addition, it is possible for the bank to have two customers with the same name (e.g., Bill Smith). In such cases, the unique CustomerID can always be used to tell them apart.

In a similar fashion, all accounts are assigned a unique account number. An account can be either a checking account or a savings account. Savings accounts earn interest but the only transactions allowed are deposits and withdrawals. Checking accounts do not earn interest. We maintain the date that the account was opened. This helps us track our customers and can be useful for marketing purposes. Finally, we maintain the current balance of an account.

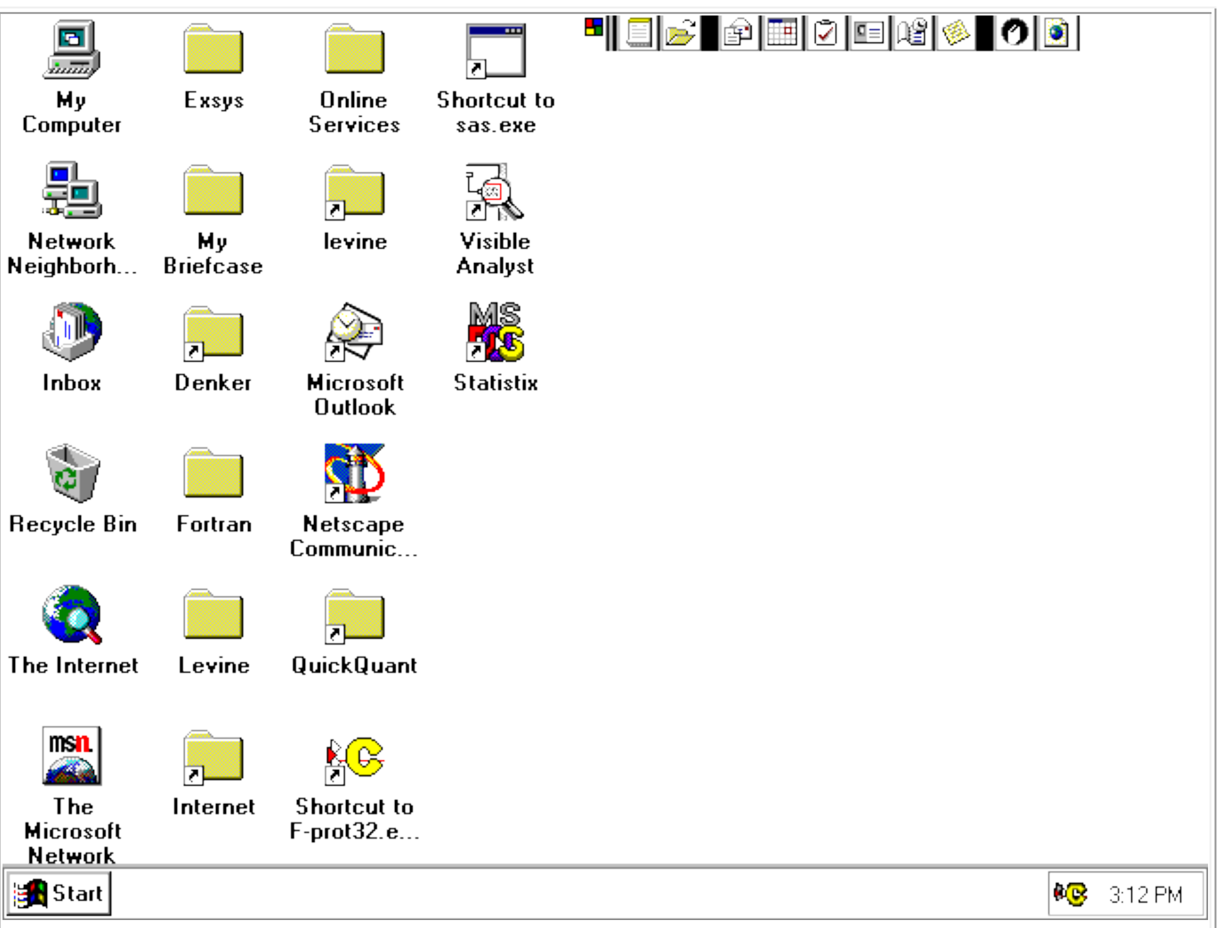
In any database application, each of the tables requires a means to get data into them and retrieve the data at a later time. The primary way to get data into tables is to use data entry forms. The primary ways to get data back out of tables or to display data in tables.

2.7.5 Starting Microsoft Access

As with most Windows 95/98/NT/2000 programs, Access can be executed by navigating the Start menu in the lower left-hand corner of the Windows Desktop. A view of a Windows Desktop is given here:

(Note that your Windows desktop may look slightly different).

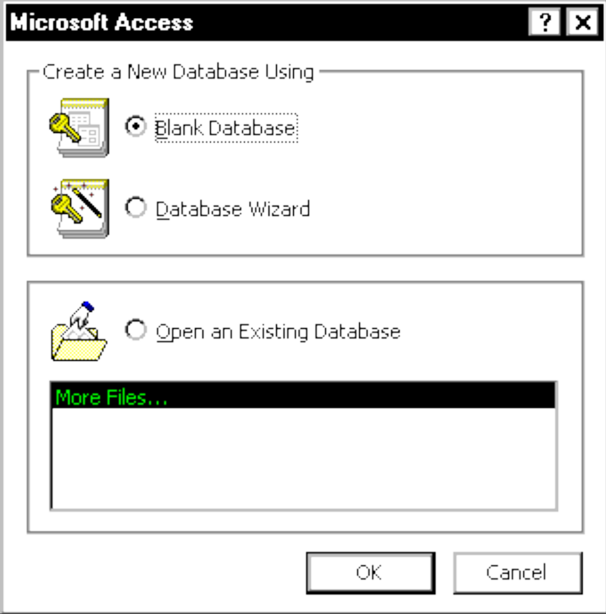
To start Access, click on the Start button, then the Programs menu, then move to the MS-Office menu and finally click on the Microsoft Access menu item.



The MS Office Professional menu is shown below.

Note that this arrangement of menus may vary depending on how MS Office was installed on the PC you are using.

Once Access is running, an initial screen will be displayed:



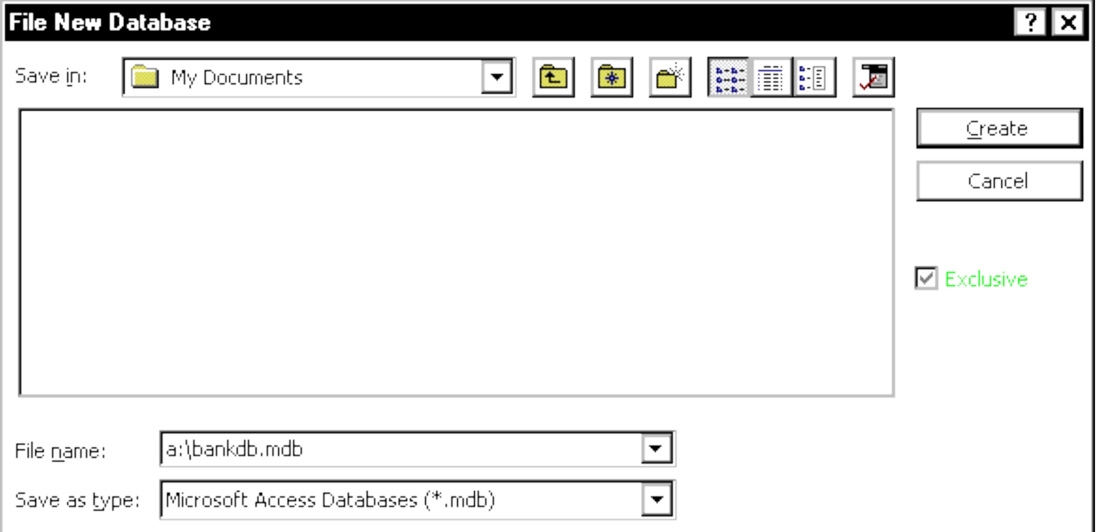
From this initial screen, the user can create a new database (either blank or with some tables created with the database wizard), or open up an existing database.

In general, the first time one begins a project, a new, blank database should be created. After that point, use the *Open existing database* option to re-open the database created previously.

Warning - If you have previously created a database, and then create it again using the same name, you will overwrite any work you have done.

For the purposes of this lesson, if you are going through these steps for the first time, choose the option to create a new, blank database as shown in the above figure.

By selecting **Blank Database** and clicking on the OK button, the following screen will appear. In order to give the new database a file name, fill in *File Name* as a:\bankdb.mdb and click on the Create button to create the database as in the following figure:

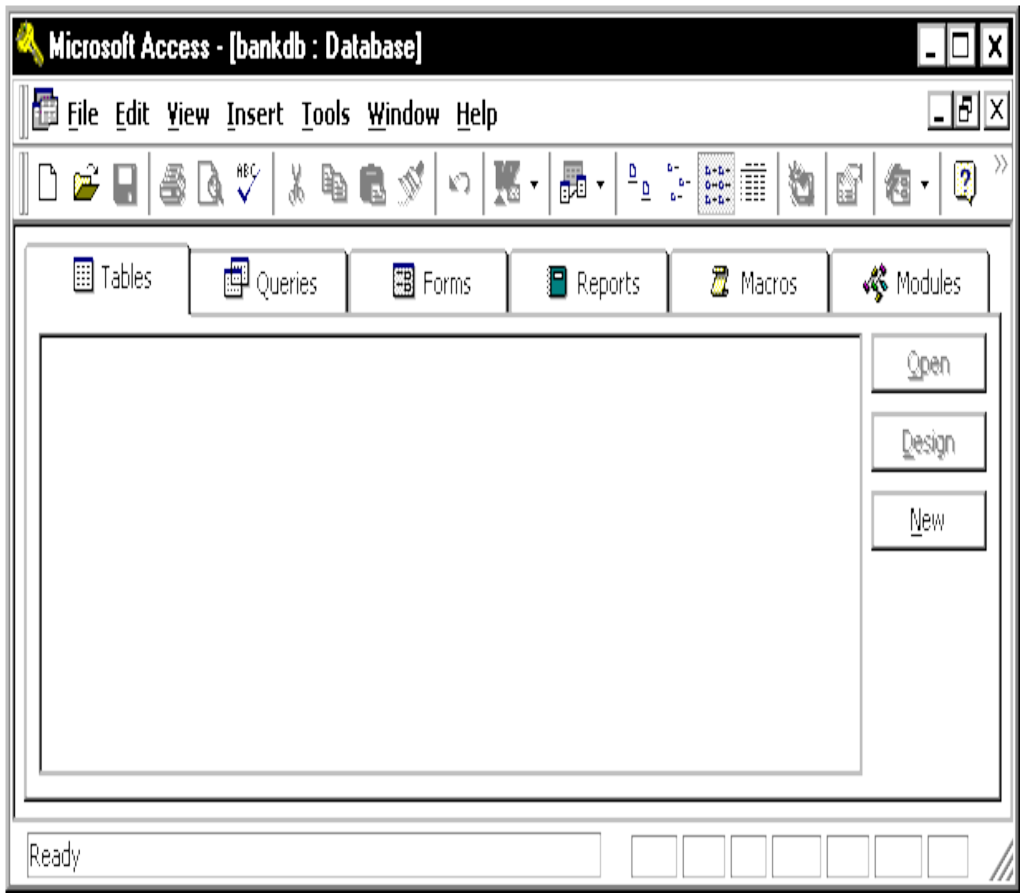


In the above file name, the a:\ indicates that the new database will be created on the A: disk drive. bankdb is the name chosen for this particular database and .mdb is the three letter extension given for *Microsoft DataBase* files.

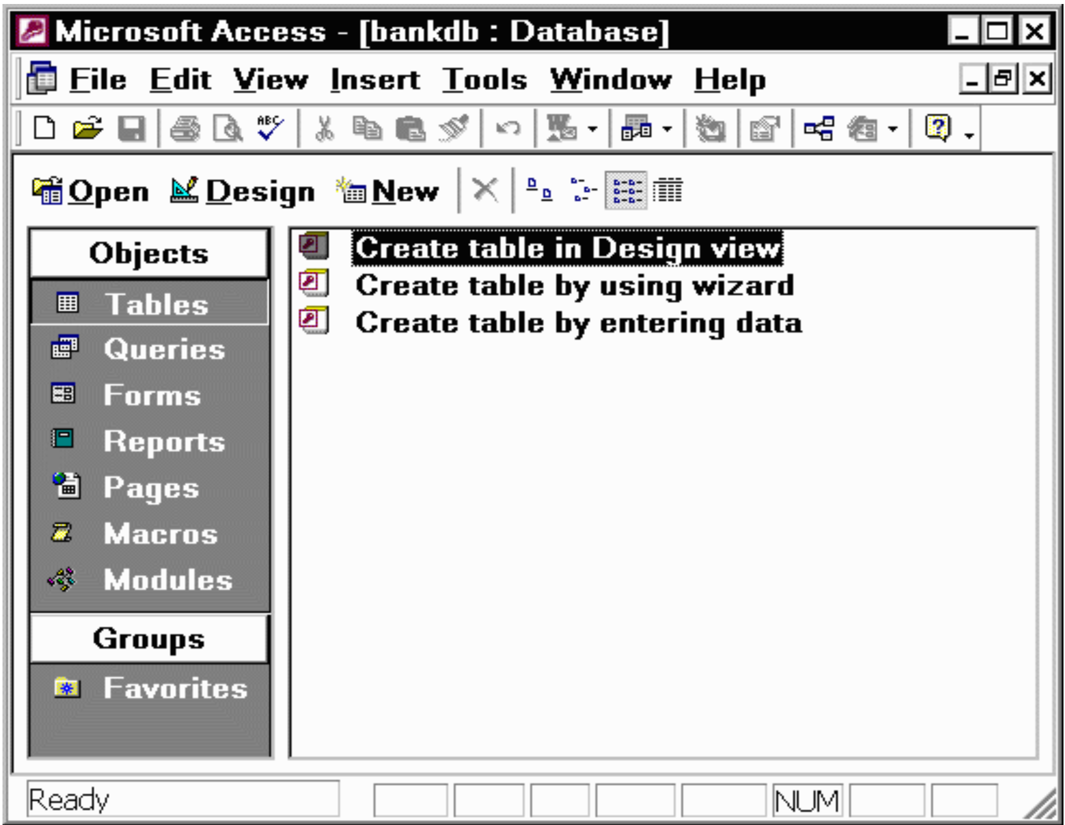
It is advisable to keep the name of the database (bankdb in the above example) relatively short and do not use spaces or other punctuation in the name of the database. Also, the name of the database should reflect the database's contents.

Once the new database is created, the following main Access screen will appear. There are different versions of Ms-Access. Now how it will appear depends upon which version of MS-Access is installed on your system. Following are the appearance of two different version i.e. MS-Access 97 and MS-Access 2000 -

MS-Access 97



MS-Access 2000



The two main features of this main screen are the menu bar that runs along the top of the window and the series of *tabs* in the main window. The menu bar is similar to other Microsoft Office products such as Excel. The menus include:

- File - Menu items to Open, Close, Create new, Save and Print databases and their contents. This menu also has the Exit item to exit Access.
- Edit - Cut, Copy, Paste, Delete
- View - View different database objects (tables, queries, forms, reports)
- Insert - Insert a new Table, Query, Form, Report, etc.
- Tools - A variety of tools to check spelling, create relationships between tables, perform analysis and reports on the contents of the database.
- Window - Switch between different open databases.
- Help - Get help on Access.

The tabs in the main window for the database include:

- Tables - Displays any tables in the database.
- Queries - Displays any queries saved in the database.
- Forms - Displays any forms saved in the database.

- Reports - Displays any reports saved in the database.
- Macros - Displays any macros (short programs) stored in the database.
- Modules - Displays any modules (Visual Basic for Applications procedures) stored in the database.

In MS Access 2000, these tabs appear along the left hand side of the window by default. MS Access 2000 also adds some selections such as Web Pages and Favorites.

2.7.6 Creating and Viewing Tables

Tables are the main units of data storage in Access. Recall that a table is made up of one or more *columns* (or *fields*) and that a given column may appear in more than one table in order to indicate a relationship between the tables.

From the business example discussed earlier, we concluded that two tables would be sufficient to store the data about **Customers** and their bank **Accounts**. We now give the step-by-step instructions for creating these two tables in Access.

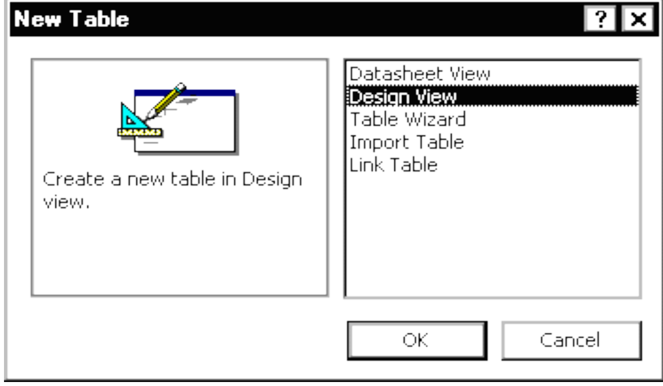
There are a number of ways to create a table in Access. Access provides *wizards* that guide the user through creating a table by suggesting names for tables and columns. The other main way to create a table is by using the *Design View* to manually define the columns (fields) and their data types.

While using the wizards is a fast way to create tables, the user has less control over the column names (fields) and data types. In this lesson, we will describe the steps to create a table using the *Design View*. Students are encouraged to experiment on their own with using the Create Table by using wizard.

Creating a Table Using the Design View

To create a table in Access using the Design View, make sure the Tables tab is displayed (that is, Access should be set to work with tables rather than with queries, forms, reports, etc.) and perform the following steps:

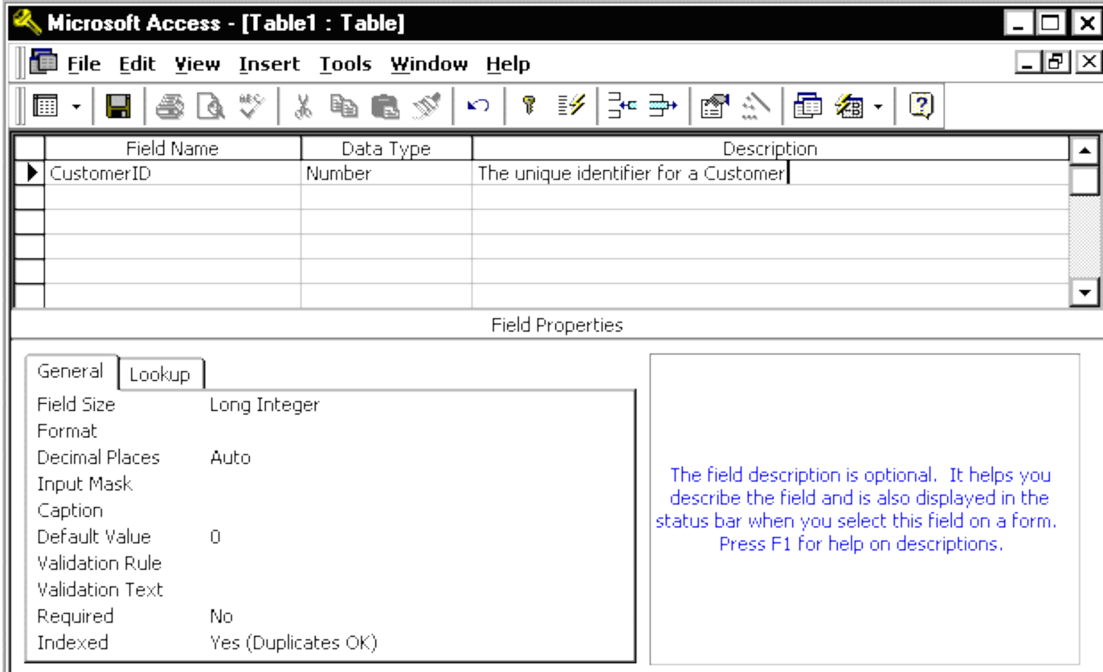
1. For Access '97, Click on the New button and highlight *Design View* in the dialog box that appears:



Then click on the OK button.

For Access 2000, double click on the "Create Table in Design View" item.

2. The Table Design View will appear. Fill in the **Field Name**, **Data Type** and **Description** for each column/field in the table. The CustomerID field is filled in below:

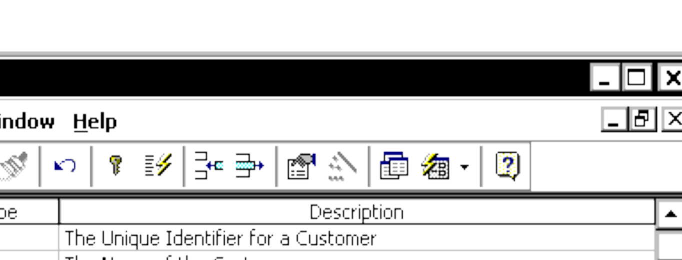


Note that the default name given for the table is Table17. In a later step, we will assign an appropriate name for this table.

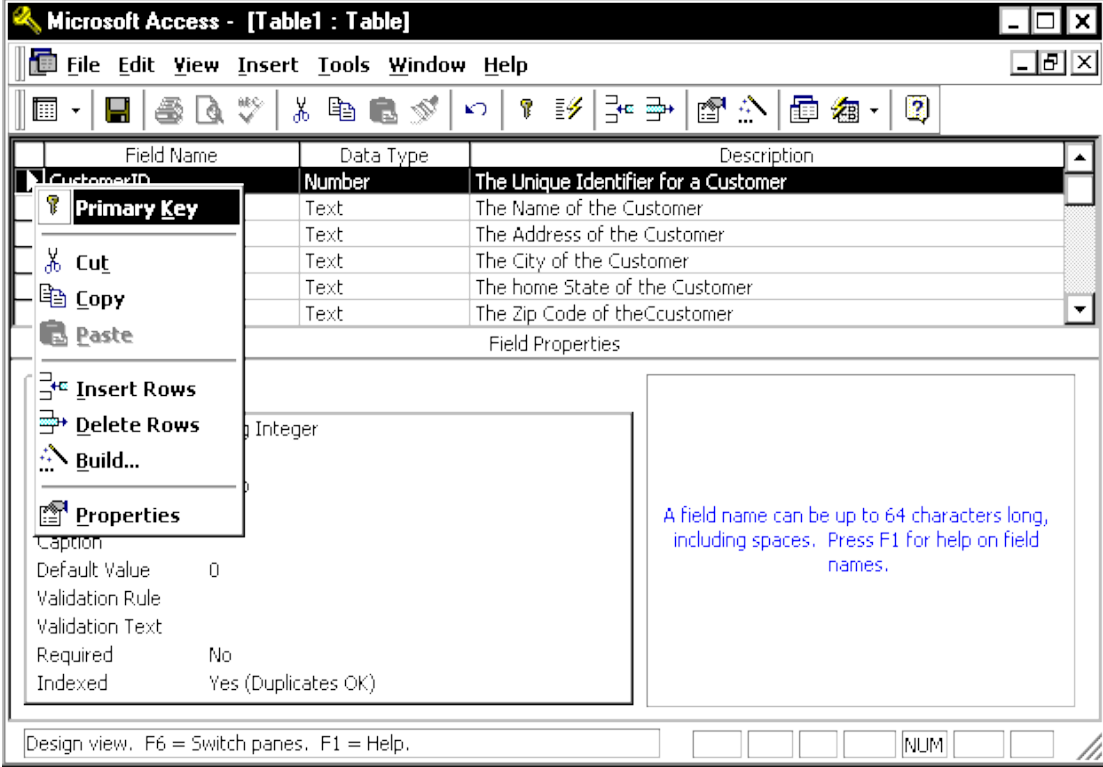
Fill in the information for the fields as follows:

Field Name	**Data Type**	**Description**
CustomerID	Number	The Unique Identifier for a Customer
Name	Text	The Name of the Customer
Address	Text	The Address of the Customer
City	Text	The City of the Customer
State	Text	The home State of the Customer
Zip	Text	The Zip Code of the Customer

A figure showing the design view with the new table definition filled in is given below:



3. Now that all of the fields have been defined for the table, a Primary Key should be defined. Click on the **CustomerID** field with the *Right* mouse button and choose Primary Key from the pop-up menu.

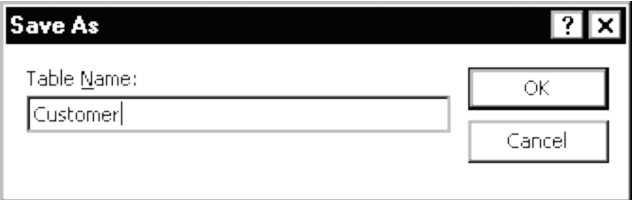


Notice that a small key appears next to the field name on the left side. Note: To remove a primary key, simply repeat this procedure to toggle the primary key off.

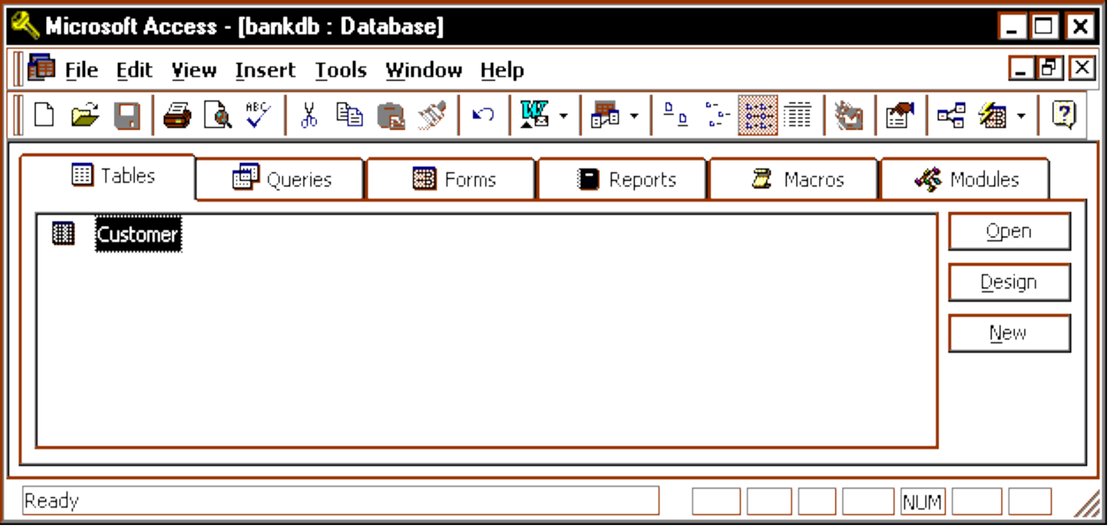
4. As a final step, the table must be saved. Pull down the File menu and choose the Save menu item. A dialog box will appear where the name

of the new table should be specified. Note that Access gives a default name such as **Table1** or **Table2**. Simply type over this default name with the name of the table.

For this example, name the table: **Customer** Then click on the OK button.



At this point, the new Customer table has been created and saved. Switch back to the Access main screen by pulling down the File menu and choosing the Close menu item. This will *close* the Design View for the table and display the Access main screen. Notice that the new Customer table appears below the Table tab.



When defining the fields (columns) for a table, it is important to use field names that give a clear understanding of the data contents of the column. For example, does the field CNO indicate a Customer Number or a Container Number ?

Field names in Access can be up to 64 characters long and may contain spaces. **However, *the use of spaces in field names and table names is strongly discouraged*.** If you wish to make field names easier to read, consider using an underscore character to separate words. However be certain no spaces appear before or after the underscore.

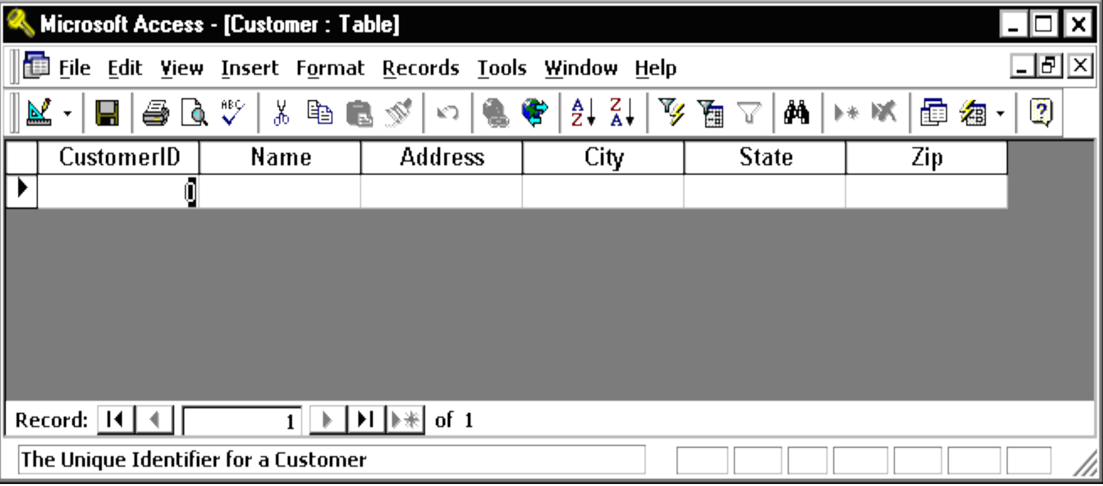
The following table summarizes some different ways to give field names:

Description	**Bad**	**Good**
Unique identifier for a customer	CID	CustomerID or Customer_ID
Description for a product	PDESC	ProductDescription
Employee's home telephone number	Employee_home_telephone_number	HomePhone
Bank account number	BA#	AccountNumber

2.7.7 Viewing and Adding Data to a Table

Data can be added, deleted or modified in tables using a simple spreadsheet-like display. To bring up this view of a single table's data, highlight the name of the table and then click on the Open button.

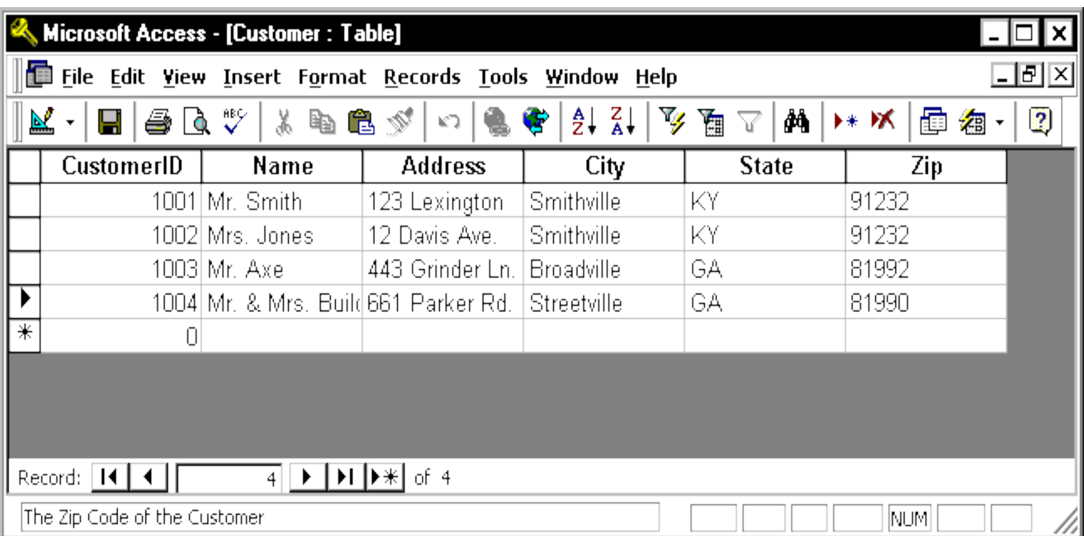
In this view of the table, shown in the figure below, the fields (columns) appear across the top of the window and the rows or records appear below. This view is similar to how a spreadsheet would be designed.



Note at the bottom of the window the number of records is displayed. In this case, since the table was just created, only one blank record appears.

To add data to the table, simply type in values for each of the fields (columns). Press the Tab key to move between fields within a record. Use the up and down arrow keys to move between records. Enter the data as given below:

CustomerID	Name	Address	City	State	Zip
1001	Mr. Smith	123 Lexington	Smithville	KY	91232
1002	Mrs. Jones	12 Davis Ave.	Smithville	KY	91232
1003	Mr. Axe	443 Grinder Ln.	Broadville	GA	81992
1004	Mr. & Mrs. Builder	661 Parker Rd.	Streetville	GA	81990



To save the new data, pull down the File menu and choose Save.

To navigate to other records in the table, use the navigation bar at the bottom of the screen: **Record: 2 of 4**

To modify existing data, simply navigate to the record of interest and tab to the appropriate field. Use the arrow keys and the delete or backspace keys to change the existing data.

To delete a record, first navigate to the record of interest. Then pull down the Edit menu and choose the Delete menu item.

To close the table and return to the Access main screen, pull down the File menu and choose the Close menu item.

2.7.8 Summary

Microsoft Access is a relational database management system (DBMS). Microsoft Access is a powerful program to create and manage your databases. It has many built in features to assist you in constructing and viewing your information. In the above section we learnt about the basics of the MS-Access, how we can create

databases. Further how tables can be created in the database. In the last we learnt how data can be read, modified and deleted from tables.

2.7.9 Self Understanding

Q1. Explain the Features of MS-Access.

Q2. Explain the various components of MS-Access

Q3. List down the steps taking suitable examples how database can be created in MS-Access.

Q4. Explain the steps to be followed in creating the tables in database and how the data can be fed in the tables.

Q5. Explain how the data in the tables can be viewed, modified and deleted.

QUERIES IN MS-ACCESS

Structure:

2.8.0 Introduction
2.8.1 Objectives
2.8.2 Queries in MS-Access
2.8.3 Setting Query Properties
2.8.4 Setting Field Properties
2.8.5 Viewing Results and SQL Equivalent
2.8.6 Creating and Running Queries
2.8.7 Integrity Constraints
2.8.8 Referential Integrity
2.8.9 Join Types
2.8.10 Summary
2.8.11 Self Understanding

2.8.0 Introduction

Queries offer the ability to retrieve and filter data, calculate summaries (totals), and update, move and delete records in bulk. Mastering Microsoft Access queries will improve your ability to manage and understand your data and simplify application development. In the following section we will explain how to retrieve data from database using queries. In addition to it, how to create and run queries after creating queries will be explained. Moreover, how integrity and referential constraints are implemented in the database.

2.8.1 Objectives

After reading this lesson you will be able to learn

- How to create Queries
- How to run queries
- How to set Integrity and referential constraints
- Various types of Joins available in MS-Access

2.8.2 Queries in MS-Access

Microsoft Access is now the most popular Windows database program. A major reason for its success is its revolutionary query interface. Once data is collected in a database, analysis and updates need to be performed. Queries offer the ability to retrieve and filter data, calculate summaries (totals), and update, move and delete records in bulk. Mastering Microsoft Access queries will improve your ability to manage and understand your data and simplify application development.

The visual representation of tables and the graphical links between them makes Microsoft Access queries extremely easy to use. Fortunately, the nice user interface also allows very powerful and advanced analysis. The entire query engine is modeled on SQL systems and allows switching between the graphical query design and SQL syntax. Many Microsoft Access users and developers learned SQL from this feature.

Knowing the many features of Microsoft Access queries will allow you to perform advanced analysis quickly without programming.

Query Types

Microsoft Access supports many types of queries. Here is a description of the major categories:

- **Select Queries**

  Retrieve records or summaries (totals) across records. Also includes cross-tabulations.
- **Make Table Queries**

  Similar to Select queries but results are placed in a new table.
- **Append Queries**

  Similar to Select queries but results are added to an existing table.
- **Update Queries**

  Modify data in the records.
- **Delete Queries**

  Records are deleted from a table.

Select queries are the most common queries and can be used for viewing and a data source for forms, reports, controls, and other queries. The other queries create or change data and are known collectively as Action queries.

Basic Select Queries

The most basic Select queries retrieve the records you specify from a table. You can choose the fields from a table to display, and specify the criteria for selecting records. In the most cases, while viewing the query results you can modify the data and update the original records. These updateable views are extremely powerful.

Selecting Table and Fields

The first step in creating a query is to specify the table or tables to use and the fields to display. Selecting tables is simple. Just choose the table from the list when the query is first created or use the Add Table command from the Query menu. The selected table is placed on the upper portion of the query design window. From there you can select the fields for the query by double clicking on them or selecting several fields (using Shift-Click or Ctrl-Click) and dragging them to the bottom portion: the query by example (QBE) grid. Make sure the Show option is checked to display the field.

Sorting and Reordering Fields

Once the fields are placed on the QBE grid, you can reorder the fields by clicking on the column and dragging it to the place you want. To sort the results, specify the Sort option under the fields to sort. You can choose Ascending or Descending order. Note that you can turn off the Show setting and sort on a field that does not appear in the display.

Renaming Fields

A very nice feature of Microsoft Access queries is the ability to rename fields. You may have stored your data in fields that are not easily understood by users. By using a query expression, you can change the field name the user sees. For instance, a field named "CustID" could be changed to "Customer ID" by placing the new name followed by a colon and the original name in the QBE field cell: Customer ID:[CustID].

Using Calculated Fields (Expressions)

In addition to retrieving fields from a table, a Select query can also display calculations (expressions). Of course, expressions cannot be updated since they do not exist in the original table. Expressions are extremely powerful and allow you to easily display complex calculations. There is an Expression Builder that simplifies the selection of fields and functions. By default, expression fields are named "Expr1", "Expr2", etc.; therefore, you usually want to rename them to something more understandable.

2.8.3 Setting Query Properties

While designing a query, you can choose View | Properties or right click on the top portion of the query and choose Properties to see and modify the query properties.

Description

This property lets you provide a description of the query. This should help you in remembering the purpose of the query.

Output All Fields

This option is usually set to No. If it is changed to Yes, all the fields of all the tables in the query are shown. In general, you should leave this property alone and specify the fields desired in the QBE grid.

Top Values

Rather than retrieving all records, you can specify the top n records or n percent, where n is the value specified here.

Unique Values

By default this is set to No and all records are retrieved. If this is changed to Yes, every record retrieved contains unique values (SQL uses the SELECT DISTINCT command). That is, no retrieved records are identical. For instance, you can run a query for the State field of the Patient table. With this set to No, the result is a record

for each patient. When set to Yes, only the list of unique states is displayed. When set to Yes, the query is not updateable.

Unique Records

By default this is set to Yes and all records are retrieved. For one table queries, this property is ignored. For multi-table queries, if it is set to No, (similar to using a DISTINCTROW in a SQL statement) only the Unique Records and Unique Values properties are linked and only one can be set to Yes (both can be No). When Unique Records is Yes, Unique Values is automatically set to No. When both properties are set to No, all records are returned.

Other Properties

The Other properties are more technical and rarely need to be modified (unless you are using SQL tables). For more information refer to Microsoft Access' on-line help system.

2.8.4 Setting Field Properties

In addition to query properties, each field also has properties that can be set. Move to a field in the QBE grid and right click. Depending on the field type, different properties are available. The most important properties are for numeric and date fields. You can specify how the fields are formatted when the query is run.

2.8.5 Viewing Results and SQL Equivalent

Once the query is completed, you can view its results by switching from Design to DataSheet view. You can also view the SQL equivalent. You can even edit the SQL syntax directly and view the results and/or switch to Design view.

Setting Criteria

The bottom section of the QBE grid is several rows for Criteria. These are optional entries to specify which records are retrieved. If you want all the Patients from the state of Virginia, just enter "VA" in the State's criteria. To further narrow the scope, you can enter criteria for several fields.

Multi-Field Query Criteria

Entering criteria on the same row for several fields performs an AND query between the fields. That is, records that match the criteria in field 1 AND the criteria in field 2, etc. are retrieved. If criteria is placed in different rows, an OR query is performed: retrieve all records matching criteria in field 1 OR criteria in field 2, etc.

Criteria Types

The simplest criteria is the exact match. Just enter the value desired in the field's criteria section. Remember that by using the Show option to eliminate the field from the display, you can specify criteria in fields the user never sees.

<>, <, >, Between .. And ..

You can also retrieve records where a field does not have a particular value by using "< >" followed by the value you don't want. Similarly, you can use >, <, >=, or

<= for ranges. To select records with values between two values, use the BETWEEN .. AND .. syntax.

Nulls

To select records with Null values, enter Is Null. The opposite is Not Is Null. For text fields, remember that zero length strings ("") are not nulls.

OR and IN(.., .., ..)

To select records where a field can have one of several values, use the OR command. You can simply say: "MD" or "DC" or "VA". Alternatively, the IN command performs the same function: IN("MD", "DC", "VA"). The second syntax is easier if you have many values. Of course, if you have a very large number of values, it is better to keep those values in a table and link your query to it. That is easier to maintain than OR or IN clauses inside queries.

Wildcard Searches

Sometimes, you need to search for a particular letter or digit. Combined with the Like command, wildcards let you specify such criteria. These are the wildcard characters Microsoft Access uses:

- ? Single Character
- * Any number of Characters
- # Single Digit
- [..] Character List
- [!..] not in Character List

For instance, if you are interested in a text field where the second letter is "a", the criteria would be: Like "?a*". If you were seeking values where the second letter could be an "a" or "e", the criteria would be: Like "?[ae]*". The opposite of this (all values that do not have "a" or "e" as the second letter) is performed by adding an "!": Like "?[!ae]*". Finally, to select a range of letters (say "a" through "e"), add a dash between the letters: Like "?[a-e]*".

To search for a wildcard character, enclose the value in brackets. For instance, to find values that end in a question mark, use this: Like "*[?]"

2.8.6 Creating and Running Queries

Queries are a fundamental means of accessing and displaying data from tables. Queries can access a single table or multiple tables. Examples of queries for our bank database might include:

- Which Customers live in Georgia ?
- Which Accounts have less than a $500 balance ?

In this section, we show how to use the Access Wizards to create queries for a single table and for multiple tables.

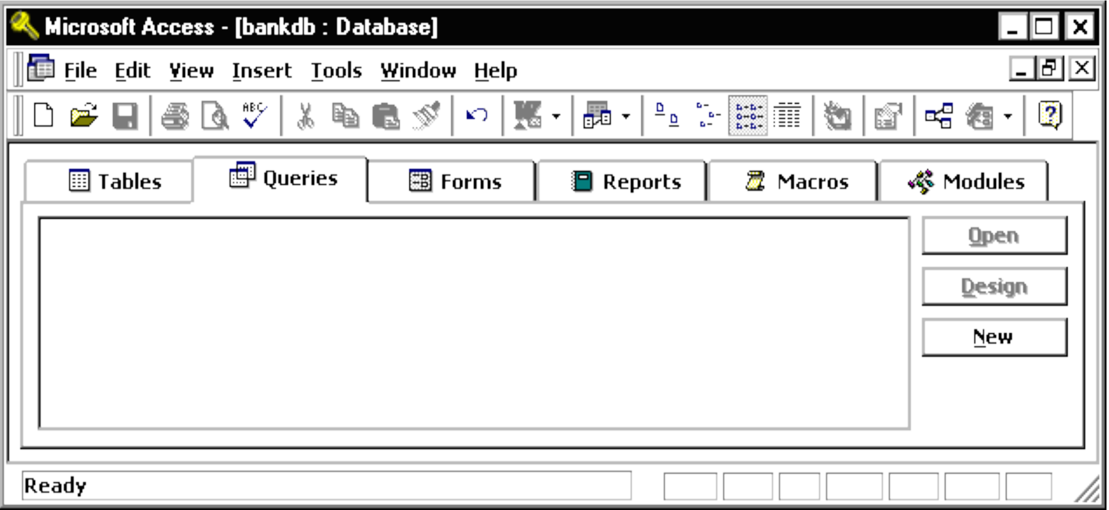
Single Table Queries

In this section, we demonstrate how to query a single table. Single table queries are useful to gain a view of the data in a table that:

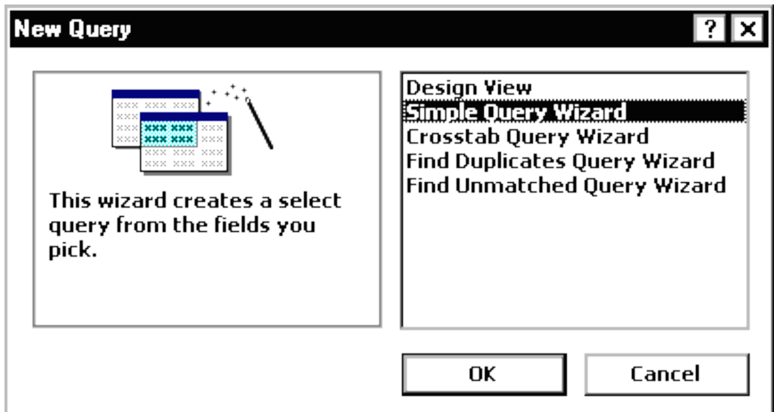
- only displays certain fields (columns) in the output
- sorts the records in a particular order
- performs some statistics on the records such as calculating the sum of data values in a column or counting the number of records, or
- filters the records by showing only those records that match some criteria. For example, show only those bank customers living in GA.

Creating a query can be accomplished by using either the query design view or the Query wizard. In the following example, we will use the query wizard to create a query.

Queries are accessed by clicking on the **Queries** tab in the Access main screen. This is shown below:



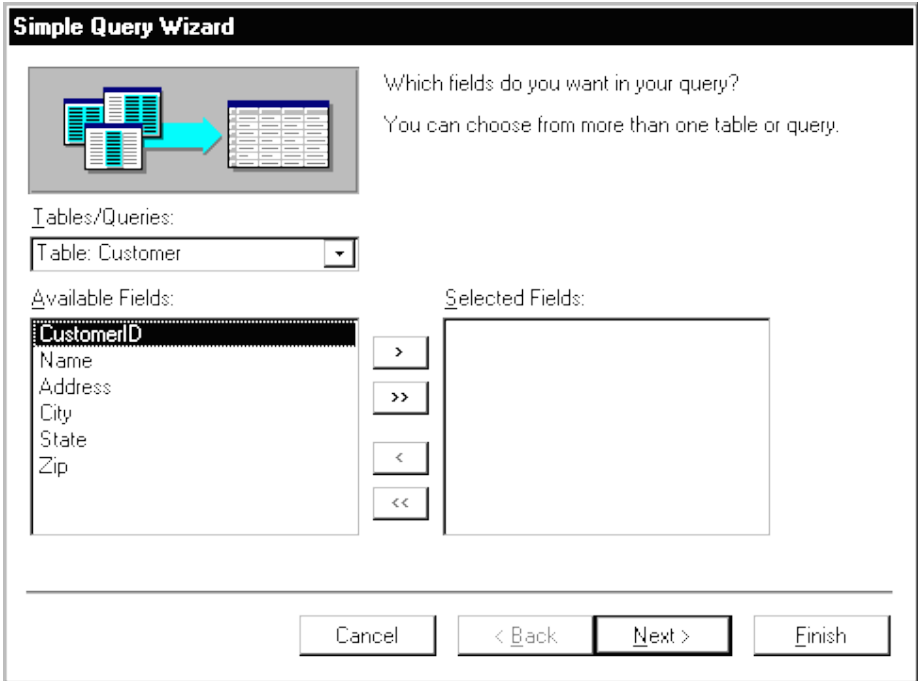
To create a new query, click on the New button. The New Query menu will appear as below. Select the Simple Query wizard option and click the OK button.



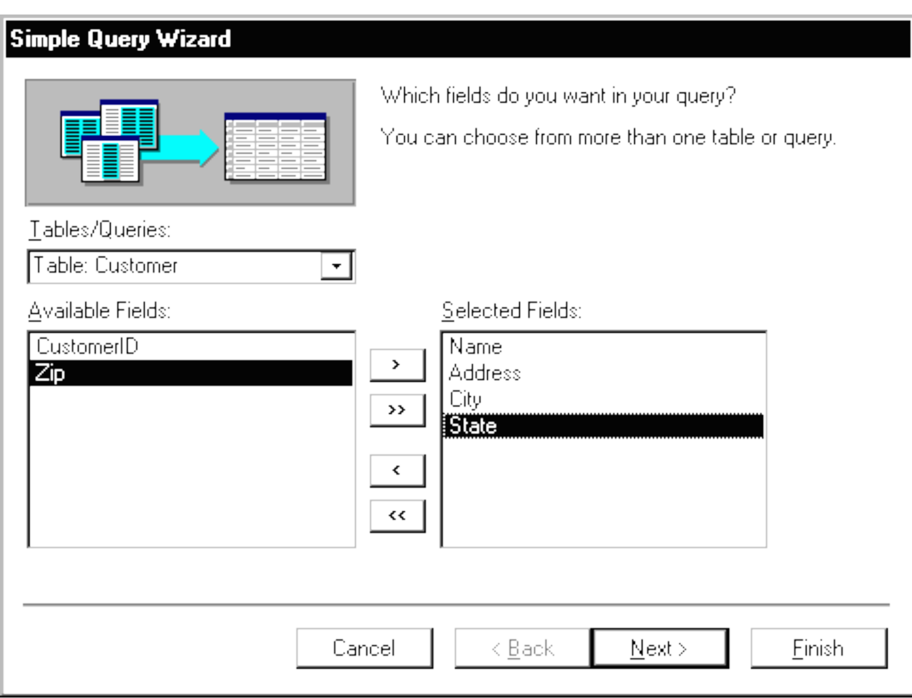
The first step in the Simple Query wizard is to specify the table for the query and which fields (columns) should be displayed in the query output. Three main sections of this step are:

1. Tables/Queries - A pick list of tables or queries you have created.
2. Available Fields - Those fields from the table that can be displayed.
3. Selected Fields - Those fields from the table that *will* be displayed.

For this example, pull down the Tables/Queries list and choose the Customer table. Notice that the available fields change to list only those fields in the Customer table. This step is shown below:



From the list of Available fields on the left, move the Name, Address, City and State fields over to the Selected Fields area on the right. Highlight one of the fields and then click on the right arrow button in the center between the two areas. Repeat this for each of the four fields to be displayed. When done with this step, the wizard should appear as below:

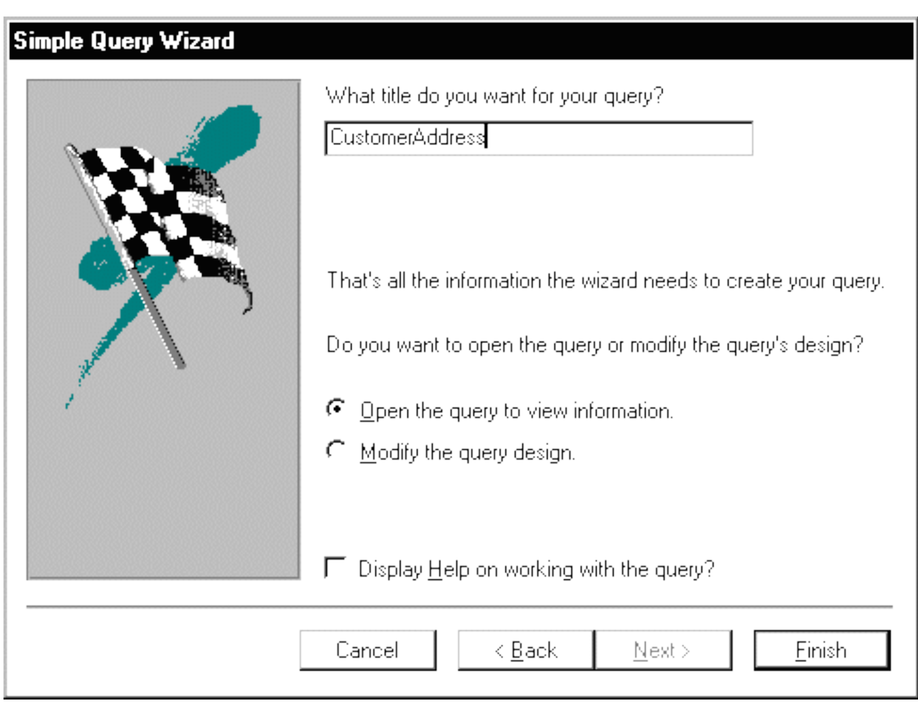


Click on the Next button to move to the next and final step in the Simple Query wizard.

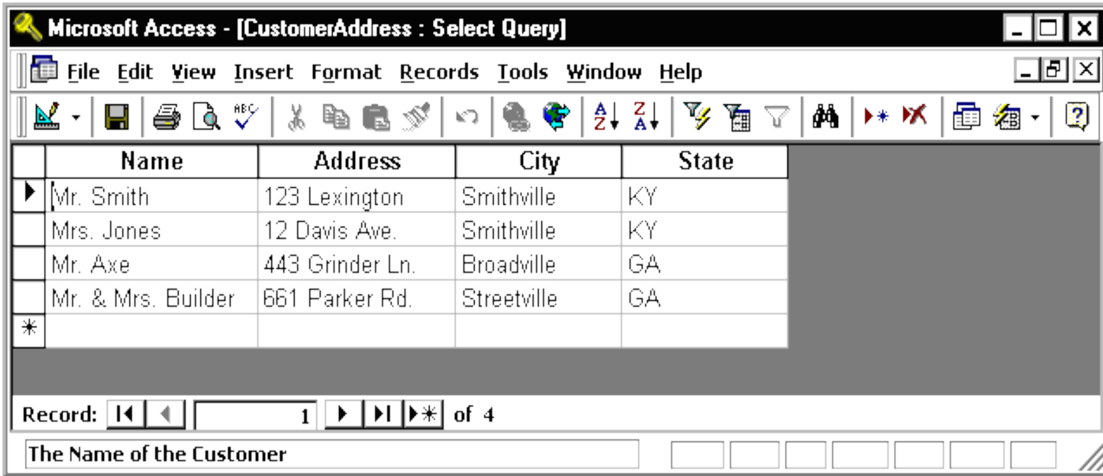
In the final step, give your new query a name. For this example, name the query: Customer Address

At this point, the wizard will create the new query with the option to either:

- Open the query to view information - that is, the wizard will execute the query and show the data.
- Modify the query design - the wizard will switch to the Design View to allow further modification of the query.



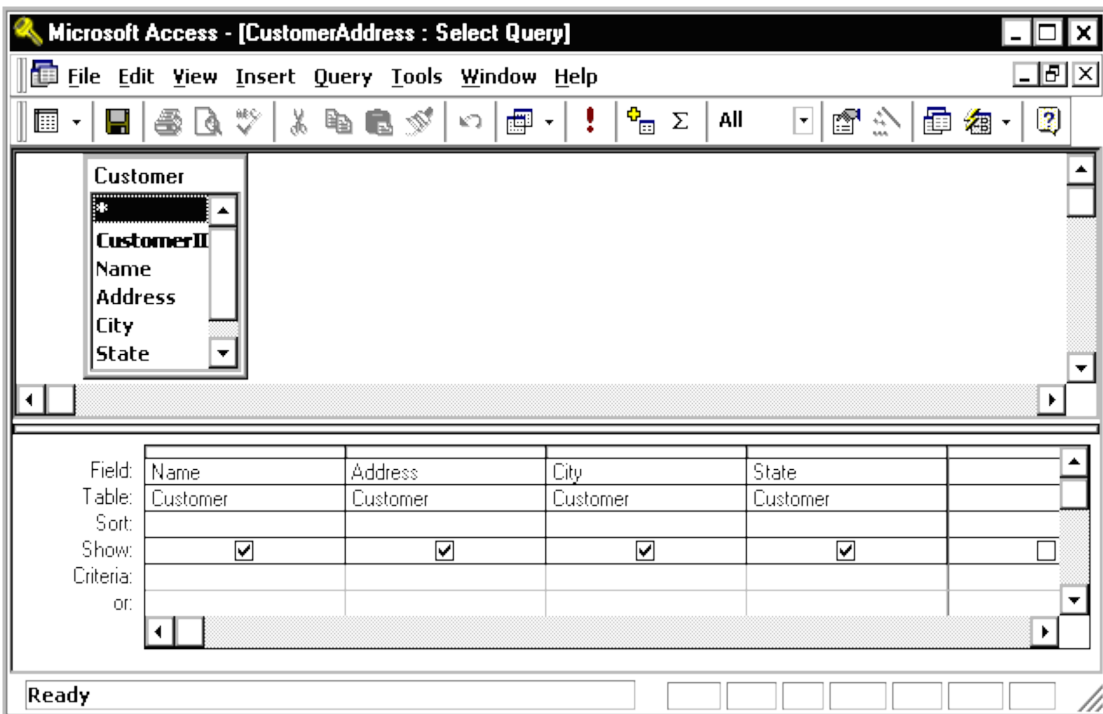
For this example, choose Open the query to view information and click on the Finish button. When this query executes, only the customer's name, address, city and state fields appear, however, all of the rows appear as shown in the figure below:



Close this query by pulling down the File menu and choosing the Close menu item. The Access main screen showing the Queries tab should appear. Note the new query CustomerAddress appears under the Queries tab.

In the following example, we will modify the CustomerAddress query to only display customers in a certain state. To accomplish this, we will make use of the Query Design View.

Open up the CustomerAddress query in the design view by highlighting the name of the query and clicking on the Design button. The design view will appear as in the figure below:



The Query Design view has two major sections. In the top section, the table(s) used for the query are displayed along with the available fields. In the bottom section, those fields that have been selected for use in the query are displayed.

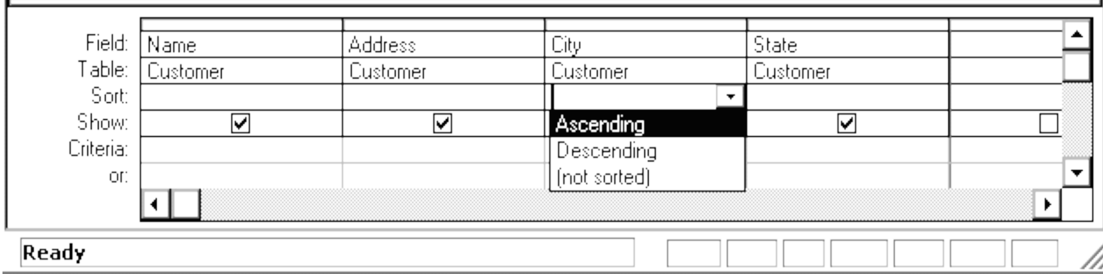
Each field has several options associated with it:

- Field - The name of the field from the table
- Table - The table the field comes from
- Sort - The order in which to sort on this field (Ascending, Descending or Not Sorted)
- Show - Whether or not to display this field in the query output

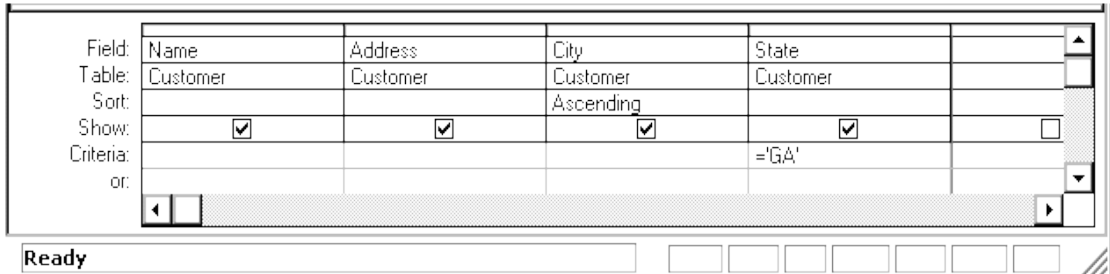
- Criteria - Indicates how to filter the records in the query output.

For this example, we will filter the records to only display those customers living in the State of Georgia (GA). We will also sort the records on the City field.

To sort the records on the **City** field, click in the Sort area beneath the **City** field. Choose Ascending from the list as shown in the figure below:

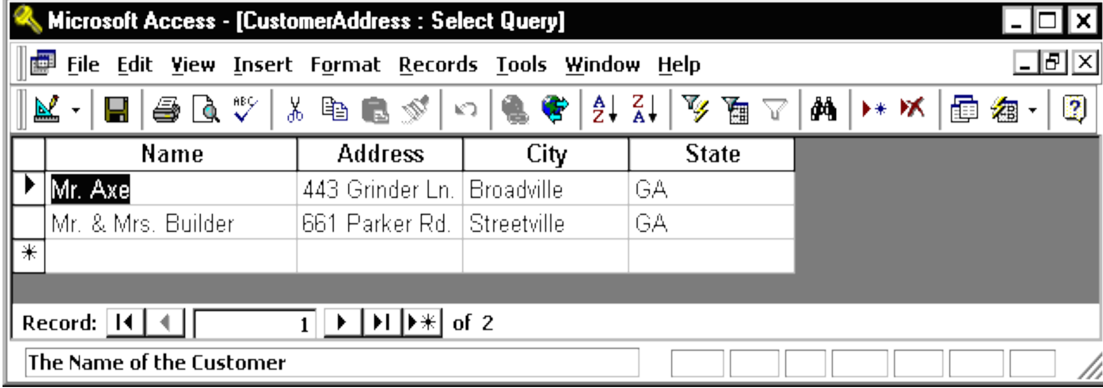


To filter the output to only display Customers in Georgia, click in the Criteria area beneath the **State** field and type the following statement:= 'GA'



The = 'GA' statement tells Access to only show those records where the value of the **State** field is equal to 'GA'.

Run the query by pulling down the Query menu and choosing the Run menu item. The output is shown in the figure below:



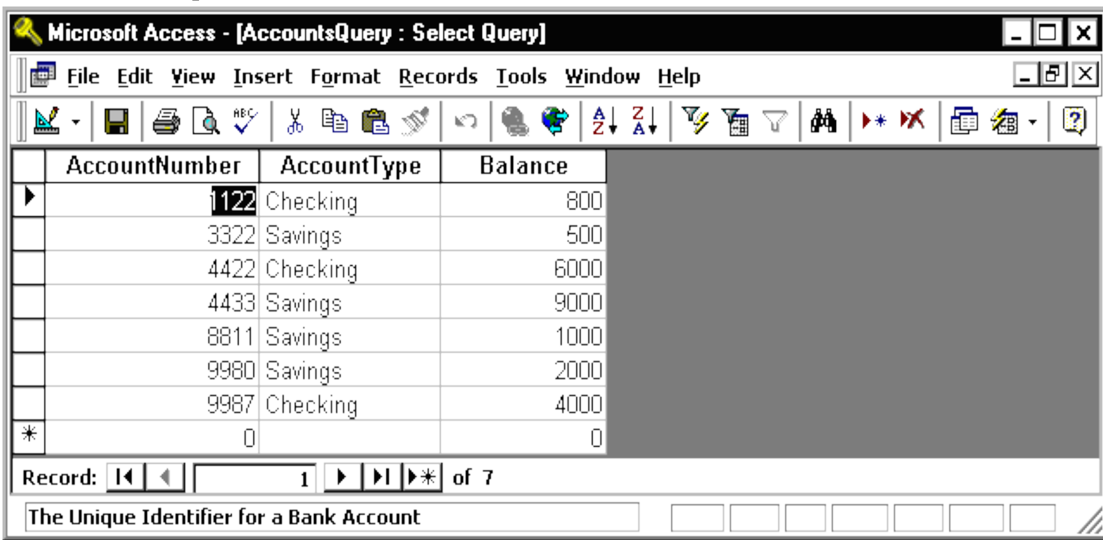
Finally, save and close this query to return to the Access main screen.

Exercise: Single Table Queries

For this exercise, use the Simple Query wizard to create a query on the Accounts table showing just the AccountNumber, AccountType and Balance fields.

1. From the Access main screen, click on the Queries tab. Then click on the New button.
2. Choose the Simple Query wizard option and click on the OK button.
3. Under Table/Queries: choose the Accounts table. Then move the AccountNumber, AccountType and Balance fields over to the Selected fields area. Then click the Next button.
4. In the next panel, you will be asked to choose between a detail or summary query. Choose detailed query and click on the Next button.
5. Name the new Query : AccountsQuery and click on the Finish button.

The output is shown below:



AccountNumber	AccountType	Balance
1122	Checking	800
3322	Savings	500
4422	Checking	6000
4433	Savings	9000
8811	Savings	1000
9980	Savings	2000
9987	Checking	4000
0		0

Close this query by pulling down the File menu and choosing Close.

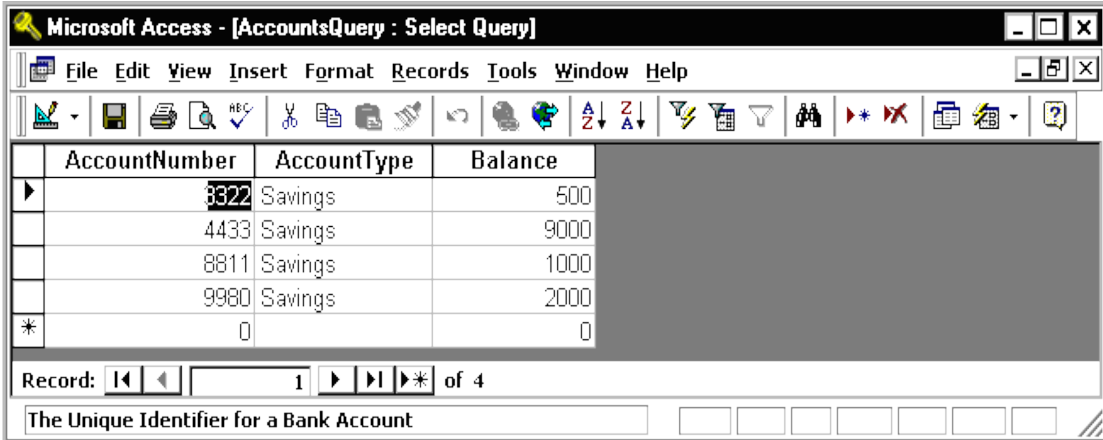
In the next part of the exercise, we will modify the query to sort the output on the account number and only display the Savings accounts.

1. From the Queries tab on the Access main screen, highlight the AccountsQuery and click on the Design button.
2. Change the Sort order for the **AccountNumber** field to Ascending. Add the following statement to the Criteria: are under the **AccountType** field:
3. = 'Savings'

Field:	AccountNumber	AccountType	Balance		
Table:	Accounts	Accounts	Accounts		
Sort:	Ascending				
Show:	☑	☑	☑	☐	☐
Criteria:		='Savings'			

Ready

4. Run the query by pulling down the Query menu and choosing the Run menu item. The output is shown below:



AccountNumber	AccountType	Balance
3322	Savings	500
4433	Savings	9000
8811	Savings	1000
9980	Savings	2000
0		0

Finally, save and close the query to return to the Access main screen.

2.8.7 Integrity constraints

Now we describe how to define relationships in a Microsoft Access database.

What Are Table Relationships

In a relational database, relationships enable you to prevent redundant data. For example, if you are designing a database that will track information about books, you might have a table called Titles that stores information about each book, such as the book's title, date of publication, and publisher. There is also information you might want to store about the publisher, such as the publisher's phone number, address, and zip code. If you were to store all of this information in the titles table, the publisher's phone number would be duplicated for each title that the publisher prints. A better solution is to store the publisher information only once in a separate table, Publishers. You would then put a pointer in the Titles table that references an entry in the Publishers table.

To make sure that your data is not out of sync, you can enforce referential integrity between the Titles and Publishers tables. Referential integrity relationships help ensure that information in one table matches information in another. For example, each title in the Titles table must be associated with a specific publisher in the Publishers table. A title cannot be added to the database for a publisher that does not exist in the database.

Types of Table Relationships

A relationship works by matching data in key columns, usually columns with the same name in both tables. In most cases, the relationship matches the primary key from one table, which provides a unique identifier for each row, with an entry in the foreign key in the other table. For example, sales can be associated with the specific titles sold by creating a relationship between the title_id column in the Titles table (the primary key) and the title_id column in the Sales table (the foreign key).

There are three types of relationships between tables. The type of relationship that is created depends on how the related columns are defined.

One-To-Many Relationships

A one-to-many relationship is the most common type of relationship. In this type of relationship, a row in table A can have many matching rows in table B, but a row in table B can have only one matching row in table A. For example, the Publishers and Titles tables have a one-to-many relationship: each publisher produces many titles, but each title comes from only one publisher. A one-to-many relationship is created if only one of the related columns is a primary key or has a unique constraint. In Access, the primary key side of a one-to-many relationship is denoted by a key symbol. The foreign key side of a relationship is denoted by an infinity symbol.

Many-To-Many Relationships

In a many-to-many relationship, a row in table A can have many matching rows in table B, and vice versa. You create such a relationship by defining a third table, called a junction table, whose primary key consists of the foreign keys from both table A and table B. For example, the Authors table and the Titles table have a many-to-many relationship that is defined by a one-to-many relationship from each of these tables to the TitleAuthors table. The primary key of the TitleAuthors table is the combination of the au_id column (the authors table's primary key) and the title_id column (the Titles table's primary key).

One-To-One Relationships

In a one-to-one relationship, a row in table A can have no more than one matching row in table B, and vice versa. A one-to-one relationship is created if both of the related columns are primary keys or have unique constraints.

This type of relationship is not common because most information related in this way would be all in one table. You might use a one-to-one relationship to:

- Divide a table with many columns.
- Isolate part of a table for security reasons.
- Store data that is short-lived and could be easily deleted by simply deleting the table.
- Store information that applies only to a subset of the main table.

In Access, the primary key side of a one-to-one relationship is denoted by a key symbol. The foreign key side is also denoted by a key symbol.

Defining Relationships Between Tables

When you create a relationship between tables, the related fields do not have to have the same names. However, related fields must have the same data type unless the primary key field is an AutoNumber field. You can match an AutoNumber field with a Number field only if the **FieldSize** property of both of the matching fields is the same. For example, you can match an AutoNumber field and a Number field if the **FieldSize** property of both fields is Long Integer. Even when both matching fields are Number fields, they must have the same **FieldSize** property setting.

Defining a One-To-Many or One-To-One Relationships

To create a one-to-many or a one-to-one relationship, follow these steps:

1. Close any tables that you have open. You cannot create or modify relationships between open tables.
2. Press F11 to switch to the Database window.
3. On the **Tools** menu, click **Relationships**.
4. If you have not yet defined any relationships in your database, the **Show Table** dialog box is automatically displayed. If you want to add the tables that you want to relate, but the **Show Table** dialog box is not displayed, click **Show Table** on the **Relationships** menu.
5. Double-click the names of the tables that you want to relate, and then close the **Show Table** dialog box. To create a relationship between a table and itself, add that table twice.
6. Drag the field that you want to relate from one table to the related field in the other table. To drag multiple fields, press CTRL, click each field, and then drag them.

   In most cases, you drag the primary key field (which is displayed in bold text) from one table to a similar field (often with the same name) called the foreign key in the other table.
7. The **Edit Relationships** dialog box is displayed. Ensure that the field names displayed in the two columns are correct. You can change them if necessary.

   Set the relationship options if necessary. If you need information about a specific item in the **Edit Relationships** dialog box, click the question mark button, and then click the item. These options will be explained in detail later in this article.
8. Click **Create** to create the relationship.
9. Repeat steps 5 through 8 for each pair of tables that you want to relate. When you close the **Edit Relationships** dialog box, Microsoft Access asks if you want to save the layout. Whether you save the layout or not, the relationships that you create are saved in the database.

NOTE: You can create relationships in queries as well as tables. However, referential integrity is not enforced with queries.

Defining a Many-To-Many Relationships

To create a many-to-many relationship, follow these steps:

1. Create the two tables that will have a many-to-many relationship.
2. Create a third table, called a junction table, and then add to the junction table new fields with the same definitions as the primary key fields from each of the other two tables. In the junction table, the primary key fields function as foreign keys. You can add other fields to the junction table, just as you can to any other table.
3. In the junction table, set the primary key to include the primary key fields from the other two tables. For example, in an TitleAuthors junction table, the primary key would be made up of the OrderID and ProductID fields.

   NOTE: To create a primary key, follow these steps:

   a. Open a table in Design view.
   b. Select the field or fields that you want to define as the primary key. To select one field, click the row selector for the desired field.

      To select multiple fields, hold down the CTRL key, and then click the row selector for each field.
   c. Click **Primary Key** on the toolbar.

   NOTE: If you want the order of the fields in a multiple-field primary key to be different from the order of those fields in the table, click **Indexes** on the toolbar to display the **Indexes** dialog box, and then reorder the field names for the index named **PrimaryKey**.
4. Define a one-to-many relationship between each of the two primary tables and the junction table.

2.8.8 Referential Integrity

Referential integrity is a system of rules that Microsoft Access uses to ensure that relationships between records in related tables are valid, and that you do not accidentally delete or change related data. You can set referential integrity when all of the following conditions are met:

- The matching field from the primary table is a primary key or has a unique index.
- The related fields have the same data type. There are two exceptions. An AutoNumber field can be related to a Number field with a **FieldSize** property setting of Long Integer, and an AutoNumber field with a **FieldSize** property setting of Replication ID can be related to a Number field with a **FieldSize** property setting of Replication ID.
- Both tables belong to the same Microsoft Access database. If the tables are linked tables, they must be tables in Microsoft Access format, and

you must open the database in which they are stored to set referential integrity. Referential integrity cannot be enforced for linked tables from databases in other formats.

The following rules apply when you use referential integrity:

- You cannot enter a value in the foreign key field of the related table that does not exist in the primary key of the primary table. However, you can enter a Null value in the foreign key, specifying that the records are unrelated. For example, you cannot have an order that is assigned to a customer that does not exist, but you can have an order that is assigned to no one by entering a Null value in the CustomerID field.
- You cannot delete a record from a primary table if matching records exist in a related table. For example, you cannot delete an employee record from the Employees table if there are orders assigned to the employee in the Orders table.
- You cannot change a primary key value in the primary table, if that record has related records. For example, you cannot change an employee's ID in the Employees table if there are orders assigned to that employee in the Orders table.

Cascading Updates and Deletes

For relationships in which referential integrity is enforced, you can specify whether you want Microsoft Access to automatically cascade update or cascade delete related records. If you set these options, delete and update operations that would normally be prevented by referential integrity rules are allowed. When you delete records or change primary key values in a primary table, Microsoft Access makes the necessary changes to related tables to preserve referential integrity. If you click to select the **Cascade Update Related Fields** check box when you define a relationship, any time that you change the primary key of a record in the primary table, Microsoft Access automatically updates the primary key to the new value in all related records. For example, if you change a customer's ID in the Customers table, the CustomerID field in the Orders table is automatically updated for every one of that customer's orders so that the relationship is not broken. Microsoft Access cascades updates without displaying any message.

NOTE: If the primary key in the primary table is an AutoNumber field, selecting the **Cascade Update Related Fields** check box will have no effect, because you cannot change the value in an AutoNumber field.

If you select the **Cascade Delete Related Records** check box when you define a relationship, any time that you delete records in the primary table, Microsoft Access automatically deletes related records in the related table. For example, if you delete a customer record from the Customers table, all the customer's orders are automatically

deleted from the Orders table (this includes records in the Order Details table related to the Orders records). When you delete records from a form or datasheet with the **Cascade Delete Related Records** check box selected, Microsoft Access warns you that related records may also be deleted. However, when you delete records using a delete query, Microsoft Access automatically deletes the records in related tables without displaying a warning.

2.8.9 Join Types

There are three join types, as follows:

Option 1 defines an inner join. An inner join is a join where records from two tables are combined in a query's results only if values in the joined fields meet a specified condition. In a query, the default join is an inner join that selects records only if values in the joined fields match.

Option 2 defines a left outer join. A left outer join is a join in which all the records from the left side of the LEFT JOIN operation in the query's SQL statement are added to the query's results, even if there are no matching values in the joined field from the table on the right.

Option 3 defines a right outer join. A right outer join is a join in which all the records from the right side of the RIGHT JOIN operation in the query's SQL statement are added to the query's results, even if there are no matching values in the joined field from the table on the left.

2.8.10 Summary

In this lesson you have learnt how to create and run queries in MS-Access, how integrity and referential constraints can be implemented on tables in MS-Access database.

2.8.11 Self Understanding

Q1. Explain the steps in MS-Access for creating and running queries.

Q2. Explain the various steps for Defining Relationships Between Tables

Q3. What do you mean by Referential Integrity. How it can be implemented in MS-Access. Explain taking suitable example.

Q4. What are the various join types available in MS-Access database.

Lesson No. 2.9 **Author : Vishal Goyal**

INTRODUCTION TO FORMS

Structure:

2.9.0 Introduction
2.9.1 Objectives
2.9.2 Create Form by Using Wizard
2.9.3 Create Form in Design View
2.9.4 Adding Records Using A Form
2.9.5 Editing Forms
2.9.6 Form Control
2.9.7 Sorting and filtering
2.9.8 Summary
2.9.9 Self Understanding

2.9.0 Introduction

A form is nothing more than a graphical representation of a table. You can add, update, and delete records in your table by using a form. Although a form can be given different from a table, they both still manipulate the same information and the same data. Hence, if you change a record in a form, it will be changed in the table also. A form is very good to use when you have numerous fields in a table. This way you can see all the fields in one screen, whereas if you were in the table view (datasheet) you would have to keep scrolling to get the field you desire.

Forms are used as an alternative way to enter data into a database table. In the following sections of this lesson, we will explain how to create forms through wizard and design view. In addition, how to add various controls to forms, filtering and sorting data.

2.9.1 Objectives

After reading this lesson we will be able to

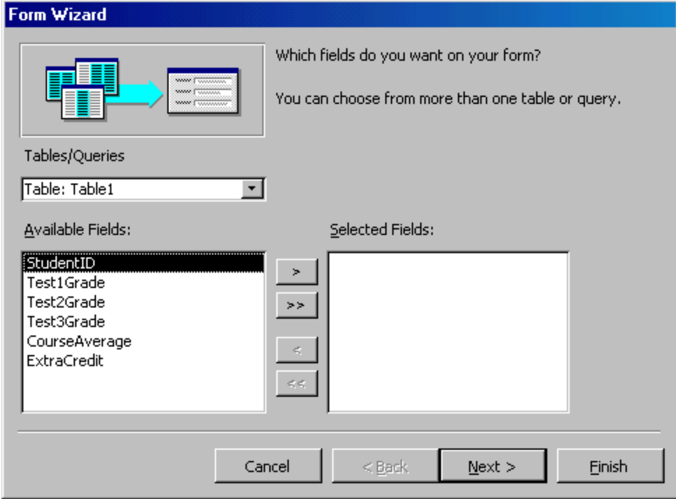
- Create and edit Forms
- How to add records using forms
- Steps for adding various Form Controls to form
- Filter and Sort Data

2.9.2 Create Form by Using Wizard

To create a form using the assistance of the wizard, follow these steps:

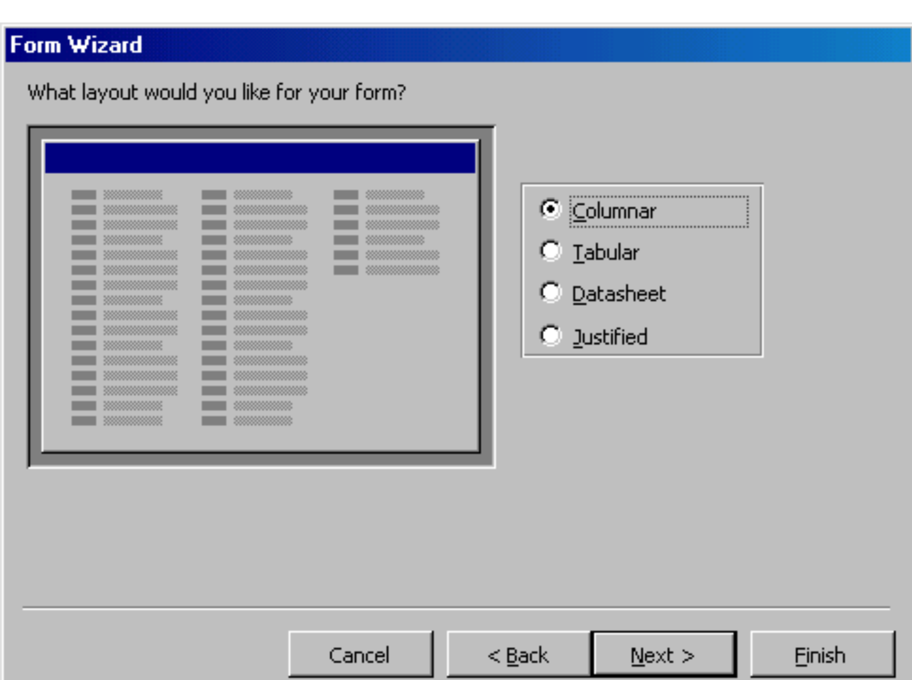
1. Click the Create form by using wizard option on the database window.
2. From the Tables/Queries drop-down menu, select the table or query whose datasheet the form will modify. Then, select the fields that will be included on

the form by highlighting each one on the Available Fields window and clicking the single right arrow button > to move the field to the Selected Fields window. To move all of the fields to Selected Fields, click the double right arrow button >>. If you make a mistake and would like to remove a field or all of the fields from the Selected Fields window, click the left arrow < or left double arrow << buttons. After the proper fields have been selected, click the Next > button to move on to the next screen.

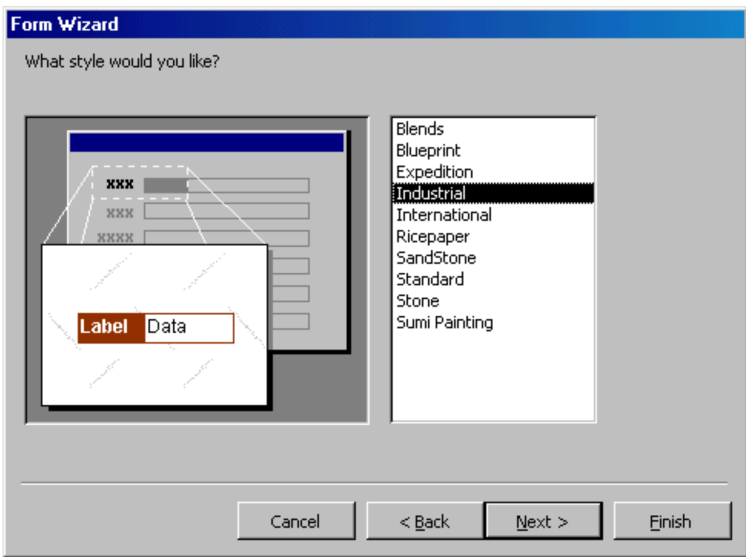


3. On the second screen, select the layout of the form.
   - Columnar - A single record is displayed at one time with labels and form fields listed side-by-side in columns
   - Justified - A single record is displayed with labels and form fields are listed across the screen
   - Tabular - Multiple records are listed on the page at a time with fields in columns and records in rows
   - Datasheet - Multiple records are displayed in Datasheet View

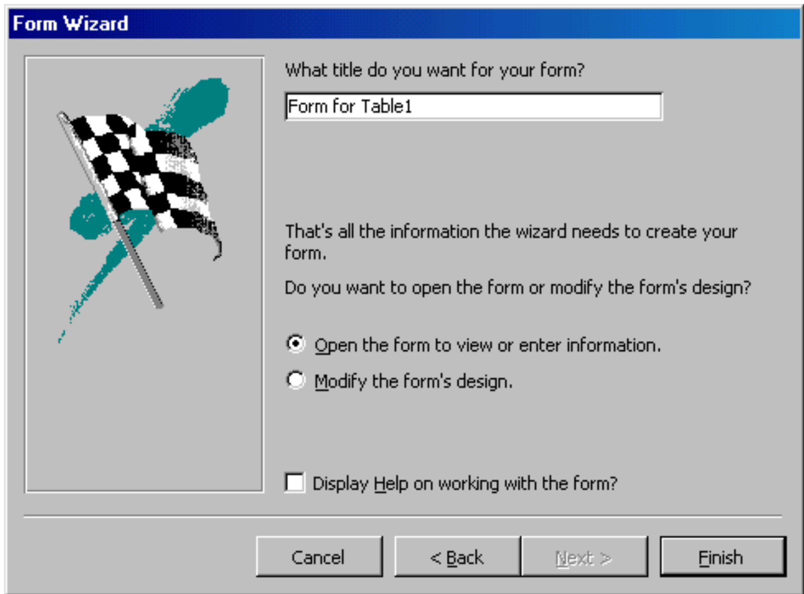
   Click the Next > button to move on to the next screen.



4. Select a visual style for the form from the next set of options and click Next >.



5. On the final screen, name the form in the space provided. Select "Open the form to view or enter information" to open the form in Form View or "Modify the form's design" to open it in Design View. Click Finish to create the form.

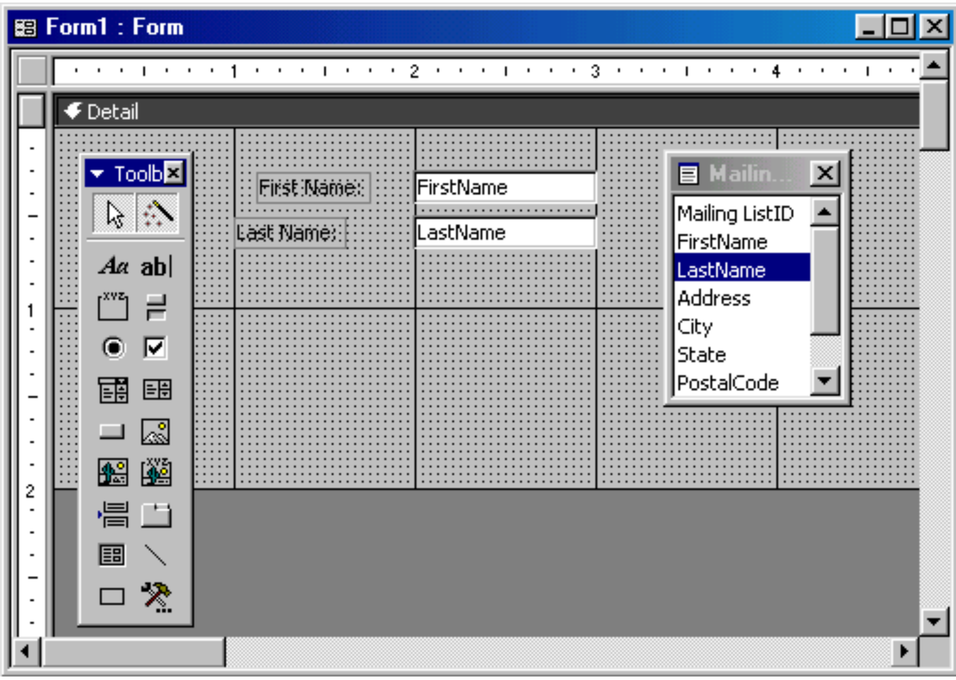


2.9.3 Create Form in Design View

To create a form from scratch without the wizard, follow these steps:

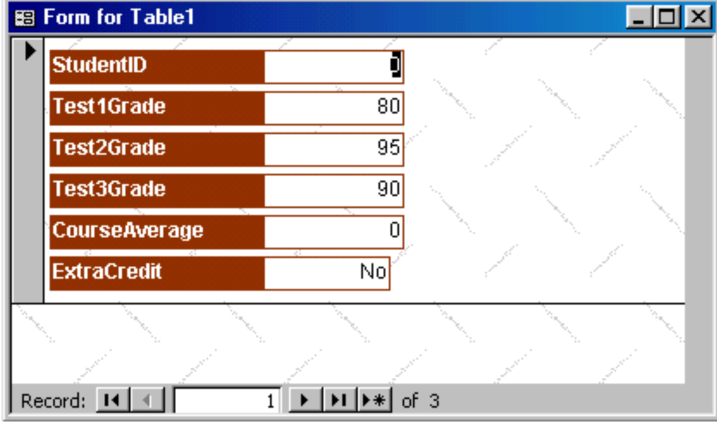
1. Click the New button on the form database window.
2. Select "Design View" and choose the table or query the form will be associated with the form from the drop-down menu.
3. Select View|Toolbox from the menu bar to view the floating toolbar with additional options.

4. Add controls to the form by clicking and dragging the field names from the Field List floating window. Access creates a text box for the value and label for the field name when this action is accomplished. To add controls for all of the fields in the Field List, double-click the Field List window's title bar and drag all of the highlighted fields to the form.



2.9.4 Adding Records Using A Form

Input data into the table by filling out the fields of the form. Press the Tab key to move from field to field and create a new record by clicking Tab after the last field of the last record. A new record can also be created at any time by clicking the New Record button ▸* at the bottom of the form window. Records are automatically saved as they are entered, so no additional manual saving needs to be executed.

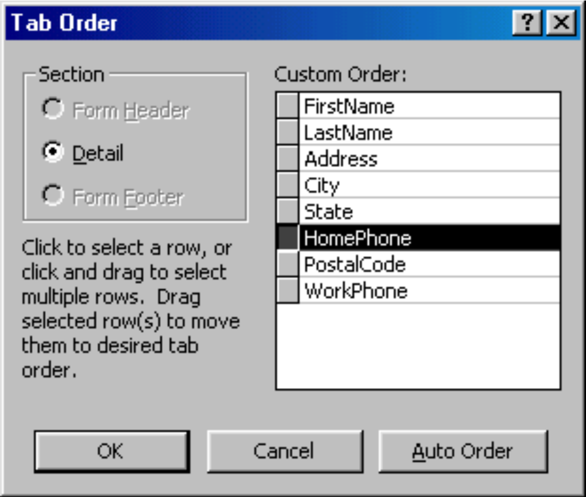


2.9.5 Editing Forms

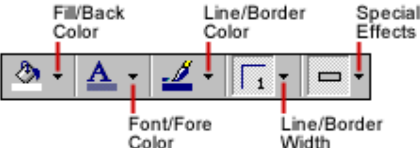
The follow points may be helpful when modifying forms in Design View.

- *Grid lines* - By default, a series of lines and dots underlay the form in Design View so form elements can be easily aligned. To toggle this feature on and off select View|Grid from the menu bar.
- *Snap to Grid* - Select Format|Snap to Grid to align form objects with the grid to allow easy alignment of form objects or uncheck this feature to allow objects to float freely between the grid lines and dots.
- *Resizing Objects* - Form objects can be resized by clicking and dragging the handles on the edges and corners of the element with the mouse.
- *Change form object type* - To easily change the type of form object without having to create a new one, right click on the object with the mouse and select Change To and select an available object type from the list.
- *Label/object alignment* - Each form object and its corresponding label are bounded and will move together when either one is moved with the mouse. However, to change the position of the object and label in relation to each other (to move the label closer to a text box, for example), click and drag the large handle at the top, left corner of the object or label.
- *Tab order* - Alter the tab order of the objects on the form by selecting View|Tab Order... from the menu bar. Click the gray box before the row you would like to

change in the tab order, drag it to a new location, and release the mouse button.



- *Form Appearance* - Change the background color of the form by clicking the Fill/Back Color button on the formatting toolbar and click one of the color swatches on the palette. Change the color of individual form objects by highlighting one and selecting a color from the Font/Fore Color palette on the formatting toolbar. The font and size, font effect, font alignment, border around each object, the border width, and a special effect can also be modified using the formatting toolbar:



- *Page Header and Footer* – Headers and footers added to a form will only appear when it is printed. Access these sections by selecting View|Page Header/Footer on the menu bar. Page numbers can also be added to these sections by selecting Insert|Page Numbers. A date and time can be added from Insert|Date and Time.... Select View|Page Header/Footer again to hide these sections from view in Design View.

2.9.6 Form Control

This section explains the uses for other types of form controls including lists, combo boxes, checkboxes, option groups, and command buttons.

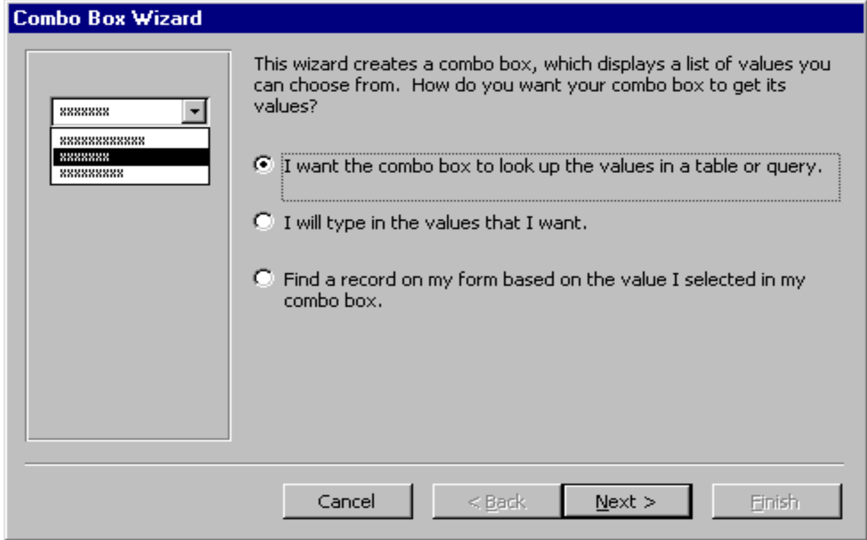
List and Combo Boxes

If there are small, finite number of values for a certain field on a form, using combo or list boxes may be a quicker and easier way of entering data. These two control types

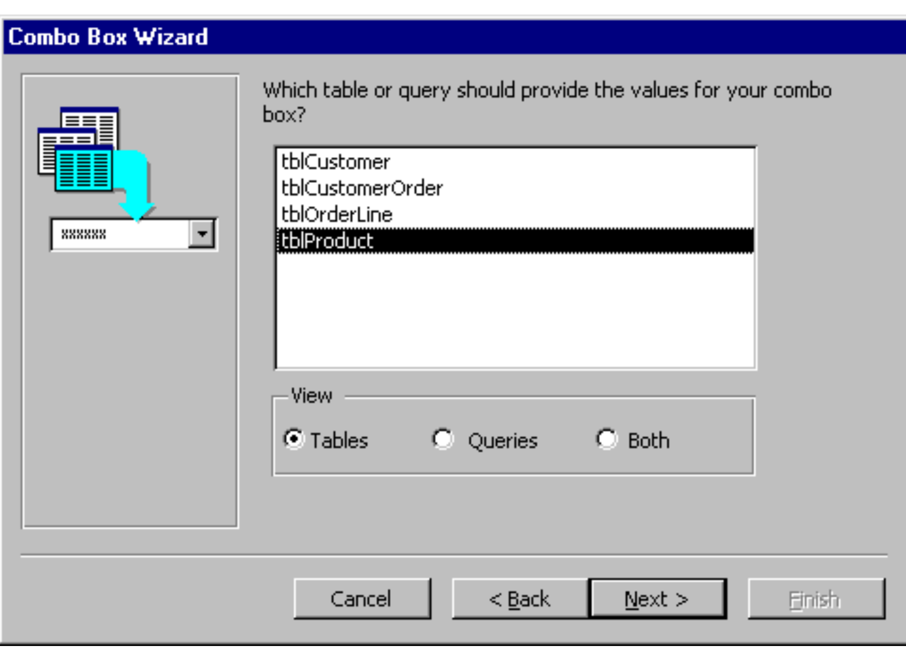
differ in the number of values they display. List values are all displayed while the combo box values are not displayed until the arrow button is clicked to open it as shown in these examples:

By using a combo or list box, the name of the academic building does not need to be typed for every record. Instead, it simply needs to be selected from the list. Follow these steps to add a list or combo box to a form:

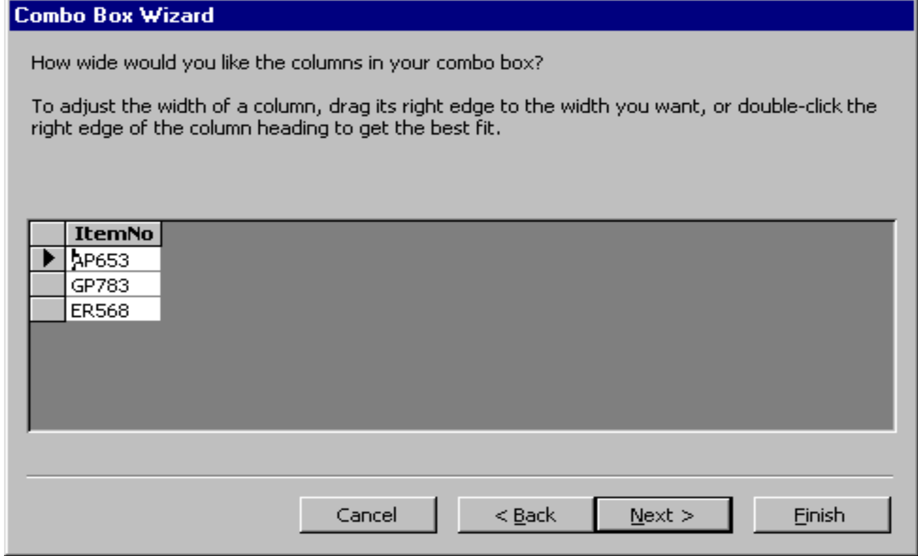
1. Open the form in Design View.
2. Select View|Toolbox to view the toolbox and make sure the "Control Wizards" button is pressed in.
3. Click the list or combo box tool button and draw the outline on the form. The combo box wizard dialog box will appear.
4. Select the source type for the list or combo box values and click Next >.



5. Depending on your choice in the first dialog box, the next options will vary. If you chose to look up values from a table or query, the following box will be displayed. Select the table or query from which the values of the combo box will come from. Click Next > and choose fields from the table or query that was selected. Click Next > to proceed.

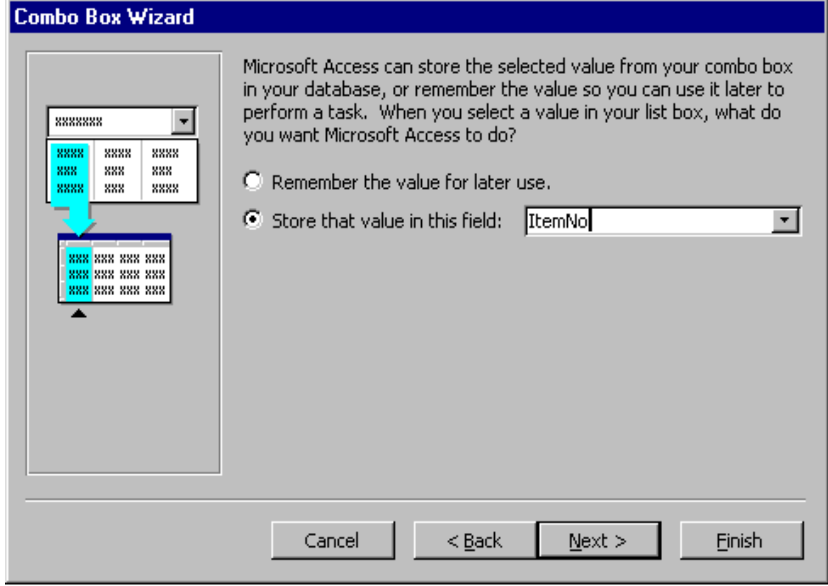


6. On the next dialog box, set the width of the combo box by clicking and dragging the right edge of the column. Click Next >.



7. The next dialog box tells Access what to do with the value that is selected. Choose "Remember the value for later use" to use the value in a macro or procedure (the value is discarded when the form is closed), or select the

field that the value should be stored in. Click Next > to proceed to the final screen.



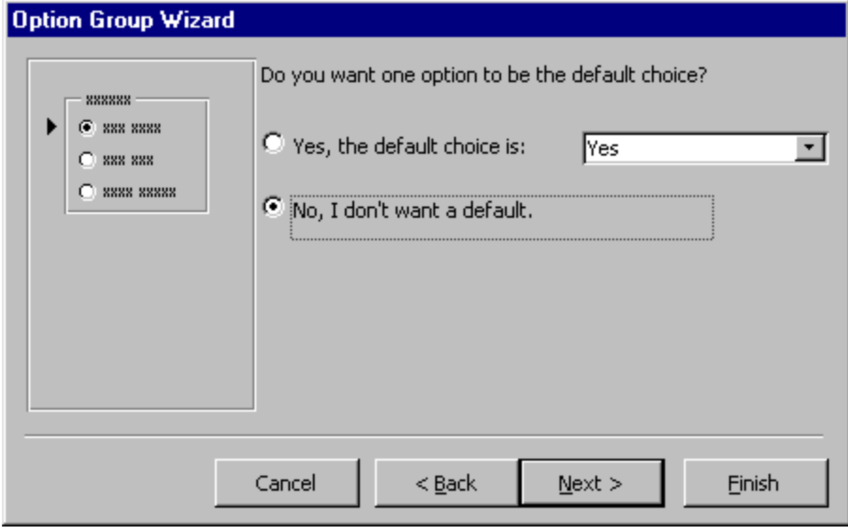
8. Type the name that will appear on the box's label and click Finish.

Check Boxes and Option Buttons

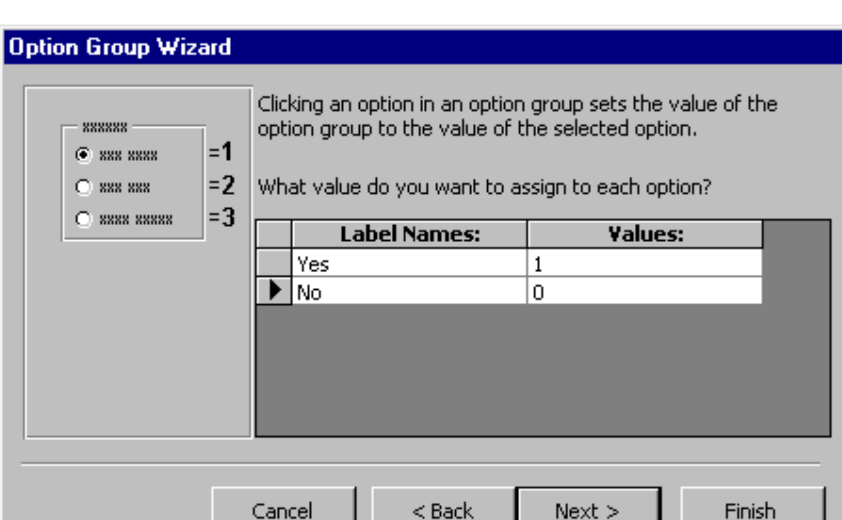
Use check boxes and option buttons to display yes/no, true/false, or on/off values. Only one value from a group of option buttons can be selected while any or all values from a check box group can be chosen. Typically, these controls should be used when five or less options are available. Combo boxes or lists should be used for long lists of options. To add a checkbox or option group:

1. Click the Option Group tool on the toolbox and draw the area where the group will be placed on the form with the mouse. The option group wizard dialog box will appear.
2. On the first window, enter labels for the options and click the tab key to enter additional labels. Click Next > when finished typing labels.

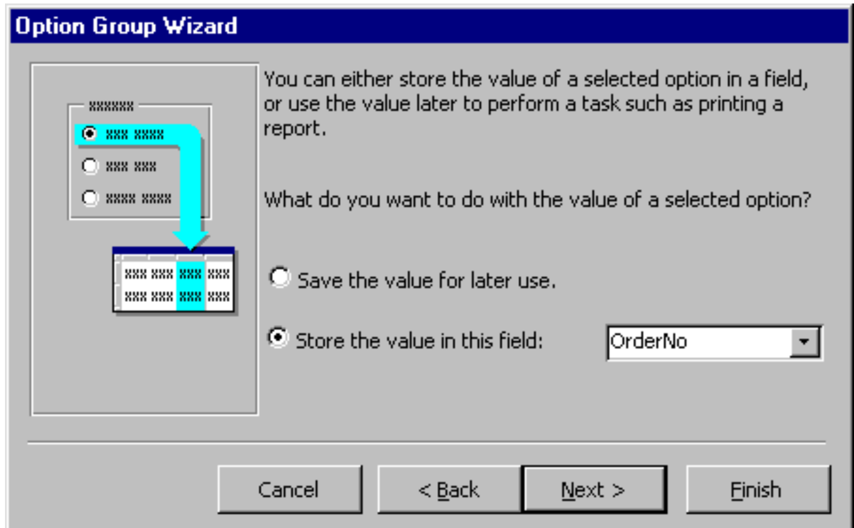
3. On the next window, select a default value if there is any and click Next >.



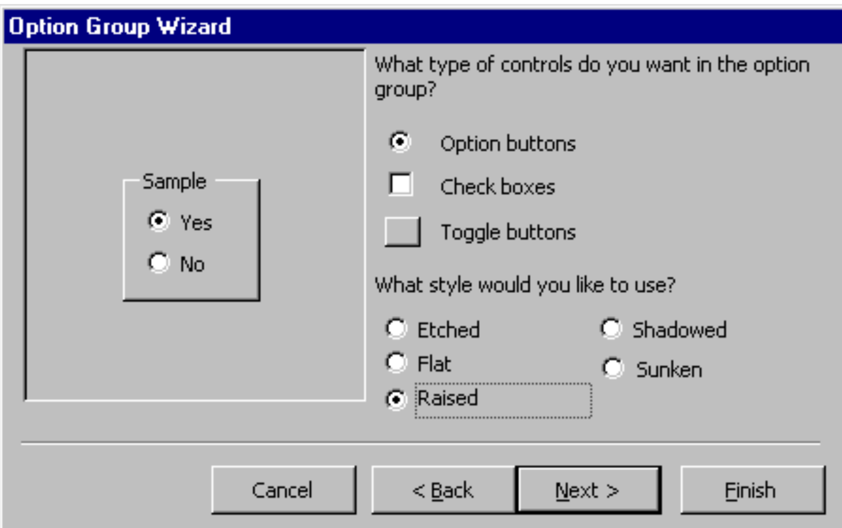
4. Select values for the options and click Next >.



5. Choose what should be done with the value and click Next >.



6. Choose the type and style of the option group and click Next >.

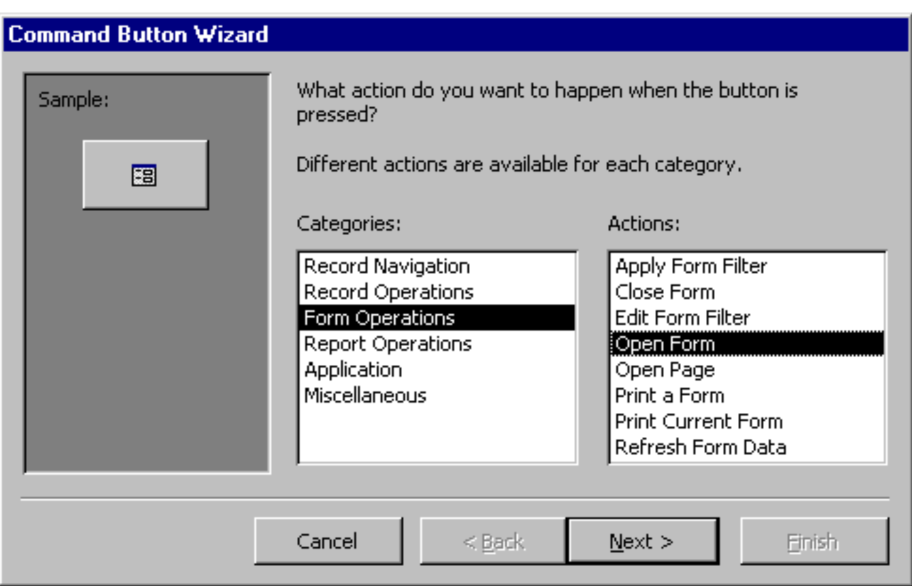


7. Type the caption for the option group and click Finish.

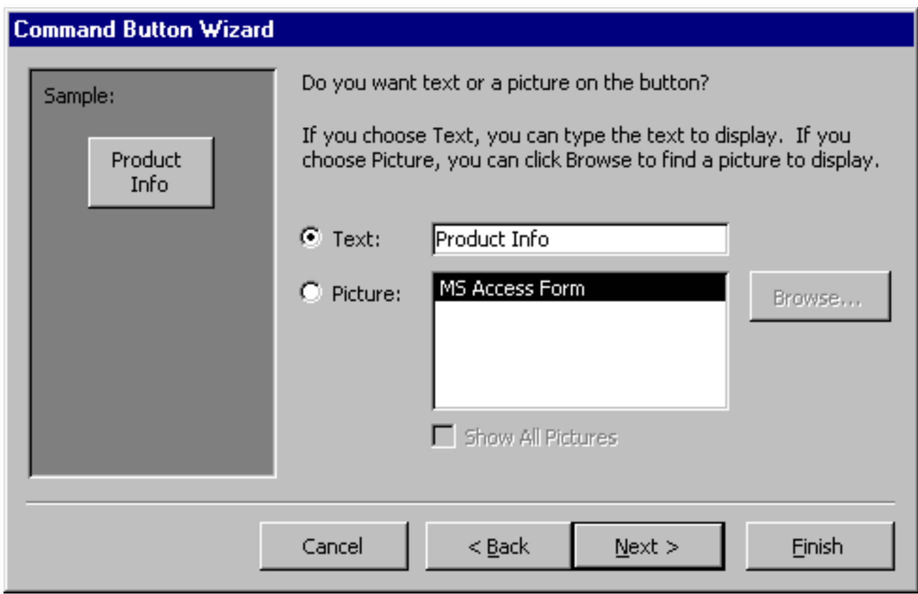
Command Buttons

In this example, a command button beside each record is used to open another form.

1. Open the form in Design View and ensure that the Control Wizard button on the toolbox is pressed in.
2. Click the command button icon on the toolbox and draw the button on the form. The Command Button Wizard will then appear.
3. On the first dialog window, action categories are displayed in the left list while the right list displays the actions in each category. Select an action for the command button and click Next >.



4. The next few pages of options will vary based on the action you selected. Continue selecting options for the command button.
5. Choose the appearance of the button by entering caption text or selecting a picture. Check the Show All Pictures box to view the full list of available images. Click Next >.



6. Enter a name for the command button and click Finish to create the button.

2.9.7 Sorting and filtering

Sorting and filtering allow you to view records in a table in a different way either by reordering all of the records in the table or view only those records in a table that meet certain criteria that you specify.

Sorting

You may want to view the records in a table in a different order than they appear such as sorting by a date or in alphabetical order, for example. Follow these steps to execute a simple sort of records in a table based on the values of one field:

1. In table view, place the cursor in the column that you want to sort by.
2. Select Records|Sort|Sort Ascending or Records|Sort|Sort Descending from the menu bar or click the Sort Ascending or Sort Descending buttons on the toolbar.

To sort by more than one column (such as sorting by date and then sorting records with the same date alphabetically), highlight the columns by clicking and dragging the mouse over the field labels and select one of the sort methods stated above.

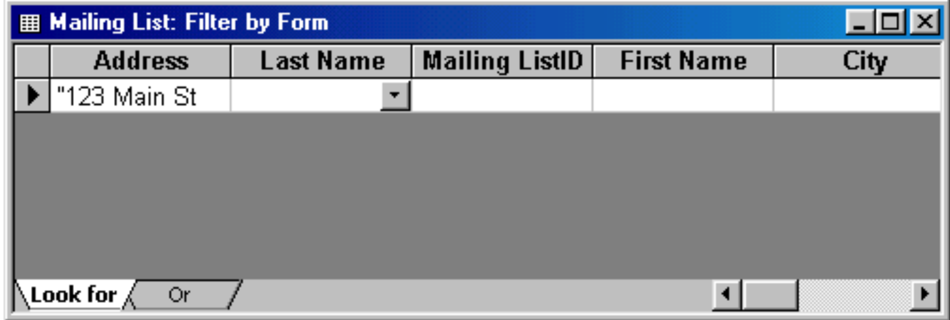
Filter by Selection

This feature will filter records that contain identical data values in a given field such as filtering out all of the records that have the value "Smith" in a name field. To Filter by Selection, place the cursor in the field that you want to filter the other records by and click the Filter by Selection button on the toolbar or select Records|Filter|Filter By Selection from the menu bar. In the example below, the cursor is placed in the City field of the second record that displays the value "Ft. Myers" so the filtered table will show only the records where the city is Ft. Myers.

Filter by Form

If the table is large, it may be difficult to find the record that contains the value you would like to filter by so using Filter by Form may be advantageous

instead. This method creates a blank version of the table with drop-down menus for each field that each contain the values found in the records of that field. Under the default Look for tab of the Filter by Form window, click in the field to enter the filter criteria. To specify an alternate criteria if records may contain one of two specified values, click the Or tab at the bottom of the window and select another criteria from the drop-down menu. More Or tabs will appear after one criteria is set to allow you to add more alternate criteria for the filter. After you have selected all of the criteria you want to filter, click the Apply Filter button on the toolbar.

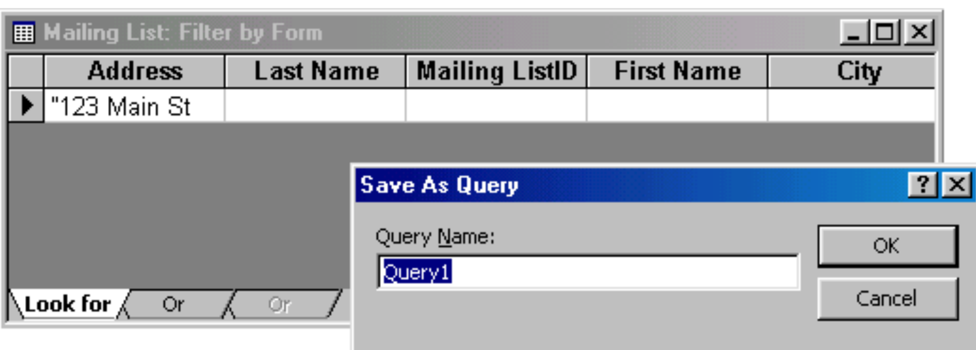


The following methods can be used to select records based on the record selected by that do not have exactly the same value. Type these formats into the field where the drop-down menu appears instead of selecting an absolute value.

Filter by Form	
Format	Explanation
Like "*Street"	Selects all records that end with "Street"
<="G"	Selects all records that begin with the letters A through G
>1/1/00	Selects all dates since 1/1/00
<> 0	Selects all records not equal to zero

Saving a Filter

The filtered contents of a table can be saved as a query by selecting File|Save As Query from the menu bar. Enter a name for the query and click OK. The query is now saved within the database.



Remove a Filter

To view all records in a table again, click the depressed Apply Filter toggle button on the toolbar.

2.9.8 Summary

In this lesson you have learnt how to create and edit forms, how form controls can be added to forms, how the data in the tables can sorted, how data can be entered through forms, how the data can filtered using filters.

2.9.9 Self Understanding

Q1. Explain the various steps involved in creating the forms using Wizard.

Q2. Explain step to step process in creating forms in Design View.

Q3. Explain various Form Controls.

Q4. Explain in detail using suitable example about Filters.

Q5. How data can be sorted in tables ?

REPORTS AND MACRO

Structure:

2.10.0 Introduction
2.10.1 Objectives
2.10.2 Creating Reports by Using Wizard
2.10.3 Creating Reports in Design View
2.10.4 Printing Reports
2.10.5 What is Macro?
2.10.6 Creating Macros
2.10.7 Running Macros
2.10.8 Modules
2.10.9 Macro vs. Modules (or VBA)
2.10.10 Jumping to the Internet
2.10.11 Summary
2.10.12 Self Understanding

2.10.0 Introduction

A report is an effective way to present your data in a printed format. Because you have control over the size and appearance of everything on a report, you can display the information the way you want to see it. Reports are similar to queries in that they retrieve data from one or more tables and display the records. Unlike queries, however, reports add formatting to the output including fonts, colors, backgrounds and other features. Reports are often printed out on paper rather than just viewed on the screen. In this section, we cover how to create simple reports using the Report wizard. Reports will organize and group the information in a table or query and provide a way to print the data in a database.

2.10.1 Objectives

After reading this lesson you will be able to

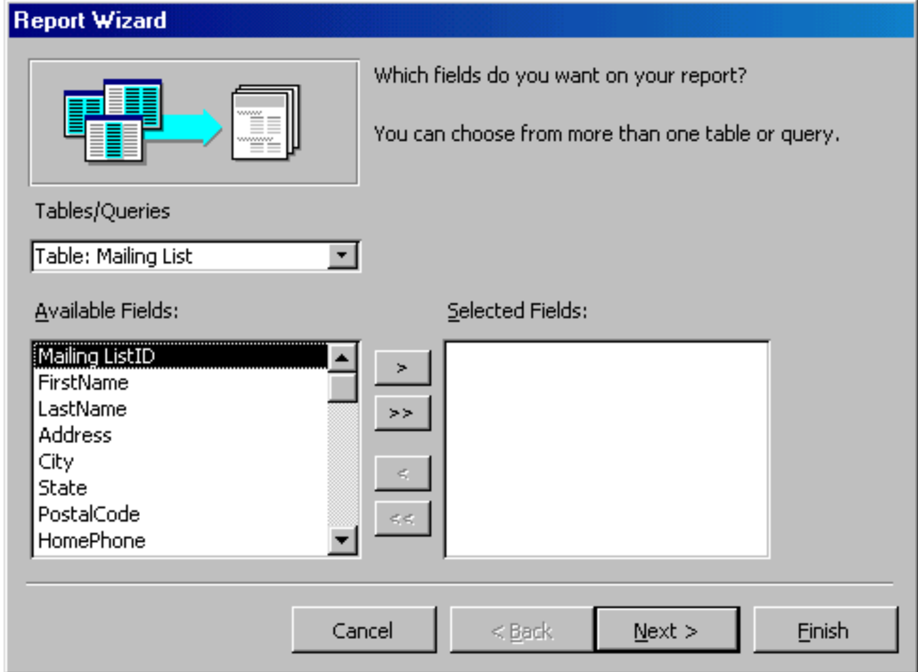
- Create reports using the Wizard and through Design View
- Printing Reports
- Creating and running Macros
- What are Modules and how to use them?

2.10.2 Creating Reports by using Wizard

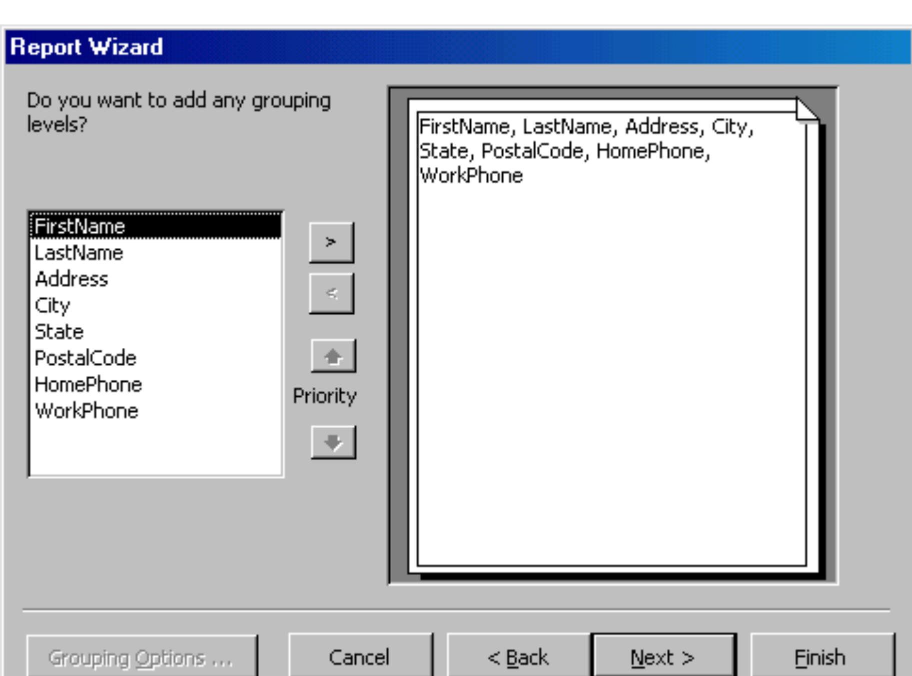
Create a report using Access' wizard by following these steps:

1. Double-click the "Create report by using wizard" option on the Reports Database Window.

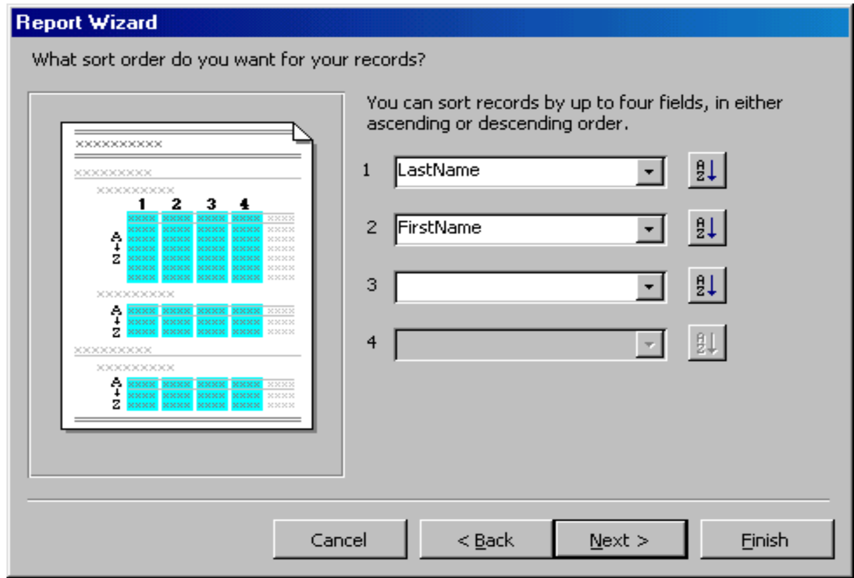
2. Select the information source for the report by selecting a table or query from the **Tables/Queries** drop-down menu. Then, select the fields that should be displayed in the report by transferring them from the **Available Fields** menu to the **Selected Fields** window using the single right arrow button **>** to move fields one at a time or the double arrow button **>>** to move all of the fields at once. Click the **Next >** button to move to the next screen.



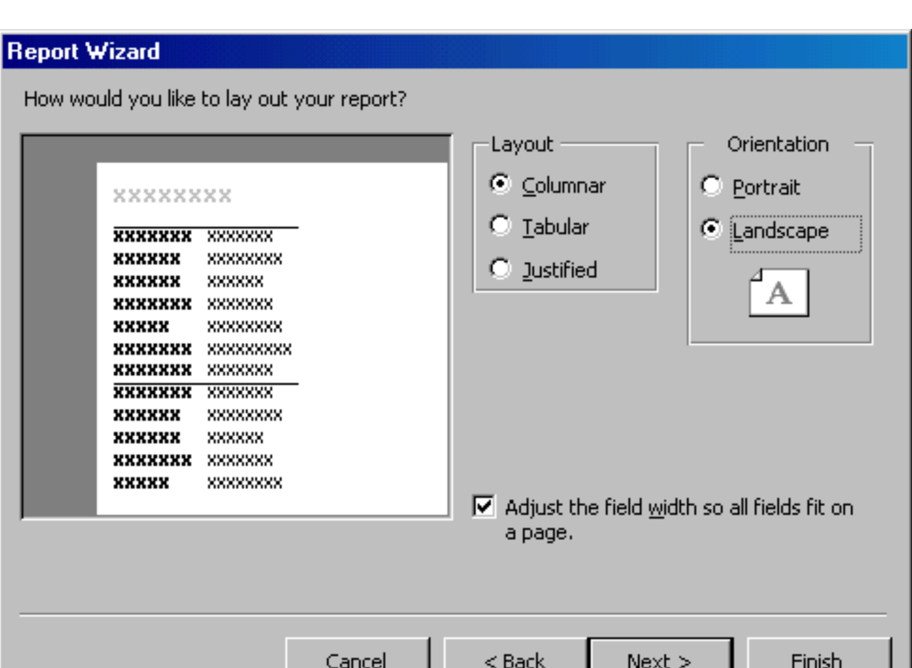
Select fields from the list that the records should be grouped by and click the right arrow button **>** to add those fields to the diagram. Use the **Priority** buttons to change the order of the grouped fields if more than one field is selected. Click **Next >** to continue.



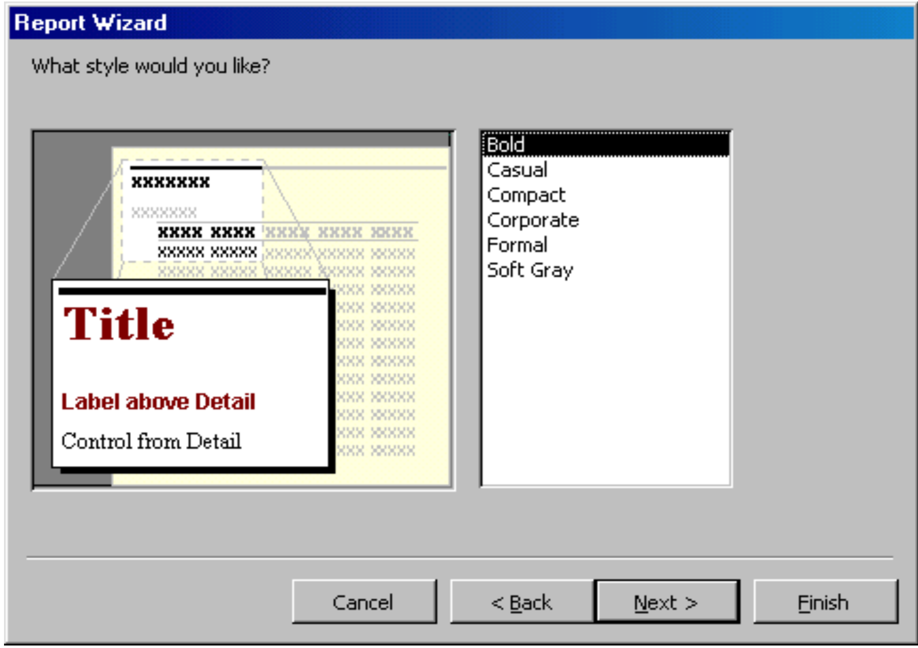
3. If the records should be sorted, identify a sort order here. Select the first field that records should be sorted by and click the A-Z sort button to choose from ascending or descending order. Click **Next >** to continue.



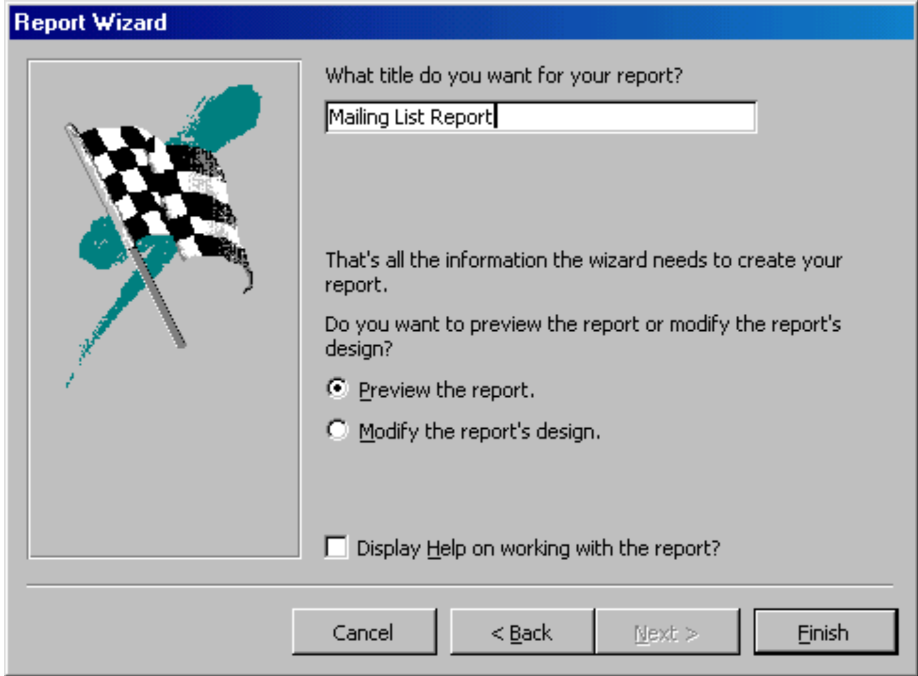
4. Select a layout and page orientation for the report and click **Next >**.



5. Select a color and graphics style for the report and click **Next >**.



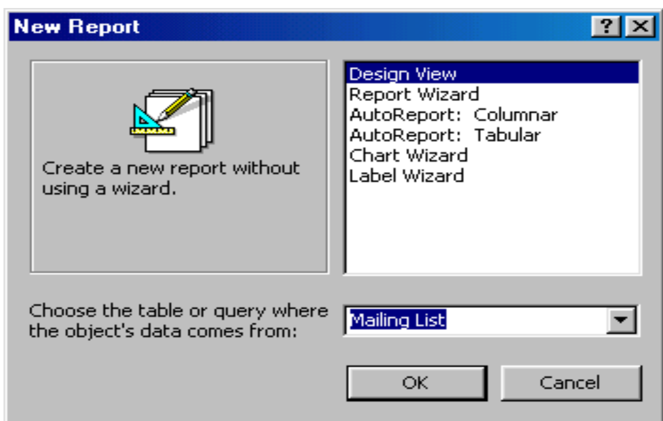
6. On the final screen, name the report and select to open it in either Print Preview or Design View mode. Click the **Finish** button to create the report.



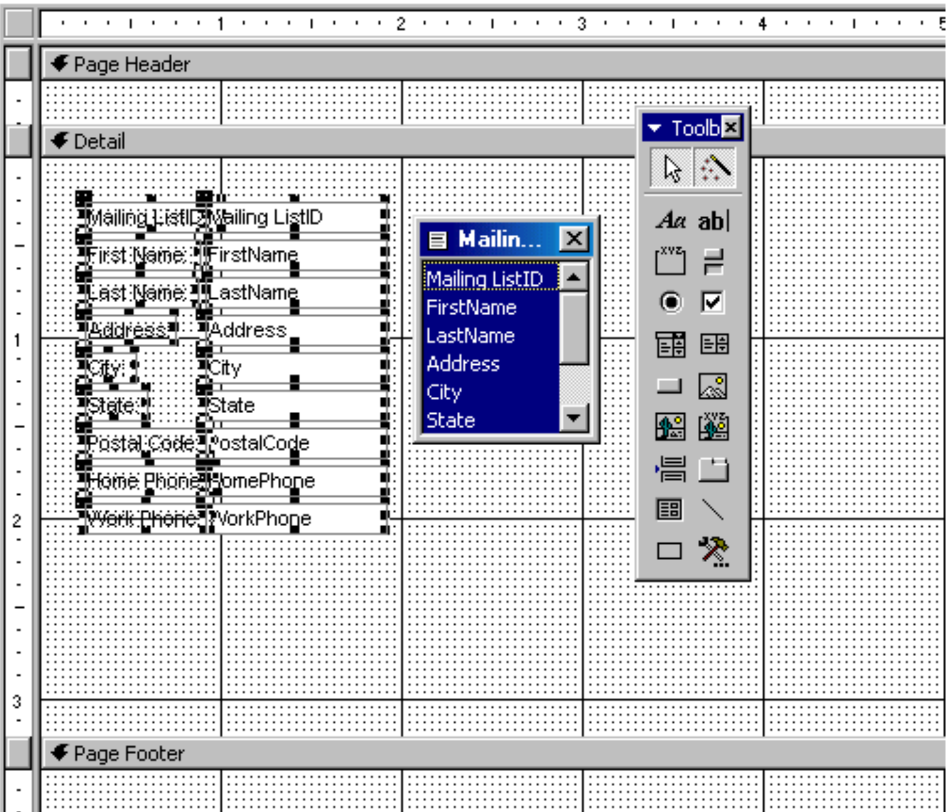
2.10.3 Creating Reports in Design View

To create a report from scratch, select Design View from the Reports Database Window.

1. Click the **New** button on the Reports Database Window. Highlight "Design View" and choose the data source of the report from the drop-down menu and click **OK**.



2. You will be presented with a blank grid with a Field Box and form element toolbar that looks similar to the Design View for forms. Design the report in much the same way you would create a form. For example, double-click the title bar of the Field Box to add all of the fields to the report at once. Then, use the handles on the elements to resize them, move them to different locations, and modify the look of the report by using options on the formatting toolbar. Click the Print View button at the top, left corner of the screen to preview the report.



2.10.4 Printing Reports

Select **File|Page Setup** to modify the page margins, size, orientation, and column setup. After all changes have been made, print the report by selecting **File|Print** from the menu bar or click the **Print** button on the toolbar.

Macros

2.10.5 What is Macro?

A macro is a set of one or more actions that each performs a particular operation, such as opening a form or printing a report. Macros can help you to

automate common tasks. For example, you can run a macro that prints a report when a user clicks a command button.

A macro can be one composed of a sequence of actions, or it can be a macro group. You can also use a conditional expression[1] to determine whether in some cases an action will be carried out when a macro runs.

When you create a macro, you enter the actions you want to carry out in this portion of the Macro window.

You can specify arguments for an action in this portion of the window. Arguments provide additional information on how to carry out the action, such as which object or data to use.

Print Invoice : Macro

Action	
OpenReport	Open the Invoice report

Action Arguments

Report Name	Invoice
View	Print
Filter Name	
Where Condition	[OrderID]=[Forms]![Orders]

2.10.6 Creating Macros

1. In the Database window, click the **Macros** tab and then click **New**.
2. In the Action column, click in the first cell and then click the arrow to display the action list.
3. Click the action you want to use.
4. Type a comment for the action. Comments are optional, but make your macro easier to understand and maintain.
5. In the lower part of the window, specify arguments[2] for the action, if any are required.
6. To add more actions to the macro, move to another action row, and repeat steps 2 through 5. Microsoft Access carries out the actions in the order you list them.

Tip : To quickly create a macro that carries out an action on a specific database object[3], drag the object from the Database window to an action row in the Macro window. For example, you can create a macro that opens a form by dragging the form to an action row. To do this, click **Tile Vertically on the Windows** menu to position the Macro window and Database window so that they're both visible on your screen;

---

1. Conditional expression is an expression that Microsoft Access evaluates and compares to a specific value ¾ for example, If...Then and Select Case statements. If the condition is met, one or more operations are carried out. If the condition isn't met, Microsoft Access skips the operations associated with the expression and moves to the next expression. You can use conditional expressions in macros and Visual Basic code.
2. Argument is a constant, variable, or expression that supplies information to an action, event, method, property, or procedure.
3. Database objects include tables, queries, forms, reports, macros, and modules.

then click the tab in the Database window for the type of object you want to drag, click the object, and drag it to an action row. Dragging a macro adds an action that runs the macro, while dragging other objects (tables, queries, forms, reports, or modules) adds an action that opens the object.

2.10.7 Running Macros

When you run a macro, Microsoft Access starts at the beginning of the macro and carries out all the actions in the macro until it reaches either another macro (if the macro is in a macro group) or the end of the macro.

You can run a macro directly, from another macro or an event procedure[4], or in response to an event[5] that occurs on a form, report, or control. For example, you can attach a macro to a command button on a form so that the macro runs when a user clicks the button. You can also create a custom menu command or toolbar button that runs a macro, assign a macro to a key combination, or run a macro automatically when you open a database.

Create a command button running a Macro

1. Open a form in Design view.
2. Click the Control Wizards tool in the toolbox if it's pressed in. This turns off the wizard.
3. In the toolbox, click the Command Button tool .
4. On the form, click where you want to place the command button.
5. Make sure the command button is selected, and then click Properties on the toolbar to open its property sheet.
6. In the OnClick property box, enter the name of the macro or event procedure that you want to run when the button is clicked, or click the Build button to use the Macro Builder or Code Builder.
7. If you want to display text on the command button, type the text in the Caption property box. If you don't want text on the button, you can use a picture instead.

2.10.8 Modules

A module is a collection of Visual Basic for Applications (**VBA**) declarations and procedures that are stored together as a unit. Refer to **Fig. 1**.

---

4. Event procedure is a procedure automatically executed in response to an event initiated by the user or program code, or triggered by the system.
5. Event is an action recognized by an object, such as a mouse click or key press, for which you can define a response. An event can be caused by a user action or a Visual Basic statement, or it can be triggered by the system. Using properties associated with events, you can tell Microsoft Access to run a macro, call a Visual Basic function, or run an event procedure in response to an event.

There are two basic types of modules: class modules[6] and standard modules[7]. Each procedure in a module can be a Function procedure[8] or a Sub procedure[9].

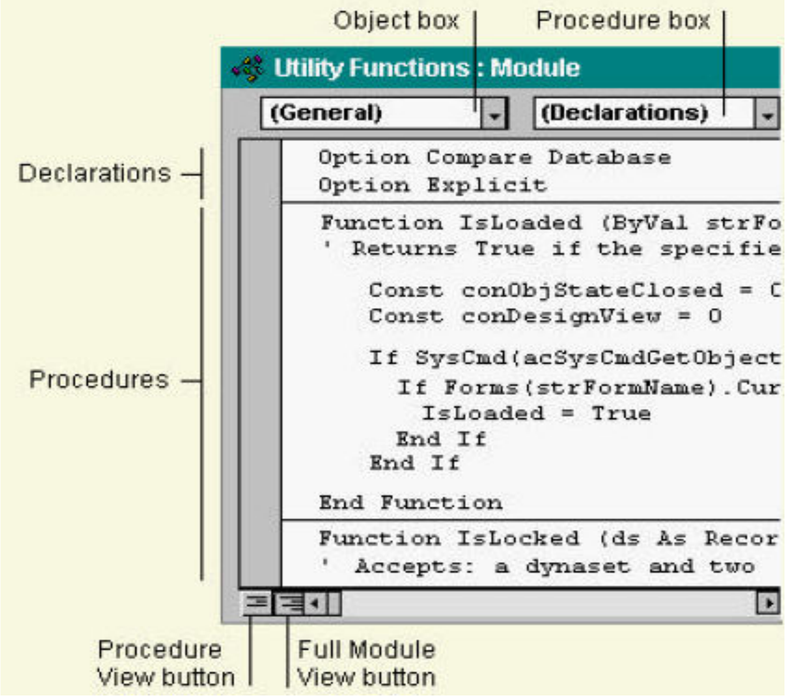


Fig. 1 Module

2.10.9 Macro vs. Modules (or VBA)

In Microsoft Access, you can accomplish many tasks with macros or through the user interface. In many other database programs, the same tasks require programming. Whether to use a macro or Visual Basic for Applications often depends on what you want to do.

When should I use a macro?

Macros are an easy way to take care of simple details such as opening and closing forms, showing and hiding toolbars, and running reports. You can quickly

---

6. Class module is a module that can contain the definition for a new object. When you create a new instance of a class, you create the new object. Any procedures defined in the module become the properties and methods of the object. Class modules in Microsoft Access exist both independently and in association with forms and reports.
7. Standard module is a module in which you can place Sub and Function procedures that you want to be available to other procedures throughout your database.
8. Function procedure is a procedure that returns a value and that can be used in an expression. You declare a function with the Function statement and end it with the End Function statement
9. Sub procedure is a procedure that carries out an operation. Unlike a Function procedure, a Sub procedure doesn't return a value. You declare a Sub procedure with the Sub keyword and end it with an End Sub statement.

and easily tie together the database objects you've created because there's little syntax to remember; the arguments for each action are displayed in the lower part of the Macro window.

In addition to the ease of use macros provide, you must use macros to:

- Make global key assignments.
- □□□Carry out an action or series of actions when a database first opens. However, you can use the Startup dialog box to cause certain things to occur when a database opens, such as open a form.

When should I use VBA?

You should use Visual Basic instead of macros if you want to:

- Make your database easier to maintain. Because macros are separate objects from the forms and reports that use them, a database containing many macros that respond to events on forms and reports can be difficult to maintain. In contrast, Visual Basic event procedures are built into the form's or report's definition. If you move a form or report from one database to another, the event procedures built into the form or report move with it.
- Create your own functions. Microsoft Access includes many built-in functions, such as the IPmt function, which calculates an interest payment. You can use these functions to perform calculations without having to create complicated expressions. Using Visual Basic, you can also create your own functions either to perform calculations that exceed the capability of an expression or to replace complex expressions. In addition, you can use the functions you create in expressions to apply a common operation to more than one object.
- Mask error messages. When something unexpected happens while a user is working with your database, and Microsoft Access displays an error message, the message can be quite mysterious to the user, especially if the user isn't familiar with Microsoft Access. Using Visual Basic, you can detect the error when it occurs and either display your own message or take some action.
- Create or manipulate objects. In most cases, you'll find that it's easiest to create and modify an object in that object's Design view. In some situations, however, you may want to manipulate the definition of an object in code. Using Visual Basic, you can manipulate all the objects in a database, as well as the database itself.
- Perform system-level actions. You can carry out the RunApp action in a macro to run another Windows-based or MS-DOS–based application from your application, but you can't use a macro to do much else outside Microsoft Access. Using Visual Basic, you can check to see if a file exists on

the system, use Automation or dynamic data exchange (DDE) to communicate with other Windows-based applications such as Microsoft Excel, and call functions in Windows dynamic-link libraries (DLLs).

- Manipulate records one at a time. You can use Visual Basic to step through a set of records one record at a time and perform an operation on each record. In contrast, macros work with entire sets of records at once.
- Pass arguments to your Visual Basic procedures. You can set arguments for macro actions in the lower part of the Macro window when you create the macro, but you can't change them when the macro is running. With Visual Basic, however, you can pass arguments to your code at the time it is run or you can use variables for arguments — something you can't do in macros. This gives you a great deal of flexibility in how your Visual Basic procedures run.

2.10.10 Jumping to the Internet

There are three types of Web pages Microsoft Access creates:

- Data access pages
- Server-generated HTML
- Static HTML

Data access pages

- You create a data access pages as a database object that contains a shortcut to the location of the page's corresponding HTML file.
- Use pages to view, edit, update, delete, filter, group, and sort live data from either a Microsoft Access database or a Microsoft SQL Server database, in Microsoft Internet Explorer 5 or later. A page can also contain additional controls called including a **spreadsheet**, a **PivotTable** list, and a **chart**.
- To make your pages available on the World Wide Web, you publish the pages to Web Folders or a Web server, and make the Access database or SQL Server database available to users of the page. Internet Explorer needs to download the page only once from the Web server to let you view and interact with the data on the page. Because a page uses Dynamic HTML, access to the database is generally very efficient in a client/server environment.

Server-generated HTML

- You can create server-generated HTML files, either ASP or IDC/HTX, from tables, queries, and forms. Server-generated HTML files are displayed in a table format in a Web browser. Use server-generated HTML files when you want to use any Web browser, your data changes

frequently, you need to see live data in a table connected to an ODBC data source, but you only need to see read-only data.

- Once you output ASP or IDC/HTX files, you must publish these files to be processed on a supported Web server product and platform.
- Each time a user opens or refreshes an ASP or HTX file from a Web browser, the Web server dynamically creates an HTML file, and then sends that HTML file to the Web browser.

Static HTML

- You can create static HTML files from tables, queries, forms, and reports. In a Web browser, reports display in a report format, and tables, queries, and forms display in a datasheet format.
- Use static HTML files when you want to use any Web browser that supports HTML version 3.2 or later and your data does not change frequently.
- To make your static HTML files available on the World Wide Web, you publish the files to Web Folders or a Web server.
- When you access the data through a Web browser, the browser only needs to download the static HTML file once from the Web server to let you view the data. However, the r.esulting HTML files are a snapshot of the data at the time you published your files. There is no ODBC data source connected to the static HTML file, and If your data changes, you must export your files again to be able to view new data in a Web browser.

2.10.11 Summary

In this lesson you have learnt how to create and print reports. You have also learnt what are modules and how to use the modules. You have also learnt about macros, creating macros and running macros.

2.10.12 Self Understanding

Q1. What is a report and how it can be created using Wizard?

Q2. How Reports can be created using Design View?

Q3. What is a macro? How it is created and run?

Q4. What do you mean by Modules and how can it be used?

Q5. Differentiate Macros with Modules.

Type Setting :
Department of Distance Education, Punjabi University, Patiala.