ଓଡ଼ିଶା ରାଜ୍ୟ ମୁକ୍ତ ବିଶ୍ୱବିଦ୍ୟାଳୟ, ସମ୍ବଲପୁର, ଓଡ଼ିଶା
Odisha State Open University, Sambalpur, Odisha
Established by an Act of Government of Odisha.

DIPLOMA IN COMPUTER APPLICATION

DCA-05 DATABASE SYSTEMS

BLOCK

2 DATABASE SYSTEM DESIGN

EXPERT COMMITTEE

Dr. P.K Behera(Chairman)
Reader in Computer Science
Utkal University
Bhubaneswar, Odisha

Dr.J.R Mohanty(Member)
Professor and HOD
KIIT University
Bhubaneswar, Odisha

Sri Pabitrananda Pattnaik(Member)
Scientist-E, NIC
Bhubaneswar, Odisha

Sri Malaya Kumar Das (Member)
Scientist-E, NIC
Bhubaneswar, Odisha

Dr. Bhagirathi Nayak (Member)
Professor and Head (IT & System)
Sri Sri University
Bhubaneswar,Odisha

Dr.Manoranjan Pradhan(Member)
Professor and Head (IT & System)
G.I.T.A
Bhubaneswar, Odisha

Sri Chandrakant Mallick(Convener)
Consultant (Academic)
School of Computer and Information Science
Odisha State Open University
Sambalpur, Odisha

DIPLOMA IN COMPUTER APPLICATION

Course Writers

Chandrakant Mallick
Odisha State Open University, Sambalpur, Odisha
Bijay Kumar Paikaray
Centurion University of Technology and Management, Odisha

UNIT-1 RELATIONAL DATABASE OPERATIONS

Unit Structure

1.0 Introduction

The Relational database was proposed by Edgar Codd (of IBM Research) around 1969. It has since become the dominant database model for commercial applications (in comparison with other database models such as hierarchical, network and object models). Today, there are many commercial *Relational Database Management System* (RDBMS), such as Oracle, IBM DB2 and Microsoft SQL Server. There are also many free and open-source RDBMS, such as MySQL, mSQL (mini-SQL) and the embedded JavaDB (Apache Derby).
In this unit we will briefly introduce the topics of relational databases, and will hopefully these will help you in learning the higher concepts.

1.1 Learning Objectives

After learning this unit you should be able to

- Know the fundamental concepts of relational database languages
- Define relational schema.
- Identify the operators of relational algebra
- Understand the seven basic relational algebra operations are Selection, Projection, Joining, Union, Intersection, Difference and Division.
- Differentiate between relational algebra and relational calculus.
- Know how to translate the relational query or SQL query in Relational Algebra expression.

1.2 Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks.

A Relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd.
Examples of RDBMS are: MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access etc.
Key Difference between DBMS and RDBMS: The key difference is that RDBMS (relational database management system) applications store data in a tabular form, while DBMS applications store data as files. Today, all database management systems by default are relational.
DBMS as used in this book is a general term that includes RDBMS.

1.2.1 Overview of a Relational Table

A relational **database** is a set of tables containing data fitted into predefined categories. Each table (which is sometimes called a **relation**) contains one or more data categories in columns. Each row contains a unique instance of data for the categories defined by the columns.

A relational table has the following properties:

- The table has a name that is distinct from all other tables in the database.
- Each cell of the table contains exactly one value. (For example, it would be wrong to store several telephone numbers for a single branch in a single cell. In other words, tables don't contain repeating groups of data. A relational table that satisfies this property is said to be *normalized* or in *first normal form*.)
- Each column has a distinct name.
- The values of a column are all from the same domain.
- The order of columns has no significance. In other words, provided a column name is moved along with the column values, we can interchange columns.
- Each record is distinct; there are no duplicate records.
- The order of records has no significance, theoretically.

1.3 Relational Languages

We have so far considered the structure of a database; the relations and the associations between relations. In this section we consider how useful data

may be extracted and filtered from database tables. A *relational language* is needed to express these *queries* in a well defined way. A *relational language* is an abstract language which provides the database user with an interface through which they can specify data to be retrieved according to certain selection criteria. The two main relational languages are relational algebra and relational calculus. Relational algebra, which we focus on here, provides the user with a set of operators which may be used to create new (temporary) relations based on information contained in existing relations. Relational calculus, on the other hand, provides a set of key words to allow the user to make ad hoc inquiries.

1.4 Introduction to Relational Algebra and Relational Calculus

Relational Algebra (RA) and Relational Calculus (RC) are formal languages for the database relational model while SQL is the practical language in the database relational model. In these formal languages a conceptual database model is expressed in mathematical terms and notations while in the practical language – SQL, the mathematical expressions of the functionality and transaction of the database operations are implemented physically. Formal languages provide a medium through which to optimize and implement queries in database transactions.

Of course the first notable differences between these languages in the syntax and notation used in the expressions. Each language, that RA, RC, and SQL have their own notations to express their notations.

1.4.1 What is Algebra?

- A language based on operators and a domain of values
- Operators map values taken from the domain into other domain values
- Hence, an expression involving operators and arguments produces a value in the domain
- When the domain is a set of all relations (and the operators are as described later), we get the *relational algebra*
- We refer to the expression as a *query* and the value produced as the *query result*

1.4.2 Definition of Relational Algebra

Relational algebra presents the basic set of operations for relational model. It is a procedural language, which describes the procedure to obtain the result. Relational algebra is prescriptive because it describes the order of operations in the query that specifies *how* to retrieve the result of a query.

The sequence of operations in a relational algebra is called relational algebra expression. The Relational Algebra Expression either it takes one relation or two relations as an input to the expression and produces a new relation as a result. The resultant relation obtained from the relational algebra expressions can be further composed to the other relational algebra expression whose result will again be a new relation.

The Relation Algebra forms the framework for implementing and optimizing queries while query processing. Relational algebra is an integral part of relational DBMS. The fundamental operation included in relational algebra are {**Select (σ), Project (π), Union (∪), Set Difference (-), Cartesian product (×) and Rename (ρ)**}.

1.4.3 Definition of Relational Calculus

Unlike Relational Algebra, Relational Calculus is a higher level Declarative language. In converse to the relational algebra, relational calculus defines *what* result is to be obtained. Like Relational Algebra, Relational Calculus does not specify the sequence of operations in which query will be evaluated.

The sequence of relational calculus operations is called relational calculus expression that also produces a new relation as a result. The Relational Calculus has two variations namely Tuple Relational Calculus and Domain Relational Calculus.

The Tuple Relational Calculus list the tuples to selected from a relation, based on a certain condition provided. It is formally denoted as:

{ t | P(t) }, Where **t** is a set of tuples for which the condition **P** is true.

The next variation is Domain Relational Calculus, which in contrast to Tuple Relational Calculus **list the attributes** to be selected from a relation, based on certain **condition**. The formal definition of Domain Relational Calculus is as follow:

{<X1, X2, X3, . . .Xn> | P(X1, X2, X3, . . . Xn)}

Where **X1, X2, X3, . . .Xn** are the attributes and **P** is the certain condition.

1.4.4 Key Differences between Relational Algebra and Relational Calculus

1. The basic difference between Relational Algebra and Relational Calculus is that Relational Algebra is a Procedural language whereas, the Relational Calculus is a Non-Procedural, instead it is a declarative language.
2. The Relational Algebra defines how to obtain the result whereas, the Relational Calculus define what information the result must contain.
3. Relational Algebra specifies the sequence in which operations have to be performed in the query. On the other hands, Relational calculus does not specify the sequence of operations to perform in the query.
4. The Relational Algebra is not domain dependent whereas, the Relational Calculus can be domain dependent as we have Domain Relational Calculus.
5. The Relational Algebra query language is closely related to programming language whereas; the Relational Calculus is closely related to the Natural Language.

1.5 Operations of Relational Algebra

A query language is a language in which user requests information from the database. It can be categorized as either procedural or nonprocedural. In a procedural language the user instructs the system to do a sequence of operations on database to compute the desired result. In nonprocedural language the user describes the desired information without giving a specific procedure for obtaining that information.

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produces a new relation as output.

1.5.1 Fundamental Operations

- SELECT
- PROJECT
- UNION
- SET DIFFERENCE
- CARTESIAN PRODUCT
- RENAME

Select and project

Select and project operations are unary operation as they operate on a single relation. Union, set difference, Cartesian product and rename operations are binary operations as they operate on pairs of relations.

Other Operations

- SET INTERSECTION
- NATURAL JOIN
- DIVISION
- ASSIGNMENT

1.5.2 Schema

The schema of a relation is similar to the format or structure of a table. The schema of a relation is the set of attributes that forms a tuple for that relation. For example, the following describes the schema for the relation student:

Student(student #, first name, last name, street address, city, state, zip, phone, major, GPA)

When we wish to obtain information from a database, we use a language like SQL to create a query for the database management system to process. The response from the database will be a result set with a particular schema. The definition above says "Relational algebra is a formal language describing how new relations are formed from existing relations." If we think of the tables in the database as the "existing relations", and the result set of the query as the "new relations", then relational algebra is a language that can be used to describe data base queries that will return a result set from the existing database.

Result set = query (existing database)

By working with relational algebra, you will learn how to formulate database queries to return the results you desire.

There are five basic operations in relational algebra, and several derived operations. The derived operations are called derived because they can be

derived from a combination of the basic operations, similar to the way multiplication can be derived from addition in simple arithmetic.

1.5.3 The Basic Operations:

Union, difference, selection, projection, and Cartesian product (sometimes referred to as Cartesian join).

The derived operations important to database management are: intersection, complement, natural join, equi-join and theta join.

If you have ever studied set theory then some of these operations might be familiar to you. In fact, relational algebra and the operations it discusses are closely associated with the set theory studied in elementary school, where children learn about things like the union and intersection of sets.

Union

In simple set theory, the union of Set A and Set B includes all of the elements of Set A and all of the elements of Set B. In relational algebra, the union operation is similar:

The Union of relation A and relation B is a new relation containing all of the tuples contained in either relation A or relation B. Union can only be performed on two relations that have the same schema.

The symbol for union is ∪. In relational algebra we would write something like R3 = R1 ∪ R2.

Example:

The set of students majoring in Communications includes all of the Acting majors and all of the Journalism majors.

Let A = the relation with data for all Acting majors
Let J = the relation with data for all Journalism majors
Let C = the relation with data for all Communications majors

$$C = A \cup J$$

The union operation is both commutative and associative.

Commutative law of union: $A \cup B = B \cup A$

Associative law of union: $(A \cup B) \cup C = A \cup (B \cup C)$

Intersection

Like union, intersection means pretty much the same thing in relational algebra that it does in simple set theory:

The Intersection of relation A and relation B is a new relation containing all of the tuples that are contained in both relation A and relation B. Intersection can only be performed on two relations that have the same schema.

The symbol for intersection is ∩. In relational algebra we would write something like R3 = R1 ∩ R2.

Example:
We might want to define the set of all students registered for both Database Management and Linear Algebra.
Let D = the relation with data for all students registered for Database Management
Let L = the relation with data for all students registered for Linear Algebra
Let B = the relation with data for all students registered for both Database Management and Linear Algebra
B = D ∩ L
The intersection operation is both commutative and associative.
Commutative law of intersection: $A \cap B = B \cap A$
Associative law of intersection: $(A \cap B) \cap C = A \cap (B \cap C)$

Unlike union, however, intersection is not considered a basic operation, but a derived operation, because it can be derived from the basic operations. We will look at difference next, but the derivation looks like this:

$R1 \cap R2 = R1 - (R1 - R2)$

For our purposes, however, it really doesn't matter that intersection is a derived operation.

Difference

The difference operation also means pretty much the same thing in relational algebra that it does in simple set theory:

The difference between relation A and relation B is a new relation containing all of the tuples that are contained in relation A but not in relation B. Difference can only be performed on two relations that have the same schema.

The symbol for difference is the same as the minus sign - We would write R3 = R1 – R2.

Example:

We might want to define the set of all players sitting on the bench during a basketball game.

Let T = the relation with data for all players currently on the team
Let G = the relation with data for all players currently in the game
Let B = the relation with data for all players on the bench; that is, on the team but not playing

B = T - G

The difference operation is neither commutative nor associative.

$A - B \neq B - A$
$(A - B) - C \neq A - (B - C)$

Complement

Union, Intersection, and difference were binary operations; that is, they were performed on two relations with the result being a third relation. Complement is a unary operation; it is performed on a single relation to form a new relation, as follows:

The complement of relation A is a relation composed all possible tuples not in A, which have the same schema as A, derived from the same range of values for each attribute of A.

Complement is sometimes shown by using a superscripted C, like this: B = AC.

Example:
Let A = the relation containing data on members of the U.S. Supreme Court, with the schema (first name, last name, state of birth, year of birth).

A = {(William, Rehnquist, Arizona, 1924), (John, Stevens, Illinois, 1920), (Sandra, O'Connor, Arizona, 1930), (Antonin, Scalia, New Jersey, 1936), (Anthony, Kennedy, California, 1936), (David, Souter, Massachusetts, 1939), (Clarence, Thomas, Georgia, 1948), (Ruth, Ginsburg, New York, 1933), (Stephen, Breyer, California, 1938) }
AC would be the set of all possible tuples with the same schema as A but not in A, that are derived from the same domains for the attributes. It would be very large and include tuples like (William, Rehnquist, Arizona, 1920), (William, Rehnquist, Arizona, 1930), (Ruth, Rehnquist, Illinois, 1924), (William, Stevens, Illinois, 1930), (John, O'Connor, Georgia, 1938), (Clarence, Ginsberg, California, 1936), and so on.

If we perform the complement operation twice, we get back the original relation, just like we would when using the negation operation on numbers in simple arithmetic. (AC) C = A, which means that if B = AC, then A = B C.

Selection

Selection is another unary operation. It is performed on a single relation and returns a set of tuples with the same schema as the original relation. The selection operation must include a condition, as follows:

A selection of Relation A is a new relation with all of the tuples from Relation A that meets a specified condition. The condition must be a logical comparison of one or more of the attributes of Relation A and their possible values. Each tuple in the original relation must be checked one at a time to see if it meets the condition. If it does, it is included in the result set, if not, it is not included in the result set. The logical comparisons are the same as those used in Boolean conditions in most computer programming languages.

The symbol for selection is σ, a lowercase Greek letter sigma.

A selection operation is written as B = σ c (A), where c is the condition.

Example:
We wish to find all members of the U.S. Supreme Court born before 1935 in Arizona

Let A = the relation containing data on members of the Supreme Court, as defined above.

B = σ ((year of birth < 1935) $\wedge$ (state of birth = "Arizona")) (A).

B = {(William, Rehnquist, Arizona, 1924), (Sandra, O'Connor, Arizona, 1930)}

The allowable operators for the logical conditions are the six standard logical comparison operators: equality, inequality, less than, greater than, not less than (equality or greater than), not greater than (equality or less than) Simple logical comparisons may be connected via conjunction, disjunction and negation (and, or, and not).

The symbols for the six logical comparison operators are:

Comparison Operation	Symbol
equal to	=
not equal to	≠ or <>
less than	<
greater than	>
not less than (greater than or equal to)	≥ or >=
not greater than (less than or equal to)	≤ or <=

The symbols for the three logical modifiers used to build complex conditions are:

Modifier	Symbol
And	∧
or	∨
not	¬ or ~

Some definition or collating sequence must exist to determine how values compare to one another for each of the data types used, just as in computer

programming languages. For example, 2 comes before 11 if the values are integers, but “11” comes before “2” if the values are character strings.

Each time we perform a selection operation, the result set will have the same number or fewer tuples. Most often, the result set gets smaller with each selection operation.

Projection

Projection is another unary operation, performed on a single relation with a result set that is a single relation, defined as follows:

The projection operation returns a result set with all rows from the original relation, but only those attributes that are specified in the projection.

Projection is shown by $\prod$, the upper case Greek letter Pi. A selection operation is written as

B = $\prod$ attributes (A), where attributes is a list of the attributes to be included in the result set.

Example:
We wish to show only the names of the U. S. Supreme Court Justices from the example above.

Let A = the relation containing data on members of the Supreme Court, as defined above.

B = $\prod$ first name, last name (A)

B = { (William, Rehnquist), (John, Stevens), (Sandra, O'Connor), (Antonin, Scalia), (Anthony, Kennedy), (David, Souter), (Clarence, Thomas), (Ruth, Ginsburg), (Stephen, Breyer) }

A projection operation will return the same number of tuples, but with a new schema that will include either the same columns or fewer columns. Most often, the number of columns shrinks with each projection.

Combining Selection and Projection

Often the selection and projection operations are combined to select certain data from a single relation. We can nest the operation with parenthesis, just like in ordinary algebra.

Example:
We wish to show only the names of the U.S. Supreme Court justices born in Arizona before 1935.

Let A = the relation containing data on members of the U. S. Supreme Court, as defined above.

B = $\prod$first name, last name (σ ((year of birth < 1935) $\wedge$ (state of birth = "Arizona")) (A))

B = { (William, Rehnquist), (Sandra, O'Connor) }

When performing both a projection and a selection, which would be more efficient to do first, a projection or a selection? Imagine that we have a database of 100,000 student records. We wish to find the student #, name, and GPA for student # 111-11-1111. Should we find the student's record first and then pull out the name and GPA, or should we pull out the name and GPA for all students, then search that set for the student we are seeking? Usually it is best to do the selection first, thereby limiting the number of tuples, but this is not always the case. Fortunately, most good database management systems have optimizing compilers that will perform the operations in the most efficient way possible.

The most common queries on single tables in modern data base management systems are equivalent to a combination of the selection and projection operations.

1.6 Joins

Joins are operations that "cross reference" the data. That is, tuples from one relation are somehow matched with tuples from another relation to form a third relation. There are several types of joins, but the most basic type is the Cartesian join, sometimes called a Cartesian product or cross product. Other joins, including the natural join, the equi-join and the theta join, are

variations of the Cartesian join in which special rules are applied. Each of these four types of joins in described below.

Cartesian Join

Imagine that we have two sets, one composed of letters and one composed of numbers, as follows:

S1 = {a, b, c, d} and S2 = {1, 2, 3}

The cross product of the two sets is a set of ordered pairs, matching each value from S1 with each value from S2.

S1 x S2 = { (a,1), (a2), (a3), (b1), (b2), (b3), (c,1), (c2), (c3), (d1), (d2), (d3) }

The cross product of two sets is also called the Cartesian product, after René Descartes, the French mathematician and philosopher, who among other things developed the Cartesian Coordinates used in quantifying geometry. In Cartesian coordinates, we have an X-axis and a Y-axis, which means we have a set of X values and a set of Y values. Each point on the Cartesian plane can be referenced by its coordinates, with an X-Y ordered pair: (x, y). The set of all possible X and Y coordinates is the cross product of the set of all X coordinates with the set of all Y coordinates.

In relational algebra, the cross product of two relations is also called the Cartesian product or Cartesian Join, and is defined as follows:

The Cartesian Join of relation A and relation B is composed by matching each tuple from relation A one at time, with each tuple from relation B one at a time to form a new relation containing tuples with all of the attributes from tuple A and all of the attributes from tuple B. If tuple A has AT tuples and AA attributes and tuple B has BT tuples and BA attributes, then the new relation will have (AT * BT) tuples, and (AA + BA) attributes.

The symbol for a Cartesian Join is **×.**
For Example, we would write C = A × B.

Example:

We wish to match a group of drivers with a group of trucks.

Let D = the relation with data for all of the drivers
D has the schema D (name, years of service)
D = {(Joe Smith, 12), (Mary Jones, 4),(Sam Wilson, 20), Bob Johnson, 8) }

Let T = the relation with data for all of the trucks
T has the schema D (make, model, year purchased)
T = {(White, Freightliner, 1996), (Ford, Econoline, 2002), (Mack, CHN 602, 2004)}

Let A = the relation with data for all possible assignments of driver to trucks

$A = D \times T$
A has the schema A (name, years of service, make, model, year purchased)

A = { (Joe Smith, 12, White, Freightliner, 1996), (Joe Smith, 12, Ford, Econoline, 2002), (Joe Smith, 12, Mack, CHN 602, 2004), (Mary Jones, 4, White, Freightliner, 1996), (Mary Jones, 4, Ford, Econoline, 2002), (Mary Jones, 4, Mack, CHN 602, 2004), (Sam Wilson, 20, White, Freightliner, 1996), (Sam Wilson, 20, Ford, Econoline, 2002), (Sam Wilson, 20, Mack, CHN 602, 2004), (Bob Johnson, 8, White, Freightliner, 1996), (Bob Johnson, 8, Ford, Econoline, 2002), (Bob Johnson, 8, Mack, CHN 602, 2004) }

Although Cartesian Joins form the conceptual basis for all other joins, they are rarely used in actual database management systems because they often result in a relation with a large amount of data. Consider the case of a table with data for 40,000 students, with each row needing 300 bytes of storage space, and a table for 2,000 advisors, with each row needing 200 bytes. The two original tables would need about 12,000,000 and 400,000 bytes of storage space (12 megabytes and 400 kilobytes). The Cartesian join of these two would have 80,000,000 records, each with nearly 600 bytes of storage space for a total of 48,000,000,000 bytes (48 gigabytes).

Another reason that Cartesian joins are not used often is this: What is the value of a Cartesian join? How often do we really need to create such a table?

The other types of joins, which are based on the Cartesian join, are used more often, and are commonly applied in combination with projection and selection operations.

Natural Join

A natural join is performed on two relations that share at least one attribute, and is defined as follows:

The natural join of relation A with relation B is a new relation formed by matching all tuples from relation A one by one with all tuples from relation B one by one, but only where the value of the shared attributes are the same. Each shared attribute is only included once in the schema of the result set. A natural join can only be performed on two relations that have at least one shared attribute.

The symbol for a natural join is $\bowtie$ We would write $C = A \bowtie B$.

Theta Join

A theta join is similar to a Cartesian join, except that only those tuples are included that meet a specified condition, as follows:

The theta join of relation A with relation B is a new relation formed by matching all tuples from relation A one by one with all tuples from relation B one by one, but only where the tuples meet a specified condition, called the theta predicate. If the relations share any attributes, then each shared attribute is only included once in the schema of the result set.

The general symbol for a theta join is composite symbol, similar to the symbol for a natural join subscripted with the Greek letter theta: $\bowtie_\theta$ We would write $C = A \bowtie_\theta B$. In practice, the theta is replaced with the actual condition.

Equi-Join

An equi-join, which is similar to both a theta join and a natural join, is defined as follows:

The equi-join of relation A with relation B is a new relation formed by matching all tuples from relation A one by one with all tuples from relation B one by one, but only where the tuples meet a specified condition of equality, called the equi-join predicate. If the relations share any attributes, then each shared attribute is only included once in the schema of the result set.

The symbolism for an equi-join is similar to the symbol for a natural join subscripted with the equal sign: $\bowtie_{=}$ We would write $C = A \bowtie_{=} B$. Just as with the theta join, in practice the equal sign is replaced with the actual condition.

The difference between an equi-join and a theta join is that the condition must be one of equality in an equi-join. The difference between an equi-join and a natural join is that the two relations do not need to have a common attribute in an equi-join.

Example:

We wish to match groups of people waiting for tables at a restaurant with the available tables, on the condition that the number of people in the group equals the number of seats at the table.

Let W = the relation with data for all the groups waiting for tables
Let T = the relation with data for all of the available tables
Let M = the relation with data assigning groups to tables

$M = W \bowtie_{group.size = table.seats} T$

In this notation the attribute names are shown in their more complex form, relation. attribute, so that group. size refers to the size attribute of the group relation, and table. Seats refer to the seats attribute of the table relation.

1.7 Minus

If P is a result of an operation and Q is a result of another operation, P - Q is the set of all tuples that are in P and not in Q.
For example, to list all the departments which do not have an ongoing project (projects with status = ongoing) −

AllDept←πDepartment(EMPLOYEE)AllDept←πDepartment(EMPLOYEE
)
ProjectDept←πDepartment(σStatus="ongoing"(PROJECT))ProjectDept←π
Department(σStatus="ongoing"(PROJECT))
Result←AllDept−ProjectDept

1.8 Translating SQL Queries into Relational Algebra

SQL queries are translated into equivalent relational algebra expressions before optimization. A query is at first decomposed into smaller query blocks. These blocks are translated to equivalent relational algebra expressions. Optimization includes optimization of each block and then optimization of the query as a whole.

Examples
Let us consider the following schemas −

EMPLOYEE

Emp ID	Name	City	Department	Salary

PROJECT

PId	City	Department	Status

WORKS

EmpID	PID	Hours

Example 1

To display the details of all employees who earn a salary LESS than the average salary, we write the SQL query −

SELECT * FROM EMPLOYEE
WHERE SALARY < (SELECT AVERAGE (SALARY) FROM EMPLOYEE) ;
This query contains one nested sub-query. So, this can be broken down into two blocks.

The inner block is −

SELECT AVERAGE (SALARY) FROM EMPLOYEE;
If the result of this query is AvgSal, then outer block is −

SELECT * FROM EMPLOYEE WHERE SALARY < AvgSal;
Relational algebra expression for inner block −

AvgSal←AVERAGE (Salary) EMPLOYEE
AvgSal←□AVERAGE(Salary)EMPLOYEE
Relational algebra expression for outer block −

σSalary<AvgSal>EMPLOYEE

1.9 Division Operation

In its simplest form, this operation has a binary relation R(X,Y) as the dividend and a divisor that includes Y. The output is a set, *S*, of values of X such that $x \in S$ if there is a row (*x,y*) in R for each *y* value in the divisor.

As an example, suppose we have two relations R6 and R7:

RELATION: R6

Store-ID	**Location**
C1	Boston
C2	Chicago
C1	Washinton
C3	Boston
C2	New York
C3	New York
C3	Chicago

RELATION: R7

Location
Boston
New York

The operation:

R8 = R6 / R7

will give the result:

RELATION: R8

Company
C3

This is because C3 is the only company for which there is a row with Boston and New York. The other companies, C1 and C2, do not satisfy this condition.

Note on Relational Calculus

As we have seen, relational algebra can be used to specify the sequence of operations necessary to answer a given query, in other words relational algebra is a procedural language. In contrast, *relational calculus* is a way of specifying only the information required, not a method for specifying how it is obtained.

1.10 Let us Sum Up

Relational Algebra and Relational Calculus both have equivalent expressive power. The main difference between them is just that Relational Algebra specifies how to retrieve data and Relational Calculus defines what data is to be retrieved.

Now that we've looked at the structure of a relational database, we will look at its fundamental operations — that is, what you can do with that database.

Relational Algebra Operators are critical mathematical tools used to retrieve queries by describing a sequence operation on tables or even databases (schema) involved mathematically. With relational algebra operators, a query is always composed of a number of operators, which each in turn are composed of relations as variables and return an individual abstraction as the end product. Relational Algebra operations can easily be translated into SQL commands to retrieve query results, making it a powerful tool in the hands of any Database designer, user, and administrator.

1.11 Self assessment Questions

1. What do you mean by RDBMS? Give Examples of it..

……
……
……
……
……
……

2. What is a relation? What are its characteristics?

……
……
……
……
……
……
……
……

3. What are the relational algebra operations that can be performed? What are the conditions for different set operations in RA?

……
……
……
……
……
……
……
……

4. What are the conditions for insertion operation?

……
……
……
……
……
……
……

5. Write short notes on natural join, theta joins.

……
……
……

……
……
……
……

1.12 Model Questions

1. Define the five basic operators of relational algebra with an example of each.
2. What is relational calculus? Differentiate relational algebra and relational calculus.
3. Consider the following relation schemes:
   Project (Project#, Project_name, Chief_architect)
   Employee (Emp#, Empname)
   Assigned_To (Project#, Emp#)
   Write the relational algebra expressions for each of the queries below:
   (i) Get the employee numbers of employees who work on all projects.
   (ii) Get the employee numbers of those employees who do not work on the TELCOM project.
4. Explain the basic features, advantages, disadvantages and importance of relational databases.
5. Discuss different types of join operations used in relational algebra with example.

1.13 References &Suggested Readings

1. Date, C.J., Introduction to Database Systems (7th Edition) Addison Wesley.
2. Leon, Alexis and Leon, Mathews, Database Management Systems, LeonTECH World
3. Elamasri R .and Navathe, S., Fundamentals of Database Systems (3rd Edition), Pearson Education
4. http://www.databasejournal.com/
5. Kato Mivule, "An Introduction to Relational Algebra", By C. Herbert, 2005.

UNIT-2 RELATIONAL QUERY LANGUAGES

Unit Structure

2.0 Introduction

Relational query languages allows user to interactively interrogate the database, analyze its data and update it according to the users privileges on data. It also controls the security of the database. Data security prevents unauthorized users from viewing or updating the database. Using passwords, users are allowed access to the entire database. It is the language by which user communicates with the database. Today almost all RDBMS (MySQL, Oracle, Informix, Sybase, MS Access, and SQL Server) uses SQL as the standard database language. SQL is used to perform all type of data operations in RDBMS. These relational query languages can be procedural or non-procedural.

2.1 Learning Objectives

At the end of this unit the reader will be able to:

- Differentiate between procedural and nonprocedural languages
- Know the relational database rule called Codd's rule.
- Characteristics of SQL commands
- Understand how various SQL commands are used?
- Identify different DDL, DML and DCL commands.
- Understand the constraints and indexes in SQL

2.2 Procedural Query Language

A procedural query language is a collection of queries binds in a single block called procedure, instructing the DBMS to perform various transactions in the sequence to meet the user request. For example, Calculate_Net_Sal is a procedure (which gives you the Net salary of an employee) will have various queries to get the no. of present days, no of leave, calculate total present day, add TA, DA etc. , deduct LIC, hr and then calculate the Net salary of an employee. This procedural query language tells the database what is required from the database and how to get them from the database. Relational algebra is a procedural query language.

2.3 Non-Procedural Query Language

Non-Procedural Language was a relational database language. Non-procedural queries will have a single query based on one or more tables to get the result from database. For example, get the name and address of an Employee with particular EMP_ID will have single query on Employee table. Relational Calculus is a non-procedural language which informs what to do with the tables, but doesn't inform how to accomplish this.
These query languages basically will have queries on tables in the database (or relations). A table is made up of rows and columns. A row is also called a record (or tuple). A column is also called a field (or attribute).

2.4 Codd's Rule

E. F. Codd's introduced the term in his seminar paper "A relational Model of Data for Large Shared Data Banks", published in 1970. In this unit he defined what he meant by relational. One well-known definition of what constitutes a relational database system is Codd's 12 rules.

Rule 1: Information rule
This rule states that all information, which is stored in the database, must be a value of some table cell.

Rule 2: Guaranteed Access rule
This rule states that every single data element (value) is guaranteed to be accessible logically with combination of table-name, primary-key (row value) and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL values
This rule states the NULL values in the database must be given a systematic treatment. As a NULL may have several meanings, i.e. NULL can be interpreted as one the following: data is missing, data is not known, data is not applicable etc.

Rule 4: Active online catalog
This rule states that the structure description of whole database must be stored in an online catalog, i.e. data dictionary, which can be accessed by

the authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive data sub-language rule
This rule states that a database must have a support for a language which has linear syntax which is capable of data definition, data manipulation and transaction management operations. Database can be accessed by means of this language only, either directly or by means of some application. If the database can be accessed or manipulated in some way without any help of this language, it is then a violation.

Rule 6: View updating rule
This rule states that all views of database, which can theoretically be updated, must also be updatable by the system.

Rule 7: High-level insert, update and delete rule
This rule states the database must employ support high-level insertion, updating and deletion. This must not be limited to a single row that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical data independence
This rule states that the application should not have any concern about how the data is physically stored. Also, any change in its physical structure must not have any impact on application.

Rule 9: Logical data independence
This rule states that the logical data must be independent of its user's view (application). Any change in logical data must not imply any change in the application using it. For example, if two tables are merged or one is split into two different tables, there should be no impact the change on user application. This is one of the most difficult rule to apply.

Rule 10: Integrity independence
This rule states that the database must be independent of the application using it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes database independent of the front-end application and its interface.

Rule 11: Distribution independence
This rule states that the end user must not be able to see that the data is distributed over various locations. User must also see that data is located at one site only. This rule has been proven as a foundation of distributed database systems.

Rule 12: Non-subversion rule
This rule states that if a system has an interface that provides access to low level records, this interface then must not be able to subvert the system and bypass security and integrity constraints.

2.5 Introduction to Structured Query Languages (SQL)

In this unit we want to emphasize that SQL is both deep and wide. Deep in the sense that it is implemented at many levels of database communication, from a simple Access form list box right up to high-volume communications between mainframes. SQL is widely implemented in that almost every DBMS supports SQL statements for communication. The reason for this level of acceptance is partially explained by the amount of effort that went into the theory and development of the standards.
Current State
So the ANSI-SQL group has published three standards over the years:
• SQL89 (SQL1)
• SQL92 (SQL2)
• SQL99 (SQL3)
The vast majority of the language has not changed through these updates. We can all profit from the fact that almost all of the code we wrote to SQL standards of 1989 is still perfectly usable. Or in other words, as a new student of SQL there is over ten years ofSQL code out there that needs your expertise to maintain and expand.
Most DBMS are designed to meet the SQL92 standard. Since many of the advanced features of SQL92 have yet to be implemented by DBMS vendors, there has been little pressure for a new version of the standard. Nevertheless a SQL99 standard was developed to address advanced issues in SQL. All of the core functions of SQL, such as adding, reading and modifying data, are the same. Therefore, the topics in this unit are not affected by the new standard. As of early 2001, no vendor has implemented theSQL99 standard.
There are three areas where there is current development in SQL standards. First entails improving Internet access to data, particularly to meet the needs

of the emerging XMLstandards. Second is integration with Java, either through Sun's Java Database Connectivity (JDBC) or through internal implementations. Last, the groups that establishSQL standards are considering how to integrate object- based programming models.

2.5.1 What is SQL?

Structured Query Language, commonly abbreviated to SQL and pronounced as “sequel”, is not a conventional computer programming language in the normal sense of the phrase. It allows users to access data in relational database management systems. SQL is about data and results; each SQL statement returns a result, whether that result is a query, anupdate to a record or the creation of a database table. SQL is most often used to address a relational database, which is what some people refer to as a SQL database. So in brief we can describe SQL as follows:

• SQL stands for Structured Query Language
• SQL allows you to access a database
• SQL can execute queries against a database
• SQL can retrieve data from a database
• SQL can insert new records in a database
• SQL can delete records from a database
• SQL can update records in a database
• SQL is easy to learn

In other word we can say SQL is a database computer language designed for the retrieval and management of data in relational database management system (RDBMS), database schema creation and modification and database object access control management.

SQL is not a single language of its own. It is the combination of 5 different sub languages those are DQL (Data query Language), DDL (Data definition Language), DML (Data manipulation Language), DCL (Data control Language) and TCL (Transaction control Language).

2.5.2 Why SQL?

- Allows users to access data in relational database management systems.
- Allows users to describe the data.
- Allows users to define the data in database and manipulate that data.
- Allows embedding within other languages using SQL modules, libraries & pre-compilers.
- Allows users to create and drop databases and tables.

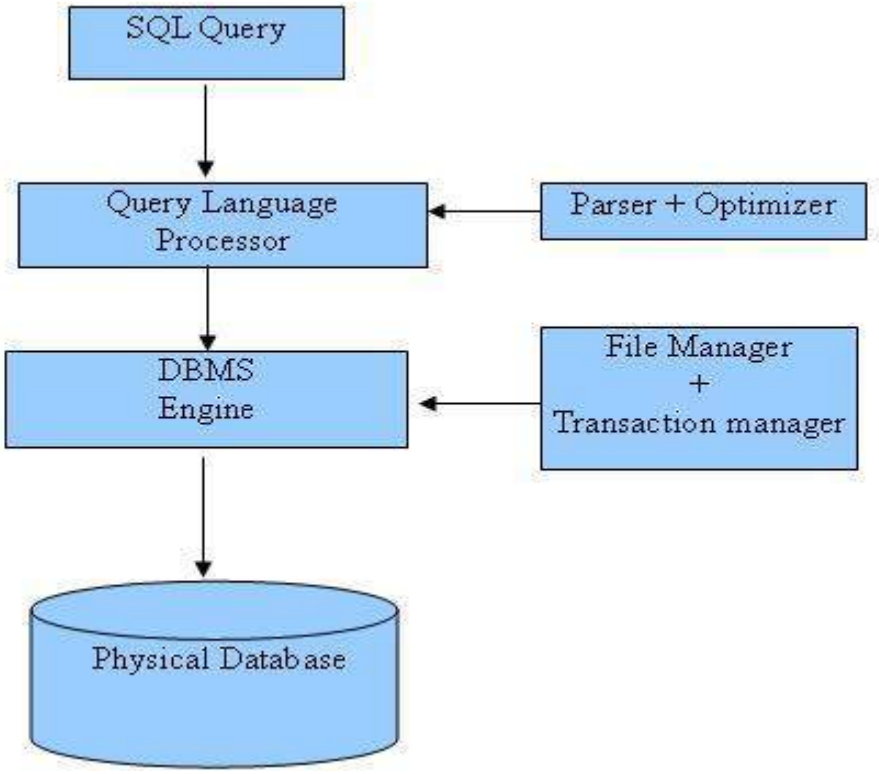
- Allows users to create view, stored procedure, functions in a database.
- Allows users to set permissions on tables, procedures, and views

2.5.3 SQL Process

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

There are various components included in the process. These components are Query Dispatcher, Optimization Engines, Classic Query Engine and SQL Query Engine, etc. Classic query engine handles all non-SQL queries but SQL query engine won't handle logical files.

Following is a simple diagram showing SQL Architecture:



2.5.4 Advantages of SQL

- **High Speed**

SQL Queries can be used to retrieve large amounts of records from a database quickly and efficiently.

- **Well Defined Standard Exist**

SQL databases use long-established standard, which is being adopted by ANSI & ISO. Non-SQL databases do not adhere to any clear standard.

- **No Coding Required**

Using standard SQL it is easier to manage database systems without having to write substantial amount of code.

- **Emergence of RDBMS**

Previously SQL databases were synonymous with relational database. With the emergence of Object Oriented DBMS, object storage capabilities are extended to relational databases.

2.5.5 Disadvantages of SQL

- **Difficulty in Interfacing**

Interfacing an SQL database is more complex than adding a few lines of code.

- **More Features Implemented in Proprietary way**

Although SQL databases conform to ANSI & ISO standards, some databases go for proprietary extensions to standard SQL to ensure vendor lock-in.

2.6 Data Types in SQL

SQL data type is an attribute that specifies type of data of any object. Each column, variable and expression has related data type in SQL.
You would use these data types while creating your tables. You would choose a particular data type for a table column based on your requirement.
SQL Server offers six categories of data types for your use

Numeric Data Types

DATA TYPE	FROM	TO
bigint	-9,223,372,036,854,775,808	+9,223,372,036,854, 775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1
money	-922,337,203,685,477.5808	+922,337,203,685,4 77.5807

smallmoney	-214,748.3648	+214,748.3647

Character strings

- CHARACTER(n) or CHAR(n): fixed-width n-character string, padded with spaces as needed
- CHARACTER VARYING(n) or VARCHAR(n): variable-width string with a maximum size of n characters
- NATIONAL CHARACTER(n) or NCHAR(n): fixed width string supporting an international character set
- NATIONAL CHARACTER VARYING(n) or NVARCHAR(n): variable-width NCHAR string

Bit strings

- BIT(*n*): an array of *n* bits
- BIT VARYING(*n*): an array of up to *n* bits

Numbers

- INTEGER, SMALLINT and BIGINT
- FLOAT, REAL and DOUBLE PRECISION
- NUMERIC(*precision*, *scale*) or DECIMAL(*precision*, *scale*)

For example, the number 123.45 has a precision of 5 and a scale of 2. The *precision* is a positive integer that determines the number of significant digits in a particular radix (binary or decimal). The *scale* is a non-negative integer. A scale of 0 indicates that the number is an integer. For a decimal number with scale S, the exact numeric value is the integer value of the significant digits divided by 10S.

Temporal (date/time)

- **DATE**: for date values (e.g. 2011-05-03)
- **TIME:** for time values (e.g. 15:51:36). The granularity of the time value is usually a *tick* (100 nanoseconds).
- **TIME WITH TIME ZONE or TIMETZ:** the same as TIME, but including details about the time zone in question.
- **TIMESTAMP:** This is a DATE and a TIME put together in one variable (e.g. 2011-05-03 15:51:36).
- **TIMESTAMP WITH TIME ZONE or TIMESTAMPTZ:** the same as TIMESTAMP, but including details about the time zone in question.

2.7 Operator

In SQL, Operator is nothing but a symbol or preserve word or a character used to perform a specific task over the operands. Operators are used to specify conditions in an **SQL** statement and to serve as conjunctions for multiple conditions in a statement. Also Operators are used for row restriction in SQL. Row restriction is basically a concept through which those many records are displayed where a condition is satisfied.

SQL provides basically 3 types of Operator.

1. Arithmetic Operator
2. Relational Operator
3. Logical Operator

Arithmetic Operator

The symbol which is used to perform mathematical operation is called Arithmetic Operator. It must be used with numbers (integer, float and double).

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
+ (Add)	Used to perform the addition operation.	a + b will give 30
-(Subtract)	Used to perform subtraction operation.	a - b will give -10
*(Multiplication)	Used to perform multiplication operation.	a * b will give 200
/(Division)	Used to perform division operation.	b / a will give 2
%(Modules)	Used to get the reminder.	b % a will give 0

Syntax:

```
SELECT <Expression> [arithmetic operator] <expression>...
FROM [table_name]
WHERE [expression];
```

<> Compulsory
[ ] Optional

Example:

Employee

Eid	Ename	Basic_Sal	TA	DA	LIC
100	Saroj	5600	560	280	400
101	Ranjit	4600	460	230	200
102	Himansu	8000	960	400	750
103	Bijay	6200	744	310	550

- From the above table (Employee) display the employee id, name and net salary of that employee.

Net Salary = (Basic_sal+TA+DA) - LIC

Select Eid,Ename,((Basic_sal+TA+DA) – LIC) as Net_Sal
from Employee;

Result Table

Eid	Ename	Net_Sal
100	Saroj	6040
101	Ranjit	5090
102	Himansu	8610
103	Bijay	6704

Relational Operator

Relational operator is also known as Comparison Operator. These operators are used to compare data in column to the specified in SQL statement. The Relational operator should be used with WHERE clause.

Assume variable a holds 10 and variable b holds 20, then:

Operator	Description	Example
= (Equal to)	Used to compare the value of two variable or fields or not. If operand values are equal then it produces the result.	(a = b) is not true.
!= (Not Equal to)	Used to compare the value of two variable or fields or not. If operand values are not equal then it produces the result.	(a != b) is true.
<>	Used to Checks if the	(a <> b) is true.

	values of two operands are equal or not, if values are not equal then condition becomes true.	
> (Greater than)	Used to compare the value of two variable or fields, if left value/field is greater than right value/field then it produce result.	(a > b) is not true.
< (Less than)	Used to compare the value of two variable or fields, if left value/field is less than right value/field then it produce result.	(a < b) is true.
>= (Greater than or equal)	Used to compare the value of two variable or fields, if left value/field is greater than or equal to right value/field then it produce result.	(a >= b) is not true.
<= (Less than or equal)	Used to compare the value of two variable or fields, if left value/field is less than or equal to right value/field then it produce result.	(a <= b) is true.
!<	Used to the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

Example:

- From the above table (Employee) display the employee id, name and net salary of that employee whose Net Salary greater than 6500

Select Eid,Ename,((Basic_sal+TA+DA) – LIC) as Net_Sal
from Employee
where Net_Sal> 6500;

Result Table

Eid	Ename	Net_Sal
102	Himansu	8610
103	Bijay	6704

- From the above table (Employee) display the employee id, name and Basic salary of that employee whose Basic Salary Less than 7000

Select Eid, Ename, Basic_sal
from Employee
where Basic_Sal< 7000;

Result Table

Eid	Ename	Basic_Sal
100	Saroj	5600
101	Ranjit	4600
103	Bijay	6200

Logical operator

Logical operators are produces the result if and only if the condition is true. SQL provides 3 types of operator such as AND, OR, NOT.

Operator	Description
AND	If both side condition are true then AND operator produces result.
OR	If any one side of the condition or both the condition are true then OR operator produces result.
NOT	It produces result if condition is not matched or satisfied.

Other than conventional operators SQL provides some exclusive Operator those are

Operator	Description
BETWEEN AND	This operator is responsible to extract data from a table between a given ranges of value. This operator can be operated on number and Date type of values.
LIKE	This operator is used to extract data from a table depending on character position

	(Match a character pattern). This operator can be operated on 2 types of values i.e. character and Date. LIKE operator provides two wild card characters to do so. 1. **Underscore (_)**: To represent a single character. 2. **Percentage (%)**: To represent multiple character.
IN	This operator is the extended version of OR operator. It produce result when Equal to one of multiple possible values.
IS	Compare to null (missing data)
AS	Used to change a field name when viewing results

2.8 Data Query Language (DQL)

DQL is otherwise known as Data Retrieval Language (DRL), responsible to retrieve data from a table and display to the end-user, to do so SQL provides a statement called **SELECT**.

SELECT is a powerful query statement responsible to extract data from one/more than one table, condition wise or non-condition wise.

Syntax:

SELECT [DISTINCT] <col1, col2, col3 ……..col n>
From <Table Name>;

[ ] – Optional

<> - Compulsory

SELECT statement can perform 2 types of job

1. **Selection**
2. **Projection**

Projection can be possible in two different manners.

1. **Column wise projection**
   Column wise projection is basically a concept through which these many attributes are displayed which are mentioned in the Select list.
2. **Row wise projection**

Whereas Row wise projection is a concept through which those many records are displayed where a condition satisfied.

Note. **'*' represent all the columns of a particular table. Each statement in SQL is terminated by semicolon (;).**

2.9 Data Definition Language (DDL)

DDL is otherwise known as Object Manipulation Language (OML). It is responsible to manipulate objects in Database. Database objects such as Table, View, Sequence, Index, Type etc. To manipulate the mentioned objects DDL provides the following commands CREATE, ALTER, DROP, RENAME, TRUNCATE.

CREATE – To Create Objects in the Database.

> **Note:** A*database object* is any defined object in a database that is used to store or reference data. Some examples of database objects include tables, views, clusters, sequences, indexes, and synonyms. The table is this hour's focus because it is the primary and simplest form of data storage in a relational database.

Syntax for Creating a Table Object:

CREATE Object Object_Name(Column_Name1 Datatype, Column_Name2 Datatype, ... Column_NameN Datatype);

Example:

CREATE TABLE Student (SnoNUMBER(3), Sname VARCHAR (10), class VARCHAR (5));

ALTER –Used toalters the structure of the Database.

ALTER Object Object_Name ADD (new field_1 data_type (size), new field_2 data_type(size),..);

ALTER TABLE ...ADD...: This is used to add some extra fields into existing relation.
Syntax:
ALTER TABLE relation_name ADD (new field_1 data_type (size), new field_2 data_type(size),..);
Example:

ALTER TABLE std ADD (Address CHAR (10));

ALTER TABLE...MODIFY...: This is used to change the width as well as data type of fields of existing relations.
Syntax:
ALTER TABLE relation_name MODIFY (field_1 newdata_type(Size), field_2
newdata_type(Size),....field_newdata_type(Size));
Example:
ALTER TABLE student MODIFY(sname VARCHAR(10),class VARCHAR(5));

ALTER TABLE..DROP...: This is used to remove any field of existing relations.
Syntax:
ALTER TABLE relation_name DROP COLUMN (field_name);
Example:
ALTER TABLE student DROP column (sname);

ALTER TABLE..RENAME...: This is used to change the name of fields in existing relations.
Syntax:
ALTER TABLE relation_name RENAME COLUMN (OLD field_name) to (NEW field_name);
Example:
ALTER TABLE student RENAME COLUMN snametostu_name;

DROP –Used todelete objects from the database.
Syntax:
DROP Object Object_Name;
Example:
DROP Table Student;

RENAME –Used to **rename** an object.
Syntax:
RENAME Object Object_Old_name TO Object_new_name;
Example:
RENAME TABLE Student TO BStudent;

TRUNCATE –Used to remove all records from a table, including all spaces allocated for the records are removed

Syntax:

TRUNCATE Object <Object_name>;

Example:

TRUNCATE TABLE Student;

2.10 Data Manipulation Language (DML)

DML is otherwise known as Record Manipulation Language (RML). It is responsible to manipulate the records of a table and to do so DML provides the following commands INSERT, UPDATE, DELETE.

INSERT – Used toinsert data into a table.

Syntax:

INSERT INTO < relation/table name> (field_1, field_2.....field_n)VALUES (data_1,data_2,.....data_n);

Example:

INSERT INTO Student (sno, sname, class, address) VALUES (1,'Ravi','M.Tech','Palakol');

UPDATE –Used toupdates existing data within a table.

Syntax:

UPDATE relation name SET

Field_name1=data,field_name2=data, WHERE field_name=data;

Example:

UPDATE student SET sname = 'kumar' WHERE sno=1;

DELETE – Used todeletes all records from a table, the space for the records remain.

Syntax:

SQL>DELETE FROM relation_name WHERE condition;

Example:

DELETE FROM student WHERE Sno = 2;

Difference between Truncate & Delete

- By using truncate command data will be removed permanently & will not get back where as by using delete command data will be removed temporally & get back by using roll back command.

- By using delete command data will be removed based on the condition whereas by using truncate command there is no condition.
- Truncate is a DDL command & delete is a DML command.

2.11 Transaction Control Language (TCL)

Whenever any transaction is made in database table (Insert, Update, and Delete), the transaction never be save unless until a save command / a discard command is issued, to do so TCL provides the following commands COMMIT, ROLLBACK, SAVEPOINT.

ROLLBACK is used for revoking the transactions until last commit.

SYNTAX:

```
ROLLBACK [WORK]
   [ TO [SAVEPOINT] savepoint
   | FORCE 'text' ]
```

Where:

- WORK : is optional and is provided for ANSI compatibility.
- TO : rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.
- FORCE : manually rolls back an in-doubt distributed transaction. The transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. ROLLBACK statements with the FORCE clause are not supported in PL/SQL.

Example:

To rollback your current transaction, enter

```
SQL> ROLLBACK;
```

Rollback complete.

COMMIT is used for committing the transactions to the database. Once we commit we cannot rollback. Once we rollback we cannot commit. Commit and Rollback are generally used to commit or revoke the transactions that are with regard to DML commands.

SYNTAX:

```
COMMIT [WORK]
   [ COMMENT 'text'
   | FORCE 'text' [, integer] ]
```

Where:

- WORK : is supported only for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.
- COMMENT : specifies a comment to be associated with the current transaction. The 'text' is a quoted literal of up to 50 characters that Oracle stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in-doubt.
- FORCE : manually commits an in-doubt distributed transaction. The transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can also use the integer to specifically assign the transaction a system change number (SCN). If you omit the integer, the transaction is committed using the current SCN.

  COMMIT statements using the FORCE clause are not supported in PL/SQL.

Example:

To commit your current transaction, enter

SQL> **COMMIT WORK;**

Commit complete.

SAVEPOINT is used to identify a point in a transaction to which you can later roll back

Syntax:

SAVEPOINT <SavePointName>

Example:

```
DECLARE
        Total_Sal  number(9);
BEGIN
    INSERT INTO Emp  VALUES('E101', 'Aamir', 10, 7000');
    INSERT INTO Emp  VALUES('E102', 'Aatif', 11, 6500');


SAVEPOINT   no_update;

UPDATE Emp SET salary =salary+2000 WHERE Emp_Name = 'Aamir';
UPDATE Emp SET salary =salary+2000 WHERE Emp_Name = 'Aatif';

SELECT sum(Salary) INTO Total_sal  FROM Emp;

IF  Total_Sal > 15000  THEN
    ROLLBACK  To  SAVEPOINT   no_update;
 END IF;

COMMIT;
END;
```

2.12 Data Control Language (DCL)

Data Control Language is used for the control of data. That is a user can access any data based on the privileges given to him. This is done through DATA CONTROL LANGUAGE. Some of the DCL Commands are GRANT, REVOKE.

GRANT is used to allow specified users to perform specified tasks.
SQL GRANT is a command used to provide access or privileges on the database objects to the users.

The Syntax for the GRANT command is:

GRANT privilege_name
ON object_name
TO {user_name |PUBLIC |role_name}
[WITH GRANT OPTION];

privilege_name is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
object_name is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
user_name is the name of the user to whom an access right is being granted.
user_name is the name of the user to whom an access right is being granted.
PUBLIC is used to grant access rights to all users.
ROLES are a set of privileges grouped together.

Example:
GRANT SELECT ON employee TO user1;This command grants a SELECT permission on employee table to user1.

REVOKE is used to cancel previously granted or denied permissions. This command removes user access rights or privileges to the database objects.

The Syntax for the REVOKE command is:

REVOKE privilege_name

ON object_name
FROM {user_name |PUBLIC |role_name}

Example:

REVOKE SELECT ON employee FROM user1;This commmand will REVOKE a SELECT privilege on employee table from user1.When you REVOKE SELECT privilege on a table from a user, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it. You cannot REVOKE privileges if they were not initially granted by you.

2.13 Constraints in SQL

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should only be one row for each product number. To that end, SQL allows you to define constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition.

Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy an arbitrary expression. For instance, to require positive product prices, you could use:

CREATE TABLE products (product no integer, name text, price numeric CHECK (price > 0));

As you see, the constraint definition comes after the data type, just like default value definitions. Default values and constraints can be listed in any order. A check constraint consists of the key word CHECK followed by an expression in parentheses. The check constraint expression should involve

the column thus constrained; otherwise the constraint would not make too much sense.

Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value. A syntax example:

CREATE TABLE products (product no integer NOT NULL, name text NOTNULL, price numeric);

A not-null constraint is always written as a column constraint. A not-null constraint is functionally equivalent to creating a check constraint CHECK (column name IS NOTNULL), but in PostgreSQL creating an explicit not-null constraint is more efficient. The drawback is that you cannot give explicit names to not-null constraints created that way.

Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table. The syntax is

CREATE TABLE products (product no integer UNIQUE, name text, price numeric);

When written as a column constraint, and

CREATE TABLE products (productno integer, name text, price umeric, UNIQUE (product_no));

When written as a table constraint.

Primary Key Constraints

Technically, a primary key constraint is simply a combination of a unique constraint and not-null constraint. So, the following two table definitions accept the same data:

CREATE TABLE products (product_no integer UNIQUE NOT NULL, name text, price numeric);

CREATE TABLE products (product_no integer PRIMARY KEY,name text, price numeric);

Primary keys can also constrain more than one column; the syntax is similar to unique constraints:

CREATE TABLE example (a integer, b integer, c integer, PRIMARY KEY (a, c));

A primary key indicates that a column or group of columns can be used as a unique identifier for rows in the table. (This is a direct consequence of the

definition of a primary key. Note that a unique constraint does not, in fact, provide a unique identifier because it does not exclude null values.) This is useful both for documentation purposes and for client applications. For example, a GUI application that allows modifying row values probably needs to know the primary key of a table to be able to identify rows uniquely.

Foreign Keys Constraints

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the referential integrity between two related tables.

Say you have the product table that we have used several times already:
CREATE TABLE products (product_no integer PRIMARY KEY, name text, price numeric);
Let's also assume you have a table storing orders of those products. We want to ensure that the orders table only contains orders of products that actually exist. So we define a foreign key constraint in the orders table that references the products table:
CREATE TABLE orders (order_id integer PRIMARY KEY, product_nointeger REFERENCES products (product_no), quantity integer);
Now it is impossible to create orders with product_no entries that do not appear in the products table.
We say that in this situation the orders table is the referencing table and the products table is the referenced table. Similarly, there are referencing and referenced columns.

2.14 Let Us Sum Up

SQL allows users to access data in relational database management systems. There are three groups of SQL commands viz., DDL, DML and DCL. The Data Definition Language (DDL) part of SQL permits database tables to be created or deleted. SQL language also includes syntax to update, insert, and delete records. These query and update commands together form the Data Manipulation Language (DML) part of SQL. The SQL Data Control Language (DCL) provides security for your database. The DCL consists of the GRANT, REVOKE, COMMIT, and ROLLBACK statements. Constraints are a way to limit the kind of data that can be stored in a table. Relational databases like SQL Server use indexes to find data quickly when a query is processed.

2.15 Self assessment Questions

1. What does SQL stand for?

   ……
   ……
   ……
   ……
   ……
   ……
   ……
   ……
   ……...

2. Explain the function of each of the clauses in the SELECT statement.

   ……
   ……
   ……
   ……
   ……
   ……
   ……
   ……

3. Explain how the GROUP BY clause works. What is the difference between the WHERE and HAVING clauses?

   ……
   ……
   ……
   ……
   ……
   ……
   ……

……

……

……

……

……

4. Explain the use of Grant and Revoke Commands?

……

……

……

……

……

……

……

……

……

……

5. What are Transaction Control Language Commands?

……

……

……

……

……

……

……

……

……

2.16 Model Questions

1. Discuss different data types in SQL.
2. Explain the uses of different DDL Comands.
3. Write the syntax of different DML commands and give examples statements for each of them
4. Explain different types of constraints used in SQL.
5. Write a short note on DQL.

2.17 References &Suggested Readings

1. Sams Teach Yourself Microsoft SQL Server 2000 in 21 Days by Richard Waymire, Rick Sawtell
2. Microsoft SQL Server 7.0 Programming Unleashed by John Papa, et al
3. www.microsoft.com
4. Microsoft SQL Server 7.0 Resource Guide by Microsoft Corporation
5. http://www.cs.rpi.edu/~temtanay/dbf-fall97/oracle/sql_ddcmd.html
6. International Journal of Research in Science and Technology http://www.ijrst.com (IJRST) 2012, Vol. No. 2, Issue No. III, Oct-Dec ISSN: 2249-060

UNIT-3 FUNCTIONAL DEPENDENCY

Unit Structure

3.0 Introduction

As we have explained in the earlier units, the purpose of database design is to arrange the corporate data fields into an organized structure such that it generates set of relationships and stores information without unnecessary redundancy. In fact, the redundancy and database consistency are the most important logical criteria in database design. A bad database design may result into repetitive data and information and an inability to represent desired information. It is, therefore, important to examine the relationships that exist among the data of an entity to refine the database design.
In this unit, functional dependencies and decomposition concepts which are the important for a good database designs that have been discussed to achieve minimum redundancy and maximum consistencies.

3.1 Learning Objectives

After learning this unit you should be able to

- Understand the concept of Functional dependencies.
- Know about different types of Functional dependencies.
- Apply the rules for Functional dependencies
- Advantages of Functional Dependency
- Rules for constructing Functional Dependency Diagrams.
- About the Functional Decomposition and its property.
- Understand the concept of normalization and its types.

3.2 Database Dependencies

Functional dependencies are a topic that often confuses both students and database professionals alike. Fortunately, they are not that complicated and can best be illustrated through the use of a number of examples. In this unit, we will discuss common database dependency or functional dependency and its types. We will synonymously use the database dependencies and Functional dependencies as well.

3.2.1 Functional Dependencies

A dependency occurs in a database when information stored in the same database table uniquely determines other information stored in the same table.

You can also describe this as a relationship where knowing the value of one attribute (or a set of attributes) is enough to tell you the value of another attribute (or set of attributes) in the same table.

Saying that there is a dependency between attributes in a table is the same as saying that there is a functional dependency between those attributes. If there is a dependency in a database such that attributes B is dependent upon attribute A, you would write this as "A → B".

In other words, a dependency FD: $X \rightarrow Y$ means that the values of Y are determined by the values of X. Two tuples sharing the same values of X will necessarily have the same values of Y.

For example, in table listing employee characteristics including Social Security Number (SSN) and name, it can be said that name is dependent upon SSN (or SSN → name) because an employee's name can be uniquely determined from their SSN. However, the reverse statement (name → SSN) is not true because more than one employee can have the same name but different SSNs.

3.2.2 Trivial Functional Dependencies

A **trivial functional dependency** occurs when you describe a functional dependency of an attribute on a collection of attributes that includes the original attribute.

For example, "{A, B} → B" is a trivial functional dependency, as is "{name, SSN} → SSN". This type of functional dependency is called trivial because it can be derived from common sense. It is obvious that if you already know the value of B, then the value of B can be uniquely determined by that knowledge.

3.2.3 Full Functional Dependencies

A **full functional dependency** occurs when you already meet the requirements for a functional dependency and the set of attributes on the left side of the functional dependency statement cannot be reduced any further. For example, "{SSN, age} −> name" is a functional dependency, but it is not a full functional dependency because you can remove age from the left side of the statement without impacting the dependency relationship.

3.2.4 Transitive Dependencies

Transitive dependencies occur when there is an indirect relationship that causes a functional dependency. For example, "A → C" is a transitive

dependency when it is true only because both "A → B" and "B −> C" are true.

3.2.5 Multi-valued Dependencies

Multi-valued dependencies occur when the presence of one or more rows in a table implies the presence of one or more other rows in that same table. For example, imagine a car company that manufactures many models of car, but always makes both red and blue colors of each model. If you have a table that contains the model name, color and year of each car the company manufactures, there is a multi-valued dependency in that table. If there is a row for a certain model name and year in blue, there must also be a similar row corresponding to the red version of that same car.

3.2.6 Importance of Dependencies

Database dependencies are important to understand because they provide the basic building blocks used in database normalization. For example:

- For a table to be in second normal form (2NF), there must be no case of a non-prime attribute in the table that is functionally dependent upon a subset of a candidate key.
- For a table to be in third normal form (3NF), every non-prime attribute must have a non-transitive functional dependency on every candidate key.
- For a table to be in Boyce-Codd Normal Form (BCNF), every functional dependency (other than trivial dependencies) must be on a superkey.
- For a table to be in fourth normal form (4NF), it must have no multi-valued dependencies.

3.3 Study of Functional Dependency

- **Functional Dependency** is a relationship that exists between multiple attributes of a relation.
- This concept is given by **E. F. Codd.**
- Functional dependency represents formalism on the infrastructure of relation.
- It is a type of constraint existing between various attributes of a relation.
- It is used to define various normal forms.
- These dependencies are restrictions imposed on the data in database.

- If P is a relation with A and B attributes, a functional dependency between these two attributes is represented as {A → B}. It specifies that,

A	It is a determinant set.
B	It is a dependent attribute.
{A → B}	A functionally determines B. B is a functionally dependent on A.

- Each value of A is associated precisely with one B value. A functional dependency is trivial if B is a subset of A.
- 'A' Functionality determines 'B' {A → B} (Left hand side attributes determine the values of Right hand side attributes).

For example: <Employee> Table

EmpId	**EmpName**

- In the above <Employee> table, EmpName (employee name) is functionally dependent on EmpId (employee id) because the EmpId is unique for individual names.
- The EmpId identifies the employee specifically, but EmpName cannot distinguish the EmpId because more than one employee could have the same name.
- The functional dependency between attributes eliminates the repetition of information.
- It is related to a candidate key, which uniquely identifies a tuple and determines the value of all other attributes in the relation.

3.4 Basics of Functional Dependency

Functional dependency (FD) is a property of the information represented by the relation. Functional dependency allows the database designer to express facts about the enterprise that the designer is modeling with the enterprise databases. It allows the designer to express constraints, which cannot be expressed with super keys. Functional dependency is a term derived from mathematical theory, which states that for every element in the

attribute (which appears on some row), there is a unique corresponding element (on the same row).

Let us assume that rows (tuples) of a relational table T is represented by the notation r1 ,r2 ,…….., and individual attributes (columns) of the table is represented by letters A, B,…. The letters X, Y , ….., represent the subsets of attributes.

Thus, as per mathematical theory, for a given table T containing at least two attributes A and B, we can say that A −> B. The arrow notation '−>' is read as "functionally determines".

Thus, we can say that, A functionally determines B or B is functionally dependent on A. In other words, we can say that, given two rows R1, and R2, in table T, if R1 (A) =R2 (A) then R1 (B) =R2 (B).
The attributes in subset A are sometimes known as the determinant of **FD: A −> B**.
The left hand side of the functional dependency is sometimes called determinant whereas that of the right hand side is called the dependent. The determinant and dependent are both sets of attributes. A functional dependency is a many-to-one relationship between two sets of attributes X and Y of a given table T. Here X and Y are subsets of the set of attributes of table T. Thus, the functional dependency X → Y is said to hold in relation R if and only if, whenever two tuples (rows or records) of T have the same value of X, they also have the same value for Y.

3.5 Introduction to Axioms Rules

Armstrong's Axioms is a set of rules.
It provides a simple technique for reasoning about functional dependencies.
It was developed by William W. Armstrong in 1974.
It is used to infer all the functional dependencies on a relational database.
Various Axioms Rules
A. Primary Rules

Rule 1	**Reflexivity** If A is a set of attributes and B is a subset of A, then A holds B. { A → B }
Rule 2	**Augmentation**

	If A hold B and C is a set of attributes, then AC holds BC. {AC → BC} It means that attribute in dependencies does not change the basic dependencies.
Rule 3	**Transitivity** If A holds B and B holds C, then A holds C. If {A → B} and {B → C}, then {A → C} A holds B {A → B} means that A functionally determines B.

B. Secondary Rules

Rule 1	**Union** If A holds B and A holds C, then A holds BC. If{A → B} and {A → C}, then {A → BC}
Rule 2	**Decomposition** If A holds BC and A holds B, then A holds C. If{A → BC} and {A → B}, then {A → C}
Rule 3	**Pseudo Transitivity** If A holds B and BC holds D, then AC holds D. If{A → B} and {BC → D}, then {AC → D}

Sometimes Functional Dependency Sets are not able to reduce if the set has following properties,

1. The Right-hand side set of functional dependency holds only one attribute.
2. The Left-hand side set of functional dependency cannot be reduced, it changes the entire content of the set.
3. Reducing any functional dependency may change the content of the set.

A set of functional dependencies with the above three properties are also called as Canonical or Minimal.

3.5.1 Trivial Functional Dependency

Trivial	If A holds B {A → B}, where A is a subset of B, then it is called a **Trivial Functional Dependency**. Trivial always holds Functional Dependency.

Non-Trivial	If A holds B {A → B}, where B is not a subset A, then it is called as a **Non-Trivial Functional Dependency.**
Completely Non-Trivial	If A holds B {A → B}, where A intersect Y = Φ, it is called as a **Completely Non-Trivial Functional Dependency.**

Example:

Consider relation E = (P, Q, R, S, T, U) having set of Functional Dependencies (FD).

P → Q P → R
QR → S Q → T
QR → U PR → U

Calculate some members of Axioms are as follows,

1. P → T
2. PR → S
3. QR → SU
4. PR → SU

Solution:

1. P → T
   In the above FD set, P → Q and Q → T
   So, Using Transitive Rule: If {A → B} and {B → C},
   then {A → C}
   ∴ If P → Q and Q → T, then P → T.
   P → T
2. PR → S
   In the above FD set, P → Q
   As, QR → S
   So, Using Pseudo Transitivity Rule: If {A → B} and {BC → D},
   then {AC → D}
   ∴ If P → Q and QR → S, then PR → S.
   PR → S

3. QR → SU
   In above FD set, QR → S and QR → U
   So, Using Union Rule: If{A → B} and {A → C}, then {A → BC}
   ∴ If QR → S and QR → U, then QR → SU.
   QR → SU
4. PR → SU

So, Using Pseudo Transitivity Rule: If {A → B} and {BC → D}, then {AC → D}

∴ If PR → S and PR → U, then PR → SU.

PR → SU

3.6 Why Functional Dependency is Important in Database Design

Functional dependency helps ensure the validity of data. Consider a table Employees that lists characteristics including Social Security Number (SSN), name, date of birth, address and so on.

The attribute SSN will determine the value of name, date of birth, address and perhaps other values, because a social security number is unique, while a name, date of birth or address may not be. We can write it like this:

SSN → name, date of birth, address

Therefore, name, date of birth and address are functionally dependent on SSN. However, the reverse statement (name → SSN) is not true because more than one employee can have the same name but will never have the same SSN. Put another, more concrete way, if we know the value of the SSN attribute, we can find the value of name, date of birth and address. But if we instead know the value of only the name attribute, we cannot identify the SSN.

The left side of a functional dependency can include more than one attribute. Let's say we have a business with multiple locations. We might have a table Employee with attributes employee, title, department, location and manager.

The employee determines the location he's working, so there's a **dependency: employee → location**

But the location might have more than one manager, so employee and department together determine the manager: **employee, department → manager**

3.7 Main Characteristics of Functional Dependencies in Normalization

A functional dependency has a one-to-one relationship between attribute(s) on the left- and right- hand side of a dependency;

- Hold for all time; and are nontrivial.

Functional dependency is the property of the meaning or semantics of the attributes in a relation. When a functional dependency is present, the dependency is specified as a **constraint** between the attributes.

An important integrity constraint to consider first is the identification of candidate keys, one of which is selected to be the primary key for the relation using functional dependency.

3.7.1 Advantages of Functional Dependency

- Functional Dependency avoids data redundancy where same data should not be repeated at multiple locations in same database.
- It maintains the quality of data in database.
- It allows clearly defined meanings and constraints of databases.
- It helps in identifying bad designs.
- It expresses the facts about the database design.

3.8 Functional Dependency Diagrams

Drawing the diagrams: Functional dependency diagrams are a useful diagrammatic way of showing functional dependencies. They are especially helpful in deducing the closure of an attribute set or of a set of functional dependencies. These diagrams are not used by Silberschatz, Korth, and Sudarshan. They can be found in Date's book, but he does not give a formal definition of the rules for drawing these diagrams. The following are some guidelines for drawing functional dependency diagrams.

1. Each attribute in the relation schema appears only once in the diagram.
2. If the left side of a functional dependency consists of an irreducible set of attributes, these attributes are enclosed in a box from which the arrow for that FD originates.
3. All arrows terminate on single attributes. In other words, apply Armstrong's decomposition rule to turn FDs with multiple attributes on the right-hand side into multiple FDs with only one attribute on the right-hand side.

Example:

Given the set of functional dependencies

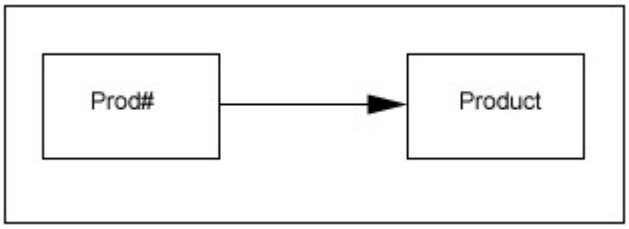
$A \rightarrow BC$

$CG \rightarrow HI$

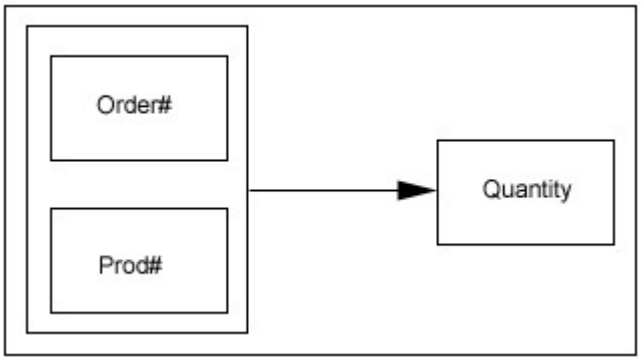
$B \rightarrow H$

A set of Functional Dependencies for a data model can be documented in a **Functional Dependency Diagram** (also known as a **Determinacy Diagram**).

In a **Functional Dependency Diagram** each attribute is shown in a rectangle with an arrow indicating the direction of the dependency. The figure below illustrates the functional dependency **Prod# > Product**.

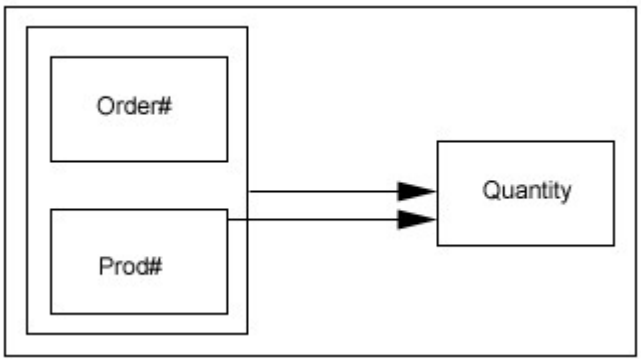


A Functional Dependency with Multiple Attributes is shown below, for the functional dependency **Order#, Prod# > Quantity**.



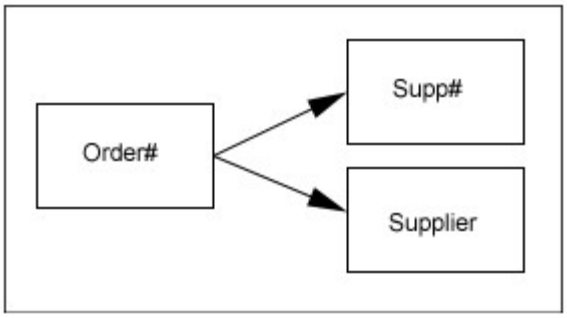
A derived Functional Dependency involving **Partial Key Dependency** is shown in the figure below.

The arrow connected to the outer rectangle, which represents **Order#, Prod# > Product** can be deleted without loss of information.



A derived Functional Dependency involving ***Transitive Dependency*** is shown in the figure below.

The arrow which represents **Order# > Supplier** can be deleted without loss of information.

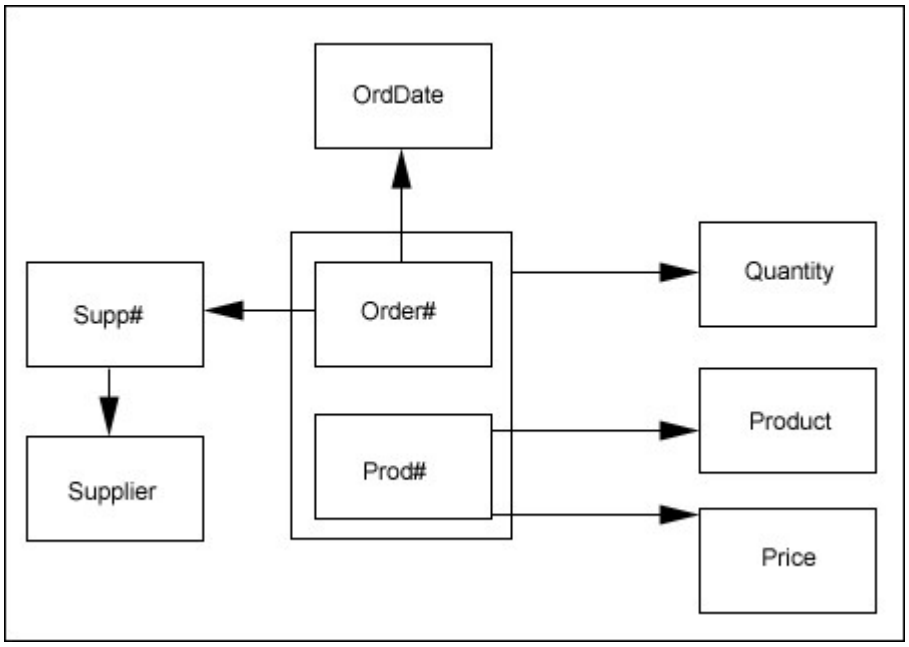


3.8.1 Rules for Functional Dependency Diagrams

The following rules apply to Functional Dependency Diagrams:

- each attribute appears only once on the Functional Dependency Diagram
- all the attributes of interest appear on the Functional Dependency Diagram
- no partial key dependencies appear on the Functional Dependency Diagram
- no transitive dependencies appear on the Functional Dependency Diagram

The complete Functional Dependency Diagram for the Purchase Order data model is shown below:



3.8.2 Coloring Algorithm

An algorithm for using a FD diagram to compute the closure of an attribute set is as follows:

Color each attribute in the given set.
while more attributes can be colored **do**
    If all attributes inside a group box are colored, color the group box.
    Color any attributes pointed to by an arrow from a colored box.

When finished, the closure of the attribute set is colored. For example, applying this procedure to the attribute set AG with the above functional dependency set, A and G are colored, then the first round colors B and C via A. Then since both C and G are colored, the box containing them is colored. Then H is colored via either B or CG, and I is colored via CG. At this point everything is colored, showing that AG^+ is the whole schema and hence AG is a candidate key.

3.9 Full Functional Dependency (FFD)

The term full functional dependency (FFD) is used to indicate the minimum set of attributes in of a functional dependency (FD). In other words, the set of attributes X will be functionally dependent on the set of attributes Y if the following conditions are satisfied:

- X is functionally dependent on Y and
- X is not functionally dependent on any subset of Y.

In relation ASSIGN of table it is true that

FD: {EMP-ID, PROJECT, PROJECT-BUDGET} → {YRS-SPENT-BY-EMP-ON PROJECT}

The values of EMP-ID, PROJECT and PROJECT-BUDGET determine a unique value of YRS-SPENT-BY-EMP-ON-PROJECT. However, it is not a full functional dependency because neither the **EMP-ID → YRS-SPENT-BY-EMP-ON-PROJECT** nor the **PROJECT → YRS-SPENT-BY-EMP-ON-PROJECT** holds true.

In fact, it is sufficient to know only the value of a subset of {EMP-ID, PROJECT, PROJECTBUDGET), namely, {EMP-ID, PROJECT}, to determine the YRS-SPENT-BY-EMP-ON PROJECT. Thus, the correct full

functional dependency (FFD) can be written as: **FD: {EMP-ID, PROJECT} → {YRS-SPENT-BY-EMP-ON-PROJECT}**

Relation R1: BUDGET

PROJECT	PROJECT-BUDGET
P1	INR 100 CR
P2	INR 150 CR
P3	INR 200 CR
P4	INR 100 CR
P5	INR 150 CR
P6	INR 300 CR

Relation R2: ASSIGN

EMP-NO	PROJECT	YRS-SPENT-BY-EMP-ON-PROJECT
106519	P1	5
112233	P3	2
106519	P2	5
123243	P4	10
106519	P3	3
111222	P1	4

3.10 Redundant Functional Dependencies

A functional dependency in the set is redundant if it can be derived from the other functional dependencies in the set. A redundant FD can be detected using the following steps:

Step 1: Start with a set of **S** of functional dependencies (FDs).

Step 2: Remove an FD f and create a set of FDs **S' = S - f** .

Step 3: Test whether f can be derived from the FDs in **S'**; by using the set of Armstrong's axioms and derived rules.

Step 4: If f can be so derived, it is redundant and hence **S' = S**. Otherwise replace f into **S'**; so that now **S' = S + f**.

Step 5: Repeat steps 2 to 4 for all FDs in **S**.

Armstrong's axioms and derived rules, as discussed in the previous section, can be used to find redundant FDs.

For example, suppose the following set of FDs is given in the algorithm:
$Z \rightarrow A \quad B \rightarrow X \quad AX \rightarrow Y \quad ZB \rightarrow Y$
Because $ZB \rightarrow Y$ Can be derived from other FDs in the set, it can be shown to be redundant.

The following argument can be given:

- $Z \rightarrow A$ by augmentation rule will yield $ZB \rightarrow AB$.
- $B \rightarrow X$ and $AX \rightarrow Y$ by pseudo-transitivity rule will yield $AB \rightarrow Y$.
- $ZB \rightarrow AB$ and $AB \rightarrow Y$ by transitivity rule will yield $ZB \rightarrow Y$.

An algorithm (called **membership algorithm**) can be developed to find redundant FDs, that is, to determine whether an **FD f (A → B)** can be derived from a set of FDs S. Using the algorithm the redundant functional dependency can be checked.

Algorithm:

Membership algorithm to find redundant functional dependency

Input: Let F be a set of FDs for relation R.

Steps:

1. F' = F - f **//find out new set of FDs by removing f from F.**

2. T = A **//set T = determinant of A → B**

3. for each FD:X → yin F' Do
a) If X ⊆ T Then **//if X is contained in T**
T = T ∪ Y **//add Y to T**
End if

4. if **B ⊆ T** then //if B is contained in T
f : A → B is redundant. //given FD f: A → B is redundant.
End if

Output: Decision whether a given FD f: A → B is redundant or not.

Example:

Suppose a relation R is given with attributes A, B, C, D, E. Also, a set of functional dependencies F is given with following FDs.
F = {A → B, C → D, BD → E, AC → E}

1. Find out whether a FD f : AC → E is redundant or not.

Step 1 : F' = {A → B, C → D, BD → E}
Step 2 : T = AC
Step 3 : T = AC + B = ACB
T = ACB + D = ACBD
T = ACBD + E = ACBDE
Step 4 : f:AC → E is redundant.

2. Find out whether a FD f: BD → E is redundant or not.

Step 1: F' = {A → B, C → D, AC → E}
Step 2: T= BD
Step 3: Nothing can be added to T, as there is no other FD : X → Y such that X⊆ T
Step 4: f: BD → E is not redundant.

3.11 Closures of a Set of Functional Dependencies

A **Closure** is a set of FDs is a set of all possible FDs that can be derived from a given set of FDs. It is also referred as a **complete** set of FDs. If F is used to donate the set of FDs for relation R, then a closure of a set of FDs implied by F is denoted by F^+. Let's consider the set F of functional dependencies given below:

F = {A → B, B → C, C → D}

From F, it is possible to derive following dependencies.
$A \rightarrow A$ ...By using Rule-4, Self-Determination.

$A \rightarrow B$ ...Already given in F.

$A \rightarrow C$ ...By using rule-3, Transitivity.

$A \rightarrow D$ ...By using rule-3, Transitivity.

Now, by applyiing Rule-6 Union, it is possible to derive $A^+ \rightarrow ABCD$ and it can be denoted using $A \rightarrow ABCD$. All such type of FDs derived from each FD of F form a closure of F. Steps to determine F^+example:

- Determine each set of attributes **X** that appears as a left hand side of some FD in F.
- Determine the set $\mathbf{X^+}$ of all attributes that are dependent on **X**, as given in above example.
- In other words, $\mathbf{X^+}$ represents a set of attributes that are functionally determined by X based on F. And, X^+ is called the **Closure of X under F.**
- All such sets of $\mathbf{X^+}$, in combine, Form a closure of F.

Algorithm: Determining X^+, the closure of X under F.

Input: Let F be a set of FDs for relation R.

Steps:

1. $X^+ = X$ //initialize X^+ to X
2. For each FD: $Y \rightarrow Z$ in F Do
If $Y \subseteq X^+$ Then //If Y is contained in X^+
$X^+ = X^+ \cup Z$ //add Z to X^+
End If
End For
3. Return X^+ //Return closure of X

Output: Closure X^+ of X under F

3.12 What is decomposition?

- Decomposition is the process of breaking down in parts or elements.
- It replaces a relation with a collection of smaller relations.
- It breaks the table into multiple tables in a database.
- It should always be lossless, because it confirms that the information in the original relation can be accurately reconstructed based on the decomposed relations.
- If there is no proper decomposition of the relation, then it may lead to problems like loss of information.

3.12.1 Properties of Decomposition

Following are the properties of Decomposition,

1. Lossless Decomposition
2. Dependency Preservation
3. Lack of Data Redundancy

1. Lossless Decomposition

- Decomposition must be lossless. It means that the information should not get lost from the relation that is decomposed.
- It gives a guarantee that the join will result in the same relation as it was decomposed.

Example:

Let's take 'E' is the Relational Schema, With instance 'e'; is decomposed into: E1, E2, E3, ... En; With instance: e1, e2, e3, en, If e1 ⋈ e2 ⋈ e3 . . . . ⋈ en, then it is called as **'Lossless Join Decomposition'.**

- In the above example, it means that, if natural joins of all the decomposition give the original relation, then it is said to be lossless join decomposition.

Example: <Employee_Department> Table

Eid	Ename	Age	City	Salary	Deptid	DeptName
E001	ABC	29	Pune	20000	D001	Finance
E002	PQR	30	Pune	30000	D002	Production
E003	LMN	25	Mumbai	5000	D003	Sales
E004	XYZ	24	Mumbai	4000	D004	Marketing
E005	STU	32	Bangalore	25000	D005	Human Resource

- Decompose the above relation into two relations to check whether decomposition is lossless or lossy.
- Now, we have decomposed the relation that is Employee and Department.

Relation 1 : <Employee> Table

Eid	Ename	Age	City	Salary
E001	ABC	29	Pune	20000
E002	PQR	30	Pune	30000
E003	LMN	25	Mumbai	5000
E004	XYZ	24	Mumbai	4000
E005	STU	32	Bangalore	25000

- Employee Schema contains (Eid, Ename, Age, City, Salary).

Relation 2 : <Department> Table

Deptid	Eid	DeptName
D001	E001	Finance
D002	E002	Production
D003	E003	Sales
D004	E004	Marketing
D005	E005	Human Resource

- Department Schema contains (Deptid, Eid, DeptName).
- So, the above decomposition is a Lossless Join Decomposition, because the two relations contains one common field that is 'Eid' and therefore join is possible.
- Now apply natural join on the decomposed relations.

Employee ⋈ Department

Eid	Ename	Age	City	Salary	Deptid	DeptName
E001	ABC	29	Pune	20000	D001	Finance
E002	PQR	30	Pune	30000	D002	Production
E003	LMN	25	Mumbai	5000	D003	Sales
E004	XYZ	24	Mumbai	4000	D004	Marketing
E005	STU	32	Bangalore	25000	D005	Human Resource

Hence, the decomposition is Lossless Join Decomposition.

- If the <Employee> table contains (Eid, Ename, Age, City, Salary) and <Department> table contains (Deptid and DeptName), then it is not possible to join the two tables or relations, because there is no common column between them. And it becomes **Lossy Join Decomposition.**

2. Dependency Preservation

- Dependency is an important constraint on the database.
- Every dependency must be satisfied by at least one decomposed table.
- If {A → B} holds, then two sets are functional dependent. And, it becomes more useful for checking the dependency easily if both sets in a same relation.
- This decomposition property can only be done by maintaining the functional dependency.
- In this property, it allows to check the updates without computing the natural join of the database structure.
- **Lack of Data Redundancy**
- Lack of Data Redundancy is also known as a **Repetition of Information.**
- The proper decomposition should not suffer from any data redundancy.
- The careless decomposition may cause a problem with the data.
- The lack of data redundancy property may be achieved by Normalization process.

Decomposition helps in eliminating some of the problems of bad design such as redundancy, inconsistencies and anomalies.
There are two types of decomposition:

1. Lossy Decomposition
2. Lossless Join Decomposition

1. Lossy Decomposition:

"The decomposition of relation R into R1 and R2 is **lossy** when the join of R1 and R2 does not yield the same relation as in R."
One of the disadvantages of decomposition into two or more relational schemes (or tables) is that some information is lost during retrieval of original relation or table.Consider that we have table STUDENT with three attribute roll_no , sname and department.

STUDENT:

Roll_no	Sname	Dept
111	parimal	COMPUTER
222	parimal	ELECTRICAL

This relation is decomposed into two relation no_name and name_dept :

No_name:

Roll_no	Sname
111	parimal
222	parimal

Name_dept :

Sname	Dept
parimal	COMPUTER
Parimal	ELECTRICAL

In lossy decomposition, spurious tuples are generated when a natural join is applied to the relations in the decomposition.

stu_joined :

Roll_no	Sname	Dept
111	parimal	COMPUTER
111	parimal	ELECTRICAL
222	parimal	COMPUTER
222	parimal	ELECTRICAL

The above decomposition is a bad decomposition or Lossy decomposition.

2. Lossless Join Decomposition:

"The decomposition of relation R into R1 and R2 is **lossless** when the join of R1 and R2 yield the same relation as in R."

A relational table is decomposed (or factored) into two or more smaller tables, in such a way that the designer can capture the precise content of the original table by joining the decomposed parts. This is called lossless-join (or non-additive join) decomposition.

This is also refferd as non-additive decomposition.

The lossless-join decomposition is always defined with respect to a specific set F of dependencies.

Consider that we have table STUDENT with three attribute roll_no, sname and department.

STUDENT:

Roll_no	**Sname**	**Dept**
111	parimal	COMPUTER
222	parimal	ELECTRICAL

This relation is decomposed into two relation Stu_name and Stu_dept :

Stu_name:

Roll_no	**Sname**
111	parimal
222	parimal

Stu_dept :

Roll_no	**Dept**
111	COMPUTER
222	ELECTRICAL

Now, when these two relations are joined on the comman column 'roll_no' ,the resultant relation will look like stu_joined.

stu_joined :

Roll_no	Sname	Dept
111	parimal	COMPUTER
222	parimal	ELECTRICAL

In lossless decomposition, no any spurious tuples are generated when a natural joined is applied to the relations in the decomposition.

3.13 Let Us Sum Up

In relational database theory, a functional dependency is constraint between two sets of attributes in a relation from a database. In other words, functional dependency is a constraint that describes the relationship between attributes in a relation. Discovering Functional Dependency (FD) is employed to extract the Functional Dependency from a datasets.

The determination of functional dependencies is an important part of designing databases in the relational model, and in database normalization and demoralization.

3.14 Self assessment Questions

1. What do you mean by Functional Dependency? Explain.

……
……
……
……
……
……
……
……
……
……

2. What do you mean by fully functional dependency?

……
……
……
……
……
……
……
……
……
……

3. What are the advantages of Functional Dependency?

……
……
……
……
……
……
……
……
……
……

4. What is decomposition? Write the properties of decomposition

……
……
……
……
……
……
……
……
……
……

5. What are the design goals for relational database design? Explain why each is desirable.

……
……

……
……
……
……
……
……
……

3.15 Model Questions

1. Write short notes on tuple relational calculus.
2. Explain trivial dependency
3. Discuss the Rules for drawing Functional Dependency Diagrams.
4. How to find closures of a Set of Functional Dependencies. Explain with examples.
5. What do you mean by lossless join decomposition? Explain with example.

3.16 References &Suggested Readings

1. Yao, H., Hamilton, H., and Butz, C., FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences, Canada.
2. Wyss, C., Giannella, C., and Robertson, E. (2001), Fast FDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances, U.S.A.
3. Huhtala, Y., Karkkainen, J., Porkka, P., and Toivonen, H., (1999), TANE: An Efficient Algorithm for discovering Functional and Approximate Dependencies, Computing Journal, V.42, No.20, pp.100-107.
4. Elmasri, R. and Navathe, S. (2004), Fundamentals of Database Systems, Addison Wesley, Fourth edition.
5. https://en.wikipedia.org/wiki/Functional_dependency
6. www.safaribooksonline.com
7. http://tutorialink.com/dbms

UNIT-4 NORMALIZATION

Unit Structure

4.0 Introduction

The normal forms defined in relational database theory represent guidelines for record design. The guidelines corresponding to first through fifth normal forms are presented here, in terms that do not require an understanding of relational theory. The design guidelines are meaningful even if one is not using a relational database system. We present the guidelines without referring to the concepts of the relational model in order to emphasize their generality, and also to make them easier to understand. Our presentation conveys an intuitive sense of the intended constraints on record design, although in its informality it may be imprecise in some technical details.
Suppose we are now given the task of designing and creating a database. How do we produce a good design? What relations should we have in the database? What attributes should these relations have? Good database design needless to say, is important. Careless design can lead to uncontrolled data redundancies that will lead to problems with data anomalies.

4.1 Learning Objectives

After learning this unit you should be able to

- Understand the concept of normalization
- Identify the needs of normalization in database design
- Define different types of Normal Forms
- Distinguish between different types of Normal Forms
- Understand the role of Functional Dependencies in Normalization.

4.2 What is Database Normalization

Database normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion anomalies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables. Normalization is used for mainly two purposes,

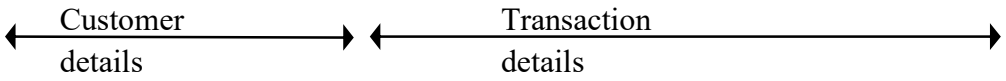
- Eliminating redundant (useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

In this unit we will examine a process known as Normalization—a rigorous design tool that is based on the mathematical theory of relations which will

result in very practical operational implementations. A properly normalized set of relations actually simplifies the retrieval and maintenance processes and the effort spent in ensuring good structures is certainly a worthwhile investment. Furthermore, if database relations were simply seen as file structures of some vague file system, then the power and flexibility of RDBMS cannot be exploited to the full.
For us to appreciate good design, let us begin by examining some bad ones.

4.2.1 A Bad Design

E.Codd has identified certain structural features in a relation which create retrieval and update problems. Suppose we start off with a relation with a structure and details like:



C#	Cname	Ccity	..	P1#	Date1	Qnt1	P2#	Date2		P9#	Date9	
1	Codd	London	..	1	21.01	20	2	23.01				
2	Martin	Paris	..	1	26.10	25						
3	Deen	London	..	2	29.01	20						

Fig: Simple Structure

This is a simple and straightforward design. It consists of one relation where we have a single tuple for every customer and under that customer we keep all his transaction records about parts, up to a possible maximum of 9 transactions. For every new transaction, we need not repeat the customer details (of name, city and telephone), we simply add on a transaction detail.

However, we note the following disadvantages:

- The relation is wide and clumsy
- We have set a limit of 9 (or whatever reasonable value) transactions per customer. What if a customer has more than 9 transactions?
- For customers with less than 9 transactions, it appears that we have to store null values in the remaining spaces. What a waste of space!

- The transactions appear to be kept in ascending order of P#s. What if we have to delete, for customer Codd, the part numbered 1—should we move the part numbered 2 up (or rather, left)? If we did, what if we decide later to re-insert part 2? The additions and deletions can cause awkward data shuffling.
- Let us try to construct a query to "Find which customer(s) bought P# 2" ? The query would have to access every customer tuple and for each tuple, examine every of its transaction looking for

  (P1# = 2) OR (P2# = 2) OR (P3# = 2) … OR (P9# = 2)

A comparatively simple query seems to require a clumsy retrieval formulation!

4.2.2 Another Bad Design

Alternatively, why don't we re-structure our relation such that we do not restrict the number of transactions per customer? We can do this with the following structure:

Transaction						
C#	Cname	Ccity	Cphone	P#	Date	Qnt
1	Codd	London	2263035	1	21.01	20
1	Codd	London	2263035	2	23.01	30
2	Martin	Paris	5555910	1	26.01	25
3	Deen	London	2234391	2	29.01	20

This way, a customer can have just any number of Part transactions without worrying about any upper limit or wasted space through null values (as it was with the previous structure). Constructing a query to "Find which customer(s) bought P# 2" is not as cumbersome as before as one can now simply state: P# = 2.

But again, this structure is not without its faults:

- It seems a waste of storage to keep repeated values of Cname, Ccity and Cphone.
- If C# 1 were to change his telephone number, we would have to ensure that we update ALL occurrences of C# 1's Cphone values. This means updating tuple 1, tuple 2 and all other tuples where there is an occurrence of C# 1. Otherwise, our database would be left in an inconsistent state.
- Suppose we now have a new customer with C# 4. However, there is no part transaction yet with the customer as he has not ordered anything yet. We may find that we cannot insert this new information because we do not have a P# which serves as part of the 'primary key' of a tuple. (A primary key cannot have null values).

- Suppose the third transaction has been canceled, i.e. we no longer need information about 20 of P# 1 being ordered on 26 Jan. We thus delete the third tuple. We are then left with the following relation:

Transaction						
C#	Cname	Ccity	Cphone	P#	Date	Qnt
1	Codd	London	2263035	1	21.01	20
1	Codd	London	2263035	2	23.01	30
3	Deen	London	2234391	2	29.01	20

But then, suppose we need information about the customer "Martin", say the city he is located in. Unfortunately as information about Martin was held in only that tuple and having the entire tuple deleted because of its P# transaction meant also that we have lost all information about Martin from the relation.

As illustrated in the above instances, we note that badly designed, un-normalized relations waste storage space. Worse, they give rise to the following storage irregularities:

1. **Update anomaly**

Data inconsistency or loss of data integrity can arise from data redundancy/repetition and partial update.

2. **Insertion anomaly**

Data cannot be added because some other data is absent.

3. **Deletion anomaly**

Data maybe unintentionally lost through the deletion of other data.

4.2.3 The Need for Normalization

Intuitively, it would seem that these undesirable features can be removed by breaking a relation into other relations with desirable structures. We shall attempt by splitting the above Transaction relation into the following two relations, Customer and Transaction, which can be viewed as entities with a one –to- many relationship.

Customer			
C#	Cname	Ccity	Cphone
1	Codd	London	2263035
2	Martin	Paris	5555910
3	Deen	London	2234391

Transaction			
C#	P#	Date	Qnt
1	1	21.01	20
1	2	23.01	30
2	1	26.01	25
3	2	29.01	20

Let us see if this new design will alleviate the above storage anomalies:

1. Update anomaly

If C# 1 were to change his telephone number, as there is only one occurrence of the tuple in the Customer relation, we need to update only that one tuple as there are no redundant/duplicate tuples.

2. Addition anomaly

Adding a new customer with C# 4 can be easily done in the Customer relation of which C# serves as the primary key. With no P# yet, a tuple in Transaction need not be created.

3. Deletion anomaly

Canceling the third transaction about 20 of P# 1 being ordered on 26 Jan would now mean deleting only the third tuple of the new Transaction relation above. This leaves information about Martin still intact in the new Customer relation.

This process of reducing a relation into simpler structures is the process of Normalization. Normalization may be defined as a step by step reversible process of transforming an un-normalized relation into relations with progressively simpler structures. Since the process is reversible, no information is lost in the transformation.

Normalization removes (or more accurately, minimizes) the undesirable properties by working through a series of stages called Normal Forms. Originally, Codd defined three types of undesirable properties:

1. Data aggregates
2. Partial key dependency
3. Indirect key dependency

And the three stages of normalization that remove the associated problems are known, respectively, as the:

- First Normal Form (1NF)
- Second Normal Form (2NF), and
- Third Normal Form (3NF)

We shall now show a more formal process on how we can decompose relations into multiple relations by using the Normal Form rules for structuring.

4.3 First Normal Form (1NF)

The purpose of the First Normal Form (1NF) is to simplify the structure of a relation by ensuring that it does not contain data aggregates or repeating groups. By this we mean that no attribute value can have a set of values. In the example below, any one customer has a group of several telephone entries:

Customer			
C#	Cname	Ccity	Cphone
1	Codd	London	2263035 5555910
2	Deen	London	2234391 832551

Fig: Presence of Repeating Groups

This is thus not in 1NF. It must be "flattened". This can be achieved by ensuring that every tuple defines a single entity by containing only atomic values. One can either re-organize into one relation as in:

Customer			
C#	Cname	Ccity	Cphone
1	Codd	London	2263035
1	Codd	London	5555910
2	Deen	London	2234391
2	Deen	London	832551

Fig: Atomic values in tuples or split into multiple relations as in:

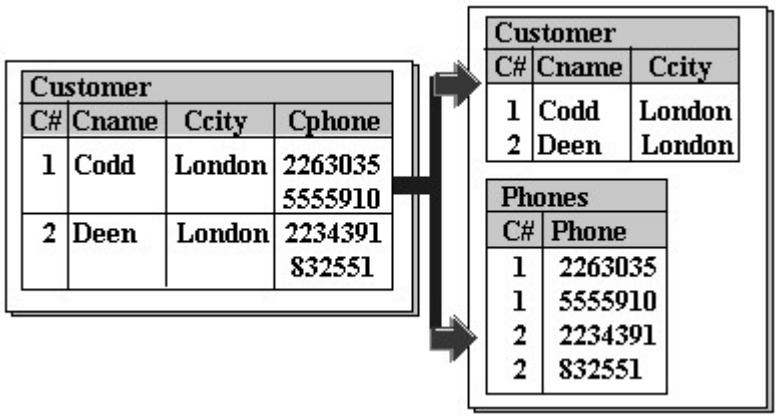


Fig: Reduction to 1NF

Note that earlier we defined 1NF as one of the characteristics of a relation (Lesson 2). Thus we consider that every relation is at least in the first normal form. The Transaction relation of Figure above is in 1NF relation.
We may thus generalize by saying that "A relation is in the 1NF if the values in the relation are atomic for every single attribute of the relation".
Before we can look into the next two normal forms, 2NF and 3NF, we need to first explain the notion of 'functional dependency' as these two forms are constrained by functional dependencies.

4.4 Functional Dependencies

4.4.1 Determinant

The value of an attribute can uniquely determine the value in another attribute. For example, in every tuple of the Transaction relation in Figure above:

- C# uniquely determines Cname
- C# also uniquely determines Ccity as well as Cphone

Given C# 1, we will know that its Cname is 'Codd' and no other. On the other hand, we cannot say that given Ccity 'London', we will know that its Cname is 'Codd' because Ccity 'London' will also give Cname of 'Deen'. Thus Ccity cannot uniquely determine Cname (in the same way that C# can).
Additionally, we see that:

- (C#, P#, Date) uniquely determines Qnt

We can now introduce the definition of a "determinant" as being an attribute (or a set of non-redundant) attributes which can act as a unique identifier of another attribute (or another set of attributes) of a given relation.
We may thus say that:

- C# is a unique key for Cname, Ccity and Cphone.
- (C#, P#, Date) is a unique key for Qnt.

These keys are non-redundant keys as no member of the composite attribute can be left out of the set. Hence, C# is a determinant of Cname, Ccity, and Cphone. (C#, P#, Date) is a determinant of Qnt.
A determinant is written as:
A $\rightarrow$ B
And can be read as "**A** determines **B**" (or **A** is a determinant of **B**). If any two tuples in the relation **R** have the same value for the **A** attribute, then they must also have the same value for their **B** attribute.
Applying this to the Transaction relation above, we may then say:

C# → Cname
C# → Ccity
C# → Cphone
(C#, P#, Date) → Qnt

The value of the attribute on the left-hand side of the arrow is the determinant because its value uniquely determines the value of the attribute on the right.

Note also that:

(Ccity, Cphone) → Cname
(Ccity, Cphone) → C#

4.4.2 The Converse Notation

A X⟶ **B**

Can be read as **A** "does not determine" **B**.
Taking again the Transaction relation, we may say therefore that Ccity cannot uniquely determine Cname

Ccity X⟶ **Cname**

Because there exists a number of customers living in the same city.

Likewise:

Cname X ⟶ **(Ccity, Cphone)**
Cname X ⟶ **C#**

As there may exist customers with the same name.

4.4.3 Functional Dependency

The role of determinants is also expressed as "functional dependencies" whereby we can say:
"If an attribute **A** is a determinant of an attribute **B**, then **B** is said to be functionally dependent on **A**" and likewise
"Given a relation **R**, attribute **B** of **R** is functionally dependent on attribute **A** if and only if each **A**-value in **R** has associated with it one **B**-value in **R** at any one time".
"C# is a determinant of Cname, Ccity and Cphone" is thus also "Cname, Ccity and Cphone are functionally dependent on C#. Given a particular value of Cname value, there exists precisely one corresponding value for each of Cname, Ccity and Cphone.

This is more clearly seen via the following functional dependency diagram:

Transaction						
C#	Cname	Ccity	Cphone	P#	Date	Qnt
1	Codd	London	2263035	1	21.01	20
1	Codd	London	2263035	2	23.01	30
2	Martin	Paris	5555910	1	26.01	25
3	Deen	London	2234391	2	29.01	20

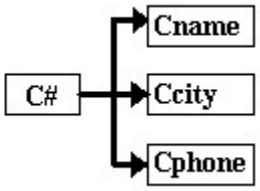


Fig: Functional dependencies in the Transaction relation

Similarly, "(C#, P#, Date) is a determinant of Qnt" is thus also "Qnt is functionally dependent on the set of attributes (C#, P#, Date)". The set of attributes is also known as a composite attribute.

Transaction						
C#	Cname	Ccity	Cphone	P#	Date	Qnt
1	Codd	London	2263035	1	21.01	20
1	Codd	London	2263035	2	23.01	30
2	Martin	Paris	5555910	1	26.01	25
3	Deen	London	2234391	2	29.01	20

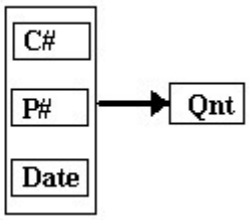


Fig: Functional dependency on a composite attribute

4.4.4 Full Functional Dependency

"If an attribute (or a set of attributes) **A** is a determinant of an attribute (or a set of attributes) **B**, then **B** is said to be fully functionally dependent on **A**" and likewise

"Given a relation **R**, attribute **B** of **R** is fully functionally dependent on attribute **A** of **R** if it is functionally dependent on **A** and not functionally dependent on any subset of **A** (**A** must be composite)".

Transaction						
C#	Cname	Ccity	Cphone	P#	Date	Qnt
1	Codd	London	2263035	1	21.01	20
1	Codd	London	2263035	2	23.01	30
2	Martin	Paris	5555910	1	26.01	25
3	Deen	London	2234391	2	29.01	20

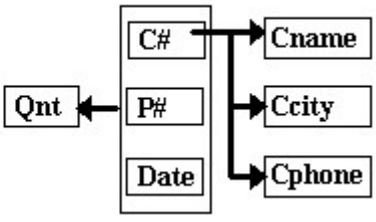


Fig: Functional dependencies in the Transaction relation

For the Transaction relation, we may now say that:

- Cname is fully functionally dependent on C#
- Ccity is fully functionally dependent on C#
- Cphone is fully functionally dependent on C#
- Qnt is fully functionally dependent on (C#, P#, Date)
- Cname is not fully functionally dependent on (C#, P#, Date), it is only partially dependent on it (and similarly for Ccity and Cphone).

Having understood about determinants and functional dependencies, we are now in a position to explain the rules of the second and third normal forms.

4.5 Second Normal Form (2NF)

Consider again the Transaction relation which was in 1NF. Recall the operations we tried to do in the previous section and the problems encountered:

1. Update

What happens if Codd's telephone number changes and we update only the first tuple (but not the second)?

Transaction						
C#	Cname	Ccity	Cphone	P#	Date	Qnt
1	Codd	London	2263035	1	21.01	20
1	Codd	London	2263035	2	23.01	30
2	Martin	Paris	5555910	1	26.01	25
3	Deen	London	2234391	2	29.01	20

2. Insertion

If we wish to introduce a new customer, we cannot do so unless an appropriate transaction is affected.

Transaction						
C#	Cname	Ccity	Cphone	P#	Date	Qnt
1	Codd	London	2263035	1	21.01	20
1	Codd	London	2263035	2	23.01	30
2	Martin	Paris	5555910	1	26.01	25
3	Deen	London	2234391	2	29.01	20
4	Smith	Vienna	?	?	?	?

Entity Integrity Violation: P# is a part of primary key !

3. Deletion

If the data about a transaction is deleted, the information about the customer is also deleted. If this happens to the last transaction for that customer the information about the customer will

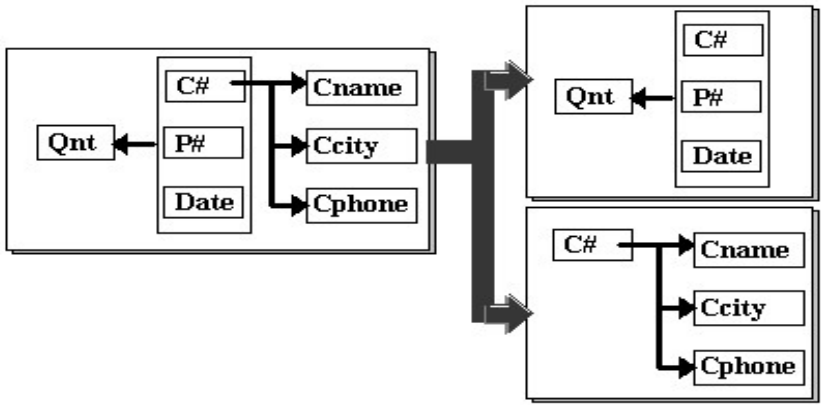
Transaction							
	C#	Cname	Ccity	Cphone	P#	Date	Qnt
	1	Codd	London	2263035	1	21.01	20
	1	Codd	London	2263035	2	23.01	30
Delete	2	Martin	Paris	5555910	1	26.01	25
	3	Deen	London	2234391	2	29.01	20

be lost.

Clearly, the Transaction relation although it is normalized to 1NF still has storage anomalies. The reason for such violations to the database's integrity and consistency rules is because of the partial dependency on the primary key.

The determinant (C#, P#, Date) is the composite key of the Transaction relation - its value will uniquely determine the value of every other non-key attribute in a tuple of the relation. Note that whilst Qnt is fully functionally dependent on all of (C#, P#, Date), Cname, Ccity and Cphone are only partially functionally dependent on the composite key (as they each depend only on the C# part of the key only but not on P# or Date).

The problems are avoided by eliminating partial key dependence in favour of full functional dependence, and we can do so by separating the dependencies as follows:



The source relation into thus split into two (or more) relations whereby each resultant relation no longer has any partial key dependencies:

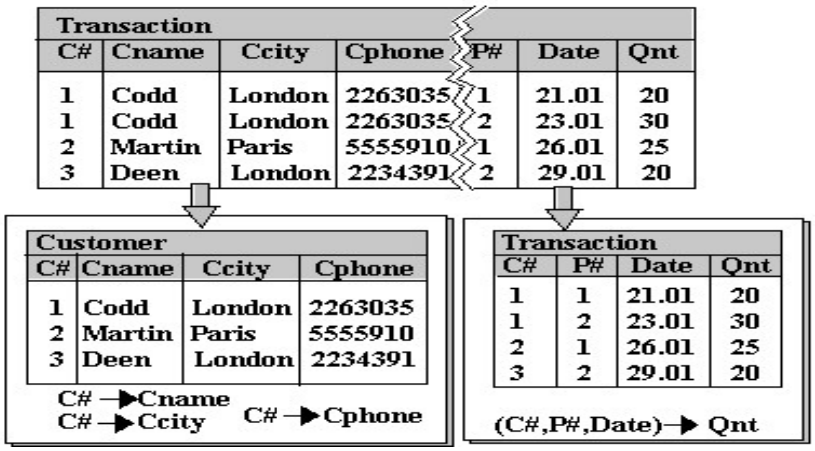


Fig: Relations in 2NF

We now have two relations, both of which are in the second normal form. These are the same relations of Figure above, and the discussion we had earlier clearly shows that the storage anomalies caused by the 1NF relation have now been eliminated:

1. **Update anomaly**

There are no redundant/duplicate tuples in the relation, thus updates are done just at one place without nay worry for database inconsistencies.

2. **Addition anomaly**

Adding a new customer can be done in the Customer relation without concern whether there is a transaction for a part or not

3. **Deletion anomaly**

Deleting a tuple in Transaction does not cause loss of information about Customer details.

More generally, we shall summaries by stating the following:

R: (K1	K2	I1	I2)
1	1	a	x
1	2	b	y
2	1	c	x
2	2	d	y

1. Suppose, there is a relation **R**

Where the composite attribute (K1, K2) is the Primary Key. Suppose also that there exist the following functional dependencies:

(K1, K2) → I1 i.e. a full functional dependency on the composite key (K1, K2)..

K2 → I2 i.e. a partial functional dependency on the composite key (K1, K2).

The partial dependencies on the primary key must be eliminated. The reduction of 1NF into 2NF consists of replacing the 1NF relation by appropriate "projections" such that every non-key attribute in the relations are fully functionally dependent on the primary key of the respective relation. The steps are:

1. Create a new relation R2 from R. Because of the functional dependency **K2 → I2,** R2 will contain K2 and I2 as attributes. The determinant, K2, becomes the key of R2.
2. Reduce the original relation R by removing from it the attribute on the right hand side of K2 **→ I2**. The reduced relation R1 thus contain all the original attributes but without I2.

3. Repeat steps 1. and 2. if more than one functional dependency prevents the relation from becoming a 2NF.
4. If a relation has the same determinant as another relation, place the dependent attributes of the relation to be non-key attributes in the other relation for which the determinant is a key.

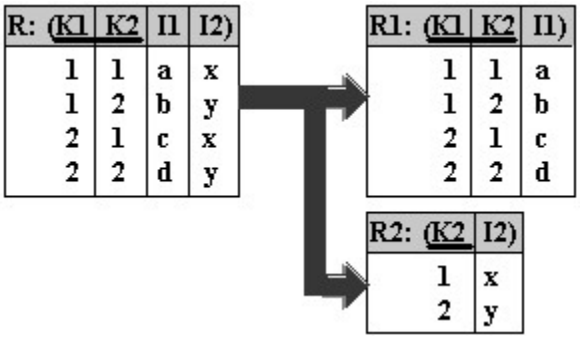


Fig: Reduction of 1NF into 2NF

Thus, "A relation R is in 2NF if it is in 1NF and every non-key attribute is fully functionally dependent on the primary key".

4.6 Third Normal Form (3NF)

A relation in the Second Normal Form can still be unsatisfactory and show further data anomalies. Suppose we add another attribute, Salesperson, to the Customer relation who attends to the needs of the customer.

Customer				
C#	Cname	Ccity	Cphone	Salesperson
1	Codd	London	2263035	Smith
2	Martin	Paris	5555910	Ducruer
3	Deen	London	2234391	Smith

Its associated functional dependencies are:

C# → Cname, Ccity, Cphone, Salesperson

Consider again operations that we may want to do on the data

1. Update

Can we change the salesperson servicing customers in London? Here, we find that there are several occurrences of London customers (e.g. Codd and Deen). Thus we must ensure that we update all tuples such that 'Smith' is now replaced by the new salesperson, say 'Jones', otherwise we end up with a database inconsistency problem again.

2. Insertion

Can we enter data that 'Fatimah' is the salesperson for the city of 'Sarawak' although no customer exists there yet?

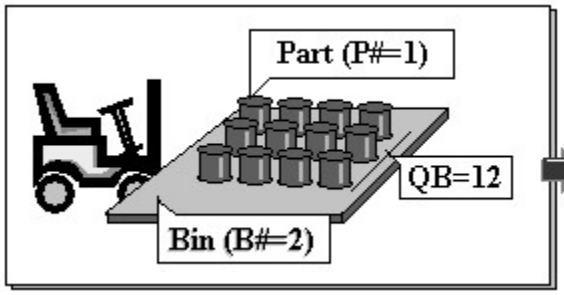
Customer				
C#	Cname	Ccity	Cphone	Salesperson
1	Codd	London	2263035	Smith
2	Martin	Paris	5555910	Ducruer
3	Deen	London	2234391	Smith
?	?	Sarawak	?	Fatimah

As C# is the primary key, a null value in C# cannot be allowed. Thus the tuple cannot be created.

3. Deletion

What happens if we delete the second tuple, i.e. we no longer need to keep information about the customer Martin, his city, telephone and salesperson ? If this tuple is removed, we will also lose all information that the salesperson Ducruet services the city of Paris as no other tuple holds this information.

As another, more complex example, consider keeping information about parts that are being kept in bins, where the following relation called Stock



Stock:				
B#	P#	QB	LT	RL
1	1	10	12	1
2	1	12	12	1
3	2	8	18	2
4	2	9	18	2

Which contains information on?

- the bin number (B#)
- the part number (P#) of the part kept inside a given bin
- the quantity of pieces of the part in a given bin (QB)
- the lead time (LT) taken to deliver a part after an order has been placed for it
- the re-order level (RL) of the part number which indicates the an order must be placed to re-order new stock of that part whenever the existing stock quantity gets too low, i.e. when QB ≤ RL

We further assume that:

- parts of a given part number may be stored in several bins

- the same bin holds only one type of part, i.e. it does not hold parts of more than one part number
- the lead time and re-order level are fixed for each part number

The only candidate key for this relation is B#, hence it must be selected as the primary key. Since the B# is a single attribute, the question of partial dependency does not arise (the relation is in Second Normal Form).

Its associated functional dependencies (which are fully functional dependencies) are:

B# → P#, QB, LT, RL

But in this case, we also have data anomalies:

1. **Update**

Suppose the re-order level for part number 1 is updated, i.e. RL for P# 1 must be changed from 1 to 4. We must ensure that we update all tuples containing P#1, i.e. tuples 1 and 2; any partial updates will lead to an inconsistent database inconsistency.

2. **Insertion**

Stock:

B#	P#	QB	LT	RL
1	1	10	12	1
2	1	12	12	1
3	2	8	18	2
4	2	9	18	2
?	3	?	24	3

Null Value is not permitted since B# is a primary key

We cannot store LT and RL information for a new expected part number in the database unless we actually have a bin number allocated to it.

3. **Deletion**

If the data (tuple) about a particular bin is deleted, the information about the part is also deleted. If this happens to the last bin containing that part, the information about the concrete part (LT, RL) will also be lost.

From the two examples above, it is still evident that despite having relations in 2NF, problems can still arise and we should now try to eliminate them. It would seem we need to further normalize them, i.e. We need a third normal form to eliminate these anomalies.

Scrutinizing the functional dependencies of these examples, we notice the existence of "other" dependencies:

Customer				
C#	Cname	Ccity	Cphone	Salesperson
1	Codd	London	2263035	Smith
2	Martin	Paris	5555910	Ducruer
3	Deen	London	2234391	Smith

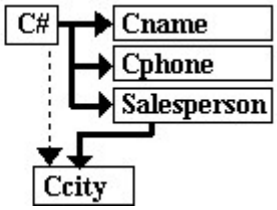


Fig: All functional dependencies in the Customer relation

Notice for example that the dependency of the attribute Salesperson on the key C#, i.e.

C# → Salesperson

is only an indirect or transitive dependency, which is also indicated in the diagram as a dotted arrow .

This is considered indirect because **C# → Ccity** and **Ccity → Salesperson**, and thus

C# → Salesperson.

Thus for the Stock relation:

B# → P#, QB and **P# → LT, RL**

then **B# → P#, QB, LT, RL**

Stock:				
B#	P#	QB	LT	RL
1	1	10	12	1
2	1	12	12	1
3	2	8	18	2
4	2	9	18	2

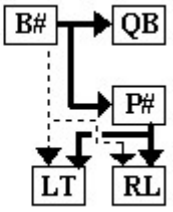
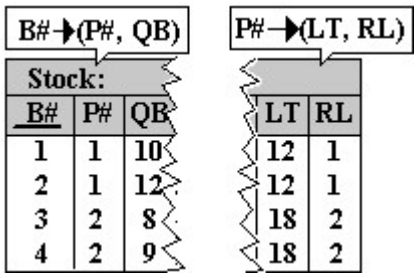


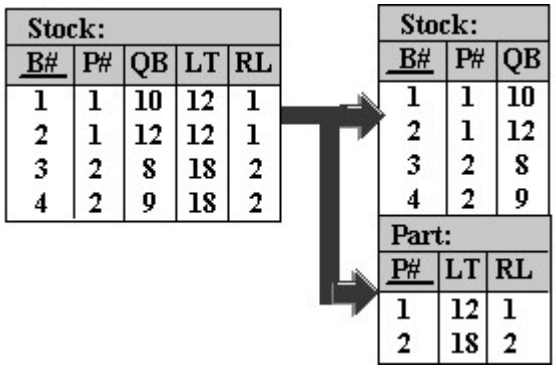
Fig: All functional dependencies in the Stock relation

The Indirect Dependency obviously causes data duplication (e.g. note the two occurrences of P#1 and LT 12 in the first two tuples of Stock), which leads to the above anomalies. This can be eliminated by removing all indirect/transitive dependencies.

We do this by splitting the source relation into two or more other relations, as illustrated in the following example:



where, we can then get



We can say that the storage anomalies caused by the 2NF relation can now be eliminated:

1. Update anomaly

To update the re-order level for part number 1, we need only change one (the first) tuple in the new Part relation without concern for other duplicates that used to exist before.

2. Addition anomaly

We can now store LT and RL information for a new part number in the database by creating a tuple in the new Part relation, without concern whether there is a bin number allocated to it or not.

3. Deletion anomaly

Deleting the tuple about a particular bin will remove a tuple form the new Stock relation. Should the part that was deleted from that bin be the only bin where it could be found, however does not mean the loss of data about that part. Information on the LT and RL of the part is still in the new Part relation.

More generally, we shall summarize by stating the following:

Suppose there is a relation R with attributes A, B and C. A is the determinant.

If **A** $\rightarrow$ **B** and **B** $\rightarrow$ **C**

Then

A $\rightarrow$ **C** is the 'Indirect Dependency'

(Of course, if **A** $\rightarrow$ **C** and B does not exist, then **A** $\rightarrow$ **C** is a 'Direct Dependency')

The Indirect Dependencies on the primary key must be eliminated. The reduction of 2NF into 3NF consists of replacing the 2NF relation by appropriate "projections" such Indirect Key Dependencies are eliminated in favour of the Direct Key Dependencies.

The steps are:

1. Reduce the original relation R by removing from it the attribute on the right hand side of any indirect dependencies **A** $\rightarrow$ **C**. The reduced relation R1 thus contain all the original attributes but without C1.
2. Form a new relation R2 that contains all attributes that are in the dependency **B** $\rightarrow$ **C**.
3. Repeat steps 1. and 2. if more than one indirect dependency prevents the relation from becoming a 3NF.
4. Verify that every determinant in every relation is a key in that relation

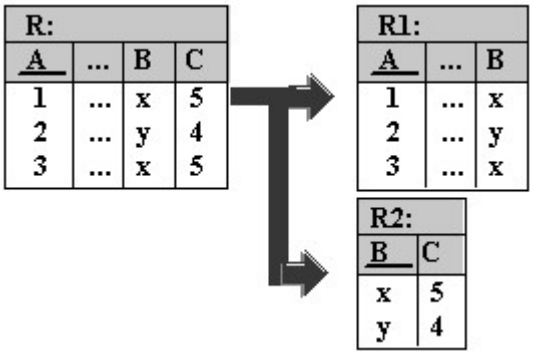


Fig. Reduction of 2NF into 3NF

Thus, "A relation R is in 3NF if it is in 2NF and every non-key attribute is fully and directly dependent on the primary key".

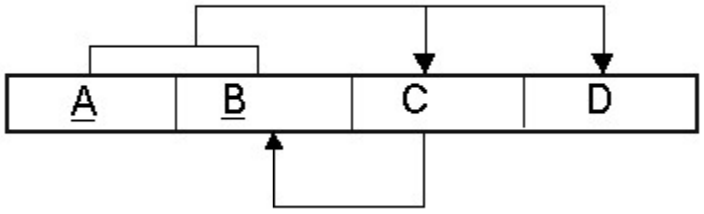
There is another definition of 3NF which states that "A relation is in third normal form if every data item outside the primary key is identifiable by the primary key, the whole primary key and by nothing but the primary key".

In order to avoid certain update anomalies, each relation declared in the data base schema, should be at least in the Third Normal Form. Structurally, 2NF is better than 1NF, and 3NF is better than 2NF. There are of course

other higher normal forms like the Boyce-Codd Normal Form (BCNF), the Fourth Normal Form (4NF) and the Fifth Normal Form (5NF).
However, the Third Normal Form is quite sufficient for most business database design purposes; although some very specialized applications may require the higher-level normalization.
It must be noted that although normalization is a very important database design component, we should not always design in the highest level of normalization, thinking that it is the best. Often at the physical implementation level, the decomposition of relations into higher normal form mean more pointer movements are required to access and the thus slower the response time of the database system. This may conflict with the end-user demand for fast performance. The designer may sometimes have to "demoralize" some portions of a database design in order to meet performance requirements at the expense of data redundancy and its associated storage anomalies.

4.7 Boyce Codd Normal Form

Boyce-Codd Normal Form (BCNF) is one of the forms of database normalization. A database table is in BCNF if and only if there are no non-trivial functional dependencies of attributes on anything other than a superset of a candidate key.
BCNF is also sometimes referred to as 3.5NF, or 3.5 Normal Form.
Usually tables that are in Third Normal Form are already in Boyce Codd Normal Form. **Boyce Codd Normal Form (BCNF)** is considered a special condition of third Normal form. **A table is in BCNF if every determinant is a candidate key.** A table can be in 3NF but not in BCNF. This occurs when a non key attribute is a determinant of a key attribute. The dependency diagram may look like the one below



The table is in 3NF. A and B are the keys and C and D depend on both A and B. There are no transitive dependencies existing between the non key attributes, C and D.

The table is not in BCNF because a dependency exists between C and B. In other words if we know the value of C we can determine the value of B.
We can also show the dependencies as

A B → C D

C → B

Example

An example table from the University Database might be as follows:
If we know the Student Number and Teacher Code we know the Offering (class) the student is in. We also know the review date for that student and teacher (Student progress is reviewed for that class by the teacher and student).

S_Num	T_Code	Offering#	Review Date
123599	FIT104	01764	2nd March
123599	PIT305	01765	12th April
123599	PIT107	01789	2nd May
346700	FIT104	01764	3rd March
346700	PIT305	01765	7th May

The dependencies are

S_Num, T_Code → Offering#, Review Date

Which means that the table is in third normal form? The table is not in BCNF as if we know the offering number we know who the teacher is.
Each offering can only have one teacher!
Offering# → T_Code
A non key attribute is a determinant.
If we look at the table we can see a combination of T_Code and offering# is repeated several times. eg FIT104 and 01764.

4.7.1 Converting to BCNF

The situation is resolved by following the steps below

1 The determinant, Offering#, becomes part of the key and the dependant attribute T_Code, becomes a non key attribute. So the Dependency diagram is now

S_Num, Offering# →T_Code, Review Date

2 There are problems with this structure as T_Code is now dependant on only part of the key. This violates the rules for 2NF, so the table needs to be divided with the partial dependency becoming a new table. The dependencies would then be **S_Num, Offering# →T_Code, Review Date Offering# →T_Code**

3 The original table is divided into two new tables. Each is in 3NF and in BCNF.

Student Review

S_Num	Offering#	Review Date
123599	01764	2nd March
123599	01765	12th April
123599	01789	2nd May
346700	01764	3rd March
346700	01765	7th May

OfferingTeacher

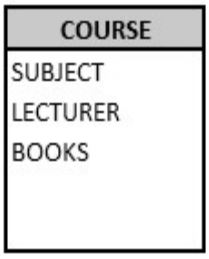
Offering#	T_Code
01764	FIT104
01765	PIT305
01789	PIT107

4.8 Fourth Normal Form (4NF)

In the fourth normal form,

- It should meet all the requirement of 3NF
- Attribute of one or more rows in the table should not result in more than one rows of the same table leading to multi-valued dependencies

To understand it clearly, consider a table with Subject, Lecturer who teaches each subject and recommended Books for each subject.



SUBJECT	LECTURER	BOOKS
Mathematics	Alex	Maths Book1
Mathematics	Bosco	Maths Book2
Physics	Rose	Physics Book
Chemistry	Adam	Chemistry Book

If we observe the data in the table above it satisfies 3NF. But LECTURER and BOOKS are two independent entities here. There is no relationship between Lecturer and Books. In the above example, either Alex or Bosco can teach Mathematics. For Mathematics subject, student can refer either 'Maths Book1' or 'Maths Book2'. i.e.

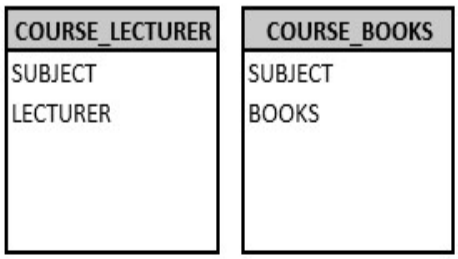
SUBJECT --> LECTURER

SUBJECT-->BOOKS

This is a multivalued dependency on SUBJECT. If we need to select both lecturer and books recommended for any of the subject, it will show up (lecturer, books) combination, which implies lecturer who recommends which book. This is not correct.

SELECT c.LECTURER, c.BOOKS FROM COURSE c WHERE SUBJECT = 'Mathematics';

To eliminate this dependency, we divide the table into two as below:



4NF

SUBJECT	LECTURER
Mathematics	Alex
Mathematics	Bosco
Physics	Rose
Chemistry	Adam

SUBJECT	BOOKS
Mathematics	Maths Book1
Mathematics	Maths Book2
Physics	Physics Book
Chemistry	Chemistry Book

Now if we want to know the lecturer names and books recommended for any of the subject, we will fire two independent queries. Hence it removes the multi-valued dependency and confusion around the data. Thus the table is in 4NF.

```
--Select the lecturer names
SELECT c.SUBJECT , c.LECTURER  FROM COURSE c WHERE c.SUBJECT = 'Mathematics';

--Select the recommended book names
SELECT c.SUBJECT , c.BOOKS FROM COURSE c WHERE c.SUBJECT = 'Mathematics';
```

4.9 Fifth Normal Form (5NF)

A database is said to be in 5NF, if and only if,

- It's in 4NF
- If we can decompose table further to eliminate redundancy and anomaly, and when we re-join the decomposed tables by means of candidate keys, we should not be losing the original data or any new record set should not arise. In simple words, joining two or more decomposed table should not lose records nor create new records.

Consider an example of different Subjects taught by different lecturers and the lecturers taking classes for different semesters.

Note: Please consider that Semester 1 has Mathematics, Physics and Chemistry and Semester 2 has only Mathematics in its academic year!!

COURSE
SUBJECT
LECTURER
CLASS

SUBJECT	LECTURER	CLASS
Mathematics	Alex	SEMESTER 1
Mathematics	Rose	SEMESTER 1
Physics	Rose	SEMESTER 1
Physics	Joseph	SEMESTER 2
Chemistry	Adam	SEMESTER 1

In above table, Rose takes both Mathematics and Physics class for Semester 1, but she does not take Physics class for Semester 2. In this case, combination of all these 3 fields is required to identify a valid data. Imagine we want to add a new class - Semester3 but do not know which Subject and who will be taking that subject. We would be simply inserting a new entry with Class as Semester3 and leaving Lecturer and subject as NULL. As we

discussed above, it's not a good to have such entries. Moreover, the entire three columns together act as a primary key, we cannot leave other two columns blank!

Hence we have to decompose the table in such a way that it satisfies all the rules till 4NF and when join them by using keys, it should yield correct record. Here, we can represent each lecturer's Subject area and their classes in a better way. We can divide above table into three - (SUBJECT, LECTURER), (LECTURER, CLASS), (SUBJECT, CLASS)

5NF

SUBJECT	LECTURER
Mathematics	Alex
Mathematics	Rose
Physics	Rose
Physics	Joseph
Chemistry	Adam

CLASS	LECTURER
SEMESTER 1	Alex
SEMESTER 1	Rose
SEMESTER 1	Rose
SEMESTER 2	Joseph
SEMESTER 1	Adam

CLASS	SUBJECT
SEMESTER 1	Mathematics
SEMESTER 1	Physics
SEMESTER 1	Chemistry
SEMESTER 2	Physics

Now, each of combinations is in three different tables. If we need to identify who is teaching which subject to which semester, we need join the keys of each table and get the result.

For example, who teaches Physics to Semester 1, we would be selecting Physics and Semester1 from table 3 above, join with table1 using Subject to filter out the lecturer names. Then join with table2 using Lecturer to get correct lecturer name. That is we joined key columns of each table to get the correct data. Hence there is no lose or new data - satisfying 5NF condition.

```
SELECT t3.Class, t3.Subject, t1.Lecturer
FROM TABLE3 t3, TABLE3 t2, TABLE3 t1,
Where t3.Class = 'SEMESTER1' and t3.SUBJECT= 'PHYSICS'
AND t3.Subject = t1.Subject
AND t3.Class = t2.Class
```

4.10 Let Us Sum Up

While we have tried to present the normal forms in a simple and understandable way, we are by no means suggesting that the data design process is correspondingly simple. The design process involves many complexities which are quite beyond the scope of this unit. In the first place, an initial set of data elements and records has to be developed, as candidates for normalization. Then the factors affecting normalization have to be assessed:

- Single-valued vs. multi-valued facts.
- Dependency on the entire key.
- Independent vs. dependent facts.
- The presence of mutual constraints.
- The presence of non-unique or non-singular representations.

Finally, the desirability of normalization has to be assessed, in terms of its performance impact on retrieval applications.

4.11 Self assessment Questions

1. What is normalization? Why it is necessary?

……
……
……
……
……
……
……
……
……

2. What is 1NF? Explain with the example.

……
……
……

……
……
……
……

3. Explain detail about 3rd Normal Form

……
……
……
……
……
……
……
……

4. Differentiate between 3NF and BCNF

……
……
……
……
……
……
……
……
……
……

5. Write short notes on 4NF.

……
……

……

……

……

……

……

……..

4.12 Model Questions

1. What is 2NF? Explain with example.
2. Define Boyce Codd's normal form.
3. Discuss different types of anomalies that arise in databases.
4. Explain detail about Fifth Normal Form
5. Write short notes on 5NF

4.13 References & Suggested Readings

1. C.J. Date, An Introduction to Database Systems (third edition), Addison-Wesley, 1981.
2. R. Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases", ACM Transactions on Database Systems 2 (3), Sept. 1977.
3. R. Fagin, "Normal Forms and Relational Database Operators", ACM SIGMOD International Conference on Management of Data, May 31-June 1, 1979, Boston, Mass. Also IBM Research Report RJ2471, Feb. 1979.
4. https://www.dlsweb.rmit.edu.au/toolbox/knowmang/content/normalisation/bcnf.htm

Answer to Self Assessment Questions (Unit-1)

1. What do you mean by RDBMS? Give Examples of it.

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

Examples of RDBMS are: MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access etc.

2. What is a relation? What are its characteristics?

A relational database is a set of tables containing data fitted into predefined categories. Each table (which is sometimes called a relation) contains one or more data categories in columns. Each row contains a unique instance of data for the categories defined by the columns.

A relational table has the following properties:

- The table has a name that is distinct from all other tables in the database.
- Each cell of the table contains exactly one value or atomic value.
- Each column has a distinct name called attribute.
- The values of a column are all from the same domain.
- The order of columns has no significance.
- Each record is distinct; there are no duplicate records.
- The order of records has no significance, theoretically.

3. What are the relational algebra operations that can be performed? What are the conditions for different set operations in RA?

The fundamental operations used in Relational Algebra are:

a. Select (unary)
b. Project (unary)
c. Rename (unary)
d. Cartesian Product (binary)
e. Union (binary)
f. Set-Difference (binary)

The conditions for set operations (union, set-intersect and set-difference) are:

i) The relations must be of the same arity. That means they must have the same number of attributes.

ii) The domains of *i*th attribute of the first set and the *i*th attribute of the second set must be the same, for all *i*.

4. What are the conditions for insertion?

The conditions for insertion are:

1. The attribute values for inserted tuples must be members of the attribute's domain.
2. Tuples inserted must be of the same arity.

5. Write short notes on natural join, theta joins.

Natural Join:

The natural join is a binary operation that allows us to combine certain selection and a Cartesian product into one operation. It is denoted by the "join" symbol □

The natural join operation forms:

a) A Cartesian product of two arguments
b) Performs a selection forcing equality on those attributes that appear in both relation schemas
c) Removes duplicate attributes

Theta Join:

The theta join operation is an extension to the natural join operation that allows us to combine a selection and a Cartesian product into a single operation. Consider relations r(R) and s(S); let θ be predicate on attributes in the schema R ∪ S. The theta join operation is defined as follows:

R □θS = σθ (r × s)

Answer to Self Assessment Questions (Unit-2)

1. What does SQL stand for? What is its use?

SQL stands for Structured Query Language. A SQL command or statement is issued to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems.

SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc. However, the standard SQL commands such as "Select", "Insert", "Update", "Delete", "Create", and "Drop" can be used to accomplish almost everything that one needs to do with a database.

2. Explain the function of each of the clauses in the SELECT statement. What restrictions are imposed on these clauses?

FROM: specifies the table or tables to be used;
WHERE: This clause filters the rows subject to some condition;
GROUP BY: This clause forms groups of rows with the same column value;
HAVING: This clause filters the groups subject to some condition;
SELECT: This clause specifies which columns are to appear in the output;
ORDER BY: This clause specifies the order of the output.

3. Explain how the GROUP BY clause works. What is the difference between the WHERE and HAVING clauses?

SQL first applies the WHERE clause. Then it conceptually arranges the table based on the grouping column(s). Next, applies the HAVING clause and finally orders the result according to the ORDER BY clause.

WHERE filters rows subject to some condition; HAVING filters groups subject to some condition.

4. Explain the use of Grant and Revoke Commands.

DCL commands are used to enforce database security in a multiple database environment.

Two types of DCL commands are: Grant and Revoke.

SQL Grant command is used to provide access or privileges on the database objects to the users.

The syntax for the GRANT command is: "GRANT privilege_name ON object_name " ""TO {user_name | PUBLIC | role_name} [with GRANT option]; Here, privilege_name: is the access right or privilege granted to the user. object_name: is the name of the database object like table, view etc. user_name: is the name of the user to whom an access right is being granted. Public is used to grant rights to all the users.

with Grant option: allows users to grant access rights to other users.

Example:

GRANT SELECT ON employee TO user1.

This command grants a SELECT permission on employee table to user1.

The revoke command removes user access rights or privileges to the database objects

The syntax for the REVOKE command is:

REVOKE privilege_name ON object_name FROM {User_name | PUBLIC | Role_name}

Example:

REVOKE SELECT ON employee FROM user1.

This command will revoke a SELECT privilege on employee table from user1.

5. What are Transaction Control Language Commands?

Transaction Control Language (TCL) commands is used to manage transactions in database. These are used to manage the changes made by DML statements. It also allows statements to be grouped together into logical transactions.

Commit Command

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

Following is Commit command's syntax,

Commit;

Rollback command

This command restores the database to last committed state. It is also use with save point command to jump to a savepoint in a transaction.

Following is Rollback command's syntax,

rollback to *savepoint-name*

Savepoint command

savepoint command is used to temporarily save a transaction so that you can rollback to that point whenever necessary.

Following is savepoint command's syntax,

savepoint *savepoint-name*;

Answer to Self Assessment Questions (Unit-3)

1. **What do you mean by Functional Dependency? Explain.**

If there is a dependency in a database such that attributes B is dependent upon attribute A, you would write this as "A → B".

In other words, a dependency FD: $A \rightarrow$ B means that the values of *B* are determined by the values of A. Two tuples sharing the same values of *X* will necessarily have the same values of B.

For example, in table listing employee characteristics including Social Security Number (SSN) and name, it can be said that name is dependent upon SSN (or SSN→ name) because an employee's name can be uniquely determined from their SSN.

2. **What do you mean by fully functional dependency?**

A fully or full functional dependency occurs when you already meet the requirements for a functional dependency and the set of attributes on the left side of the functional dependency statement cannot be reduced any further. For example, "{SSN, age} –> name" is a functional dependency, but it is not a full functional

dependency because you can remove age from the left side of the statement without impacting the dependency relationship.

3. What are the advantages of Functional Dependency?

- Functional Dependency avoids data redundancy where same data should not be repeated at multiple locations in same database.
- It maintains the quality of data in database.
- It allows clearly defined meanings and constraints of databases.
- It helps in identifying bad designs.
- It expresses the facts about the database design.

4. What is decomposition? Write the properties of decomposition

Decomposition is the process of breaking down in parts or elements. It replaces a relation with a collection of smaller relations. It breaks the table into multiple tables in a database. It should always be lossless, because it confirms that the information in the original relation can be accurately reconstructed based on the decomposed relations. If there is no proper decomposition of the relation, then it may lead to problems like loss of information. Following are the properties of decomposition

(i) Lossless Decomposition
(ii) Dependency Preservation
(iii) Lack of Data Redundancy

5. What are the design goals for relational database design? Explain why each is desirable.

The design goals for relational database design are:

i. Lossless-join decompositions
ii. Dependency preserving decompositions
iii. Minimization of repetition of information

They are desirable so we can maintain an accurate database, check correctness of updates quickly, and use the smallest amount of space possible.

Answer to Self Assessment Questions (Unit-4)

1. What is normalization? Why it is necessary?

Normalization is a systematic way of ensuring that a database structure is suitable for general-purpose querying and free of certain undesirable characteristics—insertion, update, and deletion anomalies—that could lead to a loss of data integrity. The main purpose of normalization is to minimize data redundancy and improve data integrity.

2. What is 1NF? Explain with the example.

A database is in first normal form if it satisfies the following conditions:

- Contains only atomic values
- There are no repeating groups

For example, in the table shown below, the values in the [Color] column in the first row can be divided into "red" and "green", hence [TABLE_PRODUCT] is not in 1NF

TABLE_PRODUCT

Product ID	Color	Price
1	red, green	15.99
2	yellow	23.99
3	green	17.50
4	yellow, blue	9.99
5	red	29.99

To bring this table to first normal form, we split the table into two tables and now we have the resulting tables:

TABLE_PRODUCT_PRICE

Product ID	Price
1	15.99
2	23.99
3	17.50
4	9.99
5	29.99

TABLE_PRODUCT_COLOR

Product ID	Color
1	red
1	green
2	yellow
3	green
4	yellow
4	blue
5	red

3. Explain detail about 3rd Normal Form.

3rd Normal Form:

A database is in third normal form if it satisfies the following conditions:

i) It is in second normal form

ii) There is no transitive functional dependency

By transitive functional dependency, we mean we have the following relationships in the table: A is functionally dependent on B, and B is functionally dependent on C. In this case, C is transitively dependent on A via B.

Example: Consider the following example:

TABLE_BOOK_DETAIL

Book ID	Genre ID	Genre Type	Price
1	1	Gardening	25.99
2	2	Sports	14.99
3	1	Gardening	10.00
4	3	Travel	12.99
5	2	Sports	17.99

In the table able, [Book ID] determines [Genre ID], and [Genre ID] determines [Genre Type]. Therefore, [Book ID] determines [Genre Type] via [Genre ID] and we have transitive functional dependency, and this structure does not satisfy third normal form.

To bring this table to third normal form, we split the table into two as follows:

TABLE_BOOK

Book ID	Genre ID	Price
1	1	25.99
2	2	14.99
3	1	10.00
4	3	12.99
5	2	17.99

TABLE_GENRE

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

Now all non-key attributes are fully functional dependent only on the primary key. In [TABLE_BOOK], both [Genre ID] and [Price] are only dependent on [Book ID]. In [TABLE_GENRE], [Genre Type] is only dependent on [Genre ID].

4. Differentiate between 3NF and BCNF

Both 3NF and BCNF are normal forms that are used in relational databases to minimize redundancies in tables. In a table that is in the BCNF normal form, for every non-trivial functional dependency of the form A → B, A is a super-key whereas, a table that complies with 3NF should be in the 2NF, and every non-prime attribute should directly depend on every candidate key of that table. BCNF is considered as a stronger normal form than the 3NF and it was developed to capture some of the anomalies that could not be captured by 3NF. Obtaining a table that complies with the BCNF form will require decomposing a table that is in the 3NF.

5. Write short notes on 4NF

Fourth normal form (4NF) is a level of database normalization where there are no non-trivial multivalued dependencies other than a candidate key.

It builds on the first three normal forms (1NF, 2NF and 3NF) and the Boyce-Codd Normal Form (BCNF). It states that, in addition to a database meeting the requirements of BCNF; it must not contain more than one multivalued dependency.

In other words, Attribute of one or more rows in the table should not result in more than one rows of the same table leading to multi-valued dependencies.