RELATIONAL MODEL

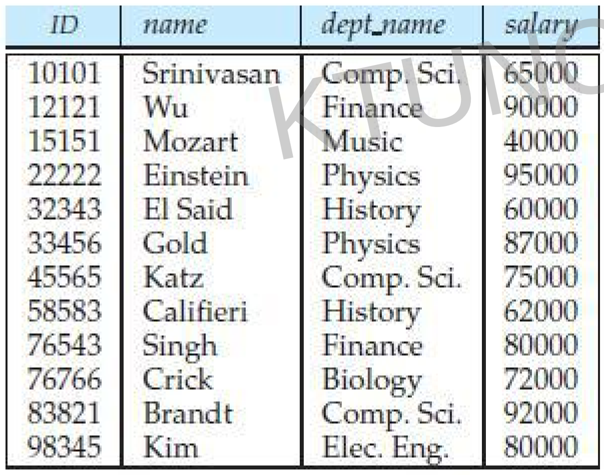
The relational model is today the primary data model for commercial data processing applications.This model is simple and it has all the properties and capabilities required to process data with storage efficiency.

Salient features of the relational model: –

- Conceptually simple.
- Fundamentals are intuitive and easy to pick up.
- Powerful underlying theory: the relational model is the only database model that is powered by formal mathematics, which results in excellent dividends when developing database algorithms and techniques.
- Easy-to-use database language: though not formally part of the relational model, part of its success is due to SQL, the de facto language for working with relational databases.

Structure

• A relational database is a collection of **tables**.
– Each table has a unique name.
– Each table consists of multiple rows.
– Each row is a set of values that by definition are related to each other in some way; these values conform to the attributes or columns of the table.
– Each attribute of a table defines a set of permitted values for that attribute; this set of permitted set is the **domain** of that attribute.



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

Figure 2.1 The *instructor* relation.

For example, consider the instructor table of Figure 2.1, which stores information about instructors. The table has four column headers: ID, name, deptname, and salary. Each row of this table records information about an instructor, consisting of the instructor's ID, name, dept name, and salary. Similarly, the course table of Figure 2.2 stores information about courses, consisting of a course id, title, dept name, and credits, for each course. Note that each instructor is identified by the value of the column ID, while each course is identified by the value of the column course id.

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

Figure 2.2 The *course* relation.

In general, a row in a table represents a relationship among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name. In mathematical terminology, a tuple is simply a sequence (or list) of values. A relationship between n values is represented mathematically by an n-tuple of values, i.e., a tuple with n values, which corresponds to a row in a table.

Thus, in the relational model the term relation is used to refer to a table, while the term tuple is used to refer to a row. Similarly, the term attribute refers to a column of a table.
We use the term relation instance to refer to a specific instance of a relation, i.e., containing a specific set of rows. The instance of instructor shown in Figure 2.1 has 12 tuples, corresponding to 12 instructors.

For each attribute of a relation, there is a set of permitted values, called the domain of that attribute. Thus, the domain of the salary attribute of the instructor relation is the set of all possible salary values, while the domain of the name attribute is the set of all possible instructor names. We require that, for all relations r, the domains of all attributes of r be atomic.
A domain is atomic if elements of the domain are considered to be indivisible units. For example, suppose the table instructor had an attribute phone number, which can store a set of phone numbers corresponding to the instructor. Then the domain of phone number would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely the individual phone numbers in the set.

The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the phone number attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code and a local number, we would be treating it as a nonatomic value. If we treat each phone number as a single indivisible unit, then the attribute phone number would have an atomic domain.

• This definition of a database table originates from the pure mathematical concept of a relation, from which the term “relational data model” originates.
– Formally, for a table r with n attributes a1 . . . an, each attribute ak has a domain Dk, and any given row of r is an n-tuple (v1, . . . , vn) such that vk ∈ Dk.
– Thus, any instance of table r is a subset of the Cartesian product D1 × · · · × Dn.
– We require that a domain Dk be atomic — that is, we do not consider the elements of Dk to be breakable into subcomponents.
– A possible member of any domain is null — that is, an unknown or non-existent value; in practice, we try to avoid the inclusion of null in our databases because they can cause a number of practical issues.

ER Model to Relational Model

ER diagrams can be mapped to relational schema, that is, it is possible to create relational schema using ER diagram. We cannot import all the ER constraints into relational model, but an approximate schema can be generated.

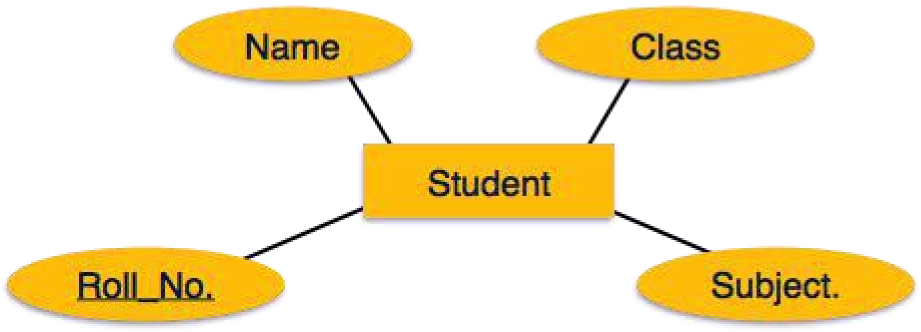
There are several processes and algorithms available to convert ER Diagrams into Relational Schema. Some of them are automated and some of them are manual. We may focus here on the mapping diagram contents to relational basics.

ER diagrams mainly comprise of −

- Entity and its attributes
- Relationship, which is association among entities.

Mapping Entity

An entity is a real-world object with some attributes.

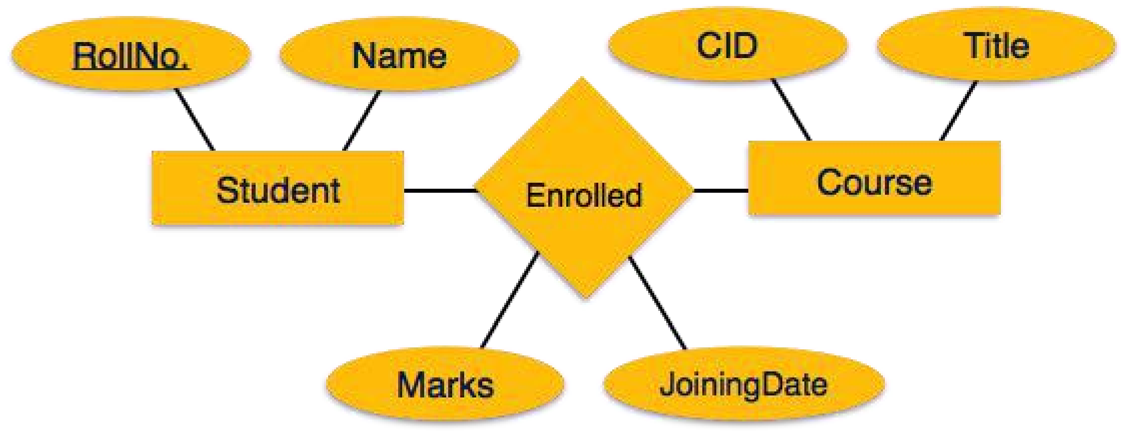


Mapping Process (Algorithm)

- Create table for each entity.
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key.

Mapping Relationship

A relationship is an association among entities.

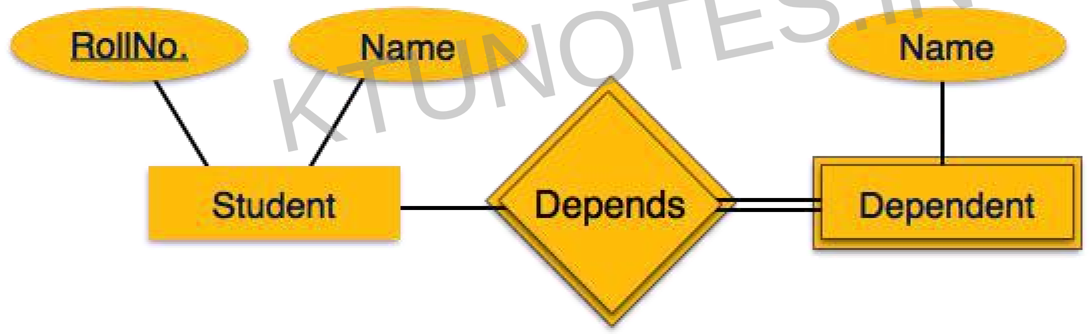


Mapping Process

- Create table for a relationship.
- Add the primary keys of all participating Entities as fields of table with their respective data types.
- If relationship has any attribute, add each attribute as field of table.
- Declare a primary key composing all the primary keys of participating entities.
- Declare all foreign key constraints.

Mapping Weak Entity Sets

A weak entity set is one which does not have any primary key associated with it.

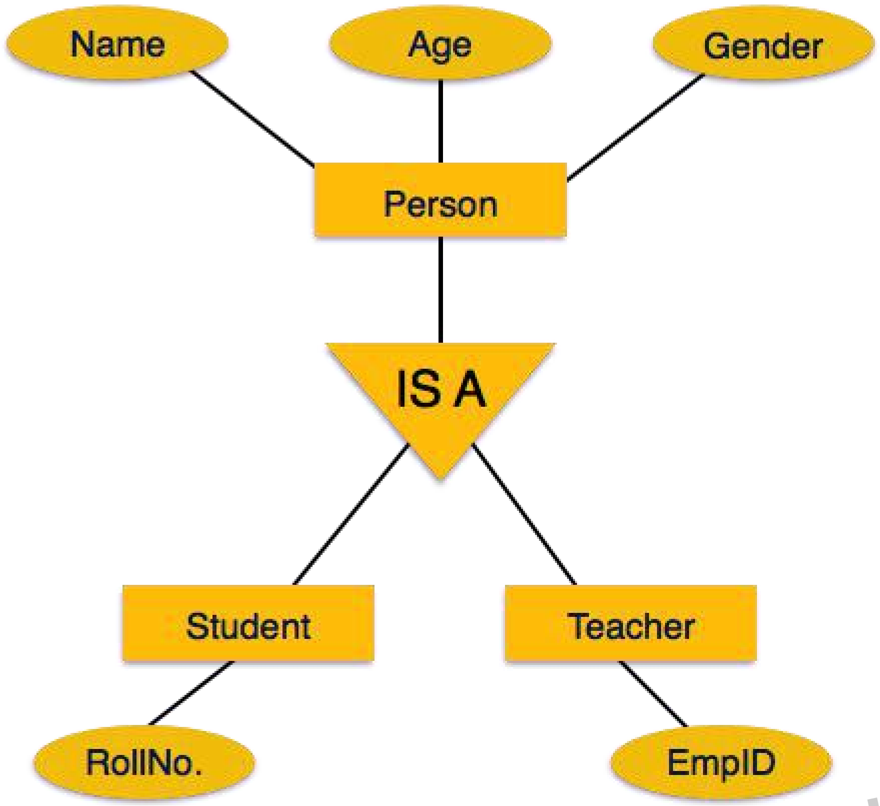


Mapping Process

- Create table for weak entity set.
- Add all its attributes to table as field.
- Add the primary key of identifying entity set.
- Declare all foreign key constraints.

Mapping Hierarchical Entities

ER specialization or generalization comes in the form of hierarchical entity sets.



Mapping Process

- Create tables for all higher-level entities.
- Create tables for lower-level entities.
- Add primary keys of higher-level entities in the table of lower-level entities.
- In lower-level tables, add all other attributes of lower-level entities.
- Declare primary key of higher-level table and the primary key for lower-level table.
- Declare foreign key constraints.

RELATIONAL MODEL -- Integrity Constraints

An **integrity constraint** (**IC**) is a condition specified on a database schema and restricts the data that can be stored in an instance of the database. If a database instance satisfies all the integrity constraints specifies on the database schema, it is a **legal** instance. A DBMS permits only legal instances to be stored in the database.
Database constraints are restrictions on the contents of the database or on database operations. It is a condition specified on a database schema that restricts the data to be inserted in an instance of the database.

Need of Constraints :
Constraints in the database provide a way to guarantee that : the values of individual columns are valid.

In a table, rows have a valid primary key or unique key values.
In a dependent table, rows have valid foreign key values that reference rows in a parent table.
Many kinds of integrity constraints can be specified in the relational model:

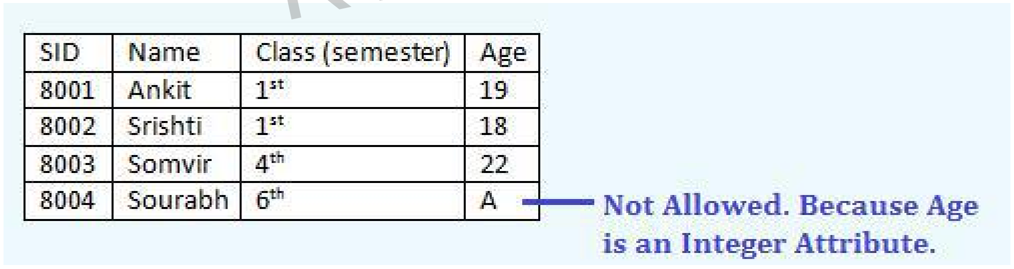
Domain Integrity

Domain integrity means the definition of a valid set of values for an attribute. You define 聽
- data type,
- length or size
- is null value allowed
- is the value unique or not for an attribute.

You may also define the default value, the range (values in between) and/or specific values for the attribute. Some DBMS allow you to define the output format and/or input mask for the attribute.
These definitions ensure that a specific attribute will have a right and proper value in the database.

Domain Constraints specifies that what set of values an attribute can take. Value of each attribute X must be an atomic value from the domain of X. The data type associated with domains include integer, character, string, date, time, currency etc. An attribute value must be available in the corresponding domain. For example, the employee ID must be unique, the employee birthday is in the range [Jan 1, 1950, Jan 1, 2000]. Such information is provided in logical statements called integrity constraints.

Consider the example below –



SID	Name	Class (semester)	Age
8001	Ankit	1st	19
8002	Srishti	1st	18
8003	Somvir	4th	22
8004	Sourabh	6th	A

Not Allowed. Because Age is an Integer Attribute.

Key Constraints

Keys are attributes or sets of attributes that uniquely identify an entity within its entity set. An Entity set E can have multiple keys out of which one key will be designated as the primary key.

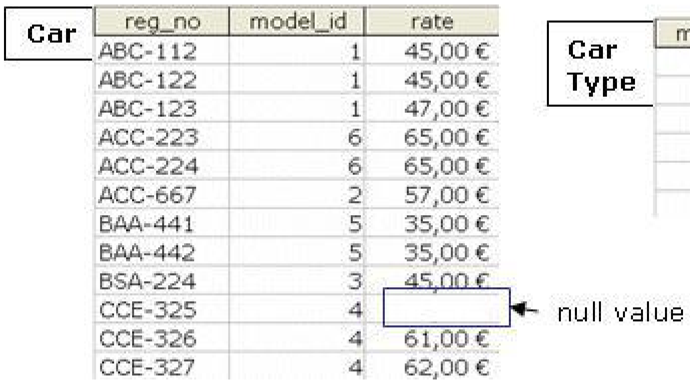
Entity Integrity Constraint

table requires a primary key. The primary key, nor any part of the primary key, can contain NULL values. This is because NULL values for the primary key means we cannot identify some rows. For example, in the EMPLOYEE table, Phone cannot be a key since some people may not have a phone.

The entity integrity constraint states that primary keys can't be null. There must be a proper value in the primary key field.

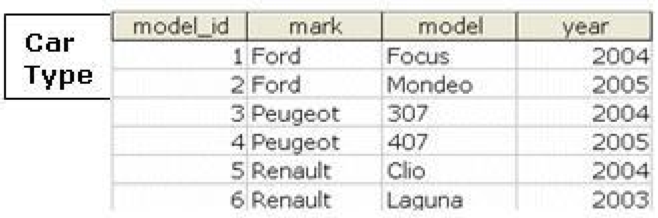
On the other hand, there can be null values other than primary key fields. Null value means that one doesn't know the value for that field. Null value is different from zero value or space.

In the Car Rental database in the Car table each car must have a proper and unique Reg_No. There might be a car whose rate is unknown - maybe the car is broken or it is brand new - i.e. the Rate field has a null value. See the picture below.



Car

reg_no	model_id	rate
ABC-112	1	45,00 €
ABC-122	1	45,00 €
ABC-123	1	47,00 €
ACC-223	6	65,00 €
ACC-224	6	65,00 €
ACC-667	2	57,00 €
BAA-441	5	35,00 €
BAA-442	5	35,00 €
BSA-224	3	45,00 €
CCE-325	4	
CCE-326	4	61,00 €
CCE-327	4	62,00 €



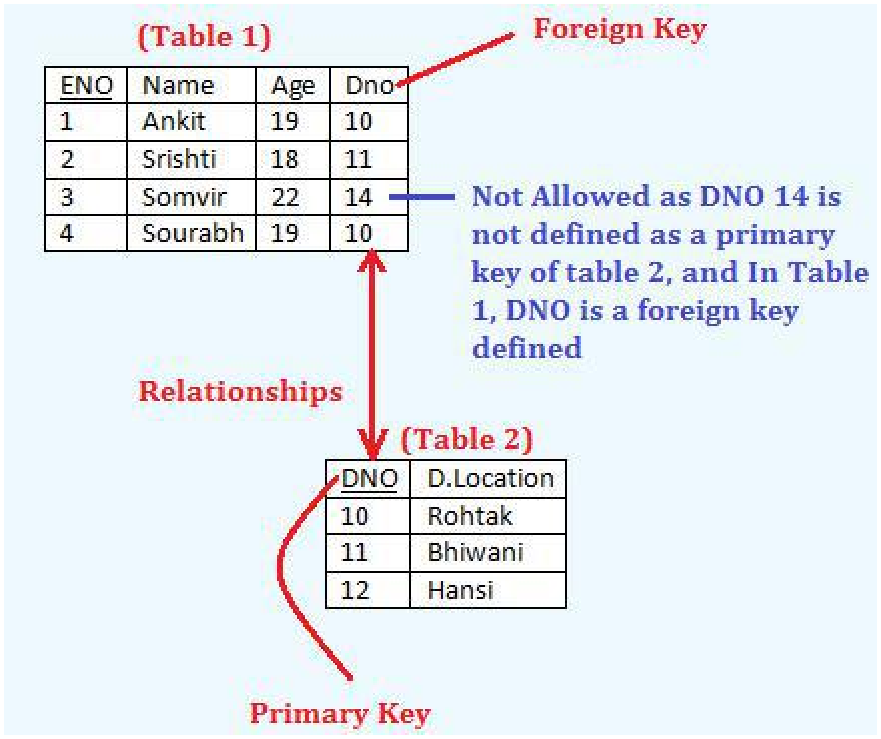
Car Type

model_id	mark	model	year
1	Ford	Focus	2004
2	Ford	Mondeo	2005
3	Peugeot	307	2004
4	Peugeot	407	2005
5	Renault	Clio	2004
6	Renault	Laguna	2003

Referential integrity

A foreign key must have a matching primary key or it must be null.This constraint is specified between two tables (parent and child); it maintains the correspondence between rows in these tables. It means the reference from a row in one table to other table must be valid. Examples of Referential integrity constraint:

This rule states that if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2. For example,


(Table 1)
Foreign Key
Not Allowed as DNO 14 is not defined as a primary key of table 2, and In Table 1, DNO is a foreign key defined
Relationships
(Table 2)
Primary Key

ENO	Name	Age	Dno
1	Ankit	19	10
2	Srishti	18	11
3	Somvir	22	14
4	Sourabh	19	10

DNO	D.Location
10	Rohtak
11	Bhiwani
12	Hansi

Some more Features of Foreign Key

Let the table in which the foreign key is defined is Foreign Table or details table i.e. Table 1 in above example and the table that defines the primary key and is referenced by the foreign key is master table or primary table i.e. Table 2 in above example. Then the following properties must be hold :

Records cannot be inserted into a Foreign table if corresponding records in the master table do not exist. Records of the master table or Primary Table cannot be deleted or updated if corresponding records in the detail table actually exist.

DATABASE LANGUAGES

Relational Query Languages

- A query language is a language in which a user requests information from the database. These languages are usually on a level higher than that of a standard programming language.

Query languages can be categorized as either procedural or nonprocedural. In a **procedural language**, the user instructs the system to perform a sequence of operations on the database to compute the desired result. In a **nonprocedural language**, the user describes the desired information without giving a specific procedure for obtaining that information.

classroom(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
section(course_id, sec_id, semester, year, building, room_number, time_slot_id)
teaches(ID, course_id, sec_id, semester, year)
student(ID, name, dept_name, tot_cred)
takes(ID, course_id, sec_id, semester, year, grade)
advisor(s_ID, i_ID)
time_slot(time_slot_id, day, start_time, end_time)
prereq(course_id, prereq_id)

Figure 2.9 Schema of the university database.

There are a number of "pure" query languages: The relational algebra is procedural, whereas the tuple relational calculus and domain relational calculus are nonprocedural.

Relational Operations

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

The operations are expressed in SQL.

- The most frequent operation is the **selection** of specific tuples from a single relation (say instructor) that satisfies some particular predicate (say salary > $85,000). The result is a new relation that is a subset of the original relation (in satisfying the predicate "salary is greater than $85000",we get the result shown in Figure 2.10.

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

Figure 2.1 The *instructor* relation.

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

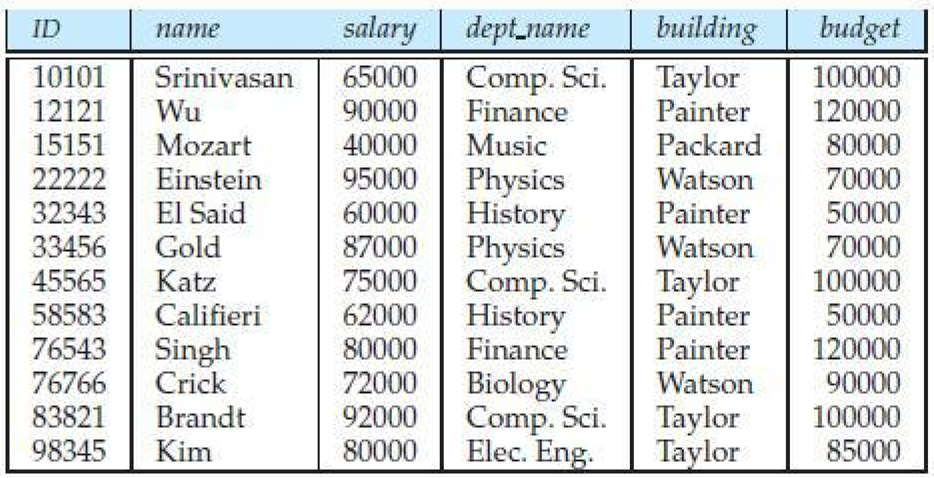
Figure 2.10 Result of query selecting *instructor* tuples with salary greater than $85000.

- Another frequent operation is to **select certain attributes (columns) from a relation**. The result is a new relation having only those selected attributes. For example, suppose we want a list of instructor IDs and salaries without listing the name and dept name values from the instructor relation of Figure 2.1, then the result, shown in Figure 2.11, has the two attributes ID and salary. in the result is derived from a tuple of the instructor relation but with only selected attributes shown.

ID	*salary*
10101	65000
12121	90000
15151	40000
22222	95000
32343	60000
33456	87000
45565	75000
58583	62000
76543	80000
76766	72000
83821	92000
98345	80000

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

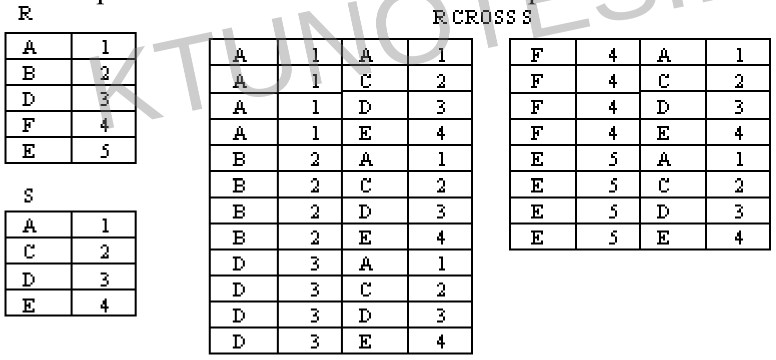
- The **join operation** allows the combining of two relations by merging pairs of tuples, one from each relation, into a single tuple. There are a number of different ways to join relations. Figure 2.12 shows an example of joining the tuples from the instructor and department tables with the new tuples showing the information about each instructor and the department in which she is working. This result was formed by combining each tuple in the instructor relation with the tuple in the department relation for the instructor's department. In the form of join shown in Figure 2.12, which is called a natural join, a tuple from the instructor relation matches a tuple in the department relation if the values of their dept name attributes are the same. All such matching pairs of tuples are present in the join result. In general, the natural join operation on two relations matches tuples whose values are the same on all attribute names that are common to both relations.



ID	name	salary	dept_name	building	budget
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
12121	Wu	90000	Finance	Painter	120000
15151	Mozart	40000	Music	Packard	80000
22222	Einstein	95000	Physics	Watson	70000
32343	El Said	60000	History	Painter	50000
33456	Gold	87000	Physics	Watson	70000
45565	Katz	75000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
76543	Singh	80000	Finance	Painter	120000
76766	Crick	72000	Biology	Watson	90000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000

Figure 2.12 Result of natural join of the *instructor* and *department* relations.

- The **Cartesian product** operation combines tuples from two relations, but unlike the join operation, its result contains all pairs of tuples from the two relations, regardless of whether their attribute values match. Because relations are sets,we can perform normal set operations on relations.
- ❖ The Cartesian Product is also an operator which works on two sets. It is sometimes called the **CROSS PRODUCT** or **CROSS JOIN**.
- ❖ It combines the tuples of one relation with all the tuples of the other relation.



R

A	1
B	2
D	3
F	4
E	5

S

A	1
C	2
D	3
E	4

R CROSS S

A	1	A	1
A	1	C	2
A	1	D	3
A	1	E	4
B	2	A	1
B	2	C	2
B	2	D	3
B	2	E	4
D	3	A	1
D	3	C	2
D	3	D	3
D	3	E	4
F	4	A	1
F	4	C	2
F	4	D	3
F	4	E	4
E	5	A	1
E	5	C	2
E	5	D	3
E	5	E	4

Figure: Cartesian product

The **union operation** performs a set union of two "similarly structured" tables (say a table of all graduate students and a table of all undergraduate students).
For example, one can obtain the set of all students in a department. Other set operations, such as intersection and set difference can be performed as well.
As we noted earlier, we can perform operations on the results of queries. For example, if we want to find the ID and salary for those instructors who have salary greater than $85,000, we would perform the first two operations in our example above. First we select those tuples from the instructor relation where the salary value is greater than $85,000 and then, from that result, select the two attributes ID and salary, resulting in the relation shown in Figure 2.13 consisting of the ID.

ID	*salary*
12121	90000
22222	95000
33456	87000
83821	92000

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

RELATIONAL ALGEBRA

The relational algebra defines a set of operations on relations, paralleling the usual algebraic operations such as addition, subtraction or multiplication, which operate on numbers. Just as algebraic operations on numbers take one or more numbers as input and return a number as output, the relational algebra operations typically take one or two relations as input and return a relation as output.

Symbol (Name)	Example of Use
σ (Selection)	$\sigma_{\text{salary}>=85000}(instructor)$ Return rows of the input relation that satisfy the predicate.
Π (Projection)	$\Pi_{ID, salary}(instructor)$ Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.
$\bowtie$ (Natural join)	$instructor \bowtie department$ Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.
$\times$ (Cartesian product)	$instructor \times department$ Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes)
$\cup$ (Union)	$\Pi_{name}(instructor) \cup \Pi_{name}(student)$ Output the union of tuples from the two input relations.

Table of Contents

2.1 Relational Databases

The relational model for database management is a data model based on set theory. It was introduced by Edgar F. Codd of IBM Research in 1970. The fundamental assumption of the relational model is that all data are represented as mathematical n-ary **relations**, an n-ary relation being a subset of the Cartesian product of n sets. This is collection of tables which is assigned a unique name. A row in the table represents a relationship among a set of values.

Basic components of relational database are

a) Set of domains and Set of relations
b) Integrity Rules
c) Set of Operations

2.1.1 Basic Concepts

a) Tuples - The rows in a relation are called tuples. Each row consists of a sequence of values, one for each column in the table. In addition, each row (or record) in a table must be unique.

b) Attributes – The column in a relation is called attribute. The attributes represent characteristics of an entity.

c) Domain – For each attribute there is a set of permitted values called domain of that attribute. For all relations 'r', the domain of all attributes of 'r' should be atomic. A domain is said to be **atomic** if elements of the domain are considered to be indivisible units. It is possible to have several attributes to have the same domain. NULL value is the member of any domain.

d) Database Schema – Logical design of the database is termed as database schema.

e) Relation schema – The concept of relation schema corresponds to the programming notion of type definition. It can be considered as the definition of a domain of values. The database schema is the collection of relation schemas that define a database. The relation has two parts a relation scheme (header) and a time varying set of tuples (body). Relation is the subset of a Cartesian product of a list of domains.

f) Degree of the relation is the number of attributes in the relation. This is also called **arity** of the relation. The arity of a relation is the number of domains in the corresponding Cartesian product.

g) Keys – A key specifying uniqueness. Keys in relational model are primary key, candidate key and super key.

Super key is defined in a relational model as a set of attributes that, taken collectively, to uniquely identify a tuple in the relation. For eg: the *social_security_no* attribute of the relation employee is sufficient to distinguish one employee from another. Thus *social_security_no* is a superkey for the relation employee.

Candidate key: Superkeys with minimal subset is known as the candidate key. For eg: it is possible to combine the attributes, *employee_id & organization_name* to form a superkey. But the *social_security_no* is sufficient to distinguish the two employees. Thus *social_security_no* is a candidate key.

Primary key is used to denote the candidate key that is chosen by the database designer to uniquely identify a tuple in a relation.

A **foreign key** is an attribute or set of attributes of a relation say R(R) such that the value of each attribute in this set is that of a primary key of the relation S(S).

2.1.2 Integrity Rules

Relational algebra includes two general integrity rules.

a) **Entity Integrity**: This is concerned with primary key values of individual relations. This specifies that instances of entities are distinguishable and no prime attribute value may be NULL.

b) **Referential Integrity**: Referential integrity is specified between two relations and is used to maintain the consistency among tuples of the two relations. The referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation. The attributes in a relation that provides reference to tuple in another relation is called foreign key.

2.1.3 Set of operations

The operations of relational model can be categorized into retrievals and updates. There are three basic update operations on relations: (a) Insert (b) Delete and (c) Update (or Modify). The update operations and the effect of these operations on the various constraints specified on the relational database schema are explained below:

Insert: The Insert operation is used to insert a new tuple or tuples in a relation. The Insert operation provides a list of attribute values for a new tuple that is to be inserted into a relation. Insert can violate any of the following four types of constraints (entity integrity and referential integrity constraints):

- Domain constraints can be violated if an attribute value doesn't exist in the corresponding domain.

- Unique Key constraints can be violated if the key value of the new tuple already exists in another tuple in the relation.
- Entity integrity can be violated if the primary key of the new tuple is null.
- Referential integrity can be violated if the value of any foreign key in the new tuple refers to a non-existing tuple in the referenced relation.

If an Insert operation violates one or more constraints, the DBMS rejects the insertion and informs the user about the reason for rejection.

Example of an Insert operation is given below:

INSERT <'Derek','D','Brown','348765439','1958-05-09','1234 Church Road, CA 80134','M',30000,null,5> INTO EMPLOYEE.

Delete: The Delete operation is used to delete tuples and the user has to provide a condition on the attributes of the relation to select the tuple(s) to be deleted. The Delete operation can violate only referential integrity, if the tuple being deleted is referenced by the foreign keys from other tuples in the database. If a Delete operation violates any referential integrity constraints, then the DBMS rejects the deletion and informs the user about the reason for rejection. Then the user will have following two options:

- Attempt to cascade the deletion by deleting tuples that reference the tuple being deleted.
- Modify the referencing attribute values that cause the violation; each value is set to null or changed to reference another valid tuple. The tuple can then be deleted without violating the referential integrity.

Example of a Delete operation is given below:

DELETE the EMPLOYEE tuple with SSN = '909880123'.

Update: The Update operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation. The user has to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Updating a non-key (neither primary key nor foreign key) attribute causes no constraint violations normally, except for the domain constraint violation if the new value of the attribute doesn't exist in the corresponding domain. Modifying a primary key value is similar to deleting one tuple and inserting another in its place. Hence, the constraint violations discussed under both Insert and Delete could happen in case of modifying a primary key value. If a foreign key attribute is modified, the referential integrity constraint violation could happen if the new value is not null and refers to a non-existing tuple in the referenced relation.

Example of an Update operation is given below:

UPDATE the SALARY of the EMPLOYEE tuple with SSN = ‘909880123’ to 35000.

The relational algebra operations are used to specify the retrieval operations in relational model. Hence, knowledge about relational algebra and its operations are required to understand retrieval operations in relational model. Relational algebra and operations are discussed in the sub section 2.2 below.

2.2 Relational Algebra

The relational algebra is a procedural query language. It consists of collection of operations to manipulate relations. It is similar to normal algebra (as in 2+3*x-y), except we use relations as values instead of numbers, and the operations and operators are different. Relations in relational algebra are seen as sets of tuples, so we can use basic set operations. The fundamental operations in relational algebra are select, project, union, set difference, Cartesian product and rename. There are several other operations namely, set intersection, natural join, division and assignment.

However, relational algebra is not used as a query language in actual DBMSs (SQL is used instead). The inner, lower level operations of a relational DBMS are, or are similar to, relational algebra operations. Knowledge about relational algebra is required to understand query execution and optimization in a relational DBMS. Also, some advanced SQL queries require explicit relational algebra operations, most commonly *outer join*. Difference between relational algebra and SQL are:

- SQL is a declarative query language, which means that you tell DBMS *what* you want, but not *how* to get it. Relational Algebra is procedural (like procedural programming languages C++ or Java), which means that you have to state, step by step, exactly how the result should be calculated. Relational algebra is (more) procedural than SQL. Actually, relational algebra is mathematical expressions.
- Relations are seen as sets of tuples, which means that no duplicates are allowed. SQL behaves differently in some cases (refer the SQL keyword *distinct*).

Basic Relational Algebra Operations

The data model also includes a set of operations to manipulate the data in addition to defining the database structure and constraints. The retrieval operations in relational model are based on relational algebra operations which are divided into two groups:

- Set Operations – UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT.
- Operations developed specifically for relational databases – SELECT, PROJECT, JOIN, etc.

The relation algebra operations have their own symbols. A summary of relational algebra operations and their symbols are given in the table below. The set of relational algebra operations **{σ, Π, U, -, X}** is a complete set; that is, any of the other relational algebra operations can be expressed as a sequence of operations from this set.

Operation	Symbol Name	Symbol
Projection	**PROJECT**	π
Selection	**SELECT**	σ
Renaming	**RENAME**	ρ
Union	**UNION**	$\cup$
Intersectio n	**INTERSECTION**	$\cap$
Assignmen t	**<-**	$\leftarrow$

Operation	Symbol Name	Symbol
Cartesian product	**X**	$\times$
Join	**JOIN**	$\bowtie$
Left outer join	**LEFT OUTER JOIN**	⟕
Right outer join	**RIGHT OUTER JOIN**	⟖
Full outer join	**FULL OUTER JOIN**	⟗
Semijoin	**SEMIJOIN**	$\ltimes$

a) SELECT Operation (σ)

The select operation selects tuples that satisfy a given condition. The selection condition appears as a subscript to σ. The SELECT is a unary operation; that is, it is applied to a single relation. It can be visualized as a horizontal partition of the relation into two sets of tuples – those tuples that satisfy the condition and are selected and those tuples that do not satisfy the condition and are discarded. The general syntax of the SELECT operation is:

$$\sigma_{\text{<selection condition>}}(R)$$

OR

$$\text{SELECT}_{\text{<selection condition>}}(R)$$

where the symbol **σ** (sigma) is used to denote the SELECT operator, and the selection condition is a Boolean expression specified on the attributes of relation R. R could be just the name of a database relation or a relational algebra expression whose result is a relation. The result of the SELECT operation will be another relation having the same attributes (same degree) as R. The number of tuples in the resulting relation is always less than or equal to the number of tuples in R. That is $|\sigma_c(R)| \leq |R|$. The Boolean expression specified in <selection condition> can be a number of clauses of the following forms:

<attribute name><comparison op><constant value>

OR

<attribute name><comparison op><attribute name>

where <attribute name> is the name of an attribute of R, <comparison op> is normally one of the operators =, $\neq$, <, $\leq$, > and $\geq$, and <constant value> is a constant value from the attribute domain. Clauses can be connected by the logical operators $\vee$ (OR) and $\wedge$ (AND) and NOT (!) to form a general selection condition.

The fraction of tuples selected by a selection condition is referred to as the **selectivity** of the condition. The SELECT operation is commutative; that is a cascade of SELECT operations on a relation gives the same result irrespective of the order of different SELECT operations. Also, a cascade of SELECT operations can be combined into a single SELECT operation with conditions of all individual SELECT operations connected using AND logical operators.

Example: Select all employees who either work in MKTG department and make over $25000 per year, or work in PROJ department and make over $30000.

$$\sigma_{\text{(DEPT='MKTG' AND SALARY>25000) OR (DEPT='PROJ' AND SALARY>30000)}}(\text{EMPLOYEE})$$

b) PROJECT Operation (∏)

The project operation is used to retrieve specific attributes/columns from a relation. The PROJECT operation is also a unary operation. The SELECT operation selects some of the rows from the table while discarding other rows whereas the PROJECT operation selects certain columns from the table and discards the other columns. ie. It *projects* only the attributes of interest to the user. The general syntax of the PROJECT operation is:

$$\Pi_{\text{<attribute list>}}(R)$$

OR

$$\text{PROJECT}_{\text{<attribute list>}}(R)$$

where the symbol **∏** (pi) is used to denote the PROJECT operator, and the <attribute list> is a list of attributes from the attributes of relation R. R could be just the name of a database relation or a relational algebra expression whose result is a relation.

The result of the PROJECT operation will be another relation having only the attributes specified in <attribute list> and in the same order as they appear in the list. Hence, its degree is equal to the number of attributes in <attribute list>. If the attribute list includes only non-key attributes of R, duplicates tuples are likely to occur and the PROJECT operation removes any duplicate tuples. If the projection list includes a super key of R, the resulting relation will have the same number of tuples as R. So, the number of tuples in the resulting relation of PROJECT operation is always less than or equal to the number of tuples in R. That is $|\Pi_a(R)| \leq |R|$. The PROJECT operation is not commutative.

Example: List each employee's first name, last name and salary.

$\Pi_{FNAME,\ LNAME,\ SALARY}$(EMPLOYEE)

c) **UNION Operation (U)**

This is a binary operation; that is the UNION operation is applied to two relations. The union operation results in a relation that includes all tuples that are in either (or both) of the two relations. The relations on which union operation is applied should be *union compatible*. Two relations are union compatible if they have the same arity (degree or number of attributes) and the domains of the corresponding attributes are identical. The UNION operation also removes the duplicate tuples from the resulting relation. The general syntax of the UNION operation is:

R=PUQ has tuples drawn from P & Q such that R={t|t € P or t € Q} and max(|P|,|Q|)**≤**|R|**≤** |P|+|Q|.

The UNION operation is commutative; that is PUQ = QUP.

Example: List the name & address of all students and faculty members. It can be written as:

$\Pi_{FNAME,ADDRESS}$(STUDENT) U $\Pi_{FNAME,ADDRESS}$(FACULTY).

d) **INTERSECTION Operation (∩)**

This is a binary operation; that is the INTERSECTION operation is applied to two relations. The intersection operation results in a relation that includes all tuples that are in both of the two relations. Ie. It selects common tuples in two relations. The relations on which intersection operation is applied should be *union compatible*. Two relations are union compatible if they have the same arity (degree or number of attributes) and the domains of the corresponding attributes are identical. The general syntax of the INTERSECTION operation is:

R=P∩Q such that R={t|t € P and t € Q} and 0≤|R|≤min(|P|,|Q|)

The INTERSECTION operation is commutative; that is P∩Q = Q∩P.

Example: List the names and address of all customers who have both a loan and an account. It can be written as: $\prod_{FNAME,ADDRESS}$(LOAN) ∩ $\prod_{FNAME,ADDRESS}$(ACCOUNT).

e) **SET DIFFERENCE Operation (-)**

This is a binary operation; that is the SET DIFFERENCE operation is applied to two relations. The SET DIFFERENCE operation results in a relation that includes all tuples that are in the first relation, but not in the second relation. The relations on which set difference operation is applied should be *union compatible*. Two relations are union compatible if they have the same arity (degree or number of attributes) and the domains of the corresponding attributes are identical. The SET DIFFERENCE operation is not commutative. The general syntax of the SET DIFFERENCE operation is:

R=P-Q such that R={t|t € P and t not € Q}and 0≤|R|≤|P|.

Example: List the names and address of all customers (account holders) who doesn't have a loan. It can be written as: $\prod_{FNAME,ADDRESS}$(ACCOUNT) - $\prod_{FNAME,ADDRESS}$(LOAN).

f) **CARTESIAN PRODUCT Operation (X)**

The CARTESIAN PRODUCT operation is used to obtain all possible combinations of tuples from two relations. This is a binary operation. In general, the result of P(A_1,A_2,… ..,A_n) X Q(B_1,B_2,… .,B_m) is a relation R with n+m attributes R(A_1,A_2,… ..,A_n, B_1,B_2,… .,B_m), in that order. The degree of resulting relation is n+m. The resulting relation R has one tuple for each combination of tuples – one from P and one from Q. So, the number of tuples in R will be number of tuples in P multiplied by number of tuples in Q; ie. |P|*|Q|. The general syntax of the CARTESIAN PRODUCT operation is:

R=PXQ such that {t1||t2|t1 € P and t2 € Q} and |R|=|P|*|Q|

g) **RENAME(ℓ) Operation**

The RENAME operation is used to rename either the relation name, or the attribute names, or both. The general RENAME operation when applied to a relation R of degree n is denoted by:

$\rho_{S(B1,B2,\ldots,Bn)}(R)$ or $\rho_{S}(R)$ or $\rho_{(B1,B2,\ldots,Bn)}(R)$

where the symbol ρ (rho) is used to denote the RENAME operator, S is the new relation name, and B1,B2,… Bn are the new attribute names. The first expression renames both the relation and its attributes; the second renames the relation only; and the third renames the attributes only. If the attributes of R are (A1,A2,… ..,An) in that order, then each Ai is renamed as Bi.

h) **JOIN Operation**

The JOIN operation, denoted by $\bowtie$, is used to combine related tuples from two relations into single tuples. In general, the result of JOIN operation of two relations $P(A_1,A_2,\ldots..,A_n)$ and $Q(B_1,B_2,\ldots.,B_m)$ is a relation R with n+m attributes $R(A_1,A_2,\ldots..,A_n, B_1,B_2,\ldots.,B_m)$, in that order. The join operation retrieves all tuples from the Cartesian product of the two relations ***with a matching condition***. The main difference between Cartesian product and JOIN operation is that the JOIN retrieves only combinations of tuples from two relations satisfying the join condition whereas Cartesian product retrieves all combinations of tuples. The join condition is specified on attributes from the two relations and is evaluated for each combination of tuples. The resulting relation R will have tuples between zero (no tuples matching the join condition) and |P|*|Q| (all tuples match the join condition). Ie. $0 \le |R| \le |P|*|Q|$.

The general syntax of the JOIN operation is:

$P \bowtie_{<join\ condition>} Q$

Example: Get the name of the manager of each department in the organization. List the department name and first & last names of department manager. This can be written as:

DEPT_MGR <- DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE

RESULT <- $\Pi_{DNAME,FNAME,LNAME}$(DEPT_MGR)

- **EQUIJOIN**: Join operations that involve join conditions with equality comparisons (=) only are called EQUIJOINs. This is the most common JOIN operation. The result of EQUIJOINs will have one or more pairs of attributes that have identical values in every tuple.

- **NATURAL JOIN (*)**: As explained above, the resulting relation of EQUIJOINs will always have one or more pairs of attributes with identical values. A new operation called NATURAL JOIN, denoted by *, is used to eliminate the duplicate superfluous attributes in an EQUIJOIN condition. The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, rename operation has to be applied first to convert the name of the join attribute of one relation same as that of the join attribute in second relation.

Example: Get a list of all employees including their department manager.

EMPLOYEE

Emp. Id	Name	Dept. Id
1001	George K John	01
1002	Prasad K	01
1003	Atul Kumar	01
1004	Harikrishnan	02
1005	Krishnakumar	02
1006	Ashok Shenoy	03
1007	Santhosh A	04
1008	Venu S Pillai	04
1009	Rahul Krishnan	
1010	Jose Prakash	04
1011	Varghese Paul	06

DEPARTMENT

Dept. Id	Dept. Name	Manager
01	Administration	George K John
02	Finance	Harikrishnan
03	Sales	Ashok Shenoy
04	Production	Santhosh A
05	Human Resource	Paul Joseph

EMPLOYEE * DEPARTMENT

Emp. Id	Name	Dept. Id	Dept. Name	Manager
1001	George K John	01	Administration	George K John
1002	Prasad K	01	Administration	George K John
1003	Atul Kumar	01	Administration	George K John
1004	Harikrishnan	02	Finance	Harikrishnan
1005	Krishnakumar	02	Finance	Harikrishnan
1006	Ashok Shenoy	03	Sales	Ashok Shenoy
1007	Santhosh A	04	Production	Santhosh A
1008	Venu S Pillai	04	Production	Santhosh A
1010	Jose Prakash	04	Production	Santhosh A

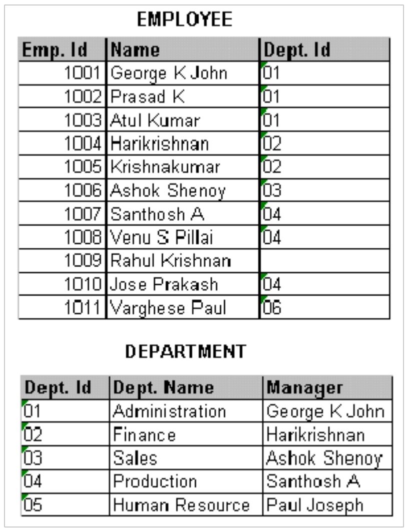
- **SEMI JOIN:** The Simi Join operation joins two relations and only keeps the attributes in the first relation. This is used to reduce the network traffic in a distributed environment. Only the joining attribute of one relation is sent to the other site where the second relation exists. A semijoin operation P $\ltimes_{A=B}$Q, where A and B are domain compatible attributes of P and Q respectively , produces the same result as the relational algebra expression Π_P(P $\bowtie_{A=B}$Q). Note that the semijoin operation is not commutative.

- **OUTER JOIN:** In the normal JOIN operation, only the tuples that matches the join condition are selected. Tuples without a matching (or related) tuple are eliminated from the JOIN result. Tuples with null in the JOIN attributes are also eliminated. A set of operations called OUTER JOINs provides the ability to include all the tuples including unmatched tuples in either or both relations in the result of the join. The outer join combines the unmatched tuple in one of the relation with an artificial tuple for the other relation; all attributes of artificial tuple set to NULL.

- **LEFT OUTER JOIN ($⟕$):** The LEFT OUTER JOIN is written as **P$⟕$Q** where P and Q are relations. The result of the left outer join is the set of all combinations of tuples in P and

Q that have matching values on their common attribute, in addition to tuples in P that have no matching tuples in Q.

Example: Get a list of all employees including their department manager.

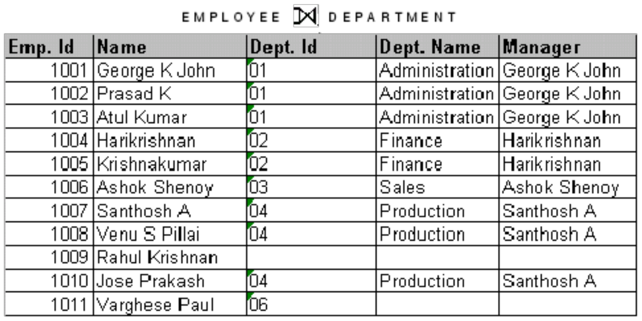


EMPLOYEE

Emp. Id	Name	Dept. Id
1001	George K John	01
1002	Prasad K	01
1003	Atul Kumar	01
1004	Harikrishnan	02
1005	Krishnakumar	02
1006	Ashok Shenoy	03
1007	Santhosh A	04
1008	Venu S Pillai	04
1009	Rahul Krishnan	
1010	Jose Prakash	04
1011	Varghese Paul	06

DEPARTMENT

Dept. Id	Dept. Name	Manager
01	Administration	George K John
02	Finance	Harikrishnan
03	Sales	Ashok Shenoy
04	Production	Santhosh A
05	Human Resource	Paul Joseph

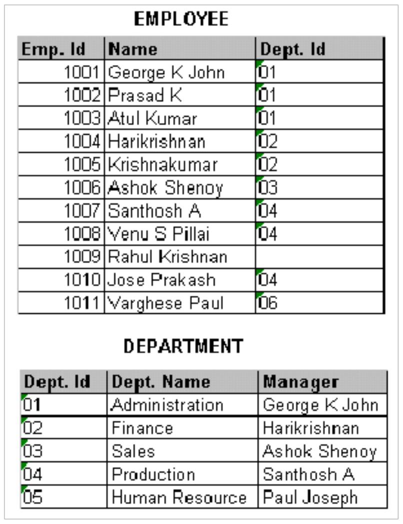


EMPLOYEE ⟕ DEPARTMENT

Emp. Id	Name	Dept. Id	Dept. Name	Manager
1001	George K John	01	Administration	George K John
1002	Prasad K	01	Administration	George K John
1003	Atul Kumar	01	Administration	George K John
1004	Harikrishnan	02	Finance	Harikrishnan
1005	Krishnakumar	02	Finance	Harikrishnan
1006	Ashok Shenoy	03	Sales	Ashok Shenoy
1007	Santhosh A	04	Production	Santhosh A
1008	Venu S Pillai	04	Production	Santhosh A
1009	Rahul Krishnan			
1010	Jose Prakash	04	Production	Santhosh A
1011	Varghese Paul	06		

- **RIGHT OUTER JOIN (⟖):** The RIGHT OUTER JOIN is written as **P⟖Q** where P and Q are relations. The result of the right outer join is the set of all combinations of tuples in P and Q that have matching values on their common attribute, in addition to tuples in Q that have no matching tuples in P.

Example: Get a list of all departments & managers including all employees under each department.

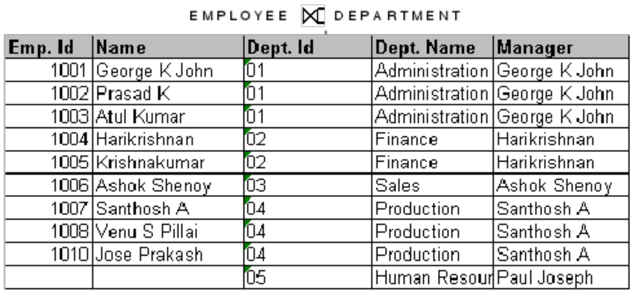


EMPLOYEE

Emp. Id	Name	Dept. Id
1001	George K John	01
1002	Prasad K	01
1003	Atul Kumar	01
1004	Harikrishnan	02
1005	Krishnakumar	02
1006	Ashok Shenoy	03
1007	Santhosh A	04
1008	Venu S Pillai	04
1009	Rahul Krishnan	
1010	Jose Prakash	04
1011	Varghese Paul	06

DEPARTMENT

Dept. Id	Dept. Name	Manager
01	Administration	George K John
02	Finance	Harikrishnan
03	Sales	Ashok Shenoy
04	Production	Santhosh A
05	Human Resource	Paul Joseph

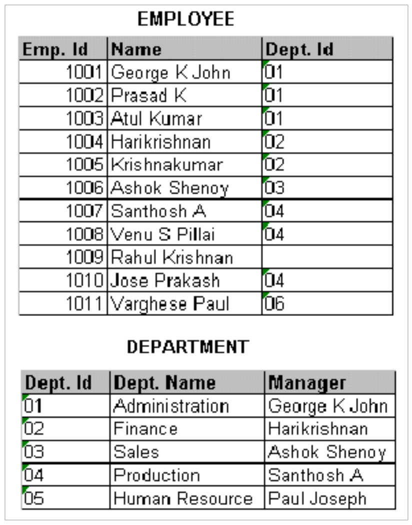


EMPLOYEE ⟖ DEPARTMENT

Emp. Id	Name	Dept. Id	Dept. Name	Manager
1001	George K John	01	Administration	George K John
1002	Prasad K	01	Administration	George K John
1003	Atul Kumar	01	Administration	George K John
1004	Harikrishnan	02	Finance	Harikrishnan
1005	Krishnakumar	02	Finance	Harikrishnan
1006	Ashok Shenoy	03	Sales	Ashok Shenoy
1007	Santhosh A	04	Production	Santhosh A
1008	Venu S Pillai	04	Production	Santhosh A
1010	Jose Prakash	04	Production	Santhosh A
		05	Human Resour	Paul Joseph

- **FULL OUTER JOIN (⟗):** The FULL OUTER JOIN in effect combines the results of the left and right outer joins. The full outer join is written as **$P ⟗ Q$** where P and Q are relations. The result of the full outer join is the set of all combinations of tuples in P and Q that have matching values on their common attribute, in addition to tuples in P that have no matching tuples in Q and tuples in Q that have no matching tuples in P.

Example: Get a list of all departments, employees & managers.

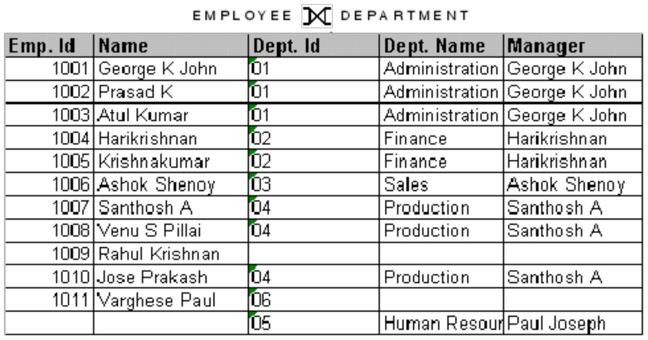


EMPLOYEE

Emp. Id	Name	Dept. Id
1001	George K John	01
1002	Prasad K	01
1003	Atul Kumar	01
1004	Harikrishnan	02
1005	Krishnakumar	02
1006	Ashok Shenoy	03
1007	Santhosh A	04
1008	Venu S Pillai	04
1009	Rahul Krishnan	
1010	Jose Prakash	04
1011	Varghese Paul	06

DEPARTMENT

Dept. Id	Dept. Name	Manager
01	Administration	George K John
02	Finance	Harikrishnan
03	Sales	Ashok Shenoy
04	Production	Santhosh A
05	Human Resource	Paul Joseph



EMPLOYEE $⟗$ DEPARTMENT

Emp. Id	Name	Dept. Id	Dept. Name	Manager
1001	George K John	01	Administration	George K John
1002	Prasad K	01	Administration	George K John
1003	Atul Kumar	01	Administration	George K John
1004	Harikrishnan	02	Finance	Harikrishnan
1005	Krishnakumar	02	Finance	Harikrishnan
1006	Ashok Shenoy	03	Sales	Ashok Shenoy
1007	Santhosh A	04	Production	Santhosh A
1008	Venu S Pillai	04	Production	Santhosh A
1009	Rahul Krishnan			
1010	Jose Prakash	04	Production	Santhosh A
1011	Varghese Paul	06		
		05	Human Resour	Paul Joseph

i) **Division Operation (÷)**

The division is a binary operation. The DIVISION operation is useful for a special kind of query that sometimes occurs in database applications. An example is, “Retrieve the names of employees who work on ***all*** the projects that Mr. George K John works on”. The DIVISION operation is written as **P÷Q** where P and Q are relations such that attributes of P are a superset of attributes of Q. The result of DIVISION operation will be a relation with attributes unique to P; that is attributes in P which are not in Q. The tuples in the resulting relation of a DIVISION operation are those tuples in P (excluding the attributes of Q) where there are tuples in combination with every tuple in Q.

Example: Retrieve the names of employees who work on ***all*** the projects that Mr. George K John works on.

PROJECT*EMPLOYEE
(relation 1)

Project Code	Emp. Id	Name
BA-100	1001	George K John
BA-100	1003	Atul Kumar
BA-100	1004	Harikrishnan
BA-100	1010	Jose Prakash
BA-100	1005	Krishnakumar
BA-101	1001	George K John
BA-101	1011	Varghese Paul
BA-101	1005	Krishnakumar
BA-102	1006	Ashok Shenoy
BA-102	1005	Krishnakumar
BA-105	1002	Prasad K
BA-105	1004	Harikrishnan
BA-107	1007	Santhosh A
BA-108	1008	Venu S Pillai
BA-109	1009	Rahul Krishnan

PROJECTS OF George K John
(relation 2)

Project Code
BA-100
BA-101

relation 1 ÷ relation 2

Emp. Id	Name
1001	George K John
1005	Krishnakumar

Let r(R) and s(S) be relations. Let $S \subseteq R$. The relation r ÷ s is a relation on schema R – S. A tuple t is in r ÷ s if for every tuple t_s in s there is a tuple t_r in r satisfying both of the following:

$$t_r[S] = t_s[S]$$
$$t_r[R-S] = t[R-S]$$

These conditions say that the $R - S$ portion of a tuple t is in $r \div s$ if and only if there are tuples with the $r - s$ portion **and** the S portion in r for **every** value of the S portion in relation S.

j) **Assignment Operation (⟵)**

It works like assignment in a programming language.

Example: Assign the result of a PROJECT operation to a relation RESULT.

RESULT <- $\Pi_{DNAME,FNAME,LNAME}$(DEPT_MGR)

2.3 Relational Calculus

Relatioal Calculus simply means 'calculating with relations'. Relational calculus is a query system wherein queries are expressed as variables and formulas on these variables. In relational calculus, a query is expressed as a formula consisting of a number of variables and an expression involving these variables. Formula describes the properties of the result relation to be obtained. If the variables represent the tuples from the specified relations, then the query or formula is referred as 'tuple relational calculus'. If the variables represent values drawn from specified domains, then the query or formula is referred as 'domain relational calculus'. The relational calculus is based on **predicate** (stands for the property) **calculus** which uses the following primitive symbols:

- **$\wedge$ - AND**
- **$\vee$ - OR**
- **$\neg$ - Negation**
- **$\rightarrow$ - Implication**
- **$\equiv$ - Equivallence**
- **$\forall$ - Universal quantifier (For All)**
- **$\exists$ - Existential quantifier (For Some)**

Laws in Predicate calculus as follows:

$\neg(\neg \mathbf{P}) \equiv \mathbf{P}$

$(\mathbf{P} \vee \mathbf{Q}) \equiv (\neg \mathbf{P} \rightarrow \mathbf{Q})$

the contrapositive law: $(\mathbf{P} \rightarrow \mathbf{Q}) \equiv (\neg \mathbf{Q} \rightarrow \neg \mathbf{P})$

de Morgan's law: $\neg(\mathbf{P} \vee \mathbf{Q}) \equiv (\neg \mathbf{P} \wedge \neg \mathbf{Q})$ and $\neg(\mathbf{P} \wedge \mathbf{Q}) \equiv (\neg \mathbf{P} \vee \neg \mathbf{Q})$

the commutative laws: $(\mathbf{P} \wedge \mathbf{Q}) \equiv (\mathbf{Q} \wedge \mathbf{P})$ and $(\mathbf{P} \vee \mathbf{Q}) \equiv (\mathbf{Q} \vee \mathbf{P})$

the associative law: $((\mathbf{P} \wedge \mathbf{Q}) \wedge \mathbf{R}) \equiv (\mathbf{P} \wedge (\mathbf{Q} \wedge \mathbf{R}))$

the associative law: $((\mathbf{P} \vee \mathbf{Q}) \vee \mathbf{R}) \equiv (\mathbf{P} \vee (\mathbf{Q} \vee \mathbf{R}))$

the distributive law: $\mathbf{P} \vee (\mathbf{Q} \wedge \mathbf{R}) \equiv (\mathbf{P} \vee \mathbf{Q}) \wedge (\mathbf{P} \vee \mathbf{R})$

the distributive law: $\mathbf{P} \wedge (\mathbf{Q} \vee \mathbf{R}) \equiv (\mathbf{P} \wedge \mathbf{Q}) \vee (\mathbf{P} \wedge \mathbf{R})$

Relational Calculus are of two types

a) Tuple Relational calculus
b) Domain Relational calculus

2.3.1 Tuple Relational Calculus

The tuple calculus is a calculus that was introduced by Edgar F. Codd. Queries in tuple relational calculus are expressed by tuple calculus expresson **{t|P(x)}** where p is a formula involving x and x represent a set of tuple variables. The expression characterizes a set of tuple of x such that the formula P(x) is true.

A tuple variable is said to be a **free variable** unless it is quantified by a $\exists$ or a $\forall$. If it is quantified by a ∃ or a ∀, it is said to be **bound variable**.

A formula is built of **atoms**. An atom is one of the following forms:

- o s ϵ r , where s is a tuple variable, and r is a relation ($\notin$ is not allowed).
- o s[x] θ u[y] where s and u are tuple variables, and x and y are attributes, and θ is a comparison operator ($<, \leq, =, \neq, >, \geq$).
- o s[x] θ c, where c is a constant in the domain of attribute x.

Examples:

1. Find the customers whose loan amount is greater than 1200.

$$\{t \mid \exists s \in borrow\ (t[cname] = s[cname] \ \wedge \ s[amount] \ > \ 1200)\}$$

In English, we may read this equation as "the set of all tuples t such that there exists a tuple s in the relation borrow for which the values of t and s for the cname attribute are equal, and the value of s for the amount attribute is greater than 1200."

2. Find all customers having a loan from the SFU branch, and the cities in which they live:

$$\{t \mid \exists s \in borrow(t[cname] = s[cname] \ \wedge \ s[bname] = \text{"SFU"} \\ \wedge \exists u \in customer(u[cname] = s[cname] \ \wedge \ t[ccity] = u[ccity]))\}$$

2.3.2 Domain Relational Calculus

The **domain relational calculus** (**DRC**) is a calculus that was introduced by Edgar F. Codd as a declarative database query language for the relational data model. This language uses the same operators as tuple calculus; Logical operators Λ (and), V(or) and ¬ (not). The existential quantifier (∃) and the universal quantifier (∀) can be used to bind the variables.

<u>Formal Definition</u>

An expression is of the form

$$\{< x_1, x_2, \ldots, x_n > \ \mid P(x_1, x_2, \ldots, x_n)\}$$

where the $x_i, 1 \leq i \leq n,$ represent domain variables, and P is a **formula**.

An atom in the domain relational calculus is of the following forms :

- o <x_1, x_2, … ., x_n> ϵ r where r is a relation on n attributes, and x_i, 1 ≤ i ≤ n, are domain variables or constants.
- o x θ y , where x and y are domain variables, and θ is a comparison operator.
- o x θ c , where c is a constant.

Formulae are built up from atoms using the following rules:

- o An atom is a formula.
- o If P is a formula, then so are $\neg P$and (P).
- o If $P_1$and P_2 are formulae, then so are $P_1 \wedge P_2$, $P_1 \vee P_2$, $P_1 \wedge P_2$ and $P_1 \Rightarrow P_2$.
- o If P(s) is a formula containing a free tuple variable s, then

$$\exists s \in r(P(s)) \text{ and } \forall s \in r(P(s))$$

are also formulae.

Examples:

1. Find branch name, loan number, customer name and amount for loans of over $1200.

$$\{< b, l, c, a > \mid < b, l, c, a > \in borrow \wedge a > 1200\}$$

2. Find all customers who have a loan for an amount greater than $1200.

$$\{< c > \mid \exists b, l, a(< b, l, c, a > \in borrow \wedge a > 1200)\}$$

3. Find all customers having a loan from the SFU branch, and the city in which they live.

$$\{< c, x > \mid \exists b, l, a(< b, l, c, a > \in borrow \wedge b = \text{“SFU”} \wedge \exists y(< c, y, x > \in customer))\}$$

4. Find all customers having a loan, an account or both at the SFU branch.

$$\{< c > \mid \exists b, l, a(< b, l, c, a > \in borrow \wedge b = \text{“SFU”}) \vee \exists b, a, n(< b, a, c, n > \in deposit \wedge b = \text{“SFU”})\}$$

5. Find all customers who have an account at **all** branches located in Brooklyn.

$$\{< c > \mid \forall x, y, z(\neg(< x, y, z > \in branch) \vee z \neq \text{“Brooklyn”} \vee (\exists a, n(< x, a, c, n > \in deposit)))\}$$

2.4 Introduction to SQL

SQL (Structured Query Language) is a database sublanguage for querying and modifying relational databases. It was developed by IBM Research in the mid 70's and standardized by ANSI in 1986. In the relational model, data is stored in structures called relations or *tables*. Each table has one or more attributes or *columns* that describe the table. In relational databases, the table is the fundamental building block of a database application.

SQL language consists of two categories of statements:

- Data Definition Language (DDL) used to create, alter and drop schema objects such as tables and indexes, and
- Data Manipulation Language (DML) used to manipulate the data within those schema objects.

Oracle's SQL*Plus is a command line tool that allows a user to type SQL statements to be executed directly against an Oracle database. SQL*Plus commands allow a user to do the following activities:

- Enter, edit, store, retrieve, and run SQL statements
- List the column definitions for any table
- Format, perform calculations on, store, and print query results in the form of reports
- Access and copy data between SQL databases

Oracle Database Objects

- TABLESPACE - A Tablespace is the physical space (in fact, a physical data file in the hard disk) where tables are stored.
- TABLE - Consists of ROWS of data separated into different COLUMNS.
- VIEW – A virtual relation for specific user group.
- INDEX – A search index created for a table to improve query performance.
- TRIGGER – A sub program stored in the database that gets executed automatically on a data manipulation event on a table.
- SEQUENCE – A database object for creating sequential numbers for use as a primary key, for example.
- PROCEDURE / FUNCTION – A sub program stored in the database for a specific application use.
- PACKAGE – A collection of stored sub programs (procedures or functions) in the database.
- USER – Stores the user name of the current session.
- SYSDATE – Stores the current system date and time.

Data Types in SQL Statements

a) CHAR(n) - Character string, fixed length n

b) VARCHAR2(n) - Variable length character string, maximum length n.

c) NUMBER (p) - Integer numbers with precision p.

d) BOOLEAN - Boolean variable used to store TRUE, FALSE, NULL

e) DATE - Stores year, month, day, hour, minute and second values

f) BLOB – Binary Large OBject (the data type in Oracle for storing binary files like executables, images etc.)

2.4.1 DDL Statements

The DDL statements are used to create and alter the database objects. Common DDL statements are:

- CREATE TABLE
- ALTER TABLE
- CREATE INDEX
- DROP TABLE
- DROP INDEX
- CREATE USER
- ALTER USER
- CREATE SEQUENCE
- CREATE TRIGGER
- CREATE PROCEDURE
- CREATE PACKAGE

1) CREATE TABLE

The CREATE TABLE Statement creates a new database table.

Syntax:

CREATE TABLE <table name> **(column1 data type (size), column2 data type (size), column3 data type (size),… … …);**

Constraints

Constraint specifications add additional restrictions on the contents of the table. They are automatically *enforced* by the DBMS.

a) **Primary Key**: Specifies the columns that uniquely identify a row in a table. One or more columns could be part of a primary key.

Syntax:

CREATE TABLE <table name> **(column1 data type (size) PRIMARY KEY, column2 data type (size), column3 data type (size),… … …);**

b) **NOT NULL**: Specifies that the column can't be set to *null*. Normally, primary key columns are declared as NOT NULL.

Syntax:

CREATE TABLE <table name> **(column1 data type (size) NOT NULL, column2 data type (size), column3 data type (size),… … …);**

c) **UNIQUE**: Specifies that this column has a unique value or *null* for all rows of the table.

Syntax:

CREATE TABLE <table name> **(column1 data type (size) UNIQUE, column2 data type (size), column3 data type (size),… … …);**

d) Foreign Key: Specifies the set of columns in a table that is used to establish the relationship with another table in the database.

Syntax:

FOREIGN KEY (column-list) REFERENCES referenced-table (column-list)

e) CHECK: specifies a user defined constraint, known as a *check condition*. The CHECK specifier is followed by a condition enclosed in parentheses.

Syntax:

CREATE Table <table name> **(column1 data type (size) CHECK (Condition), column2 data type (size), column3 data type (size),… … …);**

2) ALTER TABLE

The ALTER TABLE statement is used to change the table definition.

- To add new column

ALTER TABLE <table name> **ADD column name datatype (size);**

- To add Primary key

ALTER TABLE <table name> **ADD Column name PRIMARY KEY;**

- To add Foreign key

ALTER TABLE <table name> **ADD CONSTRAINT** <constraint name> **FOREIGN KEY (column name) REFERENCES <table name> (column name);**

- To remove a Foreign key constraint

ALTER TABLE <table name> **DROP CONSTRAINT** <constraint name>

- To rename a table

ALTER TABLE <*table name*> **RENAME TO <**new table name>

3) CREATE INDEX

The CREATE INDEX statement allows the creation of an index for an already existing relation or table.

Syntax:

CREATE [unique] Index <name of index>

On <existing table name> **(**column name **[asc**ending or **desc**ending**]**

[,column name**[asc** or **desc]) [cluster]**

4) DROP TABLE

The DROP TABLE statement removes a previously created table and its description from the catalog.

Syntax:

DROP TABLE <table-name>

5) DROP INDEX

The DROP INDEX statement removes a previously created index from the database.

Syntax:

DROP INDEX <index-name>

6) CREATE USER

The CREATE USER statement creates a new database user.

Syntax:

CREATE USER <user name> **IDENTIFIED BY** <password>

7) ALTER USER

The ALTER USER statement is used to change the password of a database user.

Syntax:

ALTER USER <user name> **IDENTIFIED BY** <new password>

8) CREATE SEQUENCE

The CREATE SEQUENCE statement is used to create a sequence to be used in DML statements.

Syntax:

```
CREATE SEQUENCE <sequence name>
 START WITH <start sequence number>
 MAXVALUE <maximum value of sequence number>
 INCREMENT BY   <increment value>
 NOCYCLE;
```

2.4.2 DML Statements

The DML statements are used to manipulate (store and maintain) the data in database tables. Common DML statements are:

- INSERT
- UPDATE

- o DELETE
- o SELECT

DML statements use arithmetic and logical operators. Arithmetic and Logical operators used in DML statements are given below:

Operators in SQL

- Arithmetic operators

+,-,*,/

- Logical operators

=,!= or <>,>,>=,<,<=

- Relational Operators

AND, OR, NOT

- Other Operators

a) LIKE - The LIKE operator implements a pattern match comparison, that is, it matches a string value against a pattern string containing wild-card characters.

The wild-card characters for LIKE are percent -- '%' and underscore -- '_'. Underscore matches any *single* character. Percent matches zero or more characters.

Example: **SELECT** * **FROM** <table name> **WHERE** <column name> **LIKE** 'abc%'

b) NOT LIKE - It does not matches a string value against a pattern string containing wild-card characters.

Example: **SELECT** * **FROM** <table name> **WHERE** <column name> **NOT LIKE** 'abc%'

c) BETWEEN - The BETWEEN operator implements a range comparison, that is, it tests whether a value is *between* two other values.

Example: **SELECT * FROM** <table name> **WHERE** qty **BETWEEN** 50 and 500

d) **NOT BETWEEN**- It tests whether a value is not *between* two other values.

Example: **SELECT** * **FROM** <table name> **WHERE** qty **NOT BETWEEN** 50 and 500

e) **IN** - The IN operator implements comparison to a list of values, that is, it tests whether a value matches any value in a list of values

Example: **SELECT** * **FROM** <table name> **WHERE** city **IN** ('Rome','Paris')

f) **NOT IN** - It tests whether a value does not matches any value in a list of values

Example: **SELECT** * **FROM** <table name> **WHERE** city **NOT IN** ('Rome','Paris')

1. INSERT

Syntax:

INSERT INTO <table name> VALUES (value-1, value-2… … .)

Examples:

- INSERT statement with direct values

  INSERT INTO EMPLOYEE (EMPLOYEE_ID, FIRST_NAME, DATE_OF_BIRTH, SEX,SALARY)
  VALUES (1090,'Anitha','10-OCT-1980','F',5000)

- INSERT statement with values from another or same table rows

  INSERT INTO EMPLOYEE (EMPLOYEE_ID, FIRST_NAME, DATE_OF_BIRTH, SEX, SALARY)
  SELECT 1091, FIRST_NAME, DATE_OF_BIRTH, SEX, SALARY FROM EMPLOYEE
  WHERE EMPLOYEE_ID = 1090

2. UPDATE

Updating allows us to change some values in a tuple without necessarily changing all. **where** clause of **update** statement may contain any construct legal in a **where** clause of a **select** statement (including nesting). A nested **select** within an update may reference the relation that is being updated. As before, all tuples in the relation are first tested to see whether they should be updated, and the updates are carried out afterwards.

Example: Update the date of birth of employee 1090 to 15-Oct-1979.

UPDATE EMPLOYEE
SET DATE_OF_BIRTH = '15-OCT-1979'
WHERE EMPLOYEE_ID = 1090

Note: WHERE clause can take different forms as listed in SELECT… above, including SUB QUERY.

3. DELETE

- Delete Specific rows

  DELETE FROM <table name> **[WHERE** predicate**]**

- Delete all rows

  DELETE * **FROM** <table name>

Example: Delete the record of the employee with employee id 1091.

DELETE FROM EMPLOYEE WHERE EMPLOYEE_ID = 1091

Note: WHERE clause can take different forms as listed in SELECT… above, including SUB QUERY.

4. SELECT

The SQL SELECT statement queries data from tables in the database. The statement begins with the SELECT keyword. The basic SELECT statement has 3 clauses:

- SELECT - specifies the table columns retrieved
- FROM - specifies the tables to be accessed
- WHERE - specifies which rows in the FROM tables to use

The WHERE clause could be followed by other clauses as listed below:

ORDER BY Clause

This clause is for retrieving rows in the sorted order.

Example: SELECT a from <t> where a LIKE 's%' ORDER BY a;

Group by Clause

This clause group rows based on distinct values that exist for specified columns.

Having Clause

This is used in conjunction with Group By clause. Filter the groups created by the group by clause.

Example: Write a SQL statement to get the list of departments having average salary less than the average salary of the company. Include Department ID and Average Salary of Dept. in the list. Exclude departments with less than 3 employees and order the list in the descending order of the department average salary.

```
SELECT DEPARTMENT_ID, AVG(SALARY) FROM EMPLOYEE
GROUP BY DEPARTMENT_ID
HAVING (AVG(SALARY) < (SELECT AVG(SALARY) FROM EMPLOYEE) AND COUNT(*) >= 3)
ORDER BY 2 DESC
```

Syntax of SELECT Statements:

SELECT <field names> FROM <table name> WHERE (condition);

To select all rows use the following syntax

SELECT * FROM <table name>

To select unique rows from the table

SELECT DISTINCT select-list

DISTINCT specifies that duplicate rows are discarded.

Examples:

- Statement to return all columns and rows from EMPLOYEE table

SELECT * FROM EMPLOYEE

- Statement to return EMPLOYEE_ID and LAST_NAME columns from EMPLOYEE table

  SELECT EMPLOYEE_ID, LAST_NAME FROM EMPLOYEE
- Statement to get only distinct employee details

  SELECT DISTINCT FIRST_NAME, LAST_NAME, DATE_OF_BIRTH FROM EMPLOYEE
- Statement to get details of employee with employee id = 1023

  SELECT FIRST_NAME, LAST_NAME, DATE_OF_BIRTH FROM EMPLOYEE WHERE EMPLOYEE_ID = 1023
- Statement to get id and last name of all employees in department 1

  SELECT EMPLOYEE_ID, LAST_NAME FROM EMPLOYEE WHERE DEPARTMENT_ID = 1
- Statement to get details of all employees whose first name start with letter A

  SELECT FIRST_NAME, LAST_NAME, DATE_OF_BIRTH FROM EMPLOYEE WHERE FIRST_NAME LIKE 'A% '
- Statement to get details of all employees with employee ids between 1030 and 1050

  SELECT EMPLOYEE_ID, LAST_NAME FROM EMPLOYEE WHERE EMPLOYEE_ID BETWEEN 1030 AND 1050
- Statement to get details of all employees with employee id greater than 1040

  SELECT EMPLOYEE_ID, LAST_NAME FROM EMPLOYEE WHERE EMPLOYEE_ID > 1040
- Statement to get the total number of employees

  SELECT COUNT (*) FROM EMPLOYEE
- Statement to get the number of employees in dept. 1

  SELECT COUNT (EMPLOYEE_ID) FROM EMPLOYEE
  WHERE DEPARTMENT_ID = 1
- Statement to get the lowest (first) employee number

  SELECT MIN(EMPLOYEE_ID) FROM EMPLOYEE
- Statement to get the highest (last) employee number

  SELECT MAX(EMPLOYEE_ID) FROM EMPLOYEE
- **SUB QUERY**
- Statement to get details of employees belonging to department 'ACCOUNTS'

  SELECT EMPLOYEE_ID, LAST_NAME FROM EMPLOYEE WHERE DEPARTMENT_ID = (SELECT DEPARTMENT_ID FROM DEPARTMENT WHERE DEPARTMENT_NAME = 'ACCOUNTS')
- Statement to get details of employees belonging to departments whose manager is ANIL KUMAR

  SELECT EMPLOYEE_ID, LAST_NAME

FROM EMPLOYEE WHERE DEPARTMENT_ID **IN** (SELECT DEPARTMENT_ID FROM DEPARTMENT WHERE DEPARTMENT_MANAGER = 'ANIL KUMAR')

- Statement to get the list of employees earning more than 20000 if atleast one employee gets less than 15000. This is called existence test.
  SELECT EMPLOYEE_ID, LAST_NAME, SALARY
  FROM EMPLOYEE WHERE SALARY > 20000 AND
  EXISTS (SELECT * FROM EMPLOYEE WHERE SALARY < 15000)
- Statement to get the list of employees sorted in ascending order of their last names.
  SELECT EMPLOYEE_ID, LAST_NAME FROM EMPLOYEE ORDER BY LAST_NAME
- Statement to get the list of employees sorted in the descending order of their salary
  SELECT EMPLOYEE_ID, LAST_NAME, SALARY FROM EMPLOYEE
  ORDER BY SALARY DESC
- Statement to get the list of employees sorted in the ascending order of their salary
  SELECT EMPLOYEE_ID, LAST_NAME, SALARY FROM EMPLOYEE
  ORDER BY 3 ASC
- Statement to get the minimum, maximum, average, sum and number of employees of departments 1 to 3 (these are called group functions)
  SELECT DEPARTMENT_ID, MIN(SALARY), MAX(SALARY), AVG(SALARY),
SUM(SALARY), COUNT(*)
  FROM EMPLOYEE WHERE DEPARTMENT_ID BETWEEN 1 AND 3
  GROUP BY DEPARTMENT_ID
  ORDER BY DEPARTMENT_ID
- Statement to get the minimum and maximum salary of departments with difference of min and max salary is greater than 5000
  SELECT DEPARTMENT_ID, MIN(SALARY), MAX(SALARY)
  FROM EMPLOYEE GROUP BY DEPARTMENT_ID
  HAVING MAX(SALARY) > (MIN(SALARY) + 5000)
  ORDER BY DEPARTMENT_ID
- **Joining Two Tables**
- Statement to get the list of employees with their department names
  SELECT EMPLOYEE.EMPLOYEE_ID, EMPLOYEE.LAST_NAME,
DEPARTMENT.DEPARTMENT_NAME
  FROM EMPLOYEE, DEPARTMENT
  WHERE DEPARTMENT.DEPARTMENT_ID = EMPLOYEE.DEPARTMENT_ID
- **Joining a Table to itself**
- Statement to get the list of employees with their supervisor names
  SELECT A.EMPLOYEE_ID, A.LAST_NAME, B.LAST_NAME

FROM EMPLOYEE A, EMPLOYEE B

WHERE B.EMPLOYEE_ID = A.SUPERVISOR_ID

- **Outer Join**
- Normal joins returns rows only if there are matching rows in both the joining tables. Outer joins returns the rows even if there are no matching rows in the joining table (with the column from the joining table as Null). Given below is a statement to get the list of employees with their department names, even if the department id is not found in department table

SELECT EMPLOYEE.EMPLOYEE_ID, EMPLOYEE.LAST_NAME, DEPARTMENT.DEPARTMENT_NAME

FROM EMPLOYEE, DEPARTMENT

WHERE DEPARTMENT.DEPARTMENT_ID (+) = EMPLOYEE.DEPARTMENT_ID

Group/Aggregate functions

- SUM () gives the total of all the rows, satisfying any conditions, of the given column, where the given column is numeric.
- AVG () gives the average of the given column.
- MAX () gives the largest figure in the given column.
- MIN () gives the smallest figure in the given column.
- COUNT() gives the number of rows satisfying the conditions.
- COUNT(*) gives the number of rows in the table.

Other SQL Functions

- ABS() gives the absolute value of the given value
- SQRT() gives the square root of the given value
- POWER(m,n) gives the result m^n
- ROUND(n,m) n rounded to m places.
- UPPER - Converts to upper case
- LOWER – Converts to lower case
- SUBSTR – Returns a sub string from the source string
- INSTR – Searches for a string in another string.
- NVL – Checks for NULL value
- TRIM, LTRIM, RTRIM – Trims the given string
- LPAD, RPAD – Pads the character string with specified number of characters.
- TO_NUMBER, TO_CHAR, TO_DATE – Conversion functions

- ADD_MONTHS – Adds given number of months to a date
- MONTHS_BETWEEN – Gives number of months between two dates.
- LAST_DAY – Returns the last date of the month of a given date
- DECODE – Very powerful function for decoding.

Note: All the above functions can be used in SQL statements (both on SELECT columns and in WHERE clause).

UNION

The UNION command is used to select related information from two tables. When using the UNION command all selected columns need to be of the same data type. With UNION, only distinct values are selected.

SQL Statement 1
UNION
SQL Statement 2

- Statement to get list of all female employees or employees in department 1 without any duplicates

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME        FROM    EMPLOYEE
WHERE SEX = 'F'
UNION
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME        FROM    EMPLOYEE
WHERE DEPARTMENT_ID = 1
```

UNION ALL

The UNION ALL command is equal to the UNION command, except that UNION ALL selects all values.

- Statement to get list of all female employees or employees in department 1 with duplicates allowed

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME    FROM  EMPLOYEE
WHERE SEX = 'F'
UNION ALL
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME    FROM  EMPLOYEE WHERE
DEPARTMENT_ID = 1
```

INTERSECTION

The INTERSECT command is used to find the intersection of two tables.

SQL Statement 1
INTERSECT

SQL Statement 2

- Statement to get list of female employees in department 1

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME          FROM     EMPLOYEE
WHERE SEX = ‘F’
INTERSECT
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME          FROM     EMPLOYEE
WHERE DEPARTMENT_ID = 1
```

INTERSECT ALL

The INTERSECT ALL command is equal to the INTERSECT command, except that INTERSECT ALL selects all values.

MINUS

The MINUS command returns rows from first select statement minus the rows from second select statement.

- Statement to get list of female employees excluding those belong to department 1

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME     FROM   EMPLOYEE WHERE
SEX = ‘F’
MINUS
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME
FROM EMPLOYEE WHERE DEPARTMENT_ID = 1
```

Tips on Efficient SQL Coding

- Avoid data conversion in the WHERE clause on indexed columns.
- Avoid using SQL functions on the indexed columns used in the WHERE clause.
- BETWEEN is more efficient than using >= and <= operators.
- Avoid using % at the beginning of a search string in LIKE predicate.
- Avoid doing arithmetic within a WHERE clause.
- Avoid using the same column for comparison in the WHERE clause more than once.
- When joining tables, put the indexed columns on the right side of join.
- Make sure that the high order index column is used in the WHERE clause in case of a multi column indexed table search.
- When using ORDER BY or GROUP BY, choose columns that have an index defined on them.
- Avoid using literals or variables with greater size or precision for comparison in the WHERE clause.
- If possible, avoid using combination of IN, LIKE or OR predicates in the WHERE clause.

- When LIKE or IN predicates are used in a WHERE clause along with ORDER BY or GROUP BY clauses, a sort of the entire table will occur first before the WHERE clause is executed. This can be avoided by dynamically creating a temporary table, loading with selected rows and then applying ORDER BY or GROUP BY clause to the temporary table.

2.4.3 Views

The DML statements given above help the users to manipulate the data stored in the 'conceptual' relations in the database. Such conceptual relations are sometimes referred to as base relations. For security and other concerns, sometimes it may be necessary to prevent direct access to the base relations by users. It is possible to create 'virtual relations' from base relations for different groups of users, exposing only the relevant data elements to different groups of users. This is possible in SQL by creating views. A relation in a view is virtual since no corresponding physical relation exists. A view represents a different perspective of a base relation or relations. The general syntax of create view statement is:

Create view <view name> **as** <query expression>

Views are not stored, since the data in the base relations may change. But, the definition of a view in a create view statement is stored in the system catalog. Whenever, such a virtual relation defined by a view is referred by a query, the view is recomputed to refresh the tuples in the virtual relation.

Example: Pay_Rate of an employee is confidential in nature and hence not all users are permitted to see the Pay_Rate of an employee. So, DBA can create a view EMP_VIEW to prevent viewing Pay_Rate by normal users and give access to this view rather than giving direct access to the EMPLOYEE table. EMP_VIEW can be defined as:

Create view EMP_VIEW **as** (**select** Emp_Id, Name, Skill **from** EMPLOYEE)

A view can be deleted from the database by using drop view statement. Syntax of drop view statement is: **Drop view** <view name>

The tuples in a view can be updated only under certain conditions. When the tuple in a view doesn't map to a single tuple in the base relation, the update operation may not be possible. Views that involve a join may not be updatable, if they do not include the primary keys of the base relations.