Program : **B.Tech**

Subject Name: **Database Management Systems**

Subject Code: **CS-502**

Semester: **5th**

Subject Notes (DBMS CS 502)

UNIT-2:

Relational Data models:

Domain: A (usually named) set/universe of *atomic* values, where by "atomic" we mean simply that, from the point of view of the database, each value in the domain is indivisible (i.e., cannot be broken down into component parts).

Examples of domains

- SSN: string of digits of length nine
- Name: string of characters beginning with an upper-case letter
- GPA: a real number between 0.0 and 4.0
- Sex: a member of the set {female, male}
- Dept_Code: a member of the set {CMPS, MATH, ENGL, PHYS, PSYC, ...}

These are all *logical* descriptions of domains. For implementation purposes, it is necessary to provide descriptions of domains in terms of concrete **data types** (or **formats**) that are provided by the DBMS (such as String, int, Boolean), in a manner analogous to how programming languages have intrinsic data types.

Attribute: the *name* of the role played by some value (coming from some domain) in the context of a **relational schema**. The domain of attribute A is denoted Dom (A).

Tuple: A tuple is a mapping from attributes to values drawn from the respective domains of those attributes. A tuple is intended to describe some entity (or relationship between entities) in the inworld.

As an example, a tuple for a PERSON entity might be

{Name --> "Keerthy", Sex -->Male, IQ -->786}

Relation: A (named) set of tuples all of the same form (i.e., having the same set of attributes). The term **table** is a loose synonym.

- **Relational Schema**: used for describing (the structure of) a relation. E.g., R (A_1, A_2, ..., A_n) says that R is a relation with *attributes*A_1 ... A_n. The **degree** of a relation is the number of attributes it has, here *n*.

Example: STUDENT (Name, SSN, Address)

One would think that a "complete" relational schema would also specify the domain of each attribute.

- **Relational Database**: A collection of **relations**, each one consistent with its specified relational schema.

Characteristics of Relations

Ordering of Tuples: A relation is a *set* of tuples; hence, there is no order associated with them. That is, it makes no sense to refer to, for example, the 5th tuple in a relation. When a relation is depicted as a table, the tuples are necessarily listed in *some* order, of course, but you should attach no significance to that order. Similarly, when tuples are represented on a storage device, they must be organized in *some* fashion, and it may be advantageous, from a performance standpoint, to organize them in a way that depends upon their content.

Ordering of Attributes: A tuple is best viewed as a mapping from its attributes (i.e., the names we give to the roles played by the values comprising the tuple) to the corresponding values. Hence, the order in which the attributes are listed in a table is irrelevant. (Note that, unfortunately, the set theoretic operations in relational algebra (at least how Elmasri & Navathe define them) make implicit use of the order of the attributes. Hence, E & N view attributes as being arranged as a sequence rather than a set.)

The **Null** value: used for *don't know*, *not applicable*.

Interpretation of a Relation: Each relation can be viewed as a **predicate** and each tuple an assertion that that predicate is satisfied (i.e., has value **true**) for the combination of values in it. In other words, each tuple represents a fact. Keep in mind that some relations represent facts about entities whereas others represent facts about relationships (between entities).

Relational database

A relational database (RDB) is a collective set of multiple data sets organized by tables, records and columns. RDBs establish a well-defined relationship between database tables. Tables communicate and share information, which facilitates data searchability, organization and reporting.

RDBs use Structured Query Language (SQL), which is a standard user application that provides an easy programming interface for database interaction.

RDB is derived from the mathematical function concept of mapping data sets and was developed by Edgar F. Codd.

RDBs organize data in different ways. Each table is known as a relation, which contains one or more data category columns. Each table record (or row) contains a unique data instance defined for a corresponding column category. One or more data or record characteristics relate to one or many records to form functional dependencies. These are classified as follows:

- One to One: One table record relates to another record in another table.
- One to Many: One table record relates to many records in another table.
- Many to One: More than one table record relates to another table record.
- Many to Many: More than one table record relates to more than one record in another table.
- RDB performs "select", "project" and "join" database operations, where select is used for data retrieval, project identifies data attributes, and join combines relations.

RDBs advantages:

- Easy extendibility, as new data may be added without modifying existing records. This is also known as scalability.
- New technology performance, power and flexibility with multiple data requirement capabilities.
- Data security, which is critical when data sharing is based on privacy. For example, management may share certain data privileges and access and block employees from other data, such as confidential salary or benefit information.

Database Schema

A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.

A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It’s the database designers who design the schema to help programmers understand the database and make it useful.

A database schema can be divided broadly into two categories −

Physical Database Schema − This schema pertains to the actual storage of data and its form of storage like files, indices, etc. It defines how the data will be stored in a secondary storage.

Logical Database Schema − This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.

Keys are the attributes of the entity, which uniquely identifies the record of the entity. For example, STUDENT_ID identifies individual students, passport#, license # etc.
As we have seen already, there are different types of keys in the database.
Super Key is the one or more attributes of the entity, which uniquely identifies the record in the database.
Candidate Key is one or more set of keys of the entity. For a person entity, his SSN, passport#, license# etc can be a super key.
Primary Key is the candidate key, which will be used to uniquely identify a record by the query. Though a person can be identified using his SSN, passport# or license#, one can choose any one of them as primary key to uniquely identify a person. Rest of them will act as a candidate key.

Comparison of primary key and foreign key

- Primary key is unique but foreign key need not be unique.
- Primary key is not null and foreign key can be null, foreign key references a primary key in another table.
- Primary key is used to identify a row; where as foreign key refers to a column or combination of columns.
- Primary key is the parent table and foreign key is a child table.

Constraints
Every relation has some conditions that must hold for it to be a valid relation. These conditions are called **Relational Integrity Constraints**. There are three main integrity constraints −

- Key constraints
- Domain constraints
- Referential integrity constraints

Key Constraints
There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called **key** for that relation. If there is more than one such minimal subset, these are called **candidate keys**.
Key constraints force that −

- In a relation with a key attribute, no two tuples can have identical values for key attributes.
- A key attribute cannot have NULL values.

Key constraints are also referred to as Entity Constraints.
Domain Constraints
Attributes have specific values in real-world scenario. For example, age can only be a positive integer. The same constraints have been tried to employ on the attributes of a relation. Every attribute is bound to have a specific range of values. For example, age cannot be less than zero and telephone numbers cannot contain a digit outside 0-9.
Referential integrity Constraints
Referential integrity constraints work on the concept of Foreign Keys. A foreign key is a key attribute of a relation that can be referred in other relation.
Referential integrity constraint states that if a relation refers to a key attribute of a different or same relation, then that key element must exist.
Intension and Extension-
Extension
The set of tuples appearing at any instant in a relation is called the extension of that relation. In other words, instance of Schema is the extension of a relation. The extension varies with time as instance of schema or the value in the database will change with time.
Intension
The intension is the schema of the relation and thus is independent of the time as it does not change once created. So, it is the permanent part of the relation and consists of –

1. **Naming Structure –** Naming Structure includes the name of the relation and the attributes of the relation.
2. **Set of Integrity Constraints –** The Integrity Constraints are divided into Integrity Rule 1 (or entity integrity rule), Integrity Rule 2 (or referential integrity rule), key constraints, domain constraints etc.

For example:

Employee (EmpNo Number (4) NOT NULL, EName Char(20), Age Number(2), Dept Char(4)

Relational Query languages- Relational query languages use relational algebra to break the user requests and instruct the DBMS to execute the requests. It is the language by which user communicates with the database. These relational query languages can be procedural or non-procedural.

Procedural query language will have set of queries instructing the DBMS to perform various transactions in the sequence to meet the user request. For example, *get_CGPA* procedure will have various queries to get the marks of student in each subject, calculate the total marks, and then decide the CGPA based on his total marks. 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.

Non-procedural queries will have single query on one or more tables to get result from the database. For example, get the name and address of the student with particular ID will have single query on STUDENT table. Relational Calculus is a non-procedural language which informs what to do with the tables, but doesn’t inform how to accomplish.

SQL

SQL language is divided into four types of primary language statements: DML, DDL, DCL and TCL. Using these statements, we can define the structure of a database by creating and altering database objects, and we can manipulate data in a table through updates or deletions. We also can control which user can read/write data or manage transactions to create a single unit of work.

The four main categories of SQL statements are as follows:

I. DDL (Data Definition Language)
II. DML (Data Manipulation Language)
III. DCL (Data Control Language)
IV. TCL (Transaction Control Language)

DDL (Data Definition Language)

DDL statements are used to alter/modify a database or table structure and schema. These statements handle the design and storage of database objects.

CREATE – create a new Table, database, schema

Example:

create table vendor_master(vencode varchar(5) unique, venname varchar(7) not null);

ALTER – alter existing table, column description

Example:

alter table vendor_masteradd(productprice int(10) check(productprice<10000));

DROP – delete existing objects from database

Example:

drop table vendor_master;

DML (Data Manipulation Language)

DML statements affect records in a table. These are basic operations we perform on data such as selecting a few records from a table, inserting new records, deleting unnecessary records, and updating/modifying existing records.

DML statements include the following:

SELECT – select records from a table

Example:

```
select * from order_master
```

INSERT – insert new records

Example:

```
insert into order_mastervalues('&oredrno','&odate','&vencode', '&o_status','&deldate');
```

UPDATE – update/Modify existing records

Example:

```
update person set address='United States'wherepid=5;
```

DELETE – delete existing records

Example:

```
delete from person where lastname='Kumar';
```

DCL (Data Control Language)

DCL statements control the level of access that users have on database objects.

GRANT – allows users to read/write on certain database objects

Example:

```
Grant select, update on dept to ABC;
```

REVOKE – keeps users from read/write permission on database objects

Example:

```
Reovek delete on emp form admin;
```

TCL (Transaction Control Language)

TCL statements allow you to control and manage transactions to maintain the integrity of data within SQL statements.

BEGIN Transaction – opens a transaction

COMMIT Transaction – commits a transaction

Example:

```
Set autocommit=0;
```

ROLLBACK Transaction – ROLLBACK a transaction in case of any error.

```
BEGIN TRAN @TransactionName
    INSERT INTO ValueTableVALUES(1), (2);
```

ROLLBACK TRAN @TransactionName;

Complex queries-

Complex SQL is the use of SQL queries which go beyond the standard SQL of using the SELECT and WHERE commands. Complex SQL often involves using complex joins and sub-queries, where queries are nested in WHERE clauses. Complex queries frequently involve heavy use of AND clauses and OR clauses. These queries make it possible for perform more accurate searches of a database.

Various 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.

Indexing -

Indexing is a way to optimize performance of a database by minimizing the number of disk accesses required when a query is processed. An index or database index is a data structure which is used to quickly locate and access the data in a database table.

Indexes are created using some database columns.

- The first column is the Search key that contains a copy of the primary key or candidate key of the table.

These values are stored in sorted order so that the corresponding data can be accessed quickly (Note that the data may or may not be stored in sorted order).

- The second column is the Data Reference which contains a set of pointers holding the address of the disk block where that particular key value can be found.

There are two kinds of indices:

1) **Ordered indices:** Indices are based on a sorted ordering of the values.
2) **Hash indices:** Indices are based on the values being distributed uniformly across a range of buckets. The bucket to which a value is assigned is determined by function called a hash function.

There is no comparison between both the techniques; it depends on the database application on which it is being applied.

- **Access Types**: e.g. value based search, range access, etc.
- **Access Time**: Time to find particular data element or set of elements.
- **Insertion Time**: Time taken to find the appropriate space and insert a new data time.
- **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead**: Additional space required by the index.

Indexing Methods

Ordered Indices

The indices are usually sorted so that the searching is faster. The indices which are sorted are known as ordered indices.

- If the search key of any index specifies same order as the sequential order of the file, it is known as primary index or clustering index.

Note: The search key of a primary index is usually the primary key, but it is not necessarily so.

- If the search key of any index specifies an order different from the sequential order of the file, it is called the secondary index or non-clustering index.

Clustered Indexing

Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.

Primary Index

In this case, the data is sorted according to the search key. It induces sequential file organization.

In this case, the primary key of the database table is used to create the index. As primary keys are unique and are stored in sorted manner, the performance of searching operation is quite efficient. The primary index is classified into two types: **Dense Index** and **Sparse Index**.

(I) Dense Index:

- For every search key value in the data file, there is an index record.
- This record contains the search key and also a reference to the first data record with that search key value.

(II) Sparse Index:

- The index record appears only for a few items in the data file. Each item points to a block as shown.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.

Non-Clustered Indexing

A non-clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For example, the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here (information on each page of book) is not organized but we have an ordered reference (contents page) to where the data points actually lie.

Secondary Index

It is used to optimize query processing and access records in a database with some information other than the usual search key (primary key). In these two levels of indexing are used in order to reduce the mapping size of the first level and in general. Initially, for the first level, a large range of numbers is selected so that the mapping size is small. Further, each range is divided into further sub ranges.

In order for quick memory access, first level is stored in the primary memory. Actual physical location of the data is determined by the second mapping level.

Trigger

A **database trigger** is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for maintaining the integrity of the information on the database.

Schema-level triggers.

- After Creation
- Before Alter
- After Alter
- Before Drop
- After Drop
- Before Insert

The four main types of triggers are:

1. Row Level Trigger: This gets executed before or after *any column value of a row* changes
2. Column Level Trigger: This gets executed before or after the *specified column* changes

3. For Each Row Type: This trigger gets executed once for each row of the result set affected by an insert/update/delete
4. For Each Statement Type: This trigger gets executed only once for the entire result set, but also fires each time the statement is executed.

Syntax

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

Details:

CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the trigger_name.

{BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.

{INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.

[OF col_name] − This specifies the column name that will be updated.

[ON table_name] − This specifies the name of the table associated with the trigger.

[REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

[FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Assertion

An **assertion** is a statement in SQL that ensures a certain condition will always exist in the database. Assertions are like column and table constraints, except that they are specified separately from table definitions. An example of a column constraint is NOT NULL, and an example of a table constraint is a compound foreign key, which, because it's compound, cannot be declared with column constraints.

Assertions are similar to check constraints, but unlike check constraints they are not defined on table or column level but are defined on schema level. (i.e., assertions are database objects of their own right and are not defined within a create table or alter table statement.)

Relational Algebra

Relational Algebra is a procedural language used for manipulating relations. The relational model gives the structure for relations so that data can be stored in that format but relational algebra enables us to retrieve information from relations. Some advanced SQL queries requires explicit relational algebra operations, most commonly outer join.

Relations are seen as sets of tuples, which means that no duplicates are allowed. SQL behaves differently in some cases. Remember the SQL keyword distinct. SQL is declarative, which means that you tell the DBMS what you want.

Set operations

Relations in relational algebra are seen as sets of tuples, so we can use basic set operations.

Review of concepts and operations from set theory

- Set Element
- No duplicate elements
- No order among the elements
- Subset
- Proper subset (with fewer elements)
- Superset
- Union
- Intersection
- Set Difference
- Cartesian product
- Relational Algebra
- Relational Algebra consists of several groups of operations

Unary Relational Operations

SELECT (symbol: s (sigma))

PROJECT (symbol: (pi))

RENAME (symbol: r (rho))

Relational Algebra Operations from Set Theory

UNION (U), INTERSECTION ( ), DIFFERENCE (or MINUS, –)

CARTESIAN PRODUCT (x)

Binary Relational Operations

JOIN (several variations of JOIN exist)

Additional Relational Operations

OUTER JOINS, OUTER UNION

AGGREGATE FUNCTIONS

Unary Relational Operations

SELECT (symbol: s (sigma))

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 meet 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 $B = \sigma_c(A)$, 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.

$B = \sigma_{((\text{year of birth} < 1935) \wedge (\text{state of birth} = \text{“Arizona”}))} (A).$

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	Λ
or	V
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.
σsubject = "database"(Books)
Output – Selects tuples from books where subject is 'database'.

σsubject = "database" and price = "450"(Books)
Output − Selects tuples from books where subject is 'database' and 'price' is 450.

σsubject = "database" and price = "450" or year > "2010"(Books)
Output − Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

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 ∏ , the upper case Greek letter Pi. A selection operation is written as B = $\prod_{\text{attributes}}$ (A), where $_{\text{attributes}}$ 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 = $\prod_{\text{first name, last name}}$ (A)

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

∏subject, author (Books)
Selects and projects columns named as subject and author from the relation Books.

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 = $\prod_{\text{first name, last name}} (\sigma_{(\text{(year of birth < 1935)} \wedge \text{(state of birth = "Arizona")})}$ (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.

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 ∪J

The union operation is both commutative and associative.

Commutative law of union: A ∪B = B ∪A
Associative law of union: (A ∪B) ∪C = A ∪(B ∪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: $B = A^C$.

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) }

A^C 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. $(A^C)^C = A$, which means that if $B = A^C$, then $A = B^C$.

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 A_T tuples and AA attributes and tuple B has B_T tuples and B_A attributes, then the new relation will have $(A_T * B_T)$ tuples, and $(A_A + B_A)$ attributes.

The symbol for a Cartesian Join is × 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 × 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 ⋈We would write C = A ⋈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 joins 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* refers to the *seats* attribute of the *table* relation.

Summary of Relation Algebra:

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	σ<selection condition>(R)
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	π<attribute list>(R)
THETA JOIN	Produces all combinations of tuples from R and R1 2 that satisfy the join condition.	R1 <join condition> R2
EQUIJOIN	Produces all the combinations of tuples from R1 and R2 that satisfy a join condition with only equality comparisons.	R1 <join condition> R2, OR (<join attributes 1>), (<join attributes 2>) R2

NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be	R1*<join condition> R2, OR R1* (<join attributes 1>), (<join attributes 2>) R2 OR R1 * R2
UNION	Produces a relation that includes all the tuples in R1 or R2 or both R1 and R2; R1 and R2 must be union-compatible.	R1 ∪ R2
INTERSECTION	Produces a relation that includes all the tuples in both R1 and R2; R1 and R2 must be union-compatible.	R1 ∩ R2
DIFFERENCE	Produces a relation that includes all the tuples in R1 that are not in R2; R1 and R2 must be union-compatible.	R1 – R2
CARTESIAN PRODUCT	Produces a relation that has the attributes of R1 and R2 and includes as tuples all possible combinations of tuples from R1 and R2.	R1 × R2
DIVISION	Produces a relation R(X) that includes all tuples t[X] in R1(Z) that appear in R1 in combination with every tuple from R(Y), where Z = X ∪ Y.	R1(Z) ÷ R2(Y)

Different Types of constraints in DBMS with Example:

- Domain Constraints
- Tuple Uniqueness Constraints
- Key Constraints
- Single Value Constraints
- Integrity Rule 1 (Entity Integrity Rule or Constraint)
- Integrity Rule 2 (Referential Integrity Rule or Constraint)
- General Constraints

Domain Constraints –

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. Consider the example below –

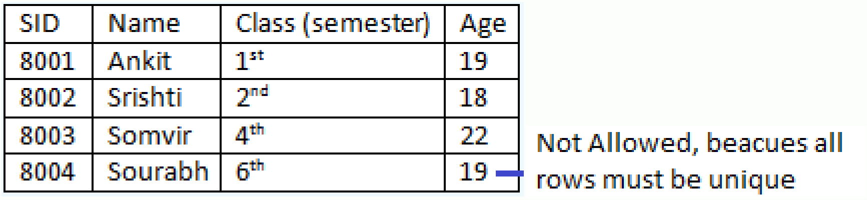
SID	Name	Class(Sem)	Age

8001	Ankit	1st	19
8002	Srishti	1st	18
8003	Somvir	4th	22
8004	Sourabh	6th	A

Not Allowed. Because age in an Integer attribute.

Tuple Uniqueness Constraints –

A relation is defined as a set of tuples. All tuples or all rows in a relation must be unique or distinct. Suppose if in a relation, tuple uniqueness constraint is applied, then all the rows of that table must be unique i.e. it does not contain the duplicate values. For example,



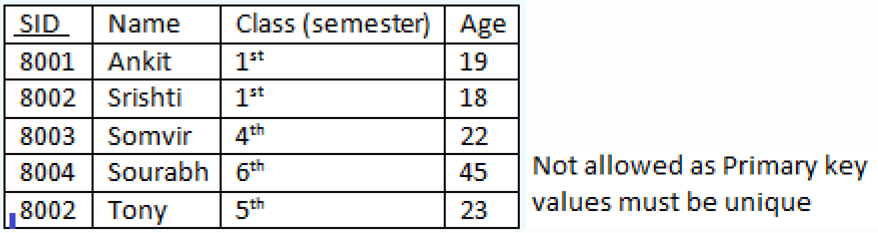
SID	Name	Class (semester)	Age
8001	Ankit	1st	19
8002	Srishti	2nd	18
8003	Somvir	4th	22
8004	Sourabh	6th	19

Not Allowed, beacues all rows must be unique

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. Primary Key must have unique and not null values in the relational table. In an subclass hierarchy, only the root entity set has a key or primary key and that primary key must serve as the key for all entities in the hierarchy.

Example of Key Constraints in a simple relational table –

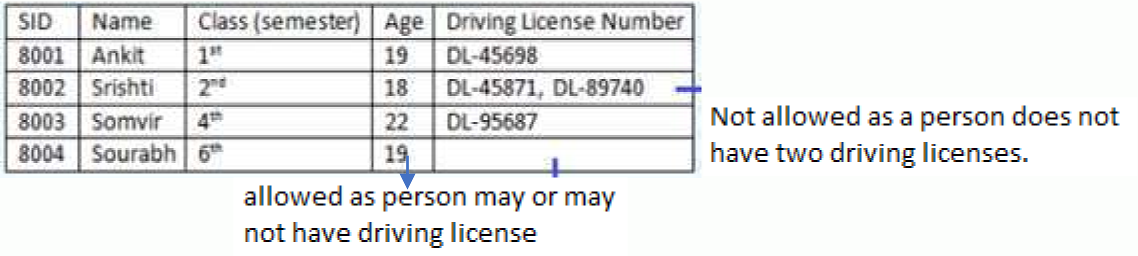


SID	Name	Class (semester)	Age
8001	Ankit	1st	19
8002	Srishti	1st	18
8003	Somvir	4th	22
8004	Sourabh	6th	45
8002	Tony	5th	23

Not allowed as Primary key values must be unique

Single Value Constraints –

Single value constraints refer that each attribute of an entity set has a single value. If the value of an attribute is missing in a tuple, then we can fill it with a "null" value. The null value for a attribute will specify that either the value is not known or the value is not applicable. Consider the below example-



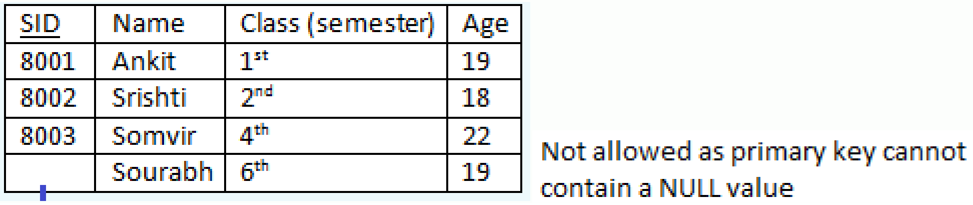
SID	Name	Class (semester)	Age	Driving License Number
8001	Ankit	1st	19	DL-45698
8002	Srishti	2nd	18	DL-45871, DL-89740
8003	Somvir	4th	22	DL-95687
8004	Sourabh	6th	19	

Not allowed as a person does not have two driving licenses.

allowed as person may or may not have driving license

Integrity Rule 1 (Entity Integrity Rule or Constraint)

The Integrity Rule 1 is also called Entity Integrity Rule or Constraint. This rule states that no attribute of primary key will contain a null value. If a relation has a null value in the primary key attribute, then uniqueness property of the primary key cannot be maintained. Consider the example below-



SID	Name	Class (semester)	Age
8001	Ankit	1st	19
8002	Srishti	2nd	18
8003	Somvir	4th	22
	Sourabh	6th	19

Not allowed as primary key cannot contain a NULL value

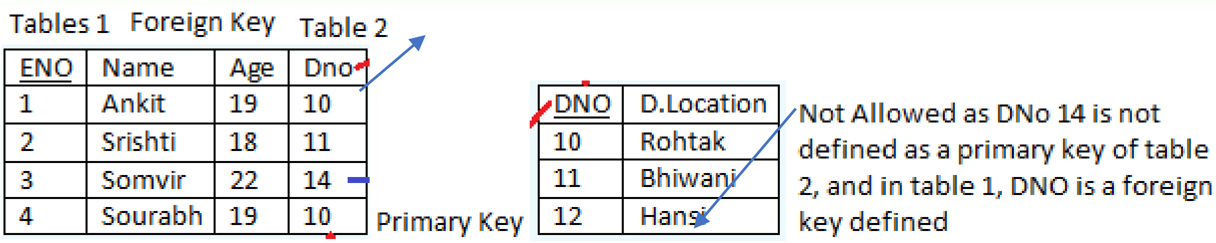
Integrity Rule 2 (Referential Integrity Rule or Constraint) –

The integrity Rule 2 is also called the Referential Integrity Constraints. 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,

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 F**oreign 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.



Tables 1 Foreign Key Table 2

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

Primary Key

DNO	D.Location
10	Rohtak
11	Bhiwani
12	Hansi

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

Relational calculus

Relational calculus is a non-procedural query language. It uses mathematical predicate calculus instead of algebra. It provides the description about the query to get the result whereas relational algebra gives the method to get the result. It informs the system what to do with the relation, but does not inform how to perform it.

For example, steps involved in listing all the students who attend 'Database' Course in relational algebra would be

- SELECT the tuples from COURSE relation with COURSE_NAME = 'DATABASE'
- PROJECT the COURSE_ID from above result
- SELECT the tuples from STUDENT relation with COUSE_ID resulted above.

In the case of relational calculus, it is described as below:

Get all the details of the students such that each student has course as 'Database'.

See the difference between relational algebra and relational calculus here. From the first one, we are clear on how to query and which relations to be queried. But the second tells what needs to be done to get the students with 'database' course. But it does tell us how we need to proceed to achieve this. Relational calculus is just the explanative way of telling the query.

There are two types of relational calculus - Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC).

Tuple Relational Calculus

A tuple relational calculus is a non-procedural query language which specifies to select the tuples in a relation. It can select the tuples with range of values or tuples for certain attribute values etc. The resulting relation can have one or more tuples. It is denoted as below:

{t | P (t)} or **{t | condition (t)} --** this is also known as expression of relational calculus

Where t is the resulting tuples, P(t) is the condition used to fetch t.

{t | EMPLOYEE (t) and t.SALARY>10000} - implies that it selects the tuples from EMPLOYEE relation such that resulting employee tuples will have salary greater than 10000. It is example of selecting a range of values.

{t | EMPLOYEE (t) AND t.DEPT_ID = 10} – this select all the tuples of employee name who work for Department 10.

The variable which is used in the condition is called **tuple variable**. In above example t.SALARY and t.DEPT_ID are tuple variables. In the first example above, we have specified the condition t.SALARY>10000. What is the meaning of it? For all the SALARY>10000, display the employees. Here the SALARY is called as bound variable. Any tuple variable with '*For All*' (?) or '*there exists*' (?) condition is **called bound variable**. Here, for any range of values of SALARY greater than 10000, the meaning of the condition remains the same. Bound variables are those ranges of tuple variables whose meaning will not change if the tuple variable is replaced by another tuple variable.

In the second example, we have used DEPT_ID= 10. That means only for DEPT_ID = 10 display employee details. Such variable is called free variable. Any tuple variable without any '*For All*' or '*there exists*' condition is called **Free Variable**. If we change DEPT_ID in this condition to some other variable, say EMP_ID, the meaning of the query changes. For example, if we change EMP_ID = 10, then above it will result in different result set. Free variables are those ranges of tuple variables whose meaning will change if the tuple variable is replaced by another tuple variable.

All the conditions used in the tuple expression are called as well formed formula – WFF. All the conditions in the expression are combined by using logical operators like AND, OR and NOT, and qualifiers like 'For All' (?) or 'there exists' (?). If the tuple variables are all bound variables in a WFF is called **closed WFF**. In an **open WFF**, we will have at least one free variable.

Domain Relational Calculus

In contrast to tuple relational calculus, domain relational calculus uses list of attributes to be selected from the relation based on the condition. It is same as TRC, but differs by selecting the attributes rather than selecting whole tuples. It is denoted as below:

$\{< a_1, a_2, a_3, \dots a_n > \mid P(a_1, a_2, a_3, \dots a_n)\}$

Where $a_1, a_2, a_3, \dots a_n$ are attributes of the relation and P is the condition.

For example, select EMP_ID and EMP_NAME of employees who work for department 10

{<EMP_ID, EMP_NAME> | <EMP_ID, EMP_NAME> ? EMPLOYEE Λ DEPT_ID = 10}

Get name of the department name that Alex works for.

{DEPT_NAME |< DEPT_NAME > ? DEPT Λ ? DEPT_ID (<DEPT_ID> ? EMPLOYEE Λ EMP_NAME = Alex)}

Here green color expression is evaluated to get the department Id of Alex and then it is used to get the department name form DEPT relation.

Let us consider another example where select EMP_ID, EMP_NAME and ADDRESS the employees from the department where Alex works. What will be done here?

{<EMP_ID, EMP_NAME, ADDRESS, DEPT_ID > | <EMP_ID, EMP_NAME, ADDRESS, DEPT_ID> ? EMPLOYEE Λ ? DEPT_ID (<DEPT_ID> ? EMPLOYEE Λ EMP_NAME = Alex)}

First, formula is evaluated to get the department ID of Alex (green color), and then all the employees with that department is searched (red color).

Other concepts of TRC like free variable, bound variable, WFF etc. remains same in DRC too. Its only difference is DRC is based on attributes of relation.