APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS

MODULE II

Structure of Relational Databases - Integrity Constraints, Synthesizing ER Diagram to Relational Schema

Introduction to Relational Algebra - Select, Project, Cartesian Product Operations, Join - Equi-join, Natural Join, Query Examples.

Introduction to Structured Query Language (SQL) - Data Definition Language (DDL), Table Definitions and Operations – CREATE, DROP, ALTER, INSERT, DELETE, UPDATE.

Reference:

Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.

STRUCTURE OF RELATIONAL DATABASES

The relational data model was first introduced by Ted Codd of IBM Research in 1970 in a classic paper (Codd 1970), and it attracted immediate attention due to its simplicity and mathematical foundation. The model uses the concept of a *mathematical relation*—which looks somewhat like a table of values—as its basic building block, and has its theoretical basis in set theory and first-order predicate logic.

The first commercial implementations of the relational model became available in the early 1980s, such as the SQL/DS system on the MVS operating system by IBM and the Oracle DBMS. Since then, the model has been implemented in a large number of commercial systems. Current popular relational DBMSs (RDBMSs) include DB2 and Informix Dynamic Server (from IBM), Oracle and Rdb (from Oracle), Sybase DBMS (from Sybase) and SQLServer and Access (from Microsoft). In addition, several open source systems, such as MySQL and PostgreSQL, are available.

Basic Concepts of Relational Data Model

The relational model represents the database as a collection of *relations.* Informally, each relation resembles a table of values or, to some extent, a *flat* file of records. It is called a **flat file** because each record has a simple linear or *flat* structure.

When a relation is thought of as a **table** of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row.

For example, a table is called **STUDENT** because each row represents facts about a particular student entity. The column names—Name, Student_number, Class, and Major—specify how to

interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

In the formal relational model terminology, a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation.* The data type describing the types of values that can appear in each column is represented by a *domain* of possible values. We now define these terms—*domain*, *tuple*, *attribute*, and *relation*— formally.

Domains, Attributes, Tuples, and Relations

- **Domain**

A **domain** *D* is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values.

Some examples of domains follow:

■ **Usa_phone_numbers** : The set of ten-digit phone numbers valid in the United States.

■ **Social_security_numbers** : The set of valid nine-digit Social Security numbers. (This is a unique identifier assigned to each person in the United States for employment, tax, and benefits purposes.)

■ **Names** : The set of character strings that represent names of persons.

■ **Academic_department_names** : The set of academic department names in a university, such as Computer Science, Economics, and Physics.

■ **Academic_department_codes** : The set of academic department codes, such as ‘CS’, ‘ECON’, and ‘PHYS’.

The preceding are called *logical* definitions of domains.

A **data type** or **format** is also specified for each domain. For example, the data type for the domain Usa_phone_numbers can be declared as a character string of the form (*ddd*)*ddddddd*, where each *d* is a numeric (decimal) digit and the first three digits form a valid telephone area code. The data type for Employee_ages is an integer number between 15 and 80.

A domain is thus given a name, data type, and format. Additional information for interpreting the values of a domain can also be given; for example, a numeric domain such as Person_weights should have the units of measurement, such as pounds or kilograms.

- **Relation Schema**

A **relation schema** *R*, denoted by *R* (*A*1, *A*2, ..., An), is made up of a relation name *R* and a list of attributes, *A*1, *A*2... *An*. Each **attribute** *Ai* is the name of a role played by some domain *D* in the relation schema *R*. A relation schema is used to *describe* a relation; *R* is called the **name** of this relation.

The **degree** (or **arity**) of a relation is the number of attributes *n* of its relation schema.

- **Relation Schema – Example**

A relation of degree seven, which stores information about university students, would contain seven attributes describing each student as follows:

STUDENT (Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)

Using the data type of each attribute, the definition is sometimes written as:

STUDENT (Name: string, Ssn: string, Home_phone: string, Address: string, Office_phone: string, Age: integer, Gpa: real)

- **Relation State**

A **relation** (or **relation state**) r of the relation schema *R*(*A*1, *A*2, ..., *An*), also denoted by *r*(*R*), is a set of *n*-tuples *r* = {*t*1, *t*2, ..., *tm*}. Each ***n*-tuple** *t* is an ordered list of *n* values *t* =<*v*1, *v*2, ...,*vn*>, where each value *vi*, $1 \leq i \leq n$, is an element of dom *(Ai*) or is a special NULL value.

Characteristics of relations

The earlier definition of relations implies certain characteristics that make a relation different from a file or a table. Those characteristics are:

- **Ordering of Tuples in a Relation**

A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. However, in a file, records are physically stored on disk (or in memory), so there always is an order among the records. This ordering indicates first, second, *i*th, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.

Tuple ordering is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level. Many tuple orders can be specified on the same relation.

- **Ordering of Values within a Tuple and an Alternative Definition of a Relation**

According to the preceding definition of a relation, an *n*-tuple is an *ordered list* of *n* values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important.

An **alternative definition** of a relation can be given, making the ordering of values in a tuple *unnecessary*. According to this definition of tuple as a mapping, a **tuple** can be considered as a **set** of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute *Ai* to a value *vi* from dom(*Ai*). The ordering of attributes is *not* important, because the *attribute name* appears with its *value*.

- **Values and NULLs in the Tuples**

Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes are not allowed. This model is sometimes called the **flat relational model.**

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases. For example, some STUDENT tuples have NULL for their office phones because they do not have an office (that is, office phone *does not apply* to these students). Another student has a NULL for home phone, presumably because either he does not have a home phone or he has one but we do not know it (value is *unknown*). In general, we can have several meanings for NULL values, such as ***value unknown***, ***value*** exists but is ***not available***, or ***attribute does not apply*** to this tuple *(*also known as ***value undefined****).*

Constraints on Relational Models

There are generally many restrictions or **constraints** on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents.

In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints. Constraints on databases can generally be divided into three main categories:

1. Constraints that are inherent in the data model. We call these **inherent model-based constraints** or **implicit constraints**.
2. Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the DDL. We call these **schema-based constraints** or **explicit constraints**.
3. Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs. We call these **application-based** or **semantic constraints** or **business rules**.

- **Domain Constraints**

Domain constraints specify that within each tuple, the value of each attribute *A* must be an atomic value from the domain dom(*A*). The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers

(float and double precision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money, or other special data types.

- **Key Constraints and Constraints on NULL Values**

In the formal relational model, a *relation* is defined as a *set of tuples.* No two tuples can have the same combination of values for *all* their attributes. Usually, there are **subsets of attributes** of a relation schema *R* with the property that no two tuples in any relation state *r* of *R* should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples *t*1 and *t*2 in a relation state *r* of *R*, we have the constraint that:

$t1[SK] \neq t2[SK]$

Any such set of attributes SK is called a **superkey** of the relation schema *R*. A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state *r* of *R* can have the same value for SK. Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key,* which has no redundancy.

A **key** *K* of a relation schema *R* is a superkey of *R* with the additional property that removing any attribute *A* from *K* leaves a set of attributes *K* that is not a superkey of *R* any more. Hence, a key satisfies two properties:

1. Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This first property also applies to a superkey.
2. It is a ***minimal superkey***—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold.

This property is not required by a superkey.

Whereas the first property applies to both keys and superkeys, the second property is required only for keys. Hence, a key is also a superkey but not vice versa.

Eg: The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn. Any set of attributes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey. However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey.

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key**.

For example**,** the CAR relation has two candidate keys: License_number and Engine_serial_number. It is common to designate one of the candidate keys as the **primary key** of the relation.

- **Integrity, Referential Integrity, and Foreign Keys**

  - **Entity Integrity Constraint**

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations.

  - **Referential Integrity Constraint**

The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, 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. For example, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, first we define the concept of a **foreign key**. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R1 and R2.

A set of attributes FK in relation schema R1 is a foreign key of R1 that references relation R2 if it satisfies the following rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R2; the attributes FK are said to reference or refer to the relation R2.

2. A value of FK in a tuple t1 of the current state r1(R1) either occurs as a value of PK for some tuple t2 in the current state r2(R2) or is NULL. In the former case, we have t1[FK] = t2[PK], and we say that the tuple t1 references or refers to the tuple t2. In this definition, R1 is called **the referencing relation** and R2 is **the referenced relation.**

If these two conditions hold, a referential integrity constraint from R1 to R2 is said to hold. In a database of many relations, there are usually many referential integrity constraints.

Synthesizing ER Diagrams to Relational Schema

After designing an ER Diagram,

- ER diagram is converted into the tables in relational model.
- This is because relational models can be easily implemented by RDBMS like MySQL , Oracle etc.

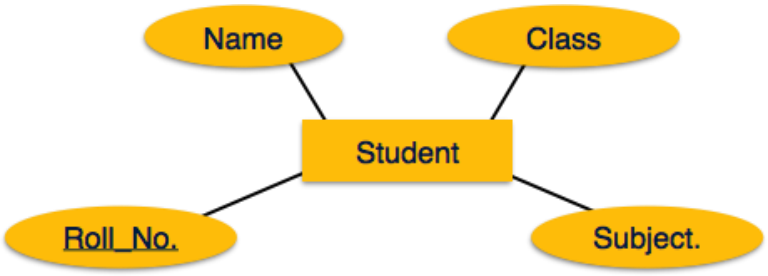
Following rules are used for converting an ER diagram into the tables:

Rule-01: For Strong Entity Set With Only Simple Attributes:

A strong entity set with only simple attributes will require only one table in relational model.

- Attributes of the table will be the attributes of the entity set.
- The primary key of the table will be the key attribute of the entity set.

Example-

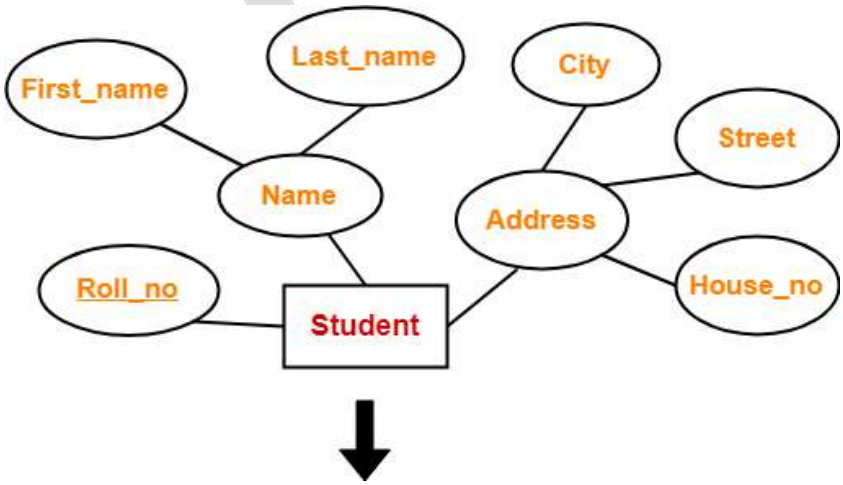


Roll no	Name	Class	Subject

Rule-02: For Strong Entity Set With Composite Attributes:

- A strong entity set with any number of composite attributes will require only one table in relational model.
- While conversion, simple attributes of the composite attributes are taken into account and not the composite attribute itself.

Example-



Roll_no	First_name	Last_name	House_no	Street	City

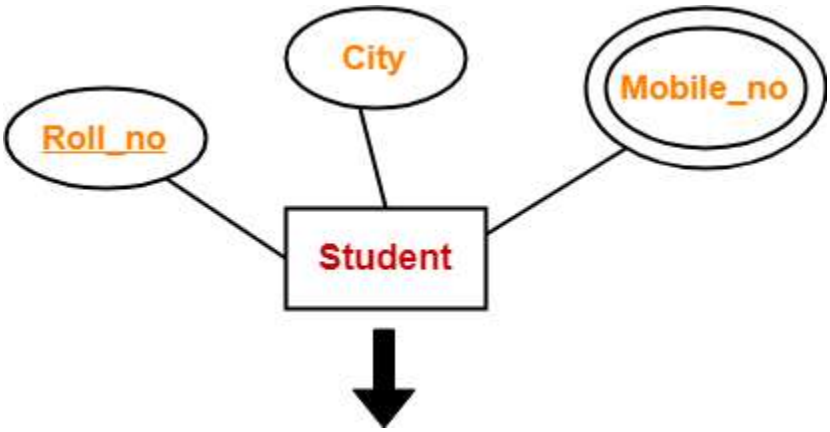
Schema: Student (Roll_no , First_name , Last_name , House_no , Street , City)

Rule-03: For Strong Entity Set With Multi Valued Attributes:

A strong entity set with any number of multi valued attributes will require two tables in relational model.

- One table will contain all the simple attributes with the primary key.
- Other table will contain the primary key and all the multi valued attributes.

Example-



Roll_no	City

Roll_no	Mobile_no

Rule-04: Translating Relationship Set into a Table:

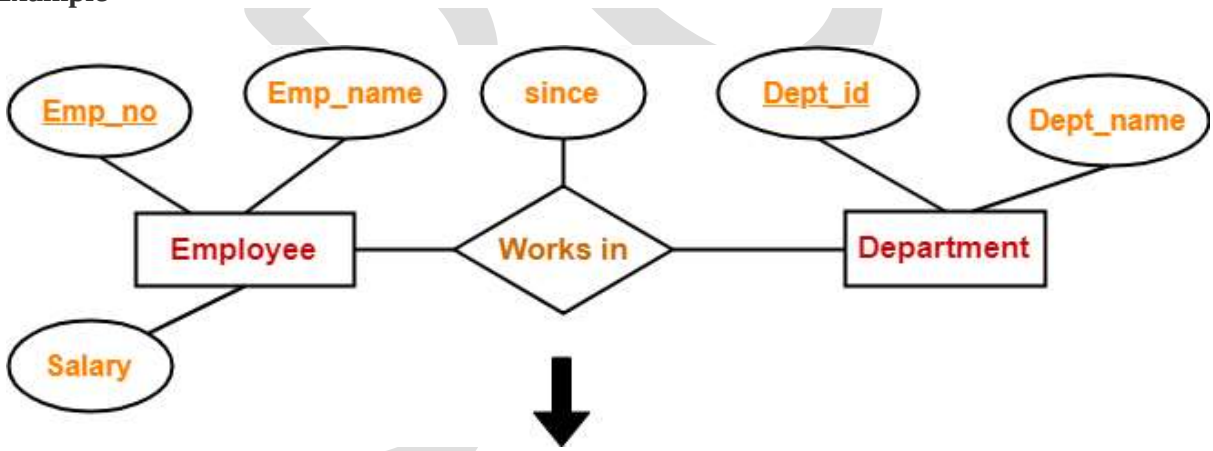
A relationship set will require one table in the relational model.

Attributes of the table are-

- Primary key attributes of the participating entity sets
- Its own descriptive attributes if any.

Set of non-descriptive attributes will be the primary key.

Example-



Emp_no	Dept_id	since

Schema : Works in (Emp_no , Dept_id , since)

NOTE-

If we consider the overall ER diagram, three tables will be required in relational model-

- One table for the entity set "Employee"
- One table for the entity set "Department"
- One table for the relationship set "Works in"

Rule-05: For Binary Relationships With Cardinality Ratios-

The following four cases are possible-

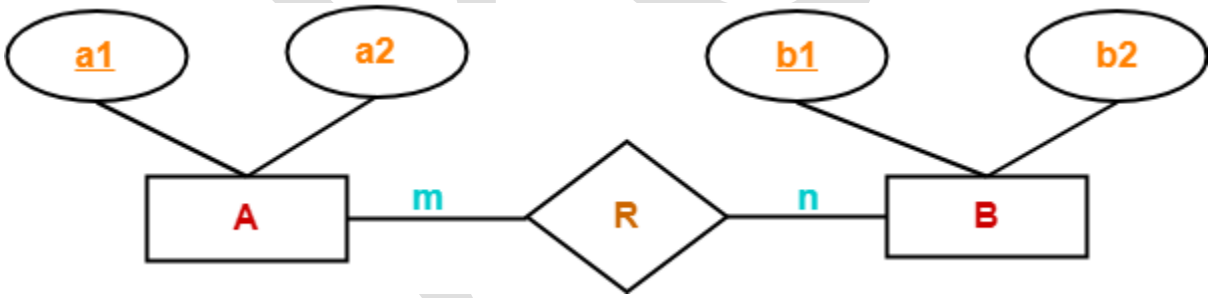
Case-01: Binary relationship with cardinality ratio m:n

Case-02: Binary relationship with cardinality ratio 1:n

Case-03: Binary relationship with cardinality ratio m:1

Case-04: Binary relationship with cardinality ratio 1:1

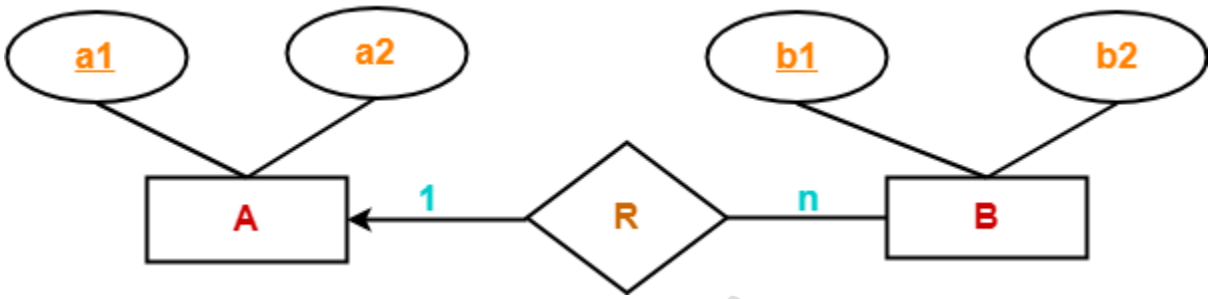
Case-01: For Binary Relationship with Cardinality Ratio m:n



Here, three tables will be required-

1. A (a1 , a2)
2. R (a1 , b1)
3. B (b1 , b2)

Case-02: For Binary Relationship with Cardinality Ratio 1:n

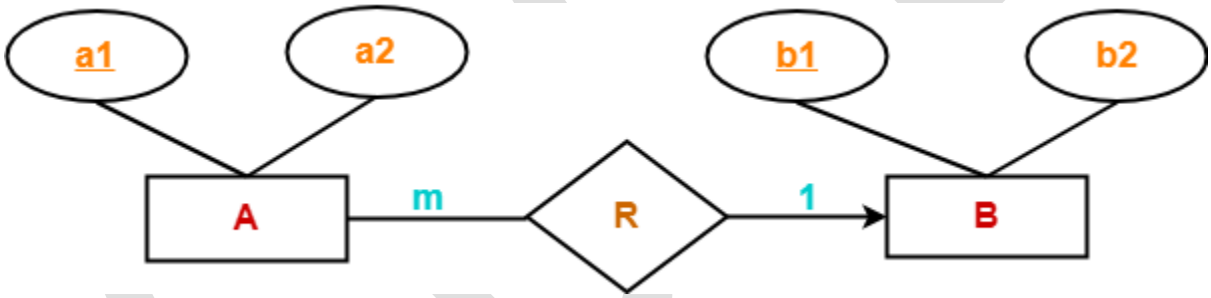


Here, two tables will be required-

1. A (a1 , a2)
2. BR (a1 , b1 , b2)

NOTE- Here, combined table will be drawn for the entity set B and relationship set R.

Case-03: For Binary Relationship with Cardinality Ratio m:1

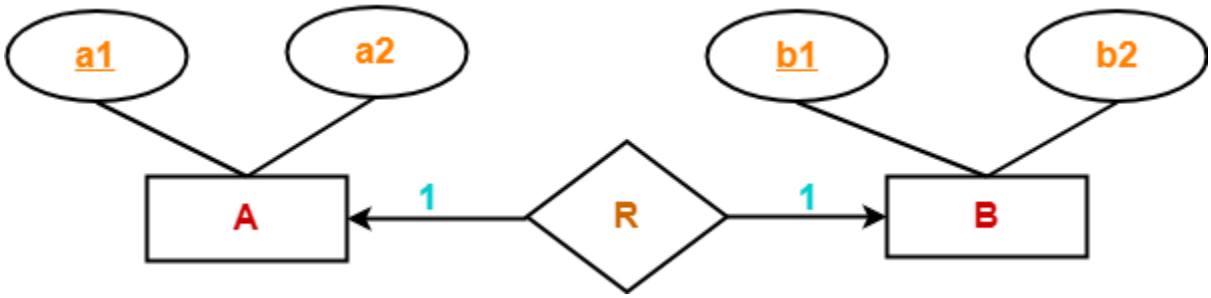


Here, two tables will be required-

1. AR (a1 , a2 , b1)
2. B (b1 , b2)

NOTE- Here, combined table will be drawn for the entity set A and relationship set R.

Case-04: For Binary Relationship with Cardinality Ratio 1:1



Here, two tables will be required. Either combine 'R' with 'A' or 'B'

Way-01:

1. AR (a1 , a2 , b1)
2. B (b1 , b2)

Way-02:

1. A (a1 , a2)
2. BR (a1 , b1 , b2)

Thumb Rules to Remember:

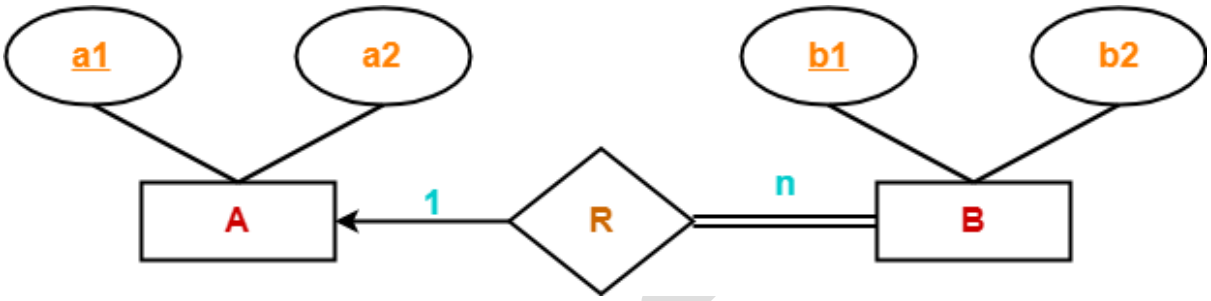
While determining the minimum number of tables required for binary relationships with given cardinality ratios, following thumb rules must be kept in mind-

- For binary relationship with cardinality ration m : n , separate and individual tables will be drawn for each entity set and relationship.
- For binary relationship with cardinality ratio either m : 1 or 1 : n , always remember "many side will consume the relationship" i.e. a combined table will be drawn for many side entity set and relationship set.
- For binary relationship with cardinality ratio 1 : 1 , two tables will be required. You can combine the relationship set with any one of the entity sets.

Rule-06: For Binary Relationship with Both Cardinality Constraints and Participation Constraints

- Cardinality constraints will be implemented as discussed in Rule-05.
- Because of the total participation constraint, foreign key acquires NOT NULL constraint i.e. now foreign key cannot be null.

Case-01: For Binary Relationship with Cardinality Constraint and Total Participation Constraint from One Side-



Because cardinality ratio = 1 : n , so we will combine the entity set B and relationship set R.

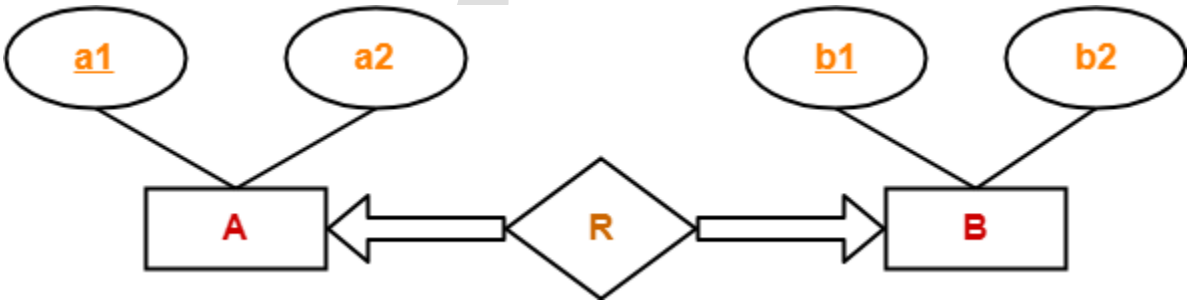
Then, two tables will be required-

1. A (a1 , a2)
2. BR (a1 , b1 , b2)

Because of total participation, foreign key a1 has acquired NOT NULL constraint, so it can't be null now.

Case-02: For Binary Relationship with Cardinality Constraint and Total Participation Constraint from Both Sides-

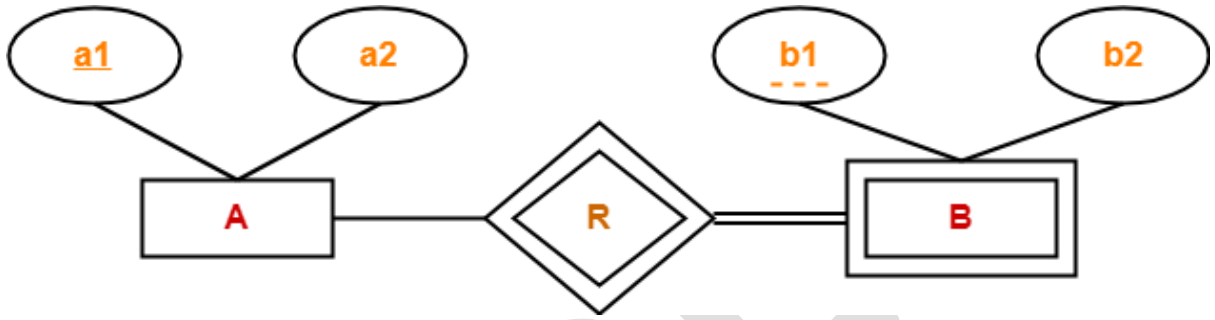
If there is a key constraint from both the sides of an entity set with total participation, then that binary relationship is represented using only single table.



Here, only one table is required. ARB (a1, a2, b1, b2)

Rule-07: For Binary Relationship with Weak Entity Set-

Weak entity set always appears in association with identifying relationship with total participation constraint.



Here, two tables will be required-

1. A (a1, a2)
2. BR (a1, b1, b2)

Introduction to Relational Algebra

The basic set of operations for the relational model is the relational algebra. These operations enable a user to specify basic retrieval requests as relational algebra expressions. The result of a retrieval is a new relation, which may have been formed from one or more relations. A sequence of relational algebra operations forms a **relational algebra expression**, whose result will also be a relation that represents the result of a database query (or retrieval request).

Importance of Relational Algebra

1. It provides a formal foundation for relational model operations.
2. It is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs).
3. Some of its concepts are incorporated into the SQL standard query language for RDBMSs.

Unary Relational Operations: SELECT and PROJECT

The SELECT Operation

The SELECT operation is used to choose a subset of the tuples from a relation that satisfies a selection condition. In general, the SELECT operation is denoted by

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

where the symbol σ (sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation R.

Query: Select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000

$$\sigma_{Dno=4}(EMPLOYEE)$$
$$\sigma_{Salary>30000}(EMPLOYEE)$$

The PROJECT Operation

If we think of a relation as a table, the SELECT operation chooses some of the rows from the table while discarding other rows. The PROJECT operation, on the other hand, selects certain columns from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to project the relation over these attributes only.

The general form of the PROJECT operation is

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

where π (pi) is the symbol used to represent the PROJECT operation, and <attribute list> is the desired sublist of attributes from the attributes of relation *R*.

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

$$\pi_{Lname,\ Fname,\ Salary}(EMPLOYEE)$$

Sequences of Operations and the RENAME Operation

In general, for most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single relational algebra expression by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results.

For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression, also known as an in-line expression, as follows:

$$\pi_{\text{Fname, Lname, Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$$

Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

$$\text{DEP5\_EMPS} \leftarrow \sigma_{\text{Dno=5}}(\text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Salary}}(\text{DEP5\_EMPS})$$

RENAME operation—To rename either the relation name or the attribute names, or both—as a unary operator. The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

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

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.

Relational Algebra Operations from Set Theory

The UNION, INTERSECTION, and MINUS Operations

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION**, **INTERSECTION**, and **SET DIFFERENCE** (also called **MINUS** or **EXCEPT**). These are **binary** operations; that is, each is applied to two sets (of tuples).

Union Compatibility

When set operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples**; this condition has been called ***union compatibility* or *type compatibility***.

Two relations $R(A1, A2, ..., An)$ and $S(B1, B2, ..., Bn)$ are said to be **union compatible** (or **type compatible**) if they have the same degree n and if $dom(Ai) = dom(Bi)$ for $1 <= i <= n$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

The three operations UNION, INTERSECTION, and SET DIFFERENCE can be defined on two union-compatible relations R and S as follows:

- **UNION:** The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.
- **INTERSECTION:** The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S.
- **SET DIFFERENCE (or MINUS):** The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S.

The set operations UNION, INTERSECTION, and MINUS. (a) Two union-compatible relations. (b) STUDENT ∪ INSTRUCTOR. (c) STUDENT ∩ INSTRUCTOR. (d) STUDENT – INSTRUCTOR. (e) INSTRUCTOR – STUDENT.

(a) **STUDENT**

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(b)

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

The CARTESIAN PRODUCT operation—also known as CROSS PRODUCT or CROSS JOIN—is denoted by ×. This is also a binary set operation, but the relations on which it is applied do not have to be union compatible. In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set). In general, the result of R(A1, A2, ..., An) × S(B1, B2, ..., Bm) is a relation Q with degree n + m attributes Q(A1, A2, ..., An, B1, B2, ..., Bm), in that order.

Query: Retrieve a list of names of each female employee's dependents.

$$\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{Sex}='F'}(\text{EMPLOYEE})$$
$$\text{EMPNAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE\_EMPS})$$
$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn}=\text{Essn}}(\text{EMP\_DEPENDENTS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Dependent\_name}}(\text{ACTUAL\_DEPENDENTS})$$

Figure 6.5
The Cartesian Product (Cross Product) operation.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

Binary Relational Operations: JOIN and DIVISION

The JOIN Operation

The JOIN operation, denoted by ⋈ is used to combine related tuples from two relations into single "longer" tuples. This operation is very important for any relational database with more than a single relation because it allows us to process relationships among relations.

Eg: To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple. We do this by using the JOIN operation and then projecting the result over the necessary attributes, as follows:

$$DEPT\_MGR \leftarrow DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE$$
$$RESULT \leftarrow \pi_{Dname,\ Lname,\ Fname}(DEPT\_MGR)$$

DEPT_MGR

Dname	Dnumber	Mgr_ssn	...	Fname	Minit	Lname	Ssn	...
Research	5	333445555	...	Franklin	T	Wong	333445555	...
Administration	4	987654321	...	Jennifer	S	Wallace	987654321	...
Headquarters	1	888665555	...	James	E	Borg	888665555	...

Figure 6.6
Result of the JOIN operation $DEPT\_MGR \leftarrow DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE$.

A general join condition is of the form <condition> AND <condition> AND...AND <condition> where each <condition> is of the form Ai θ Bj, Ai is an attribute of R, Bj is an attribute of S, Ai and Bj have the same domain, and θ (theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠}. A JOIN operation with such a general join condition is called a **THETA JOIN**.

Tuples whose join attributes are NULL or for which the join condition is FALSE do not appear in the result. In that sense, the JOIN operation does not necessarily preserve all of the information in the participating relations, because tuples that do not get combined with matching ones in the other relation do not appear in the result.

Variations of JOIN: The EQUIJOIN and NATURAL JOIN

The most common use of JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN.**

In the result of an EQUIJOIN we always have one or more pairs of attributes that have identical values in every tuple.

Because one of each pair of attributes with identical values is superfluous, a new operation called **NATURAL JOIN**—denoted by *—was created to get rid of the second (superfluous) attribute 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, a renaming operation is applied first.

Eg:- Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project.

Method 1

$$PROJ\_DEPT \leftarrow PROJECT * \rho_{(Dname,\ Dnum,\ Mgr\_ssn,\ Mgr\_start\_date)}(DEPARTMENT)$$

Method 2

$$DEPT \leftarrow \rho_{(Dname,\ Dnum,\ Mgr\_ssn,\ Mgr\_start\_date)}(DEPARTMENT)$$
$$PROJ\_DEPT \leftarrow PROJECT * DEPT$$

The attribute Dnum is called the join attribute for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations.

If the attributes on which the natural join is specified already have the same names in both relations, renaming is unnecessary. For example, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write

DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS

(a)

PROJ_DEPT

Pname	Pnumber	Plocation	Dnum	Dname	Mgr_ssn	Mgr_start_date
ProductX	1	Bellaire	5	Research	333445555	1988-05-22
ProductY	2	Sugarland	5	Research	333445555	1988-05-22
ProductZ	3	Houston	5	Research	333445555	1988-05-22
Computerization	10	Stafford	4	Administration	987654321	1995-01-01
Reorganization	20	Houston	1	Headquarters	888665555	1981-06-19
Newbenefits	30	Stafford	4	Administration	987654321	1995-01-01

(b)

DEPT_LOCS

Dname	Dnumber	Mgr_ssn	Mgr_start_date	Location
Headquarters	1	888665555	1981-06-19	Houston
Administration	4	987654321	1995-01-01	Stafford
Research	5	333445555	1988-05-22	Bellaire
Research	5	333445555	1988-05-22	Sugarland
Research	5	333445555	1988-05-22	Houston

Figure 6.7
Results of two NATURAL JOIN operations. (a) PROJ_DEPT ← PROJECT * DEPT. (b) DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS.

The DIVISION Operation

The DIVISION operation, denoted by ÷, 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 'John Smith' works on. To express this query using the DIVISION operation, proceed as follows.

First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

$$SMITH \leftarrow \sigma_{Fname='John' \text{ AND } Lname='Smith'}(EMPLOYEE)$$
$$SMITH\_PNOS \leftarrow \pi_{Pno}(WORKS\_ON \bowtie_{Essn=Ssn} SMITH)$$

Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

$$SSN\_PNOS \leftarrow \pi_{Essn,\ Pno}(WORKS\_ON)$$

Finally, apply the DIVISION operation to the two relations, which gives the desired employees' Social Security numbers:

$$SSNS(Ssn) \leftarrow SSN\_PNOS \div SMITH\_PNOS$$
$$RESULT \leftarrow \pi_{Fname,\ Lname}(SSNS * EMPLOYEE)$$

The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

(a)

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

S

A
a1
a2
a3

T

B
b1
b4

For a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with every tuple in S.

Additional Relational Operations

Some common database requests—which are needed in commercial applications for RDBMSs—cannot be performed with the original relational algebra operations. Additional operations are defined to express these requests. These operations enhance the expressive power of the original relational algebra.

Generalized Projection

The generalized projection operation extends the projection operation by allowing functions of attributes to be included in the projection list. The generalized form can be expressed as:

$$\pi_{F1, F2, ..., Fn}(R)$$

where F1, F2, ..., Fn are functions over the attributes in relation R and may involve arithmetic operations and constant values.

Eg:- Consider the relation

EMPLOYEE (Ssn, Salary, Deduction, Years_service)

A report may be required to show

Net Salary = Salary – Deduction,

Bonus = 2000 * Years_service, and

Tax = 0.25 * Salary.

Then a generalized projection combined with renaming may be used as follows:

$$\text{REPORT} \leftarrow \rho_{(\text{Ssn, Net\_salary, Bonus, Tax})}(\pi_{\text{Ssn, Salary} - \text{Deduction, } 2000 * \text{Years\_service, } 0.25 * \text{Salary}}(\text{EMPLOYEE})).$$

Aggregate Functions and Grouping

Another type of request that cannot be expressed in the basic relational algebra is to specify mathematical aggregate functions on collections of values from the database.

Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM. The COUNT function is used for counting tuples or values.

We can define an AGGREGATE FUNCTION operation, using the symbol ℑ (pronounced script F), to specify these types of requests as follows:

$$\langle \text{grouping attributes} \rangle \, \Im_{\langle \text{function list} \rangle}(R)$$

where <grouping attributes> is a list of attributes of the relation specified in R, and <function list> is a list of (<function> <attribute>) pairs.

Query: Retrieve each department number, the number of employees in the department, and their average salary.

$$\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}({}_{\text{Dno}}\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE}))$$

Figure 6.10
The aggregate function operation.

a. $\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}({}_{\text{Dno}}\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE}))$.

b. ${}_{\text{Dno}}\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

c. $\Im_{\text{COUNT Ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

R

(a)

Dno	No_of_employees	Average_sal
5	4	33250
4	3	31000
1	1	55000

(b)

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

(c)

Count_ssn	Average_salary
8	35125

OUTER JOIN Operations

A set of operations, called **outer joins**, were developed for the case where the user wants to keep all the tuples in *R*, or all those in *S*, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation.

The LEFT OUTER JOIN operation keeps every tuple in the first, or left, relation R in R ⟕ S; if no matching tuple is found in S, then the attributes of S in the join result are filled or padded with NULL values.

A similar operation, RIGHT OUTER JOIN, denoted by ⟖ , keeps every tuple in the second, or right, relation S in the result of R ⟖ S. A third operation, FULL OUTER JOIN, denoted by ⟗, keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with NULL values as needed.

Student

S_id	Name	Class	Age	C_type
1	Andrew	5	25	A
2	Angel	10	30	A
3	Anamika	8	35	C

Course

C_type	C_name
A	Foundation C
B	C++

Student ⋈ Course

S_id	Name	Class	Age	C_type	C_name
1	Andrew	5	25	A	Foundation C
2	Angel	10	30	A	Foundation C
3	Anamika	8	35	C	-

Student ⋈ Course

S_id	Name	Class	Age	C_type	C_name
1	Andrew	5	25	A	Foundation C
2	Angel	10	30	A	Foundation C
-	-	-	-	B	C++

Student ⋈ Course

S_id	Name	Class	Age	C_type	C_name
1	Andrew	5	25	A	Foundation C
2	Angel	10	30	A	Foundation C
3	Anamika	8	35	C	-
-	-	-	-	B	C++

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

$$RESEARCH\_DEPT \leftarrow \sigma_{Dname='Research'}(DEPARTMENT)$$
$$RESEARCH\_EMPS \leftarrow (RESEARCH\_DEPT \bowtie_{Dnumber=Dno} EMPLOYEE)$$
$$RESULT \leftarrow \pi_{Fname,\ Lname,\ Address}(RESEARCH\_EMPS)$$

As a single in-line expression, this query becomes:

$$\pi_{Fname,\ Lname,\ Address}\ (\sigma_{Dname='Research'}(DEPARTMENT \bowtie_{Dnumber=Dno}(EMPLOYEE))$$

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

$$\text{STAFFORD\_PROJS} \leftarrow \sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})$$
$$\text{CONTR\_DEPTS} \leftarrow (\text{STAFFORD\_PROJS} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT})$$
$$\text{PROJ\_DEPT\_MGRS} \leftarrow (\text{CONTR\_DEPTS} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(\text{PROJ\_DEPT\_MGRS})$$

Query 3. Make a list of project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

$$\text{SMITHS(Essn)} \leftarrow \pi_{\text{Ssn}}(\sigma_{\text{Lname='Smith'}}(\text{EMPLOYEE}))$$
$$\text{SMITH\_WORKER\_PROJS} \leftarrow \pi_{\text{Pno}}(\text{WORKS\_ON} * \text{SMITHS})$$
$$\text{MGRS} \leftarrow \pi_{\text{Lname, Dnumber}}(\text{EMPLOYEE} \bowtie_{\text{Ssn=Mgr\_ssn}} \text{DEPARTMENT})$$
$$\text{SMITH\_MANAGED\_DEPTS(Dnum)} \leftarrow \pi_{\text{Dnumber}}(\sigma_{\text{Lname='Smith'}}(\text{MGRS}))$$
$$\text{SMITH\_MGR\_PROJS(Pno)} \leftarrow \pi_{\text{Pnumber}}(\text{SMITH\_MANAGED\_DEPTS} * \text{PROJECT})$$
$$\text{RESULT} \leftarrow (\text{SMITH\_WORKER\_PROJS} \cup \text{SMITH\_MGR\_PROJS})$$

Query 4. Retrieve the names of employees who have no dependents.

$$\text{ALL\_EMPS} \leftarrow \pi_{\text{Ssn}}(\text{EMPLOYEE})$$
$$\text{EMPS\_WITH\_DEPS(Ssn)} \leftarrow \pi_{\text{Essn}}(\text{DEPENDENT})$$
$$\text{EMPS\_WITHOUT\_DEPS} \leftarrow (\text{ALL\_EMPS} - \text{EMPS\_WITH\_DEPS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Lname, Fname}}(\text{EMPS\_WITHOUT\_DEPS} * \text{EMPLOYEE})$$

Introduction to Structured Query Language (SQL)

The SQL language may be considered one of the major reasons for the commercial success of relational databases. The SQL language provides a higher-level *declarative* language interface, so the user only specifies *what* the result is to be, leaving the actual optimization and decisions on how to execute the query to the DBMS. The name **SQL** is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUEry Language) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the standard language for commercial relational DBMSs. SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. Hence, it is both a DDL *and* a DML. In addition, it has facilities for defining views

on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls. It also has rules for embedding SQL statements into a general-purpose programming language such as Java, COBOL, or C/C++.

SQL Data Definition and Data Types

SQL uses the terms **table**, **row**, and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*, respectively. We will use the corresponding terms interchangeably. The main SQL command for data definition is the CREATE statement, which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers).

Schema and Catalog Concepts in SQL

- The concept of an SQL schema was incorporated starting with SQL2 in order to group together tables and other constructs that belong to the same database application.
- An **SQL schema** is identified by a **schema name**, and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema.
- Schema **elements** include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema.
- A schema is created via the CREATE SCHEMA statement, which can include all the schema elements' definitions.

For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Jsmith'.

Note that each statement in SQL ends with a semicolon.

- **CREATE SCHEMA** COMPANY **AUTHORIZATION** 'Jsmith';

The CREATE DATABASE Command

- A **Database** is defined as a structured set of data. So, in SQL the very first step to store the data in a well-structured manner is to create a database.
- The **CREATE DATABASE** statement is used to create a new database in SQL.

Syntax:

CREATE DATABASE database_name;

database_name: name of the database.

Example

- CREATE DATABASE university;

The CREATE TABLE Command

- The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints.
- The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL.
- The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command.
- The relations declared through CREATE TABLE statements are called **base tables** (or base relations); this means that the relation and its tuples are actually created and stored as a file by the DBMS.
- Base relations are distinguished from **virtual relations**, created through the CREATE VIEW statement, which may or may not correspond to an actual physical file.

```
CREATE TABLE EMPLOYEE
        ( Fname             VARCHAR(15)         NOT NULL,
          Minit             CHAR,
          Lname             VARCHAR(15)         NOT NULL,
          Ssn               CHAR(9)             NOT NULL,
          Bdate             DATE,
          Address           VARCHAR(30),
          Sex               CHAR,
          Salary            DECIMAL(10,2),
          Super_ssn         CHAR(9),
          Dno               INT                 NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE DEPARTMENT
        ( Dname             VARCHAR(15)         NOT NULL,
          Dnumber           INT                 NOT NULL,
          Mgr_ssn           CHAR(9)             NOT NULL,
          Mgr_start_date    DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber           INT                 NOT NULL,
          Dlocation         VARCHAR(15)         NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname             VARCHAR(15)         NOT NULL,
          Pnumber           INT                 NOT NULL,
          Plocation         VARCHAR(15),
          Dnum              INT                 NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn              CHAR(9)             NOT NULL,
          Pno               INT                 NOT NULL,
          Hours             DECIMAL(3,1)        NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn              CHAR(9)             NOT NULL,
          Dependent_name    VARCHAR(15)         NOT NULL,
          Sex               CHAR,
          Bdate             DATE,
          Relationship      VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Figure 4.1
SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 3.7.

It is important to note that in Figure 4.1, there are some *foreign keys that may cause errors* because they are specified either via circular references or because they refer to a table that has not yet been created.

For example, the foreign key Super_ssn in the EMPLOYEE table is a circular reference because it refers to the table itself. The foreign key Dno in the EMPLOYEE table refers to the DEPARTMENT table, which has not been created yet. To deal with this type of problem, these constraints can be left out of the initial CREATE TABLE statement, and then added later using the ALTER TABLE statement.

Attribute Data Types and Domains in SQL

The basic **data types** available for attributes include numeric, character string, bit string, Boolean, date, and time.

- **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(*i*,*j*)—or DEC(*i*,*j*) or NUMERIC(*i*,*j*)—where *i*, the *precision*, is the total number of decimal digits and *j*, the *scale*, is the number of digits after the decimal point.
- **Character-string** data types are either fixed length—CHAR(*n*) or CHARACTER(*n*), where *n* is the number of characters—or varying length—VARCHAR(*n*) or CHAR VARYING(*n*) or CHARACTER VARYING(*n*), where *n* is the maximum number of characters.
- **Bit-string** data types are either of fixed length *n*—BIT(*n*)—or varying length—BIT VARYING(*n*), where *n* is the maximum number of bits.
- **Boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.
- The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS.

Specifying Constraints in SQL

Specifying Attribute Constraints and Attribute Defaults

- Because SQL allows NULLs as attribute values, a *constraint* **NOT NULL** may be specified if NULL is not permitted for a particular attribute.
- It is also possible to define a *default value* for an attribute by appending the clause **DEFAULT** <value> to an attribute definition. The default value is included in any new

tuple if an explicit value is not provided for that attribute. If no default clause is specified, the default *default value* is NULL for attributes *that do not have* the NOT NULL constraint.

- Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition.

For example, suppose that department numbers are restricted to integer numbers between 1 and 20;

- Dnumber INT **NOT NULL CHECK** (Dnumber > 0 **AND** Dnumber < 21);

Specifying Key and Referential Integrity Constraints

- The **PRIMARY KEY** clause specifies one or more attributes that make up the primary key of a relation. If a primary key has a *single* attribute, the clause can follow the attribute directly.

  For example, the primary key of DEPARTMENT can be specified as follows
  - Dnumber INT **PRIMARY KEY**;
- The **UNIQUE** clause can also be specified directly for a secondary key if the secondary key is a single attribute, as in the following example:
  - Dname VARCHAR(15) **UNIQUE**;
- Referential integrity is specified via the **FOREIGN KEY** clause
  - **FOREIGN KEY** (Super_ssn) **REFERENCES** EMPLOYEE(Ssn)
    ON DELETE SET NULL **ON UPDATE** CASCADE

A referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified. The default action that SQL takes for an integrity violation is to **reject** the update operation that will cause a violation, which is known as the RESTRICT option. However, the schema designer can specify an alternative action to be taken by attaching a **referential triggered action** clause to any foreign key constraint. The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE.

- **ON DELETE SET NULL** - means that if the tuple for a *supervising employee* is *deleted*, the value of Super_ssn is automatically set to NULL for all employee tuples that were referencing the deleted employee tuple.
- **ON UPDATE CASCADE** - if the Ssn value for a supervising employee is *updated* (say, because it was entered incorrectly), the new value is *cascaded* to Super_ssn for all employee tuples referencing the updated employee tuple.

In general, the action taken by the DBMS for SET NULL or SET DEFAULT is the same for both ON DELETE and ON UPDATE: The value of the affected referencing attributes is changed to NULL for SET NULL and to the specified default value of the referencing attribute for SET DEFAULT.

The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the referencing foreign key attribute(s) to the updated (new) primary key value for all the referencing tuples.

Giving Names to Constraints

The names of all constraints within a particular schema must be unique. A constraint name is used to identify a particular constraint in case the constraint must be dropped later and replaced with another constraint. Giving names to constraints is optional.

Specifying Constraints on Tuples Using CHECK

In addition to key and referential integrity constraints, which are specified by special keywords, other *table constraints* can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called **tuple-based** constraints because they apply to each tuple *individually* and are checked whenever a tuple is inserted or modified.

For example, suppose that the DEPARTMENT table had an additional attribute Dept_create_date, which stores the date when the department was created. Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date.

- **CHECK** (Dept_create_date <= Mgr_start_date);

The INSERT Command

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed *in the same order* in which the corresponding attributes were specified in the CREATE TABLE command.

- **INSERT INTO** EMPLOYEE
  VALUES ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak Forest, Katy, TX', 'M', 37000, '653298653', 4);

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command.

- **INSERT INTO** EMPLOYEE (Fname, Lname, Dno, Ssn)
  VALUES ('Richard', 'Marini', 4, '653298653');

The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from only one table at a time. However, the deletion may propagate to tuples in other relations if *referential triggered actions* are specified in the referential integrity constraints of the DDL.
Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all tuples in the relation are to be deleted; however, the table remains in the database as an empty table. We must use the DROP TABLE command to remove the table definition.

- **DELETE FROM** EMPLOYEE
  WHERE Lname='Brown';
- **DELETE FROM** EMPLOYEE
  WHERE Ssn='123456789';
- **DELETE FROM** EMPLOYEE
  WHERE Dno=5;
- **DELETE FROM** EMPLOYEE;

The UPDATE Command

The **UPDATE** command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a *referential triggered action* is specified in the referential integrity constraints of the DDL. An additional **SET** clause in the UPDATE command specifies the attributes to be modified and their new values.

For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively,

- **UPDATE** PROJECT
  SET Plocation = 'Bellaire', Dnum = 5
  WHERE Pnumber=10;

For example, to give all employees in the 'Research' department a 10 percent raise in salary

- **UPDATE** EMPLOYEE
  SET Salary = Salary * 1.1
  WHERE Dno = 5;

The DROP Command

DROP is used to delete a whole database or just a table.

The DROP statement destroys the objects like an existing database, table, index, or view.

A DROP statement in SQL removes a component from a relational database management system (RDBMS).

Syntax

- DROP object object_name

Examples:

- DROP TABLE table_name;

table_name: Name of the table to be deleted.

- DROP DATABASE database_name;

database_name: Name of the database to be deleted.

There are two *drop behavior* options: **CASCADE** and **RESTRICT**.

If the RESTRICT option is chosen, a table is dropped only if it is *not referenced* in any constraints (for example, by foreign key definitions in another relation) or views or by any other elements. With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.

Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the *table definition* from the catalog. If it is desired to delete only the records but to leave the table definition for future use, then the DELETE command should be used instead of DROP TABLE.

The TRUNCATE Command

TRUNCATE statement is a Data Definition Language (DDL) operation that is used to mark the extents of a table for deallocation (empty for reuse). The result of this operation quickly removes all data from a table

Syntax:

- TRUNCATE TABLE table_name;

table_name: Name of the table to be truncated.

- Truncate is normally ultra-fast and it is ideal for deleting data from a temporary table.
- Truncate preserves the structure of the table for future use, unlike drop table where the table is deleted with its full structure.
- Table or Database deletion using DROP statement cannot be rolled back, so it must be used wisely.

The ALTER Command | ADD, DROP, MODIFY

ALTER TABLE is used to add, delete/drop or modify columns in the existing table.

It is also used to add and drop various constraints on the existing table.

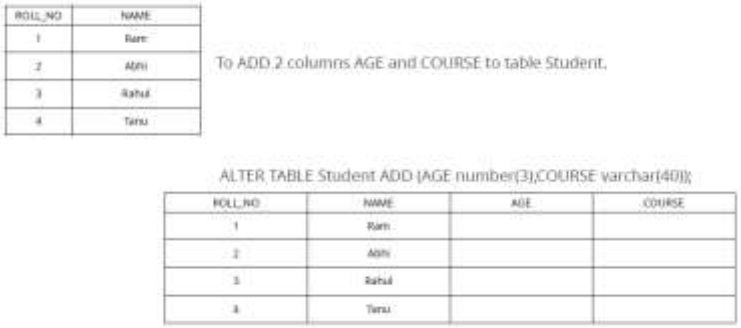
ALTER TABLE – ADD

ADD is used to add columns into the existing table.

Sometimes we may require to add additional information, in that case we do not require to create the whole database again, ADD comes to our rescue.

Syntax

- **ALTER TABLE** table_name
  ADD (Columnname_1 datatype,
  Columnname_2 datatype,
  …
  Columnname_n datatype);



ALTER TABLE – DROP

DROP COLUMN is used to drop column in a table. Deleting the unwanted columns from the table.

Syntax:

- **ALTER TABLE** table_name
  DROP COLUMN column_name;

ROLL_NO	NAME	AGE	COURSE
1	Ram		
2	Abhi		
3	Rahul		
4	Tanu		

```
ALTER TABLE Student DROP COLUMN COURSE;
```

ROLL_NO	NAME	AGE
1	Ram	
2	Abhi	
3	Rahul	
4	Tanu	

ALTER TABLE - MODIFY

It is used to modify the existing columns in a table. Multiple columns can also be modified at once.

Syntax may vary slightly in different databases.

Syntax (Oracle,MySQL,MariaDB):

- **ALTER TABLE** table_name
  MODIFY column_namecolumn_type;

Syntax (SQL Server):

- **ALTER TABLE** table_name
  ALTER COLUMN column_namecolumn_type;

Example

- ALTER TABLE Student MODIFY COURSE varchar(20);

The ALTER Command | RENAME

Sometimes we may want to rename our table to give it a more relevant name. For this purpose we can use ALTER TABLE to rename the name of table. Syntax may vary in different databases.

Syntax (Oracle,MySQL,MariaDB):

- **ALTER TABLE** table_name

  RENAME TO new_table_name;

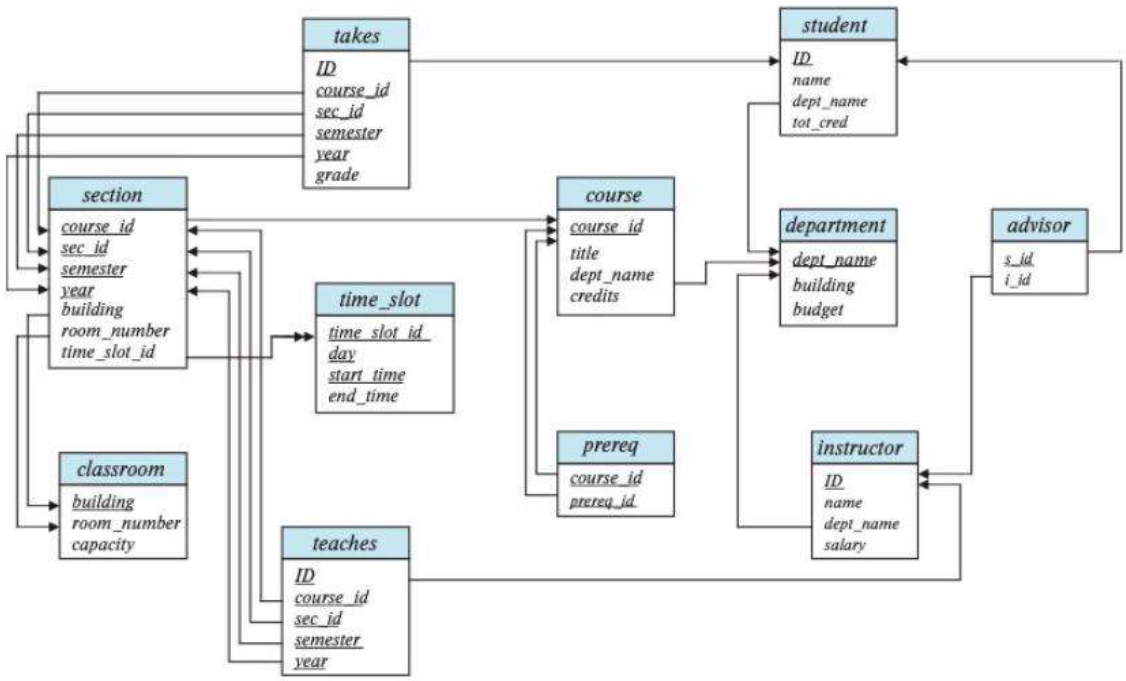
Columns can be also be given new name with the use of ALTER TABLE.

Syntax (MySQL, Oracle):

- **ALTER TABLE** table_name

  RENAME COLUMN old_name TO new_name;

Schema Diagram for University Database




```
create table instructor (
        ID          char(5),
        name        varchar(20) not null,
        dept_name   varchar(20),
        salary      numeric(8,2),
        primary key (ID),
        foreign key (dept_name) references department);
```

```
create table student (
        ID              varchar(5),
        name            varchar(20) not null,
        dept_name       varchar(20),
        tot_cred        numeric(3,0),
        primary key (ID),
        foreign key (dept_name) references department);
```

```
create table takes (
        ID              varchar(5),
        course_id       varchar(8),
        sec_id          varchar(8),
        semester        varchar(6),
        year            numeric(4,0),
        grade           varchar(2),
        primary key (ID, course_id, sec_id, semester, year) ,
        foreign key (ID) references  student,
        foreign key (course_id, sec_id, semester, year) references section);
```

```
create table course (
        course_id       varchar(8),
        title           varchar(50),
        dept_name       varchar(20),
        credits         numeric(2,0),
        primary key (course_id),
        foreign key (dept_name) references department);
```

try it now

A KTU STUDENTS PLATFORM

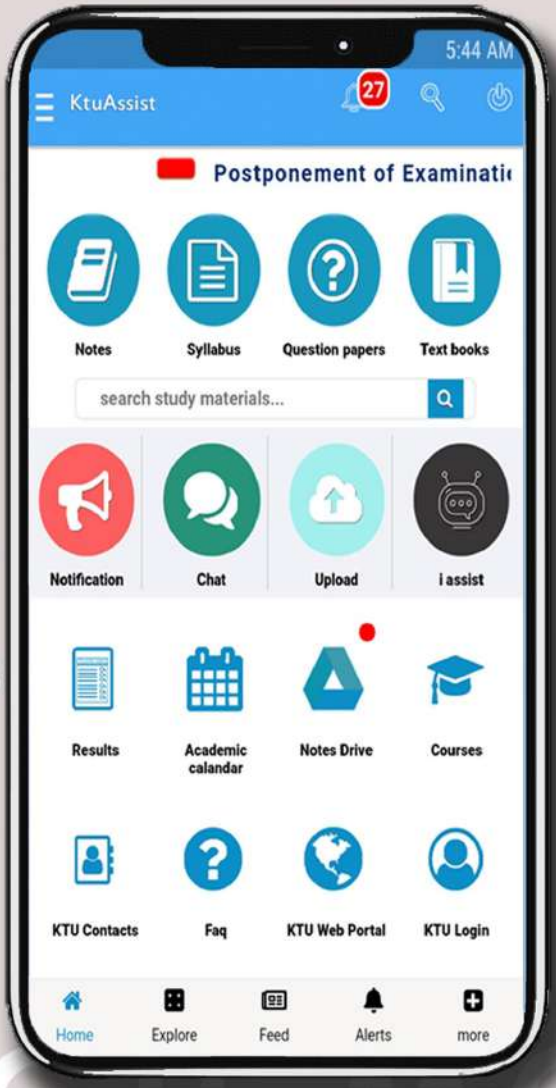
SYLLABUS

NOTES

TEXT BOOKS

QUESTION PAPERS

KTU NOTIFICATION



DOWNLOAD IT FROM GOOGLE PLAY

CHAT

FAQ

LOGIN

CALENDAR

MUCH MORE

DOWNLOAD APP

ktuassist.in

instagram.com/ktu_assist

facebook.com/ktuassist