CHAPTER 6

RELATIONAL DATA MODEL AND RELATIONAL ALGEBRA
The relational model represents the database as a collection of relations.

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.
A **domain** $D$ is a set of atomic values.

By atomic we mean that each value in the domain is indivisible.

A **relation schema** $R$, denoted by $R(A_1, A_2, \ldots, A_n)$, is made up of a relation name $R$ and a list of attributes $A_1, A_2, \ldots, A_n$.

Each attribute $A_i$ is the name of a role played by some domain $D$ in the relation schema $R$.

$D$ is called the domain of $A_i$ and is denoted by $\text{dom}(A)$. 
Characteristics of Relations

- **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.
  - Tuple ordering is not part of a relation definition, because a relation attempts to represent facts at a logical or abstract level.

- **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.
However, at a logical level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained.

An alternative definition of a relation can be given as relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a set of attributes, and a relation state $r(R)$ is a finite set of mappings $r = \{t_1, t_2, \ldots, t_m\}$, where each tuple $t_1$ is a mapping from $R$ to $D$, and $D$ is the union of the attribute domains; that is, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \ldots \cup \text{dom}(A_n)$. 
• **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.
  - An important concept is that of nulls, 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 for these cases.
Characteristics of Relations

Interpretation (Meaning) of a Relation

- The relation schema can be interpreted as a declaration or a type of assertion.
- For example, the schema of the STUDENT relation asserts that, in general, a student entity has a Name, SSN, HomePhone, Address, OfficePhone, Age, and Performance.
- Notice that some relations may represent facts about entities, whereas other relations may represent facts about relationships.
Relational Model Notation

- A relation schema $R$ of degree $n$ is denoted by $R(A_1, A_2, \ldots, A_n)$.  
- An $n$-tuple $t$ in a relation $r(R)$ is denoted by $t = <V_1, V_2, \ldots, V_n>$, where $V_i$ is the value corresponding to attribute $A_i$.  
- The letters $Q, R, S$ denote relation names.  
- The letters $q, r, s$ denote relation states.  
- The letters $t, u, V$ denote tuples.
In general, the name of a relation schema such as
STUDENT also indicates the current set of tuples in
that relation—the current relation state—whereas
STUDENT(Name, SSN, ...) refers only to the relation
schema.

An attribute A can be qualified with the relation
name R to which it belongs by using the dot notation
R.A—for example, STUDENT.Name or
STUDENT.Age.
Constraints on databases can generally be divided into three main categories:

- Constraints that are inherent in the data model. We call these inherent model based constraints.
- Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL. We call these schema-based constraints.
- 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 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, in some cases, money data types.
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.

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

\[
t_1[SK] \neq t_2[SK]
\]

Any such set of attributes SK is called a *superkey* of the relation schema R.
A key satisfies two constraints:

- Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key.
- 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.

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: LicenseNumber and EngineSerialNumber.
Entity Integrity, Referential Integrity, and Foreign Keys

- 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.
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.

To define referential integrity more formally, we first define the concept of a *foreign key*.
A set of attributes FK in relation schema R1 is a foreign key of R2 that references relation R2 if it satisfies the following two rules:

- 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.
- A value of FK in a tuple $t_1$ of the current state $r_1$ (R1) either occurs as a value of PK for some tuple $t_2$ in the current state $r_2$ (R2) or is null.
Other Types of Constraints

- The preceding integrity constraints do not include a large class of general constraints, sometimes called *semantic integrity constraints*, which may have to be specified and enforced on a relational database.
- Examples of such constraints are "the salary of an employee should not exceed the salary of the employee's supervisor" and "the maximum number of hours an employee can work on all projects per week is 56".
- Mechanisms called *triggers* and *assertions* can be used.
The types of constraints we discussed so far may be called **state constraints**, because they define the constraints that a valid state of the database must satisfy.

Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database.

An example of a transition constraint is: "the salary of an employee can only increase."
There are three basic update operations on relations: insert, delete, and modify. Insert is used to insert a new tuple or tuples in a relation, Delete is used to delete tuples, and Update (or Modify) is used to change the values of some attributes in existing tuples.
The Insert Operation

- The Insert operation provides a list of attribute values for a new tuple \( t \) that is to be inserted into a relation \( R \).
- Insert can violate any of the four types of constraints
- Insert \(<'Cecilia', 'F', 'Kolonsky', null, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, null, 4>\) into EMPLOYEE.
  - This insertion violates the entity integrity constraint (null for the primary key SSN), so it is rejected.
- Insert \(<'Alicia', 'I', 'Zelaya', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4>\) into EMPLOYEE.
  - This insertion violates the key constraint because another tuple with the same SSN value already exists in the EMPLOYEE relation, and so it is rejected.
The Insert Operation

- Insert <Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windswept, Katy, TX', F, 28000, '987654321', 7> into EMPLOYEE.
  - This insertion violates the referential integrity constraint specified on DNO because no DEPARTMENT tuple exists with DNUMBER = 7.
- Insert <Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, null, 4> into EMPLOYEE.
  - This insertion satisfies all constraints, so it is acceptable.
- If an insertion violates one or more constraints, the default option is to reject the insertion.
The Delete Operation

- The Delete operation can violate only referential integrity, if the tuple being deleted is referenced by the foreign keys from other tuples in the database.
- To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.
- **Delete the WORKS_ON tuple with ESSN = '999887777' and PNO = 10.**
  - This deletion is acceptable.
- **Delete the EMPLOYEE tuple with SSN = '999887777'.**
  - This deletion is not acceptable, because tuples in ...refer to this tuple. Hence, if the tuple is deleted, referential integrity violations will result.
Several options are available if a deletion operation causes a violation.

- The first option is to reject the deletion.
- The second option is to attempt to cascade (or propagate) the deletion by deleting tuples that reference the tuple that is being deleted.
- A third option is to modify the referencing attribute values that cause the violation; each such value is either set to null or changed to reference another valid tuple.

Combinations of these three options are also possible.
The Update Operation

- The Update (or Modify) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation R.

- For Example:
  - Update the SALARY of the EMPLOYEE tuple with SSN = '999887777' to 28000.
  - Update the DNO of the EMPLOYEE tuple with SSN = '999887777' to 1.

- Updating an attribute that is neither a primary key nor a foreign key usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain.
The Update Operation

-Modifying a primary key value is similar to deleting one tuple and inserting another in its place, because we use the primary key to identify tuples.
-If a foreign key attribute is modified, the DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is null).
-Update the DNO of the EMPLOYEE tuple with SSN = '999887777' to 7.
  ○ Unacceptable, because it violates referential integrity.
-Update the SSN of the EMPLOYEE tuple with SSN = '999887777' to '987654321'.
  ○ Unacceptable, because it violates primary key and referential integrity constraints.
Review of Previous Class

- RELATIONAL MODEL CONSTRAINTS
- Entity Integrity, Referential Integrity, and Foreign Keys
- UPDATE OPERATIONS AND DEALING WITH CONSTRAINT VIOLATIONS
  - Insert Operation
  - Delete Operation
  - Update Operation
The basic set of operations for the relational model is the relational algebra.

These operations enable a user to specify basic retrieval requests.

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).
The relational algebra is very important for several reasons.

First, it provides a formal foundation for relational model operations.

Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in relational database management systems (RDBMSs).

Third, some of its concepts are incorporated into the SQL standard query language for RDBMSs.
The SELECT operation is used to select a *subset* of the tuples from a relation that satisfy a selection condition.

One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition.

The SELECT operation can also be visualized as a *horizontal partition*. 
For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

- \( \sigma_{\text{DNO}=4}(\text{EMPLOYEE}) \)
- \( \sigma_{\text{SALARY}>30000}(\text{EMPLOYEE}) \)
The SELECT Operation

- In general, the SELECT operation is denoted by
  \( \sigma <\text{selection condition}> (R) \)
  - where the symbol \( \sigma \) (sigma) is used to denote the SELECT operator, and the selection condition is a Boolean expression specified on the attributes of relation R.

- For example, to select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

- \( \sigma (\text{DNO}=4 \ \text{AND}\ \text{SALARY}>25000) \ \text{OR} \ (\text{DNO}=5 \ \text{AND}\ \text{SALARY} > 30000) \) (EMPLOYEE)
The SELECT Operation

- The <selection condition> is applied independently to each tuple \( t \) in \( R \).
- This is done by substituting each occurrence of an attribute \( A_i \) in the selection condition with its value in the tuple \( t[A_i] \).
- If the condition evaluates to TRUE, then tuple \( t \) is selected. All the selected tuples appear in the result of the SELECT operation.
The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:

- \((\text{cond}_1 \text{ AND } \text{cond}_2)\) is TRUE if both \(\text{cond}_1\) and \(\text{cond}_2\) are TRUE; otherwise, it is FALSE.
- \((\text{cond}_1 \text{ OR } \text{cond}_2)\) is TRUE if either \(\text{cond}_1\) or \(\text{cond}_2\) or both are TRUE; otherwise, it is FALSE.
- \((\text{NOT cond})\) is TRUE if cond is FALSE; otherwise, it is FALSE. The SELECT operator is unary; that is, it is applied to a single relation.
The PROJECT Operation

- 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 result of the PROJECT operation can hence be visualized as a *vertical partition*. 
The general form of the PROJECT operation is:

\[ \pi <\text{attribute list}> (R) \]

- where \( \pi \) (pi) is the symbol used to represent the PROJECT operation, and <attribute list> is the desired list of attributes from the attributes of relation R.
- The result of the PROJECT operation has only the attributes specified in <attribute list> in the same order as they appear in the list.

For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

\[ \pi \text{ LNAME, FNAME, SALARY}( \text{EMPLOYEE}) \]
If the attribute list includes only non key attributes of R, duplicate tuples are likely to occur.

The PROJECT operation removes any duplicate tuples, so the result of the PROJECT operation is a set of tuples, and hence a valid relation.

This is known as duplicate elimination. For example, consider the following PROJECT operation:

\[ \pi \text{ SEX, SALARY( EMPLOYEE)} \]
Sometimes, we may want to apply several relational algebra operations one after the other.

Either 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.
Sequences of Operations and RENAME Operation

- 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 as follows:
  \[
  \pi_{\text{FNAME}, \text{LNAME}, \text{SALARY}}(\sigma_{\text{DNO}=5} (\text{EMPLOYEE}))
  \]
- Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation:
  \[
  \text{DEP5_EMPS} \leftarrow \sigma_{\text{DNO}=5} (\text{EMPLOYEE})
  \]
  \[
  \text{RESULT} \leftarrow \pi_{\text{FNAME}, \text{LNAME}, \text{SALARY}} (\text{DEP5_EMPS})
  \]
It is often simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression.

We can also use this technique to rename the attributes in the intermediate and result relations.

To rename the attributes in a relation, we simply list the new attribute names in parentheses, as in the following example:

- \( \text{TEMP} \leftarrow \sigma_{\text{DNO}=5} (\text{EMPLOYEE}) \)
- \( R (\text{FIRSTNAME, LASTNAME, SALARY}) \leftarrow \pi_{\text{FNAME, LNAME, SALARY}} (\text{TEMP}) \)
The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

- $\rho_{S(B_1, B_2, \ldots, B)}(R)$
- $\rho_S(R)$
- $\rho_{(B_1, B_2, \ldots, B_n)}(R)$

where the symbol $\rho$ (rho) is used to denote the RENAME operator, S is the new relation name, and $B_1$, $B_2$, $\ldots$, $B_n$ 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.
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).

These are binary operations; that is, each is applied to two sets (of tuples).

When these 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.
Two relations $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_n)$ are said to be union compatible if they have the same degree $n$ and if $\text{dom}(A) = \text{dom}(B)$ for $1 \leq i \leq n$.

This means that the two relations have the same number of attributes, and each corresponding pair of attributes has the same domain.
We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations Rand S as follows:

- **UNION**: The result of this operation, denoted by R U S, is a relation that includes all tuples that are either in R or in S or in both Rand 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 Rand 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 CARTESIAN PRODUCT operation—also known as CROSS PRODUCT or CROSS JOIN—which is denoted by \( \times \).

This is also a binary set operation, but the relations on which it is applied do not have to be union compatible.

This operation is used to combine tuples from two relations in a combinatorial fashion.

In general, the result of \( R(A_1,, A_2, ... , A_n) \times S(B_1,B_2, ... , B_m) \) is a relation \( Q \) with degree \( n + m \) attributes \( Q(A_1,A_2 ... , A_n, B_1, B_2, ... , B_m) \), in that order.
For example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

- $\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{SEX}=\text{'F'}} (\text{EMPLOYEE})$
- $\text{EMPNAMES} \leftarrow \pi_{\text{FNAME}, \text{LNAME}, \text{SSN}} (\text{FEMALE\_EMPS})$
- $\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$
- $\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{SSN}=\text{ESSN}} (\text{EMP\_DEPENDENTS})$
- $\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME, DEPENDENT\_LNAME}} (\text{ACTUAL\_DEPENDENTS})$
The JOIN Operation

The JOIN operation, is used to combine *related tuples* from two relations into single 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.

The general form of a JOIN operation on two relations $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is

$$ R \bowtie <\text{join condition}> S $$
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 MGRSSN 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 ← DEPARTMENT ⨝ MGRSSN=SSN EMPLOYEE
RESULT ← π DNAME, LNAME, FNAME (DEPT_MGR)
```
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**.

Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple.
For example the values of the attributes MGRSSN and SSN are identical in every tuple of DEPT_MGR query because of the equality join condition specified on these two attributes.

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.

In the following example, we first rename the DNUMBER attribute of DEPARTMENT to DNUM-so that it has the same name as the DNUM attribute in PROJECT-and then apply NATURAL JOIN:

```sql
PROJ_DEPT ← PROJECT * ρ_{DNAME,DNUM,MGRSSN,MGRSTARTDATE}(DEPARTMENT)
```
The attribute DNUM is called the **join attribute**.

- If the attributes on which the natural join is specified already **have the same names in both relations**, renaming is unnecessary.

- As we can see, the JOIN operation is used to combine data from multiple relations so that related information can be presented in a single table.

- Note that sometimes a join may be specified between a relation and itself such operations are also known as **inner joins**.
The DIVISION operation, denoted by $\div$, 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 'JohnSmith' works on in the intermediate relation SMITH_PNOS:
Next, create a relation that includes a tuple \(<PNO, ESSN>\) whenever the employee whose social security number is ESSN works on the project whose number is PNO in the intermediate relation SSN_PNOS:

\[
\text{SSN\_PNOS} \leftarrow \Pi_{E, F} (\text{WORKS\_ON})
\]
Finally, apply the DIVISION operation to the two relations, which gives the desired employees' social security numbers:

\[
\text{SSNS}(\text{SSN}) \leftarrow \text{SSN\_PNOS} \div \text{SMITH\_PNOS}
\]

\[
\text{RESULT} \leftarrow \pi_{\text{FNAME, LNAME}}(\text{SSNS} \times \text{EMPLOYEE})
\]
The request that cannot be expressed in the basic relational algebra can be expressed in mathematical aggregate functions on collections of values from the database.

Examples of such functions include retrieving the average or total salary of all employees or the total number of employee tuples.

These functions are used in simple statistical queries that summarize information from the database tuples.

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 $\overline{\text{F}}$ (pronounced “script F”), to specify these types of requests as follows:

$\langle\text{grouping attributes}\rangle \overline{\text{F}} \langle\text{function list}\rangle (R)$

where $\langle\text{grouping attributes}\rangle$ is a list of attributes of the relation specified in R, and $\langle\text{function list}\rangle$ is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT.

For Example: The query get the DNO, Count of Employees and Average salary of the employees.

\[\rho_{R(DNO, NO\_OF\_EMPLOYEES, AVERAGE\_SAL)}(DNO \overline{\text{F}} COUNT(SSN: AVERAGE\_SALARY(EMPLOYEE))}\]
The JOIN operations described earlier match tuples that satisfy the join condition.

For example, for a NATURAL JOIN operation \( R \times S \), only tuples from \( R \) that have matching tuples in \( S \) — and vice versa — appear in the result.

Hence, tuples without a *matching* (or *related*) tuple are eliminated from the JOIN result.

Tuples with null values in the join attributes are also eliminated. This amounts to loss of information.
A set of operations, called **outer joins**, can be used when we want 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.

This satisfies the need of queries in which tuples from two tables are to be combined by matching corresponding rows, but without losing any tuples for lack of matching values.
• For example, suppose that we want a list of all employee names and also the name of the departments they manage *if they happen* to manage a department; if they do not manage any, we can so indicate with a null value.

• We can apply an operation LEFT OUTER JOIN, denoted by, to retrieve the result as follows:

\[
\text{RESULT} \leftarrow \pi_{\text{FNAME, MINIT, LNAME, DNAME}} (\text{TEMP})
\]

\[
\text{TEMP} \leftarrow (\text{EMPLOYEE} \bowtie_{\text{SSN} = \text{MGRSSN}} \text{DEPARTMENT})
\]
Outer Join Operation

- The **LEFT OUTER JOIN** operation keeps every tuple in the *first*, or *left*, relation R in R SJ 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.
**Step 1: Mapping of Regular Entity Types.**
- For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E.
- Include only the simple component attributes of a composite attribute.
- Choose one of the key attributes of E as primary key for R. If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of R.
- In our example, we create the relations EMPLOYEE, DEPARTMENT, and PROJECT in Figure to correspond to the regular entity types EMPLOYEE, DEPARTMENT, and PROJECT.
RELATIONAL DATABASE DESIGN USING ER-TO-RELATIONAL MAPPING

Relational Data Model and Relational Algebra
The foreign key and relationship attributes, if any, are not included yet; they will be added during subsequent steps.

These include the attributes SUPERSSN and DNO of EMPLOYEE, MGRSSN and MGRSTARTDATE of DEPARTMENT, and DNUM of PROJECT.

In our example, we choose SSN, DNUMBER, and PNUMBER as primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT.

The relations that are created from the mapping of entity types are sometimes called entity relations because each tuple (row) represents an entity instance.
Step 2: Mapping of Weak Entity Types:

For each weak entity type $W$ in the ER schema with owner entity type $E$, create a relation $R$ and include all simple attributes of $W$, as attributes of $R$.

In addition, include the primary key attribute that correspond to the owner entity type(s) as foreign key attributes of $R$; this takes care of the identifying relationship type of $W$.

The primary key of $R$ is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type $W$, if any.
In our example, we create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT.

We include the primary key SSN of the EMPLOYEE relation-which corresponds to the owner entity type-as a foreign key attribute of DEPENDENT; we renamed it ESSN, although this is not necessary.

The primary key of the DEPENDENT relation is the combination {ESSN, DEPENDENT_NAME} because DEPENDENT_NAME is the partial key of DEPENDENT.
Step 3: Mapping of Binary 1:1 Relationship Types.

For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R.

There are three possible approaches:

- (1) the foreign key approach,
- (2) the merged relationship approach, and
- (3) the cross-reference or relationship relation approach.
• **Foreign key approach:**
  - Choose one of the relations-S, and include as a foreign key in S the primary key of T.
  - It is better to choose an entity type with *total participation* in R in the role of S.
  - Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S.
Merged relation option:

- An alternative mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation.
- This may be appropriate when both participations are total.
• **Cross-reference or relationship relation option:**
  - The third alternative is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types.
  - This approach is required for binary M:N relationships.
  - The relation R is called a relationship relation, (or sometimes a lookup table), because each tuple in R represents a relationship instance that relates one tuple from S with one tuple of T.
**Step 4: Mapping of Binary 1: N Relationship Types:**

- For each regular binary 1:N relationship type R, identify the relation S that represents the participating entity type at the N-side of the relationship type.
- Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R.
- This is done because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type.
- Include any simple attributes of the 1:N relationship type as attributes of S.
In our example, we now map the 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION from Figure A.

- For WORKS_FOR we include the primary key DNUMBER of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it DNO.
- For SUPERVISION we include the primary key of the EMPLOYEE relation as foreign key in the EMPLOYEE relation itself because the relationship is recursive-and call it SUPERSSN.
- The CONTROLS relationship is mapped to the foreign key attribute DNUM of PROJECT, which references the primary key DNUMBER of the DEPARTMENT relation.
Step 5: Mapping of Binary M:N Relationship Types:

For each binary M:N relationship type R, create a new relation S to represent R.

Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S.

Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S.

Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for 1:1 or I:N relationship types) because of the M:N cardinality ratio; we must create a separate relationship relation S.
In our example, we map the M:N relationship type WORKS_ON from Figure A by creating the relation WORKS_ON in Figure B.

We include the primary keys of the PROJECT and EMPLOYEE relations as foreign keys in WORKS_ON and rename them PNO and ESSN, respectively.

We also include an attribute HOURS in WORKS_ON to represent the HOURS attribute of the relationship type.

The primary key of the WORKS_ON relation is the combination of the foreign key attributes \{ESSN, PNO\}.
Step 6: Mapping of Multivalued Attributes:
For each multivalued attribute A, create a new relation R.

- This relation R will include an attribute corresponding to A, plus the primary key attribute K-as a foreign key in R-of the relation that represents the entity type or relationship type that has A as an attribute.
- The primary key of R is the combination of A and K.
- If the multivalued attribute is composite, we include its simple components.
In our example, we create a relation DEPT_LOCATIONS.
The attribute DLOCATION represents the multivalued attribute LOCATIONS of DEPARTMENT, while DNUMBER-as foreign key represents the primary key of the DEPARTMENT relation.
The primary key of DEPT_LOCATIONS is the combination of {DNUMBER, DLOCATION}.
A separate tuple will exist in DEPT_LOCATIONS for each location that a department has.
### Sahaj Computer Solutions

<table>
<thead>
<tr>
<th>Address:</th>
<th>Become a Master in</th>
</tr>
</thead>
</table>
| 3b, Hilltop Residency, Girnar Hills, Opp Gomatesh School, Hindwadi, Belgaum-590001 Ph:0831-2481703, 4200864 | - Software Development  
- Languages  
- Technologies  
- Web Development  
- Animation  
- And many more Technologies |

 Relay the data to the next page for continuous reading.