7.1 Relational Model Concepts

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. However, there are important differences between relations and files..

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

Ex: STUDENT is called table because each row represents facts about a particular student entity. The column names—Name, StudentNumber, Class, 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 called a *domain*. We now define these terms—*domain, tuple, attribute,* and *relation*—more precisely.

7.1.1 Domains, Attributes, Tuples, and Relations

A **domain** D is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the relational model is concerned. 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 10-digit phone numbers valid in the United States.
- Social_security_numbers: The set of valid 9-digit social security numbers.
- Names: The set of names of persons.
- Employee_ages: Possible ages of employees of a company; each must be a value between 15 and 80 years old.

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 Employee_ages is an integer number between 15 and 80. A domain is thus given a name, data type, and format.

Relation schema:

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 $dom(A_i)$. A relation schema is used to *describe* a relation; R is called the **name** of this relation. The **degree** of a relation is the number of attributes n of its relation schema.

An example of a relation schema for a relation of degree 7, which describes university students, is the following:

STUDENT(Name, SSN, HomePhone, Address, OfficePhone, Age, GPA)

For this relation schema, STUDENT is the name of the relation, which has seven attributes. We can specify the following domains for some of the attributes of the STUDENT relation:

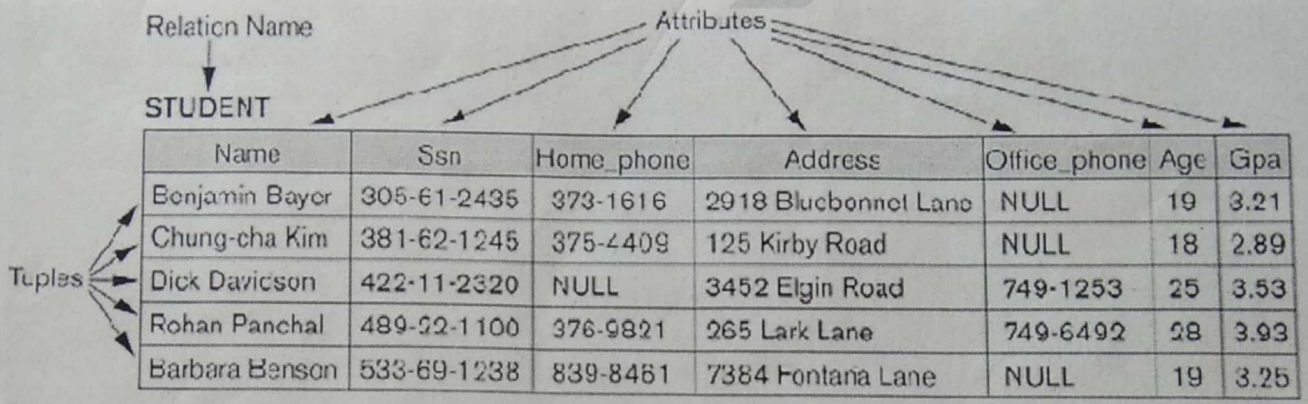
dom(Name) = Names;

dom(SSN) = Social_security_numbers etc.

Relation state:

A **relation** (or **relation state**) r of the relation schema $R(A_1, A_2, \ldots, A_n)$, also denoted by r(R), is a set of n-tuples $r = \{t_1, t_2, \ldots, t_m\}$. Each **n-tuple** t is an ordered list of n values $t = \langle v_1, v_2, \ldots, v_n \rangle$, where each value v_i is the value corresponding to Ai is an element of $dom(A_i)$ or is a special **null** value. The i^{th} value in tuple t, which corresponds to the attribute A_i, is referred to as $t[A_i]$. The terms relation **intension** for the schema R and relation **extension** for a relation state r(R) are also commonly used.

Figure below shows an example of a STUDENT relation, which corresponds to the STUDENT schema specified above. Each tuple in the relation represents a particular student entity. We display the relation as a table, where each tuple is shown as a row and each attribute corresponds to a column header indicating a role or interpretation of the values in that column. *Null values* represent attributes whose values are unknown or do not exist for some individual STUDENT tuples.



Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	749-1253	25	3.53
Rohan Panchal	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	NULL	19	3.25

Figure 5.1
The attributes and tuples of a relation STUDENT.

7.1.2 Characteristics of Relations

a)Ordering of Tuples in a Relation

b)Ordering of Values within a Tuple, and an Alternative Definition of a Relation

c)Values in the Tuples

d)Interpretation of a Relation

a) 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 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. The definition of a relation does not specify any order: there is *no preference* for one logical ordering over another.

b)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 definition—is important. However, at a logical level, the order of attributes and their values are *not* really important as long as the correspondence between attributes and values is maintained.

An **alternative definition** of a relation can be given. A relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a *set* of attributes, and a relation r(R) is a finite set of **mappings** $r = \{t_1, t_2, \ldots, t_m\}$, where each tuple t_i is a mapping from R to D, and D is the union of the attribute domains; that is, $D = dom(A_1)\ D\ dom(A_2)\ D\ldots D\ dom(A_n)$.

According to this definition, a **tuple** can be considered as a **set** of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from $dom(A_i)$. The ordering of attributes is *not* important, because the attribute name appears with its value.

c)Values 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. Multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes.

The values of some attributes within a particular tuple may be unknown or may not apply to that tuple. A special value, called **null,** is used for these cases. A table 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 types* of null values, such as "value unknown," "value exists but not available," or "attribute does not apply to this tuple." It is possible to devise different codes for different types of null values.

d)Interpretation 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 GPA. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion.

For example, the first tuple in Student table asserts the fact that there is a STUDENT whose name is Benjamin Bayer, SSN is 305-61-2435, Age is 19, and so on.

Notice that some relations may represent facts about *entities,* whereas other relations may represent facts about *relationships.*

7.1.3 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 following notation refers to **component values** of tuples:
  - Both $t[A_i]$ and $t.A_i$ refer to the value v_i in t for attribute A_i.
  - Both $t[A_u, A_w, \ldots, A_z]$ and $t.(A_u, A_w, \ldots, A_z)$, where $A_u, A_w, \ldots, A_z$ is a list of attributes from R, refer to the subtuple of values $<v_u, v_w, \ldots, v_z>$ from t corresponding to the attributes specified in the list.
- 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. This is because the same name may be used for two attributes in different relations. However, all attribute names *in a particular relation* must be distinct.

7.2 Relational Constraints and Relational Database Schemas

7.2.1 Domain Constraints

Domain constraints specify that 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, long-integer) and real numbers (float and double-precision float). Characters, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money data types.

7.2.2 Key Constraints and Constraints on Null

A *relation* is defined as a *set of tuples.* By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **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

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

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 a 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,

Ex: Any set of attributes that includes usn—for example, {usN, Name, Age}—is a superkey.

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. Hence, a key is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold.

For example, consider the STUDENT relation .The attribute set {usn} is a key of STUDENT because no two student tuples can have the same value for usn. However, the superkey {usN, Name, Age} is not a key of STUDENT, because removing Name or Age or both from the set still leaves us with a superkey.

The value of a key attribute can be used to identify uniquely each tuple in the relation.

For example, the usn value identifies uniquely the tuple corresponding to Benjamin Bayer in the STUDENT relation. we cannot and should not designate the Name attribute of the STUDENT relation as a key, because there is no guarantee that two students with identical names will never exist .

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: LicenseNumber and EngineSerialNumber. It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined .

Another constraint on attributes specifies whether null values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-null value for the Name attribute, then Name of STUDENT is constrained to be **NOT NULL.**

7.2.3 Relational Databases and Relational Database Schemas

. A **relational database schema** S is a set of relation schemas $S = \{R_1, R_2, \ldots, R_m\}$ and a set of **integrity constraints** IC.

A **relational database state** DB of S is a set of relation states $DB = \{r_1, r_2, \ldots, r_m\}$ such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC. A relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}.

Integrity constraints are specified on a database schema and are expected to hold on every database state of that schema. In addition to domain and key constraints, two other types of constraints are considered part of the relational model: entity integrity and referential integrity.

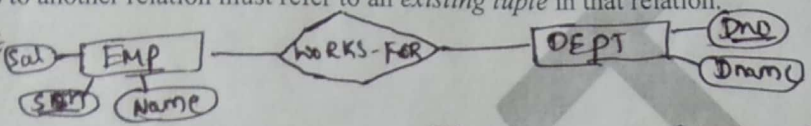
7.2.4 Entity Integrity, Referential Integrity, and Foreign Keys

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. For example, if two or more tuples had null for their primary keys, we might not be able to distinguish them.

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

The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples of 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 ex:



EMP (SSN, Name, sal, DNO) FK DEPT(Dno, Dname)

Dno in EMP represents a foreign key. The values that is indicated in Dno must be present in DEPT Dno.

DEPT

Dno	Dname
1	CS
2	IS

So in EMP Dno can take only the value 1 or 2.

EMP

SSN	Name	sal	Dno
1234	Aswath	1200	1
1235	Bharath	2000	2

} valid data

1235	chetan	2500	(3)

Invalid for Dno because it refers the content of DEPT Dno. There is no deptno 3 in DEPT table.

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, we first 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 R_1 and R_2. A set of attributes FK in relation schema R_1 is a **foreign key** of R_1 that references relation R_2 if it satisfies the following two rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R_2; the attributes FK are said to **reference** or **refer to** the relation R_2.
2. A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state $r_2(R_2)$ or is null. In the former case, we have $t_1[FK] = t_2[PK]$, and we say that the tuple t_1 **references** or **refers to** the tuple t_2. R_1 is called the **referencing relation** and R_2 is the **referenced relation.**

Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas. For example, consider the database .In the EMPLOYEE relation, the attribute DNO refers to the department for which an employee works; hence, we designate DNO to be a foreign key of EMPLOYEE, referring to the DEPARTMENT relation. This means that a value of DNO in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT—the DNUMBER attribute—in some tuple t_2 of the DEPARTMENT relation, or the value of DNO *can be null* if the employee does not belong to a department.

Notice that a foreign key can *refer to its own relation.* For example, the attribute SUPERSSN in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, SUPERSSN is a foreign key that references the EMPLOYEE relation itself.

7.3 Update Operations and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates.* we concentrate on the update operations. There are three basic update operations on relations:

(1) insert,

(2) delete, and

(3) 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. Whenever update operations are applied, the integrity constraints specified on the relational database schema should not be violated.

7.3.1 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 discussed in the previous section. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain. Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation r(R). Entity integrity can be violated if the primary key of the new tuple t is null. Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation.

Here are some examples to illustrate this discussion.

Ex: i) Insert <(null), Abhi, 2000, 1> into EMP
Insertion violates the entity integrity constraint. So it is rejected.

ii) Insert <(1235), Ganesh, 25000, 2> into EMP
Insertion violates key constraint because another tuple with the same SSN value already exists. So it is rejected.

iii) Insert <1238, Geetha, 2000, (3)>
Insertion violates Referential Integrity constraint

iv) Insert <(KA1563), Hemanth, 25000, 2>
violates Domain because domain of SSN is integer.

If an insertion violates one or more constraints, the default option is to *reject the insertion.* In this case, it would be useful if the DBMS could explain to the user why the insertion was rejected.

7.3.2 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.

i) Delete EMP with SSN = 1235 is acceptable.

ii) Delete DEPT with DNO = 1
violates referential integrity because the value is used by tuple t_1 of EMP

Ex: EMP

SSN	Name	Sal	DNO
1234	Ashish	1200	1
1235	Bharath	2000	2

FK

DEPT

PK

DNO	Dname
1	CS
2	IS

Three options are available if a deletion operation causes a violation
First option is to reject the deletion
second option is to cascade the deletion that is by deleting tuples that reference the tuple,

Third option is to modify the referential attribute value that causes the violation each such value is set to NULL or changed to reference another valid tuple.

Delete EMP where SSN = 1234 cascade;
The above statement deletes one tuple from EMP where SSN is 1234.

EMP

SSN	Name	Sal	DNO
1235	Bharath	2000	2

Three 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. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with ESSN = '999887777'.

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.

7.3.3 The Update Operation

The **Update** operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation R. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Here are some examples.

i) update Salary of the Emp tuple with ssn=1234 to 25000
   └ This is acceptable

ii) update Dno of Emp where ssn=1234 to 3
   violates referential integrity constraint.

iii) update Dno of Emp to 'A'
   violate domain constraint

iv) update ssn to NULL where ssn=1234
   violates Entity integrity constraint.

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. 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.

7.4 Basic Relational Algebra Operations

7.4.1 The SELECT Operation

7.4.2 The PROJECT Operation

7.4.3 Sequences of Operations and the RENAME Operation

7.4.4 Set Theoretic Operations

7.4.5 The JOIN Operation

.4.6 A Complete Set of Relational Algebra Operations

7.4.7 The DIVISION Operation

A basic set of relational model operations constitute the **relational algebra.** These operations enable the user to specify basic retrieval requests. The result of a retrieval is a new relation, which may have been formed from one or more relations. The algebra operations thus produce new relations, which can be further manipulated using operations of the same algebra. A sequence of relational algebra operations forms a **relational algebra expression,** whose result will also be a relation.

The relational algebra operations are usually divided into two groups. One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples. Set operations include UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT. The other group consists of operations developed specifically for relational databases; these include SELECT, PROJECT, and JOIN, among others. The SELECT and PROJECT operations are discussed first, because they are the simplest. Then we discuss set operations.

.

7.4.1 The SELECT Operation

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 general SELECT operation is denoted by

$$\sigma_{\langle \text{selection condition} \rangle}(R)$$

where the symbol σ (sigma) is used to denote the SELECT operator, and the selection condition is a Boolean expression specified on the attributes of relation R. Notice that R is generally a *relational algebra expression* whose result is a relation; the simplest expression is just the name of a database relation. The relation resulting from the SELECT operation has the *same attributes* as R. The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

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

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

where <attribute name> is the name of an attribute of R, <comparison op> is normally one of the operators {=, <, ≥, ≤, ≠ >, , }, and <constant value> is a constant value from the attribute domain. Clauses can be arbitrarily connected by the Boolean operators AND, OR, and NOT to form a general selection condition.

Ex: Emp

SSN	Name	Address	Sal
5642	Adarsh	Bng	20000
5643	Bharath	MYS	15000
5644	chetan	Bng	18000
5645	Ranjan	MYS	21000

i) $\sigma_{(Address = 'Brg')}(EMP)$

output:

SSN	Name	Address	Sal
5642	Adarsh	Brg	20000
5644	chetan	Brg	18000

ii) $\sigma_{(Address = 'Brg' \text{ and } sal > 18000)}(EMP)$

SSN	Name	Address	Sal
5642	Adarsh	Brg	20000

The SELECT operator is unary; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually;* hence, selection conditions cannot involve more than one tuple. The **degree** of the relation resulting from a SELECT operation is the same as that of R.

7.4.2 The PROJECT Operation

If we think of a relation as a table, the SELECT operation selects 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 syntax of PROJECT operation is

$$\pi_{\langle attribute\ list \rangle}(R)$$

where π (pi) is the symbol used to represent the PROJECT operation and <attribute list> is a list of attributes from the attributes of relation R. Again, notice that R is, in general, a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a database relation. The result of the PROJECT operation has only the attributes specified in <attribute list> and *in the same order as they appear in the list.* Hence, its **degree** is equal to the number of attributes in <attribute list>.

Ex: i) $\pi_{SSN, Name}(EMP)$

SSN	Name
5642	Adarsh
5643	Bharath
5644	chethan
5645	Ranjan

ii) $\pi_{Address}(EMP)$

Address
Brg
MYS

Duplicate entries are removed.

If the attribute list includes only nonkey 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.**

7.4.3 Sequences of Operations and the RENAME Operation

In general, we may want 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 name the relations that hold the intermediate results.

i) $\pi_{ssn}(\sigma_{sal>18000}(EMP))$

The above expression can be divided into two expressions.

$emp1 \leftarrow \sigma_{sal>1800}(EMP)$

$Result \leftarrow \pi_{ssn}(emp1)$

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. This can be useful in connection with more complex operations such as UNION and JOIN, as we shall see. To rename the attributes in a relation, we simply list the new attribute names in parentheses

$RESULT(socialsecurityNO) \leftarrow \pi_{ssn}(emp1)$

$\therefore$ ssn is renamed to socialsecurity NO.

RESULT	socialsecurityNO
	5642
	5645

. If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order.

We can also define a **RENAME** operation—which can rename either the relation name, or the attribute names, or both—in a manner similar to the way we defined SELECT and PROJECT. The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms.

i) $\rho_{S(B_1, B_2, \ldots B_n)}(R)$ → renames both relation & its attributes

ii) $\rho_S(R)$ → Renames the relation only.

iii) $\rho_{(B_1, B_2, \ldots B_n)}(R)$ → Renames the attributes only

where the symbol ρ (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. If the attributes of R are $(A_1, A_2, \ldots, A_n)$ in that order, then each A_i is renamed as B_i.

7.4.4 Set Theoretic Operations

The next group of relational algebra operations are the standard mathematical operations on sets.

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION, INTERSECTION,** and **SET DIFFERENCE.** These are **binary** operations; that is, each is applied to two sets. When these operations are adapted to relational databases, the two relations on which any of the above three operations are applied must have the same **type of tuples;** this condition is 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 $dom(A_i) = dom(B_i)$ for 1 1 i 1 n. This means that the two relations have the same number of attributes and that each pair of corresponding attributes have the same domain.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations R and 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 R and S. Duplicate tuples are eliminated.
- **INTERSECTION:** The result of this operation, denoted by R ∩ S, is a relation that includes all tuples that are in both R and S.
- **SET DIFFERENCE:** The result of this operation, denoted by R − S, is a relation that includes all tuples that are in R but not in S.

Ex:

STUDENT	FN	LN
	Adarsh	g
	Kiran	p
	Ranjan	An
	Rayarendra	K

INSTRUCTOR

Fname	Lname
Kiran	P
Ranjan	K

i) To display names of both student and instructor

STUDENT U INSTRUCTOR

FN	LN
Adarsh	g
Kiran	p
Ranjan	AN
Rayarendra	K
Ranjan	K

ii) To display common names

INSTRUCTOR ∩ STUDENT

Fname	Lname
Kiran	p

iii) INSTRUCTOR – STUDENT

Fname	Lname
Ranjan	K

★**CARTESIAN PRODUCT** operation:—also known as **CROSS PRODUCT** or **CROSS JOIN**—denoted by **x**, which 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 $R(A_1, A_2, \ldots, A_n)$ **x** $S(B_1, B_2, \ldots, B_m)$ is a relation Q with n + m attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$, in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has n_R tuples and S has n_S tuples, then R **x** S will have $n_R * n_S$ tuples.

Ex:- EMP

ENO	Ename	sal	DNO
1134	Adarsh	1200	1
1135	Amith	5000	2
1136	chethan	6000	1

DEPT

Dno	Dname
1	CS
2	IS

EMP X DEPT Total no of attributes (Eno, Ename, sal, Dno, Dno, Dname)

Total no of tuples $3 \times 2 = 6$

The resulting relation will be

Eno	Ename	sal	Dno	Dno	Dname
1134	Adarsh	1200	1	1	CS
1135	Amith	5000	2	1	CS
1136	chetan	6000	1	1	CS
1134	Adarsh	1200	1	2	IS
1135	Amith	5000	2	2	IS
1136	chetan	6000	1	2	IS

} cartesian product of EMP X DEPT.

To display the Eno, Ename & Dname he belongs to

$R_1 \leftarrow$ EMP X DEPT

$R_2 \leftarrow \sigma_{Dno=Dno}(R_1)$

$\pi_{Eno, Ename, Dname}(R_2)$

The operation applied by itself is generally meaningless. It is useful when followed by a selection that matches values of attributes coming from the component relations.

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations. We can then SELECT only related tuples from the two relations by specifying an appropriate selection condition. Because this sequence of CARTESIAN PRODUCT followed by SELECT is used quite commonly to identify and select related tuples from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation.

7.4.5 The JOIN Operation

The **JOIN** operation, denoted by ⋈ 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.

Ex: To display Employee no and dname from Dept.

$R_1\ (d, dname) \leftarrow \pi_{dno,\ dname}(DEPT)$

$EMP \bowtie_{dno=d} R_1$

Eno	Ename	sal	dno	d	dname
1134	Adarsh	1200	1	1	CS
1135	Anith	5000	2	2	IS
1136	Chetan	6000	1	1	CS

$\pi_{Eno,\ dname}(EMP \bowtie_{dno=d} R_1)$

Eno	dname
1134	CS
1135	IS
1136	CS

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:

$R \bowtie_{\langle join\ condition \rangle} S$

The result of the JOIN is a relation Q with n + m attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S—*whenever the combination satisfies the join condition.* This is the main difference between CARTESIAN PRODUCT and JOIN: in JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to true is included in the resulting relation Q *as a single combined tuple.*

A general join condition is of the form:

<condition> AND <condition> AND . . . AND <condition>

where each condition is of the form $A_i \theta B_j$, A_i is an attribute of R, B_j is an attribute of S, A_i and B_j 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 *do not* appear in the result. In that sense, the join operation does *not* necessarily preserve all of the information in the participating relations.

Emp $\bowtie_{dno=d}$ R1

join condition is dno=d → EQUIJOIN.

The most common JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN.** Both examples we have considered were EQUIJOINs. Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have *identical values* in every tuple.

Ex: Emp $\bowtie_{DNO=d}$ R1

Eno	Ename	sal	dno	d	Dname
1134	Adarsh	1200	1	1	CS
1135	Amith	5000	2	2	IS
1136	Chetan	6000	1	1	CS

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.

Ex: Emp * DEPT

Eno	Ename	sal	Dno	Dname
1134	Adarsh	1200	1	CS
1135	Amith	5000	2	IS
1136	Chetan	6000	1	CS

Dno is called join attribute.

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

.

The join operation is used to combine data from multiple relations so that related information can be presented in a single table.

7.4.6 A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations is a **complete** set; that is, any of the other relational algebra operations can be expressed as a *sequence of operations from this set.*

Complete set of relational algebra operations are

σ - SELECT, π - PROJECT, $\cup$ - UNION, $-$ SET DIFFERENCE, $\times$ - CROSSPRODUCT

The other operations that can be derived from this set are

i) $R \cap S = (R \cup S) - ((R - S) \cup (S - R))$

ii) $R \bowtie_{<condition>} S = \sigma_{<condition>}(R \times S)$

iii) $R * S = \pi_{<attribute\ list>}(\sigma_{<condition>}(R \times S))$

iv) DIVISION ($\div$) can be expressed as sequence of π, $\times$ and $-$ operations

$T_1 \leftarrow \pi_Y(R)$

$T_2 \leftarrow \pi_Y((S \times T_1) - R)$

$T \leftarrow T_1 - T_2$

7.4.7 The DIVISION Operation

The DIVISION operation is useful for a special kind of query that sometimes occurs in database applications. An example is "Retrieve the names of employees who work on *all* the projects that 'John Smith' works on."

In general, the DIVISION operation is applied to two relations R(Z) ÷ S(X), where X Z. Let Y = Z - X (and hence Z = X ∪ Y); that is, let Y be the set of attributes of R that are not attributes of S. The result of DIVISION is a relation T(Y) that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = t_S$ *for every tuple* t_S in S. This means that, 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.

Ex:- i)

R	A	B
	a_1	b_1
	a_2	b_1
	a_1	b_2
	a_1	b_3
	a_2	b_4
	a_2	b_3

S	A
	a_1
	a_2

Find $R \div S$

$a_1 \rightarrow \{b_1, b_2, b_3\}$ in R

$a_2 \rightarrow \{b_1, b_4, b_3\}$ in R.

$\{b_1, b_2, b_3\} \cap \{b_1, b_4, b_3\} = \{b_1, b_3\}$

$\therefore T \leftarrow R \div S$

T	B
	b_1
	b_3

ii)

P	B
	b_1
	b_4

Find $X \leftarrow R \div P$

$b_1 \rightarrow \{a_1, a_2\}$

$b_4 \rightarrow \{a_2\}$

$\{a_1, a_2\} \cap \{a_2\} = \{a_2\}$

$\therefore X \leftarrow R \div P$

X	A
	a_2

7.5 Additional Relational Operations

7.5.1 Aggregate Functions and Grouping

7.5.2 Recursive Closure Operations

7.5.3 OUTER JOIN and OUTER UNION Operations

. These operations enhance the expressive power of the relational algebra.

7.5.1 Aggregate Functions and Grouping

The first 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.

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

Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group.

An example would be to group employee tuples by DNO, so that each group includes the tuples for employees working in the same department. We can then list each DNO value along with, say, the average salary of employees within the department.

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

<grouping attribute> ℱ <function list> (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. In each such pair, <function> is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT—and <attribute> is an attribute of the relation specified by R. The resulting relation has the grouping attributes plus one attribute for each element in the function list.

Ex: EMP

Eid	Ename	ESal	Dno
1234	Adarsh	12000	1
1235	Chetan	5000	2
1236	Ranjan	8000	1
1237	Yathish	10000	2

i) Dno ℱ EMP

1234	Adarsh	12000	1
1236	Ranjan	8000	1

Dno = 1 will be grouped

1235	Chetan	5000	2
1237	Yathish	1000	2

Dno = 2 will be grouped.

ii) $_{Dno}\mathfrak{I}_{count\ Eid}(EMP)$ o/p:

Dno	count_Eid
1	2
2	2

ii) $_{Dno}\mathfrak{I}_{count\ Eid,\ SUM\ Esal}(EMP)$

Dno	count Eid	SUM-Esal
1	2	20000
2	2	15000

iii) $\mathfrak{I}_{count-Eid}(EMP)$

count-Eid
4

If grouping is not indicated it is considered as one group.

In the above example, we specified a list of attribute names—between parentheses in the rename operation—for the resulting relation R. If no renaming is applied, then the attributes of the resulting relation that correspond to the function list will each be the concatenation of the function name with the attribute name in the form <function>_<attribute>.

If no grouping attributes are specified, the functions are applied to the attribute values of *all the tuples* in the relation, so the resulting relation has a *single tuple only.*

It is important to note that, in general, duplicates are *not eliminated* when an aggregate function is applied; this way, the normal interpretation of functions such as SUM and AVERAGE is computed (Note 12). It is worth emphasizing that the result of applying an aggregate function is a relation, not a scalar number—even if it has a single value.

7.5.2 Recursive Closure Operations

Another type of operation that, in general, cannot be specified in the basic relational algebra is **recursive closure.** This operation is applied to a **recursive relationship** between tuples of the same type, such as the relationship between an employee and a supervisor. This relationship is described by the foreign key SUPERSSN of the EMPLOYEE relation , which relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor).

7.5.3 OUTER JOIN and OUTER UNION Operations

The JOIN operations described earlier match tuples that satisfy the join condition. For example, for a NATURAL JOIN operation R * 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 in the join attributes are also eliminated. A set of operations, called OUTER JOINs, can be used when we want to keep all the tuples in R, or those in S, or those in both relations in the result of the JOIN, 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.

Ex: T_1

P	Q	R
10	a	5
15	b	8
25	a	6

T_2

A	B	C
10	b	6
25	c	3
10	b	5

i) $T_1 \bowtie_{T_1.P = T_2.A} T_2$

P	Q	R	A	B	C
10	a	5	10	b	6
10	a	5	10	b	5
15	b	8	NULL	NULL	NULL
25	a	6	25	c	3

A similar operation, **RIGHT OUTER JOIN**, denoted by , keeps every tuple in the *second* or right relation S in the result of R S.

ii) $T_1 \bowtie_{T_1.Q = T_2.B} T_2$

P	Q	R	A	B	C
15	b	8	10	b	6
15	b	8	10	b	5
NULL	NULL	NULL	25	c	3

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.

iii) $T_1 \bowtie_{T_1.Q = T_2.B} T_2$

P	Q	R	A	B	C
10	a	5	NULL	NULL	NULL
15	b	8	10	b	6
15	b	8	10	b	5
25	a	6	NULL	NULL	NULL
NULL	NULL	NULL	25	c	3

The **OUTER UNION** operation was developed to take the union of tuples from two relations if the relations are *not union compatible.* This operation will take the UNION of tuples in two relations that are **partially compatible,** meaning that only some of their attributes are union compatible. It is expected that the list of compatible attributes includes a key for both relations. Tuples from the component relations with the same key are represented only once in the result and have values for all attributes in the result. The attributes that are not union compatible from either relation are kept in the result, and tuples that have no values for these attributes are padded with null values.

For example, an OUTER UNION can be applied to two relations whose schemas are STUDENT(Name, SSN, Department, Advisor) and FACULTY(Name, SSN, Department, Rank). The resulting relation schema is R(Name, SSN, Department, Advisor, Rank), and all the tuples from both relations are included in the result. Student tuples will have a null for the Rank attribute, whereas faculty tuples will have a null for the Advisor attribute. A tuple that exists in both will have values for all its attributes (Note 14).

iv) $T_1 \bowtie_{T_1.P = T_2.A} T_2$

P	Q	R	A	B	C
10	a	5	10	b	6
10	a	5	10	b	5
25	a	6	25	c	3

v) $T_1 \bowtie_{T_1.Q = T_2.B} T_2$

P	Q	R	A	B	C
15	b	8	10	b	6
15	b	8	10	b	5

vi) $T_1 \cup T_2$

P	Q	R
10	a	5
15	b	8
25	a	6
10	b	6
25	c	3
10	b	5

Table T_3

A	B	C
10	b	6
15	c	3
10	b	5

vii) $T_1 \bowtie_{(T_1.P = T_3.A \text{ AND } T_1.R = T_3.C)} T_3$

P	Q	R	A	B	C
10	a	5	10	b	5

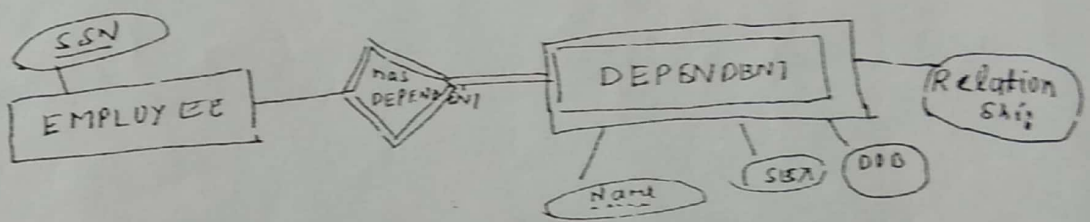
7.6 Examples of Queries in Relational Algebra
Refer class notes

SVIT

Discuss the steps followed to convert ER model to a relational Schema.

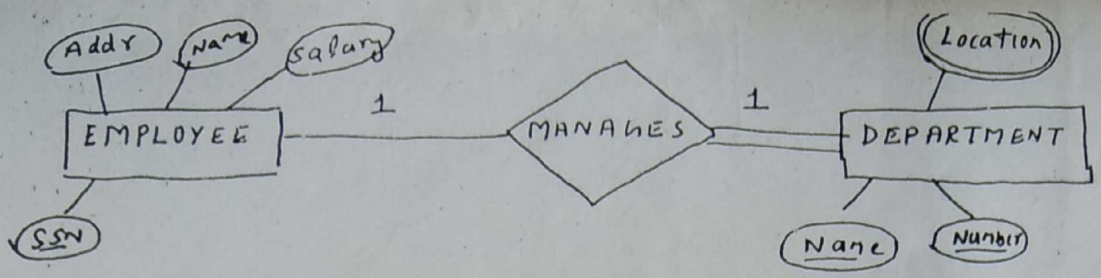
STEP1: For each regular (strong) entity type E in ER schema, create a relation R that include simple attributes. Choose one of the key attribute of E as primary key for R.

STEP2: For each weak entity type W in the ER schema with owner entity Type E, create a relation R & include all simple attributes of W. In addition, include as foreign key attribute of R the primary key attribute of relation(s) that corresponds to the owner entity type(s). primary key of R is the combination of primary key (s) of the owner & the partial key of the weak entity type.



DEPENDENT (SSN, NAME, SEX, DOB, Relationship)

STEP3: For each binary 1:1 relationship type R in the ER schema, identify the relations S & T that correspond to the entity types participating in R. Choose one of the relation say S & include as foreign key in S the primary key of T. It is better to choose an with total participation in R.



EMPLOYEE(Adr, Name, SSN , salary , DNO)-- Foreign Key

DEPARTMENT(Number , Name, location)

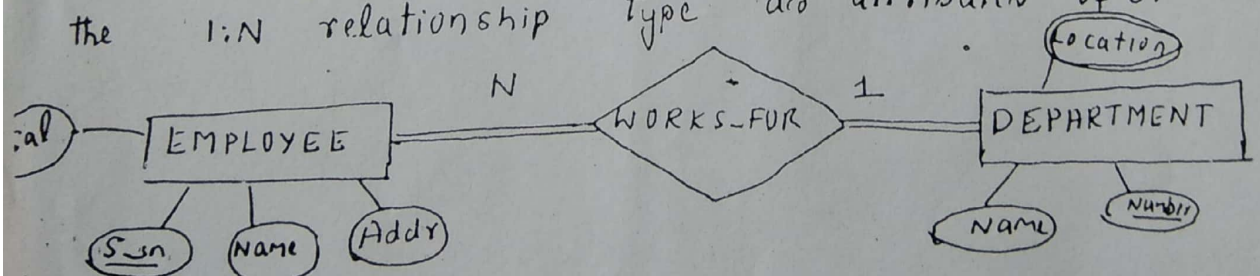
OR

prefered schema is

EMPLOYEE (Adr, Name, SSN , salary)

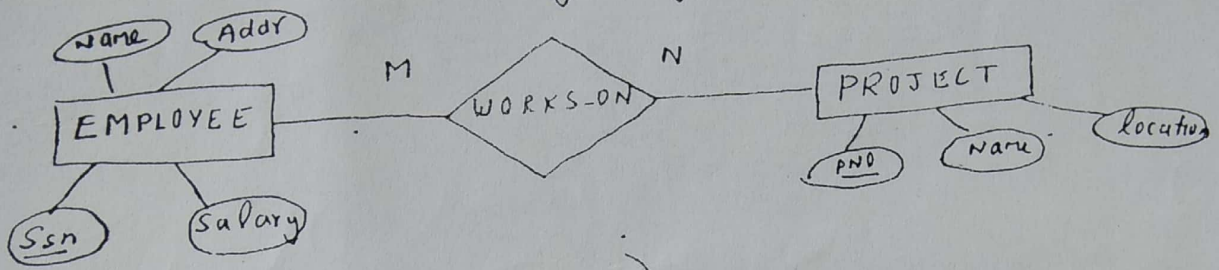
DEPARTMENT(Number , Name, location, ESSN)-- foreign key.

STEP 4: 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 . Include any simple attributes of the 1:N relationship type as attributes of s.



EMPLOYEE (Ssn , Name, sal, addr, dno)... FK

STEP5 For each binary M:N relationship type R, create a new relation S to represent R. Include as foreign key attribute in S the primary keys of the relations that represent the participating entity types, their combination will form the primary key of S.



WORKS-ON (Ssn , PNO)

STEP6 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

STEP7 For each n-ary relationship type R, where n>2, create a new relation S to represent R. Include as foreign key attribute in S the primary keys of the relations that represent the participating entity types.