Lecture Notes

DATABASE MANAGEMENT SYSTEM (BCS403)

Module – 2 **08 Hours**

Relational Model: Relational Model Concepts, Relational Model Constraints and relational database schemas,Update operations, transactions, and dealing with constraint violations.

Relational Algebra: Unary and Binary relational operations, additional relational operations (aggregate, grouping, etc.) Examples of Queries in relational algebra.

Mapping Conceptual Design into a Logical Design: Relational Database Design using ER-to-Relational mapping.

Textbook 1: Ch 5.1 to 5.3, Ch 8.1 to 8.5; Ch 9.1 to 9.2 Textbook 2: 3.5

RBT: L1, L2, L3

Teaching-Learning Process	Chalk and board, Active Learning, Demonstration

Chapter 1

Relational Model

Relational Model Concepts {8 MARKS}*

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.*

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.

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.

An Example

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

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, the first table of Figure 1.2 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.

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 formal 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.

Some examples of domains:

- Usa_phone_numbers. The set of ten-digit phone numbers valid in the United States.
- Local_phone_numbers. The set of seven-digit phone numbers valid within a particular area code in the United States. The use of local phone numbers is quickly becoming obsolete, being replaced by standard ten-digit numbers.
- 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.
- Grade_point_averages. Possible values of computed grade point averages; each must be a real (floating-point) number between 0 and 4.
- Employee_ages. Possible ages of employees in a company; each must be an integer value between 15 and 80.
- 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 examples are called *logical* definitions of domains.

A **data type** or **format** is also specified for each domain.

Usa_phone_numbers can be declared as a character string of the form (*ddd*)*ddd-dddd*, where each *d* is a numeric (decimal) digit.

A **relation schema** *R*, denoted by $R(A1, A2, \ldots, An)$, is made up of a relation name *R* and a list of attributes, $A1, A2, \ldots, An$.

Each **attribute** *Ai* is the name of a role played by some domain *D* in the relation schema *R*. *D* is called the **domain** of *Ai* and is denoted by **dom**(***Ai***).

A relation schema is used to *describe* a relation; *R* is called the **name** of this relation.

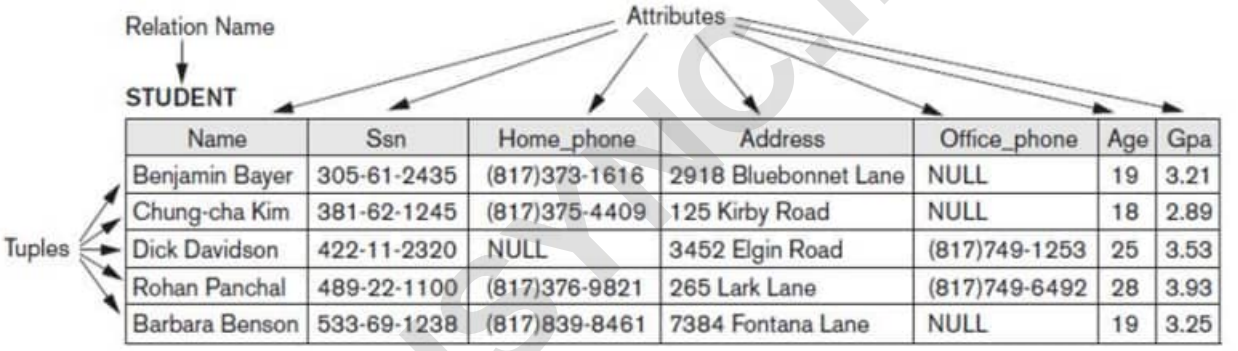
The **degree** (or **arity**) of a relation is the number of attributes *n* of its relation schema.

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 <= *i* <= *n*, is an element of dom (*Ai*) or is a special NULL value.

The *i*th value in tuple *t*, which corresponds to the attribute *Ai*, is referred to as *t*[*Ai*] or *t.Ai* (or *t*[*i*] if we use the positional notation).

The terms **relation intension** for the schema *R* and **relation extension** for a relation state *r*(*R*) are also commonly used.



Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25

A relation can be *restated* more formally using set theory concepts as follows.

A relation (or relation state) *r*(*R*) is a **mathematical relation** of degree *n* on the domains dom(*A*1), dom(*A*2), … , dom(*An*), which is a **subset** of the **Cartesian product** (denoted by ×) of the domains that define *R*:

$$r(R) \subseteq (\text{dom}(A1) \times \text{dom}(A2) \times \ldots \times (\text{dom}(An))$$

The Cartesian product specifies all possible combinations of values from the underlying domains. Hence, if we denote the total number of values, or **cardinality,** in a domain *D* by |*D*| (assuming that all domains are finite), the total number of tuples in the Cartesian product is

$$|\text{dom}(A1)| \times |\text{dom}(A2)| \times \ldots \times |\text{dom}(An)|$$

This product of cardinalities of all domains represents the total number of possible instances or tuples that can ever exist in any relation state *r*(*R*).

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.

STUDENT

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21

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.

$t = <$ (Name, Dick Davidson),(Ssn, 422-11-2320),(Home_phone, NULL),(Address, 3452 Elgin Road), (Office_phone, (817)749-1253),(Age, 25),(Gpa, 3.53)>

$t = <$ (Address, 3452 Elgin Road),(Name, Dick Davidson),(Ssn, 422-11-2320),(Age, 25), (Office_phone, (817)749-1253),(Gpa, 3.53),(Home_phone, NULL)>

Values and NULLs in the Tuples.

Each value in a tuple is an **atomic** value; i.e., 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**.

Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption.

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, Home_phone, Address, Office_phone, Age, and Gpa.

Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion. Notice that some relations may represent facts about *entities,* whereas other relations may represent facts about *relationships.*

For example, a relation schema MAJORS (Student_ssn, Department_code) asserts that students major in academic disciplines.

An alternative interpretation of a relation schema is as a **predicate**; in this case, the values in each tuple are interpreted as values that *satisfy* the predicate. For example, the predicate STUDENT (Name, Ssn, …) is true for the five tuples in relation STUDENT

Relational Model Notation

notation in our presentation

- A relation schema R of degree n is denoted by $R(A1, A2, \ldots, An)$.
- The uppercase letters Q, R, S denote relation names.
- The lowercase 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.
- An n-tuple t in a relation $r(R)$ is denoted by $t = <v1, v2, \ldots, vn>$, where vi is the value corresponding to attribute Ai. The following notation refers to **component values** of tuples:

- Both $t[Ai]$ and $t.Ai$ (and sometimes $t[i]$) refer to the value vi in t for attribute Ai.
- Both $t[Au, Aw, \ldots, Az]$ and $t.(Au, Aw, \ldots, Az)$, where $Au, Aw, \ldots, Az$ is a list of attributes from R, refer to the subtuple of values $<vu, vw, \ldots, vz>$ from t Corresponding to the attributes specified in the list.

consider the tuple t = <'Barbara Benson', '533-69-1238', '(817)839-8461', '7384 Fontana Lane', NULL, 19, 3.25> from the STUDENT relation; we have t[Name] = <'Barbara Benson'>, and t[Ssn, Gpa, Age] = <'533-69-1238', 3.25, 19>.

2. Relational Model Constraints and Relational Database Schemas {8 MARKS}*

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** or **implicit constraints**.
- Constraints that can be directly expressed in the schemas of the data model, typically by specifying them in the DDL (data definition language). We call these **schema-based constraints** or **explicit 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 or in some other way. We call these **application-based** or **semantic constraints** or **business rules**.

Another important category of constraints is *data dependencies*, which include

functional dependencies and *multivalued dependencies*. They are used mainly for testing the "goodness" of the design of a relational database and are utilized in a process called *normalization.*

The schema-based constraints include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

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 other special data types.

Domains can also be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed.

Key Constraints and Constraints on NULL Values

In the formal relational model, 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:

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

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.

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 *uniqueness* 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 hold. This *minimality* property is required for a key but is optional for a superkey.

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

The CAR relation in Figure has two candidate keys: License_number and Engine_serial_number.

CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

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.

Explain various types of constraints in relational model with examples { 6 marks}

Relational Databases and Relational Database Schemas

A **relational database schema** *S* is a set of relation schemas

S = {*R*1, *R*2, … , *Rm*} 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, … , *rm*} such that each *ri* is a state of *Ri* and such that the *ri* relation states satisfy the integrity constraints specified in IC

COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

DEPT_LOCATIONS

Dnumber	Dlocation

PROJECT

Pname	Pnumber	Plocation	Dnum

WORKS_ON

Essn	Pno	Hours

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship

Figure: shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

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

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Figure: shows a relational database state corresponding to the COMPANY schema.

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.

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, 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, *R*1 is called the **referencing relation** and *R*2 is the **referenced relation**. If these two conditions hold, a **referential integrity constraint** from *R*1 to *R*2 is said to hold. In a database of many relations, there are usually many referential integrity constraints. Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas.

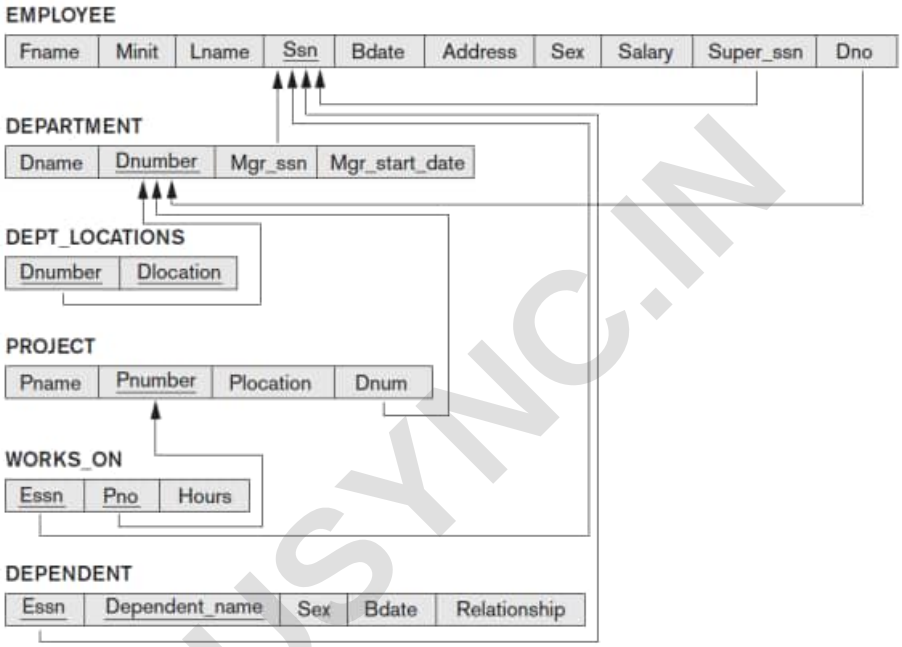


Figure: Referential integrity constraints displayed on the COMPANY relational database schema.

Explain the entity integrity and referential integrity constraints. Why is each considered important. Give examples { 6 marks}

Other Types of Constraints

Another class of general constraints, sometimes called *semantic integrity constraints,* are not part of the DDL and have to be specified and enforced in a different way.

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*. Such constraints can be specified and enforced within the application

programs that update the database, or by using a general-purpose **constraint specification language**.

Mechanisms called **triggers** and **assertions** can be used in SQL, through the CREATE ASSERTION and CREATE TRIGGER statements, to specify some of these constraints. 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.11 An example of a transition constraint is: "the salary of an employee can only increase." Such constraints are typically enforced by the application programs or specified using active rules and triggers.

3. Update Operations, Transactions, and Dealing with Constraint Violations { 8 marks}*

The operations of the relational model can be categorized into *retrievals* and *updates.* The relational algebra operations, which can be used to specify **retrievals.** A relational algebra expression forms a new relation after applying a number of algebraic operators to an existing set of relations; its main use is for querying a database to retrieve information.

There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify)

Insert is used to insert one or more new 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.

1. **Domain constraints** can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type.
2. **Key constraints** can be violated if a key value in the new tuple *t* already exists in another tuple in the relation *r*(*R*).
3. **Entity integrity** can be violated if any part of the primary key of the new tuple *t* is NULL.
4. **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.

Some of examples for insert operation are:

***Operation*:**

Insert <'Cecilia', 'F', 'Kolonsky', NULL, '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.

***Result*:** This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.

***Operation*:**

Insert <'Alicia', 'J', 'Zelaya', '999887777', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 4> into EMPLOYEE.

Result: 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.

***Operation*:**

Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windswept, Katy, TX', F, 28000, '987654321', 7> into EMPLOYEE.

***Result*:** This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in DEPARTMENT with Dnumber = 7.

***Operation*:**

Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '1960-04-05', '6357 Windy Lane, Katy, TX', F, 28000, NULL, 4> into EMPLOYEE.

***Result*:** This insertion satisfies all constraints, so it is acceptable.

The Delete Operation

The **Delete** operation can violate **only referential integrity**. This occurs if the tuple being deleted is referenced by 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.

Some of the examples for delete operations are:

***Operation*:**

Delete the WORKS_ON tuple with Essn = '999887777' and Pno = 10.

***Result*:** This deletion is acceptable and deletes exactly one tuple.

***Operation*:**

Delete the EMPLOYEE tuple with Ssn = '999887777'.

***Result*:** This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is deleted, referential integrity violations will result.

***Operation*:**

Delete the EMPLOYEE tuple with Ssn = '333445555'.

***Result*:** This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE, DEPARTMENT, WORKS_ON, and DEPENDENT relations.

Several options are available if a deletion operation causes a violation.

The first option, called **restrict,** is to *reject the deletion.*

The second option, called **cascade,** is to *attempt to cascade (or propagate) the deletion* by deleting tuples that reference the tuple that is being deleted.

A third option, called **set null** or **set default,** is to *modify the referencing attribute values* that cause the violation; each such value is either set to NULL or changed to reference another default valid tuple.

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*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified.

Some of the examples for update operation are:

***Operation*:**

Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.

***Result*:** Acceptable.

***Operation*:**

Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1.

Result: Acceptable.

***Operation*:**

Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7.

***Result*:** Unacceptable, because it violates referential integrity.

***Operation*:**

Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654321'.

***Result*:** Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates referential integrity constraints because there are other relations that refer to the existing value of Ssn.

Updating an attribute that is *neither part of a primary key nor part of 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.

The Transaction Concept

A database application program running against a relational database typically executes one or more *transactions*.

A **transaction** is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database.

At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints specified on the database schema. A single transaction may involve any number of retrieval operations and any number of update operations.

For example, a transaction to apply a bank withdrawal will typically read the user account record, check if there is a sufficient balance, and then update the record by the withdrawal amount.

A large number of commercial applications running against relational databases in **online transaction processing (OLTP**) systems are executing transactions at rates that reach several hundred per second.

Chapter 2:

The Relational Algebra

The basic set of operations for the formal 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 query is a new relation. 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 that represents the result of a database query (or retrieval request).

The relational algebra is very important for several reasons:

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.
3. Some of its concepts are incorporated into the SQL standard query language for RDBMSs

1. Relational Algebra: Unary and Binary relational operations

Unary Relational Operations: SELECT and PROJECT

The SELECT Operation {8 marks} *

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition.**

The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples—those tuples that satisfy the condition and are selected, and those tuples that do not satisfy the condition and are filtered out.

For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000.

$\sigma_{Dno=4}(EMPLOYEE)$

$\sigma_{Salary>30000}(EMPLOYEE)$

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*. 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>

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.

$\sigma_{(Dno=4\ AND\ Salary>25000)\ OR\ (Dno=5\ AND\ Salary>30000)}(EMPLOYEE)$

<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 connected by the standard Boolean operators *and*, *or*, and *not* to form a general selection condition. Notice that all the comparison operators in the set {=, <, ≤, >, ≥, ≠} can apply to attributes whose domains are *ordered values*, such as numeric or date domains.

Domains of strings of characters are also considered to be ordered based on the collating sequence of the characters. If the domain of an attribute is a set of *unordered values*, then only the comparison operators in the set {=, ≠} can be used.

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5

σ(Dno=4 AND Salary>25000) OR (Dno=5 AND Salary>30000) (EMPLOYEE).

The <selection condition> is applied independently to each *individual tuple t* in *R*. This is done by substituting each occurrence of an attribute *Ai* in the selection condition with its value in the tuple *t*[*Ai*]. 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:

- (cond1 **AND** cond2) is TRUE if both (cond1) and (cond2) are TRUE; otherwise, it is FALSE.
- (cond1 **OR** cond2) is TRUE if either (cond1) or (cond2) or both are TRUE; otherwise, it is FALSE.
- (**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. 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—its number of attributes—is the same as the degree of *R*.

Notice that the SELECT operation is **commutative**; that is,

σ<cond1>(σ<cond2>(R)) = σ<cond2>(σ<cond1>(R)).

we can always combine a **cascade** (or **sequence**) of SELECT operations into a single SELECT operation with a conjunctive (AND) condition

σ<cond1>(σ<cond2>(... (σ<condn>(R)) ...)) = σ<cond1> AND<cond2> AND...AND <condn>(R)

In SQL, the SELECT condition is typically specified in the WHERE clause of a query.

For example, the following operation:

σDno=4 AND Salary>25000 (EMPLOYEE)

would correspond to the following SQL query:

```
SELECT *
FROM EMPLOYEE
WHERE Dno=4 AND Salary>25000;
```

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.

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

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

The general form of the PROJECT operation is

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

π (pi) is the symbol used to represent the PROJECT operation

<attribute list> is the desired sublist of attributes from the attributes of relation *R*.

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

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

The PROJECT operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**. The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in *R*. If the projection list is a superkey of *R*—that is, it includes some key of *R*—the resulting relation has the *same number* of tuples as *R*

$$\pi_{\text{<list1>}}(\pi_{\text{<list2>}}(R)) = \pi_{\text{<list1>}}(R)$$

In SQL, the PROJECT attribute list is specified in the *SELECT clause* of a query. For example, the following operation:

$\pi_{\text{fname, Salary}}(\text{EMPLOYEE})$

Corresponding SQL query is

SELECT DISTINCT fname, Salary **FROM** EMPLOYEE

Sequences of Operations and the RENAME Operation

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. 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.

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

Fname	Lname	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, and using the **assignment operation,** denoted by ← (left arrow), as follows:

$DEP5\_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$

$RESULT \leftarrow \pi_{Fname, Lname, Salary}(DEP5\_EMPS)$

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston,TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston,TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble,TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

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

$TEMP \leftarrow \sigma_{Dno=5}(EMPLOYEE)$

$R(First\_name, Last\_name, Salary) \leftarrow \pi_{Fname, Lname, Salary}(TEMP)$

R

First_name	Last_name	Salary
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

We can also define a formal **RENAME** operation—which can 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: $\rho_{S(B1, B2, \dots, Bn)}(R)$ or $\rho_{S}(R)$ or $\rho_{(B1, B2, \dots, Bn)}(R)$ The symbol ρ (rho) is used to denote the RENAME operator, *S* is the new relation name, and *B*1, *B*2, … , *Bn* are the new attribute names.

In SQL, a single query typically represents a complex relational algebra expression. Renaming in SQL is accomplished by aliasing using **AS**, as in the following example:

SELECT E.Fname **AS** First_name, E.Lname **AS** Last_name, E.Salary **AS** Salary
FROM EMPLOYEE **AS** E **WHERE** E.Dno=5,

The UNION, INTERSECTION, and MINUS Operations

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

For example, to retrieve the Social Security numbers of all employees who either work in department 5 or directly supervise an employee who works in department 5, we can use the UNION operation as follows.

DEP5_EMPS ← σDno=5(EMPLOYEE)
RESULT1 ← πSsn(DEP5_EMPS)
RESULT2(Ssn) ← πSuper_ssn(DEP5_EMPS)
RESULT ← RESULT1 ∪ RESULT2

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both while eliminating any duplicates. 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**).

Definition: 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 ∪ 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 (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 relations STUDENT and INSTRUCTOR are union compatible and their tuples represent the names of students and the names of instructors, respectively.

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

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

STUDENT ∪ INSTRUCTOR

Fn	Ln
Susan	Yao
Ramesh	Shah

STUDENT ∩ INSTRUCTOR

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

STUDENT – INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

INSTRUCTOR – STUDENT

Notice that both UNION and INTERSECTION are commutative operations; that is,

$$R \cup S = S \cup R \text{ and } R \cap S = S \cap R$$

Both UNION and INTERSECTION can be treated as n-ary operations applicable to any number of relations because both are also associative operations; that is,

$$R \cup (S \cup T) = (R \cup S) \cup T \text{ and } (R \cap S) \cap T = R \cap (S \cap T)$$

The MINUS operation is not commutative; that is, in general,

$$R - S \neq S - R$$

Note that INTERSECTION can be expressed in terms of union and set difference as follows:

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

In SQL, there are three operations—UNION, INTERSECT, and EXCEPT—that correspond to the set operations described here.

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

The CARTESIAN PRODUCT operation—also known as CROSS PRODUCT or CROSS JOIN—which is denoted by ×. It is a binary set operation, but the relations on which it is applied do not have to be union compatible. This 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. If R has n_R tuples (denoted as |R| = n_R), and S has n_S tuples, then R × S will have n_R * n_S tuples.

For example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

FEMALE_EMPS ← $\sigma_{Sex='F'}$(EMPLOYEE)

EMPNAMES ← πFname, Lname, Ssn(FEMALE_EMPS)

EMP_DEPENDENTS ← EMPNAMES × DEPENDENT

ACTUAL_DEPENDENTS ← σSsn=Essn(EMP_DEPENDENTS)

RESULT ← πFname, Lname, Dependent_name(ACTUAL_DEPENDENTS)

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

The CARTESIAN PRODUCT creates tuples with the combined attributes of two relations.

We can SELECT *related tuples only* from the two relations by specifying an appropriate selection condition after the Cartesian product.

Sequence of CARTESIAN PRODUCT followed by SELECT is quite commonly used to combine *related tuples* from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation.

3. 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.

Suppose that we want to retrieve the name of the manager of each department.

DEPT_MGR ← DEPARTMENT $\bowtie_{Mgr\_ssn=Ssn}$ EMPLOYEE

RESULT ← $\pi_{Dname, Lname, Fname}$(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	...

Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation. The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation. However, JOIN is very important because it is used frequently when specifying database queries.

CARTESIAN PRODUCT operation followed by a SELECT operation.

EMP_DEPENDENTS ← EMPNAMES × DEPENDENT

ACTUAL_DEPENDENTS ← $\sigma_{Ssn=Essn}$(EMP_DEPENDENTS)

Equivalent Join operation:

ACTUAL_DEPENDENTS ← EMPNAMES $\bowtie_{Ssn=Essn}$ DEPENDENT

The general form of a JOIN operation on two relations5 R(A1, A2, … , An) and S(B1, B2, … , Bm) is

R $\bowtie_{<join\ condition>}$ S

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.

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**. Both previous examples were EQUIJOINs.

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. **NATURAL JOIN**—denoted by *—was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition.

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

PROJ_DEPT ← PROJECT * $\rho_{(Dname, Dnum, Mgr\_ssn, Mgr\_start\_date)}$(DEPARTMENT)

The same query can be done in two steps by creating an intermediate table DEPT as follows:

DEPT ← $\rho_{(Dname, Dnum, Mgr\_ssn, Mgr\_start\_date)}$(DEPARTMENT)

PROJ_DEPT ← PROJECT * DEPT

To apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write

DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS

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

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

If there is no join condition, all combinations of tuples qualify and the JOIN degenerates into a CARTESIAN PRODUCT, also called CROSS PRODUCT or CROSS JOIN.

A single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins**, to distinguish them from a different join variation called *outer joins.* The NATURAL JOIN or EQUIJOIN operation can also be specified among multiple tables, leading to an *n-way join*. For example, consider the following three-way join.

((PROJECT $\bowtie_{Dnum=Dnumber}$ DEPARTMENT) $\bowtie_{Mgr\_ssn=Ssn}$ EMPLOYEE)

A Complete Set of Relational Algebra Operations

It has been shown that the set of relational algebra operations {σ, π, ∪, ρ, –, ×} is a complete set; that is, any of the other original relational algebra operations can be expressed as a sequence of operations from this set.

For example, the INTERSECTION operation can be expressed by using UNION and MINUS as follows: $R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$

As another example, a JOIN operation can be specified as a CARTESIAN PRODUCT followed by a SELECT operation, as we discussed:

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

Similarly, a NATURAL JOIN can be specified as a CARTESIAN PRODUCT preceded by RENAME and followed by SELECT and PROJECT operations.

The DIVISION Operation

The DIVISION operation, denoted by ÷, is useful for a special kind of query that sometimes occurs in database applications.

*Retrieve the names of employees who work on **all** the projects that 'John Smith' works on*.

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

SMITH ← $\sigma_{Fname='John' \text{ AND } Lname='Smith'}$(EMPLOYEE)

SMITH_PNOS ← π_{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 ← $\pi_{Essn, Pno}$(WORKS_ON)

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

SSNS(Ssn) ← SSN_PNOS ÷ SMITH_PNOS

RESULT ← $\pi_{Fname, Lname}$(SSNS * EMPLOYEE)

(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

In general, the DIVISION operation is applied to two relations R(Z) ÷ S(X), where the attributes of S are a subset of the attributes of R; that is, X ⊆ Z.

Let Y be the set of attributes of R that are not attributes of S; that is, Y = Z – X (and hence Z = X ∪ Y).

The result of DIVISION is a relation T(Y) that includes a tuple t if tuples tR appear in R with tR [Y] = t, and with tR [X] = tS for every tuple tS 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. DIVISION operation where $X = \{A\}$, $Y = \{B\}$, and $Z = \{A, B\}$.

Notice that the tuples (values) $b1$ and $b4$ appear in R in combination with all three tuples in S; that is why they appear in the resulting relation T. All other values of B in R do not appear with all the tuples in S and are not selected: $b2$ does not appear with $a2$, and $b3$ does not appear with $a1$.

(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

The DIVISION operation can be expressed as a sequence of π, ×, and – operations as follows:

$T1 \leftarrow \pi_Y(R)$

$T2 \leftarrow \pi_Y((S \times T1) - R)$

$T \leftarrow T1 - T2$

Notation for Query Trees

Table 8.1 Operations of Relational Algebra

OPERATION	PURPOSE	NOTATION
SELECT	Selects all tuples that satisfy the selection condition from a relation R.	$\sigma_{<\text{selection condition}>}(R)$
PROJECT	Produces a new relation with only some of the attributes of R, and removes duplicate tuples.	$\pi_{<\text{attribute list}>}(R)$
THETA JOIN	Produces all combinations of tuples from R_1 and R_2 that satisfy the join condition.	$R_1 \bowtie_{<\text{join condition}>} R_2$
EQUIJOIN	Produces all the combinations of tuples from R_1 and R_2 that satisfy a join condition with only equality comparisons.	$R_1 \bowtie_{<\text{join condition}>} R_2$, OR $R_1 \bowtie_{(<\text{join attributes 1}>), (<\text{join attributes 2}>)} R_2$
NATURAL JOIN	Same as EQUIJOIN except that the join attributes of R_2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be specified at all.	$R_1 *_{<\text{join condition}>} R_2$, OR $R_1 *_{(<\text{join attributes 1}>), (<\text{join attributes 2}>)} R_2$ OR $R_1 * R_2$
UNION	Produces a relation that includes all the tuples in R_1 or R_2 or both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cup R_2$
INTERSECTION	Produces a relation that includes all the tuples in both R_1 and R_2; R_1 and R_2 must be union compatible.	$R_1 \cap R_2$
DIFFERENCE	Produces a relation that includes all the tuples in R_1 that are not in R_2; R_1 and R_2 must be union compatible.	$R_1 - R_2$
CARTESIAN PRODUCT	Produces a relation that has the attributes of R_1 and R_2 and includes as tuples all possible combinations of tuples from R_1 and R_2.	$R_1 \times R_2$
DIVISION	Produces a relation $R(X)$ that includes all tuples $t[X]$ in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$, where $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

4. Additional Relational Operations

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:

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

where F1, F2, … , Fn are functions over the attributes in relation R and may involve arithmetic operations and constant values. This operation is helpful when developing reports where computed values have to be produced in the columns of a query result.

As an example, 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:

REPORT ← ρ(Ssn, Net_salary, Bonus, Tax)($\pi_{Ssn,\ Salary\ -\ Deduction,\ 2000\ *\ Years\_service,\ 0.25\ *\ Salary}$(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.

Example: Retrieving the average or total salary of all employees or the total 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.

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

<grouping attributes> ℑ <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*.

Example: To retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes as indicated below, we write:

ρR(Dno, No_of_employees, Average_sal) (Dno ℑ COUNT Ssn, AVERAGE Salary (EMPLOYEE))

R

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

ρR(Dno, No_of_employees, Average_sal)(Dno ℑ COUNT Ssn, AVERAGE Salary (EMPLOYEE)).

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

Dno ℑ COUNT Ssn, AVERAGE Salary(EMPLOYEE).

Count_ssn	Average_salary
8	35125

ℑ COUNT Ssn, AVERAGE Salary(EMPLOYEE).

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*. 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.

Recursive Closure Operations

Another type of operation that, in general, cannot be specified in the basic original 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. Relationship is described by the foreign key Super_ssn of the EMPLOYEE relation and it relates each employee tuple (in the role of supervisee) to another employee tuple (in the role of supervisor).

An example of a recursive operation is to retrieve all supervisees of an employee e at all levels—that is, all employees e′ directly supervised by e, all employees e′ directly supervised by each employee e′, all employees e″′ directly supervised by each employee e″, and so on.

For example, to specify the Ssns of all employees e′ directly supervised—at level one—by the employee e whose name is 'James Borg' , we can apply the following operation:

BORG_SSN ← πSsn(σ$_{\text{Fname='James' AND Lname='Borg'}}$(EMPLOYEE))

SUPERVISION(Ssn1, Ssn2) ← $\pi_{Ssn,Super\_ssn}$(EMPLOYEE)

RESULT1(Ssn) ← π_{Ssn1}(SUPERVISION ⋈ $_{Ssn2=Ssn}$BORG_SSN)

To retrieve all employees supervised by Borg at level 2—that is, all employees e″ supervised by some employee e′ who is directly supervised by Borg—we can apply another JOIN to the result of the first query, as follows:

RESULT2(Ssn) ← π_{Ssn1}(SUPERVISION ⋈ $_{Ssn2=Ssn}$RESULT1)

To get both sets of employees supervised at levels 1 and 2 by 'James Borg', we can apply the UNION operation to the two results, as follows:

RESULT ← RESULT2 ∪ RESULT1

SUPERVISION

(Borg's Ssn is 888665555)

(Ssn)	(Super_ssn)
Ssn1	Ssn2
123456789	333445555
333445555	888665555
999887777	987654321
987654321	888665555
666884444	333445555
453453453	333445555
987987987	987654321
888665555	null

RESULT1

Ssn
333445555
987654321

(Supervised by Borg)

RESULT2

Ssn
123456789
999887777
666884444
453453453
987987987

(Supervised by Borg's subordinates)

RESULT

Ssn
123456789
999887777
666884444
453453453
987987987
333445555
987654321

(RESULT1 ∪ RESULT2)

OUTER JOIN Operations

Some additional extensions to the JOIN operation that are necessary to specify certain types of queries. 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 values in the join attributes are also eliminated.

This type of join, where tuples with no match are eliminated, is known as an **inner join.**

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.

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.

Example, suppose that we want a list of all employee names as well as the name of the departments they manage *if they happen to manage a department*;

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

TEMP ← (EMPLOYEE ⟕ $_{Ssn=Mgr_ssn}$DEPARTMENT)

RESULT ← $\pi_{Fname, Minit, Lname, Dname}$(TEMP)

RESULT

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

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.

The OUTER UNION Operation

The **OUTER UNION** operation was developed to take the union of tuples from two relations that have some common attributes, but are *not union (type) compatible*. This operation will take the UNION of tuples in two relations $R(X, Y)$ and $S(X, Z)$ that are **partially compatible**, meaning that only some of their attributes, say X, are union compatible. The attributes that are union compatible are represented only once in the result, and those attributes that are not union compatible from either relation are also kept in the result relation $T(X, Y, Z)$.

It is therefore the same as a FULL OUTER JOIN on the common attributes.

Two tuples $t1$ in R and $t2$ in S are said to **match** if $t1[X] = t2[X]$. These will be combined (unioned) into a single tuple in t. Tuples in either relation that have no matching tuple in the other relation are padded with NULL values.

Example: An OUTER UNION can be applied to two relations whose schemas are STUDENT(Name, Ssn, Department, Advisor) and INSTRUCTOR(Name, Ssn, Department, Rank).

STUDENT_OR_INSTRUCTOR(Name, Ssn, Department, Advisor, Rank)

5. Examples of Queries in Relational Algebra

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.

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

Query 3. Find the names of employees who work on *all* the projects controlled by department number 5.

$$DEPT5\_PROJS \leftarrow \rho_{(Pno)}(\pi_{Pnumber}(\sigma_{Dnum=5}(PROJECT)))$$
$$EMP\_PROJ \leftarrow \rho_{(Ssn,\ Pno)}(\pi_{Essn,\ Pno}(WORKS\_ON))$$
$$RESULT\_EMP\_SSNS \leftarrow EMP\_PROJ \div DEPT5\_PROJS$$
$$RESULT \leftarrow \pi_{Lname,\ Fname}(RESULT\_EMP\_SSNS * EMPLOYEE)$$

Query 4. 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.

SMITHS(Essn) ← π_{Ssn} ($\sigma_{Lname='Smith'}$(EMPLOYEE))
SMITH_WORKER_PROJS ← π_{Pno}(WORKS_ON * SMITHS)
MGRS ← $\pi_{Lname, Dnumber}$(EMPLOYEE $\bowtie_{Ssn=Mgr\_ssn}$ DEPARTMENT)
SMITH_MANAGED_DEPTS(Dnum) ← $\pi_{Dnumber}$ ($\sigma_{Lname='Smith'}$(MGRS))
SMITH_MGR_PROJS(Pno) ← $\pi_{Pnumber}$(SMITH_MANAGED_DEPTS * PROJECT)
RESULT ← (SMITH_WORKER_PROJS ∪ SMITH_MGR_PROJS)

Query 5. List the names of all employees with two or more dependents.

Strictly speaking, this query cannot be done in the *basic* (*original*) *relational algebra*. We have to use the AGGREGATE FUNCTION operation with the COUNT aggregate function. We assume that dependents of the *same* employee have *distinct* Dependent_name values.

$T1$(Ssn, No_of_dependents) ← $_{Essn}\Im_{COUNT\ Dependent\_name}$(DEPENDENT)
$T2$ ← $\sigma_{No\_of\_dependents>2}(T1)$
RESULT ← $\pi_{Lname, Fname}(T2 * EMPLOYEE)$

Query 6. Retrieve the names of employees who have no dependents.

This is an example of the type of query that uses the MINUS (SET DIFFERENCE) operation.

ALL_EMPS ← π_{Ssn}(EMPLOYEE)
EMPS_WITH_DEPS(Ssn) ← π_{Essn}(DEPENDENT)
EMPS_WITHOUT_DEPS ← (ALL_EMPS – EMPS_WITH_DEPS)
RESULT ← $\pi_{Lname, Fname}$(EMPS_WITHOUT_DEPS * EMPLOYEE)

Query 7. List the names of managers who have at least one dependent.

MGRS(Ssn) ← π_{Mgr_ssn}(DEPARTMENT)
EMPS_WITH_DEPS(Ssn) ← π_{Essn}(DEPENDENT)
MGRS_WITH_DEPS ← (MGRS ∩ EMPS_WITH_DEPS)
RESULT ← $\pi_{Lname, Fname}$(MGRS_WITH_DEPS * EMPLOYEE)

Chapter 6:

Mapping Conceptual Design into a Logical Design

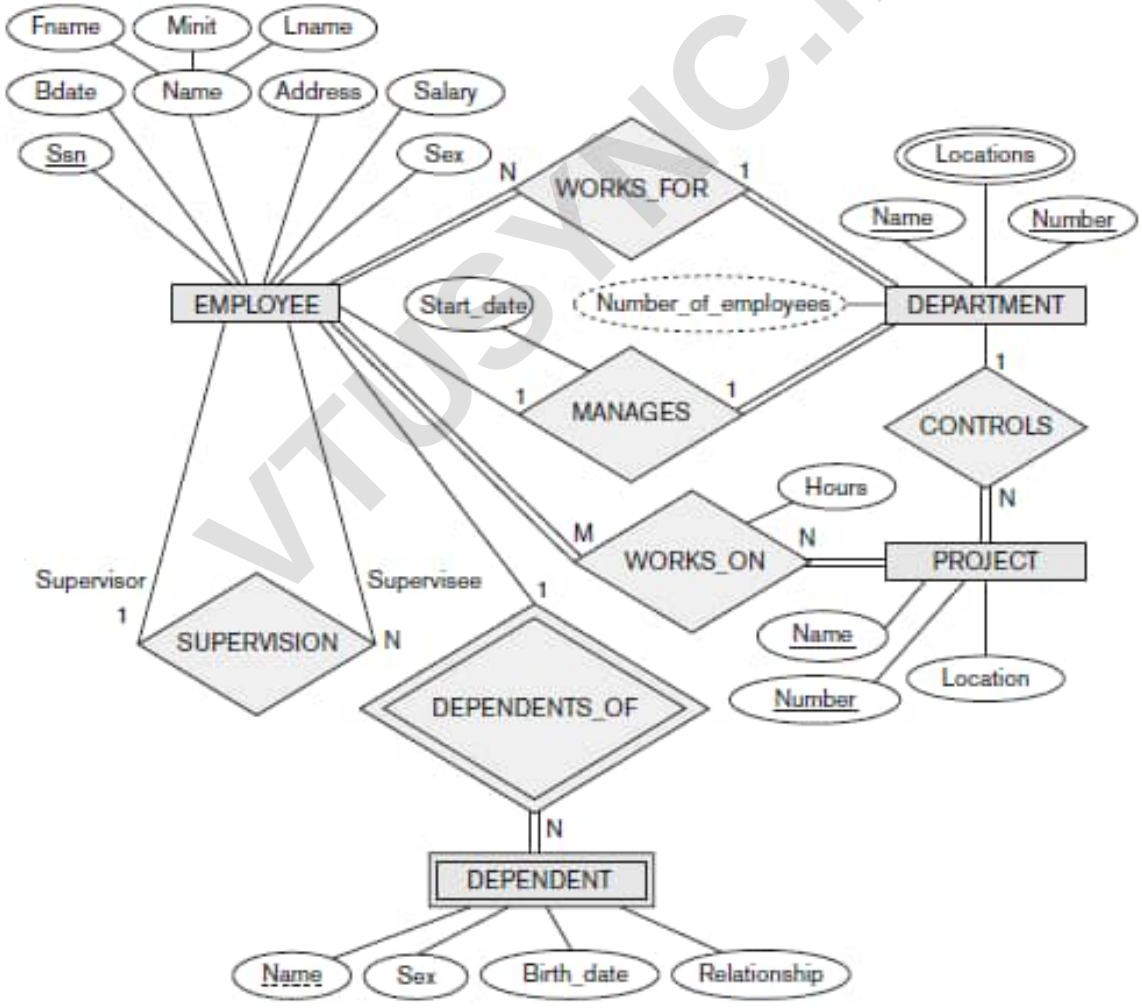
1. Relational Database Design using ER-to-Relational mapping

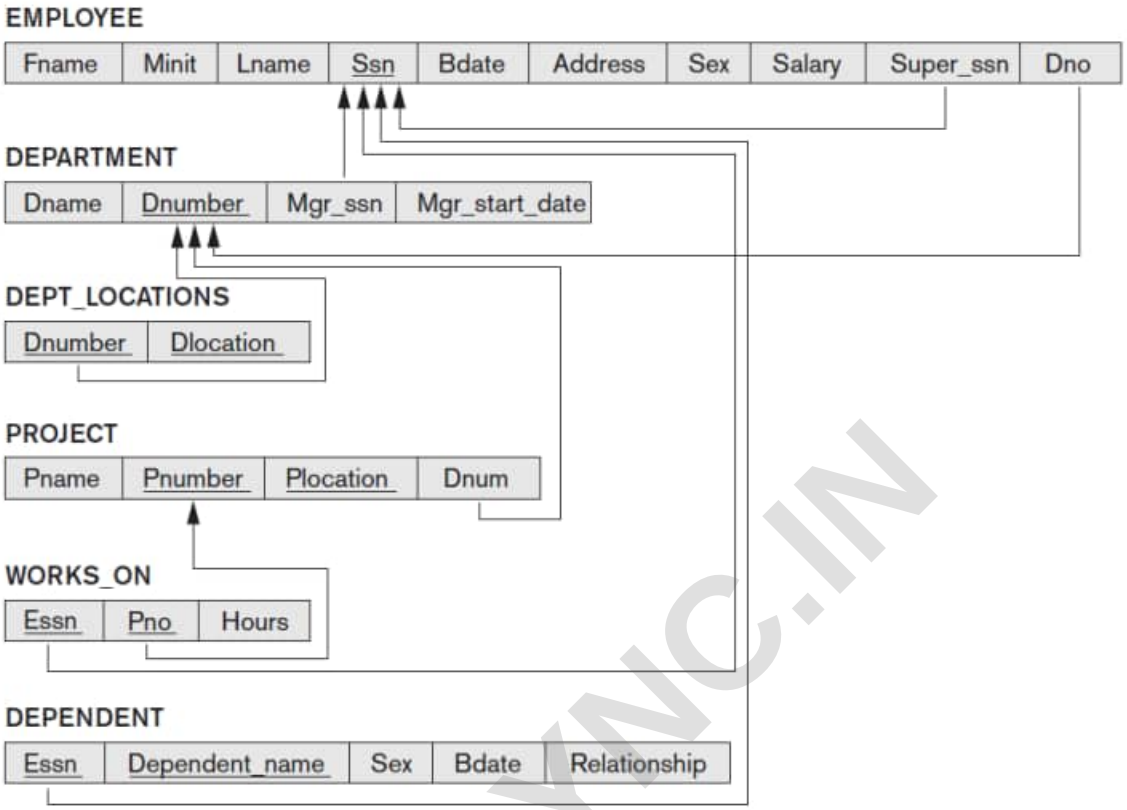
ER-to-Relational Mapping Algorithm

The steps of an algorithm for ER-to-relational mapping.

We use the COMPANY database example to illustrate the mapping procedure.

The COMPANY ER schema





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 the primary key for *R*. If the chosen key of *E* is a composite, then the set of simple attributes that form it will together form the primary key of *R*.

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 (or simple components of composite attributes) of *W* as attributes of *R*.

In addition, include as foreign key attributes of *R*, the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping 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. If there is a weak entity type *E*2 whose owner is also a weak entity type *E*1, then *E*1 should be mapped before *E*2 to determine its primary key first.

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*, say—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 approach: An alternative mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation.

This is possible when *both participations are total,* as this would indicate that the two tables will have the exact same number of tuples at all times.

Cross-reference or relationship relation approach: The third option 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. As we will see, this approach is required for binary M:N relationships.

Step 4: Mapping of Binary 1:N Relationship Types.

There are two possible approaches:

(1) the foreign key approach and

(2) the cross-reference or relationship

relation approach.

The first approach is generally preferred as it reduces the number of tables.

The foreign key approach: 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*; we do this because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type.

The relationship relation approach: An alternative approach is to use the **relationship relation** (cross-reference) option as in the third option for binary 1:1 relationships. We create a separate relation *R* whose attributes are the primary keys of *S* and *T*, which will also be foreign keys to *S* and *T*. The primary key of *R* is the same as the primary key of *S*. This option can be used if few tuples in *S* participate in the relationship to avoid excessive NULL values in the foreign key.

Step 5: Mapping of Binary M:N Relationship Types. In the traditional relational model with no multivalued attributes, the only option for M:N relationships is the **relationship relation (cross-reference) option**.

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*.

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 a multivalued attribute. The primary key of *R* is the combination of *A* and *K*. If the multivalued attribute is composite, we include its simple components.

Step 7: Mapping of *N*-ary Relationship Types. We use the **relationship relation option**. For each *n*-ary relationship type *R*, where *n* > 2, create a new relationship relation *S* to

represent *R*. Include as foreign key attributes in *S* the primary keys of the relations that represent the participating entity types.

Also include any simple attributes of the *n*-ary relationship type (or simple components of composite attributes) as attributes of *S*. The primary key of *S* is usually a combination of all the foreign keys that reference the relations representing the participating entity types.

Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	*Entity* relation
1:1 or 1:N relationship type	Foreign key (or *relationship* relation)
M:N relationship type	*Relationship* relation and *two* foreign keys
n-ary relationship type	*Relationship* relation and *n* foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key