Chapter 2: ER and Relational Model

E-R Model:

The entity-relationship(E-R) data model is based on a perception of a real world that consists of a collection of a basic objects, called entities, and of relationships among these objects. The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. It employs three basic notions: entity sets, relationship sets, and attributes.

- **Entities and Their Attributes.** The basic object that the ER model represents is an **entity**, which is a *thing* in the real world with an independent existence. An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course). Each entity has **attributes**—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job. A particular entity will have a value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.
- A **relationship** is an association among several entities. For example, a **depositor** relationship associates a customer with each account they have. The set of all entities of the same type and the set of all the relationships of the same type are termed as **entity set** and **relationship set** respectively.

Types Of Attributes:

• **Simple** and **composite** attributes. The attributes are not divided into subparts are called **Simple** attributes. **Composite** attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute *name* could be structured as a composite attribute consisting of *first-name*, *middle-initial*, and *last-name*. Using composite attributes in a design schema is a good choice if a user will wish to refer to an entire attribute on some occasions, and to only a component of the attribute on other occasions. Figure 2.2 depicts these examples of composite attributes for the *customer* entity set.

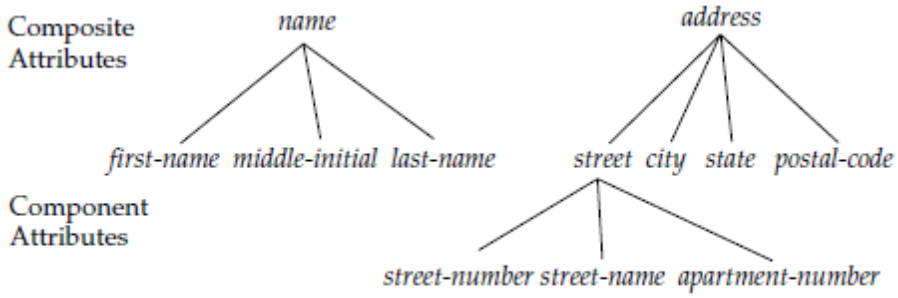


Figure 2.2 Composite attributes *customer-name* and *customer-address*.

• **Single-valued** and **multivalued** attributes. The attributes in our examples all have a single value for a particular entity. For instance, the *loan-number* attribute for a specific loan entity refers to only one loan

number. Such attributes are said to be **single valued**. There may be instances where an attribute has a set of values for a specific entity. Consider an *employee* entity set with the attribute *phone-number*. An employee may have zero, one, or several phone numbers, and different employees may have different numbers of phones. This type of attribute is said to be **multivalued**.

- **Derived** attribute. The value for this type of attribute can be derived from the values of other related attributes or entities. For example, If the *customer* entity set also has an attribute *date-of-birth*, we can calculate *age* from *date-of-birth* and the current date. Thus, *age* is a derived attribute. In this case, *date-of-birth* may be referred to as a *base* attribute, or a *stored* attribute. The value of a derived attribute is not stored, but is computed when required.
- An attribute takes a **null** value when an entity does not have a value for it. The *null* value may indicate "not applicable"—that is, that the value does not exist for the entity. For example, one may have no middle name. *Null* can also designate that an attribute value is unknown. An unknown value may be either *missing* (the value does exist, but we do not have that information) or *not known* (we do not know whether or not the value actually exists).

Constraints:

An E-R enterprise schema may define certain constraints to which the contents of a database must conform. Here, we examine mapping cardinalities and participation constraints, which are two of the most important types of constraints.

Mapping Cardinalities:

Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.

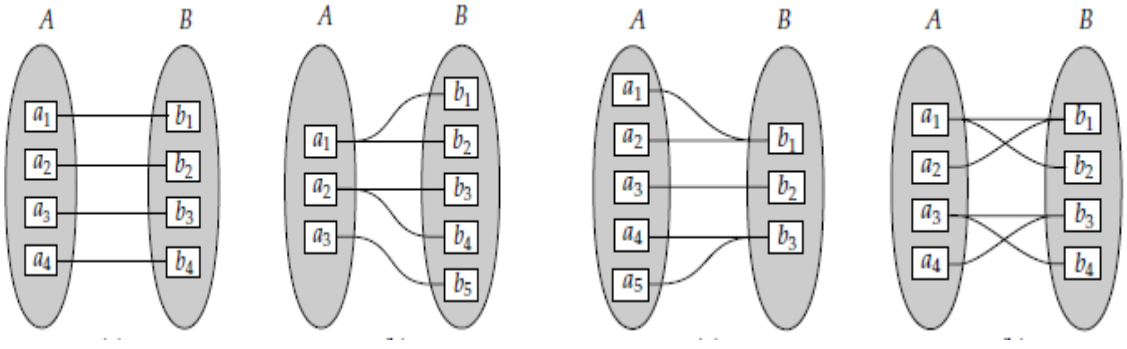
For a binary relationship set *R* between entity sets *A* and *B*, the mapping cardinality must be one of the following:

• **One to one**. An entity in *A* is associated with *at most* one entity in *B*, and an entity in *B* is associated with *at most* one entity in *A*.

• **One to many**. An entity in *A* is associated with any number (zero or more) of entities in *B*. An entity in *B*, however, can be associated with *at most* one entity in *A*.

• **Many to one**. An entity in *A* is associated with *at most* one entity in *B*. An entity in *B*, however, can be associated with any number (zero or more) of entities in *A*.

• **Many to many**. An entity in *A* is associated with any number (zero or more) of entities in *B*, and an entity in *B* is associated with any number (zero or more) of entities in *A*.



(a) **(b)** **(c)** **(d)**

Fig: Mapping Cardinalities (a) One to one. (b)One to many. (c) Many to one. (d) Many to many

Participation Constraints:

The participation of an entity set *E* in a relationship set *R* is said to be **total** if every entity in *E* participates in at least one relationship in *R*. If only some entities in *E* participate in relationships in *R*, the participation of entity set *E* in relationship *R* is said to be **partial**. For example, we expect every loan entity to be related to at least one customer through the *borrower* relationship. Therefore the participation of *loan* in the relationship set *borrower* is total. In contrast, an individual can be a bank customer whether or not she has a loan with the bank. Hence, it is possible that only some of the *customer* entities are related to the *loan* entity set through the *borrower* relationship, and the participation of *customer* in the *borrower* relationship set is therefore partial.

Entity-Relationship Diagram:

An **E-R diagram** can express the overall logical structure of a database graphically. E-R diagrams are simple and clear—qualities that may well account in large part for the widespread use of the E-R model. Such a diagram consists of the following major components:

- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationship sets
- **Lines**, which link attributes to entity sets and entity sets to relationship sets
- **Double ellipses**, which represent multivalued attributes
- **Dashed ellipses**, which denote derived attributes
- **Double lines**, which indicate total participation of an entity in a relationship set
- **Double rectangles**, which represent weak entity sets

Consider the entity-relationship diagram in Figure below, which consists of two entity sets, *customer* and l*oan*, related through a binary relationship set *borrower*. The attributes associated with *customer* are *customer-id*, *customer-name*, *customer-street*, and *customer-city*. The attributes associated with l*oan a*re *loan-number* and *amount*. In Figure, attributes of an entity set that are members of the primary key are underlined. The relationship set *borrower* may be many-to-many, one-to-many, many-to-one, or one-to-one. To distinguish among these types, we draw either a directed line (→) or an undirected line (—) between the relationship set and the entity set in question.

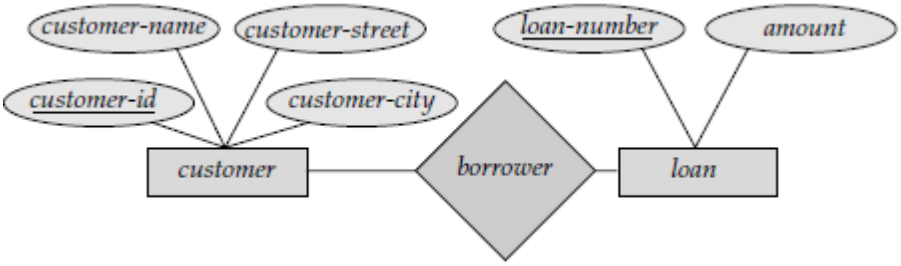


Fig: E-R Diagram corresponding to Customers and loans

- A directed line from the relationship set *borrower* to the entity set *loan* specifies that *borrower* is either a one-to-one or many-to-one relationship set, from *customer* to *loan*; *borrower* cannot be a many-to-many or a one-to-many relationship set from *customer* to *loan*.
- An undirected line from the relationship set *borrower* to the entity set *loan* specifies that *borrower* is either a many-to-many or one-to-many relationship set from *customer* to *loan*.

Returning to the E-R diagram of above Figure, we see that the relationship set *borrower* is many-to-many. If the relationship set *borrower* were one-to-many, from *customer* to *loan*, then the line from

borrower to *customer* would be directed, with an arrow pointing to the *customer* entity set. Similarly, if the relationship set *borrower* were many-to-one from *customer* to *loan*, then the line from *borrower* to *loan* would have an arrow pointing to the *loan* entity set. Finally, if the relationship set *borrower* were one-to-one, then both lines from *borrower* would have arrows: one pointing to the *loan* entity set and one pointing to the *customer* entity set.

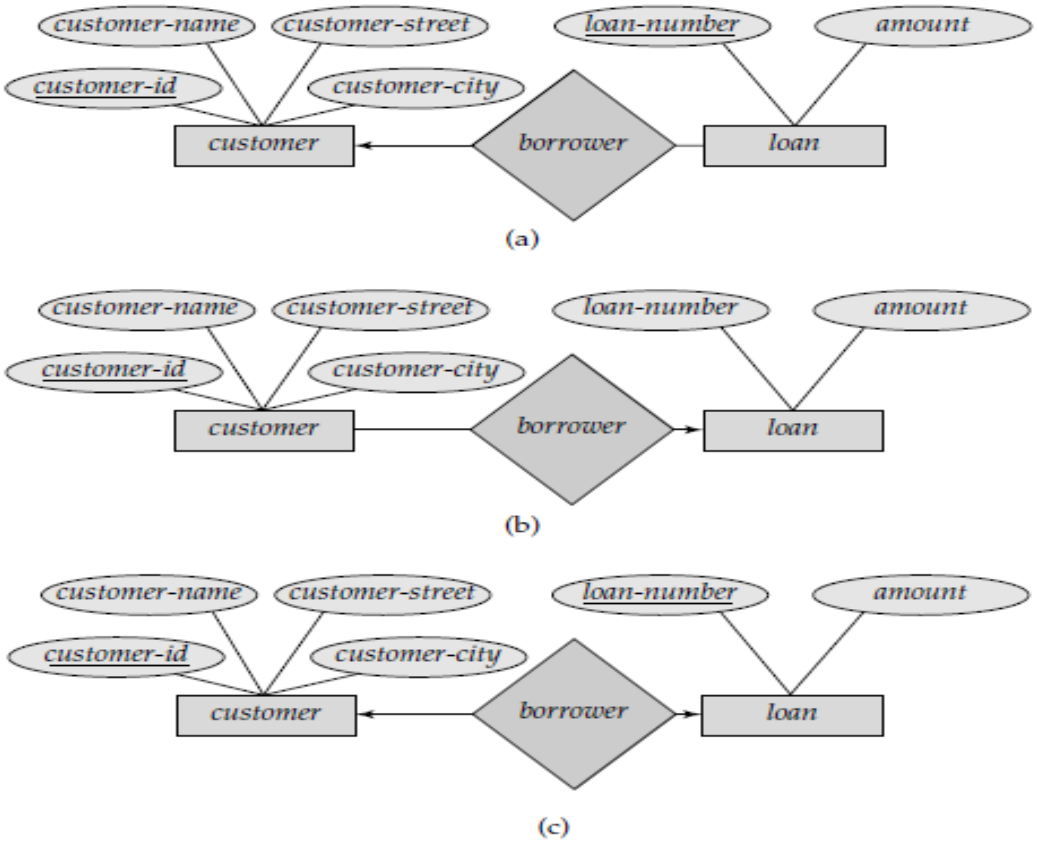


Fig: Relationships. (a) One to many. (b) Many to one. (c)One to one.

If a relationship set has also some attributes associated with it, then we link these attributes to that relationship set. For example, in Figure 2.10, we have the *access-date* descriptive attribute attached to the relationship set *depositor* to specify the most recent date on which a customer accessed that account.

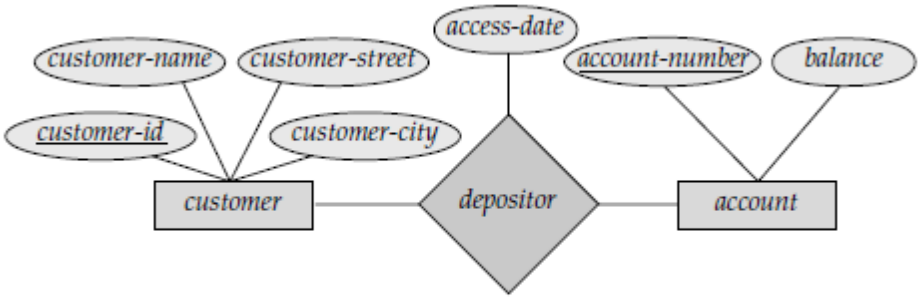


Figure 2.10 E-R diagram with an attribute attached to a relationship set.

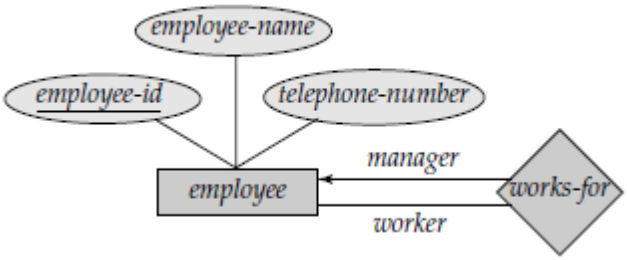


Fig: E-R diagram with role indicators.

Example: Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

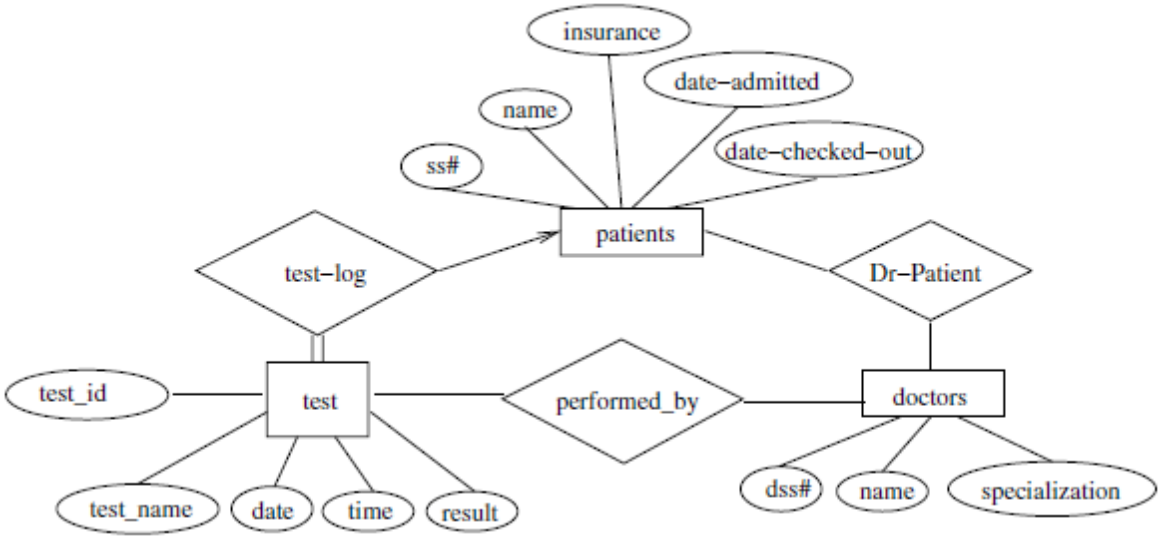


Fig: E-R diagram for a hospital

Example: Consider a database to record the marks that students get in different exams of different course offerings. Construct an E-R diagram that shows relationship, for the above database.

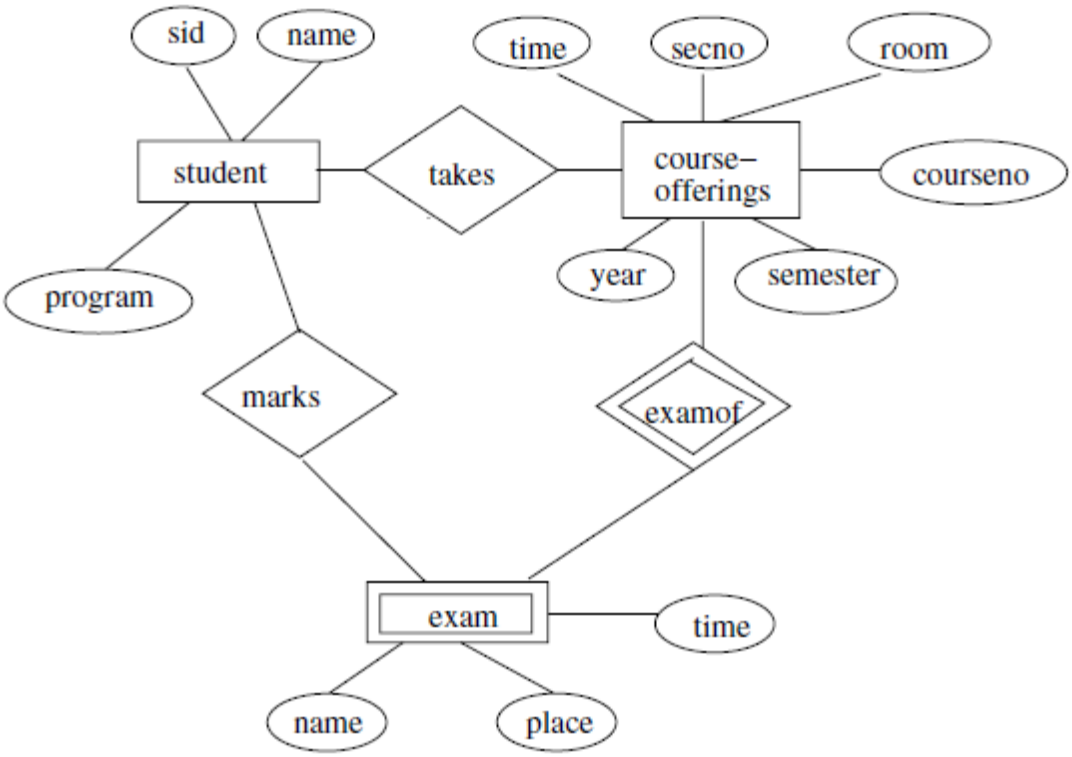


Fig: E-R diagram for marks database

Extended E-R Features:

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. Here, we discuss the extended E-R features of specialization and generalization.

Specialization:

An entity set may include subgroupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set. The E-R model provides a means for representing these distinctive entity groupings. Consider an entity set *person*, with attributes *name*, *street*, and *city*. A person may be further classified as one of the following:

- *customer*
- *employee*

Each of these person types is described by a set of attributes that includes all the attributes of entity set *person* plus possibly additional attributes. For example, *customer* entities may be described further by the attribute *customer-id*, whereas *employee* entities may be described further by the attributes *employee-id* and *salary*. The process of designating subgroupings within an entity set is called **specialization**. The specialization of *person* allows us to distinguish among persons according to whether they are employees or customers.

Generalization:

The refinement from an initial entity set into successive levels of entity subgroupings represents a **top-down** design process in which distinctions are made explicit. The design process may also proceed in a **bottom-up** manner, in which multiple entity sets are synthesized into a higher-level entity set on the

basis of common features. The database designer may have first identified a *customer* entity set with the attributes *name*, *street*, *city*, and *customer-id*, and an *employee* entity set with the attributes *name*, *street*, *city*, *employee-id*, and *salary*. There are similarities between the *customer* entity set and the *employee* entity set in the sense that they have several attributes in common. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets. In our example, *person* is the higher-level entity set and *customer* and *employee* are lower-level entity sets. Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The *person* entity set is the superclass of the *customer* and *employee* subclasses. For all practical purposes, generalization is a simple inversion of specialization.

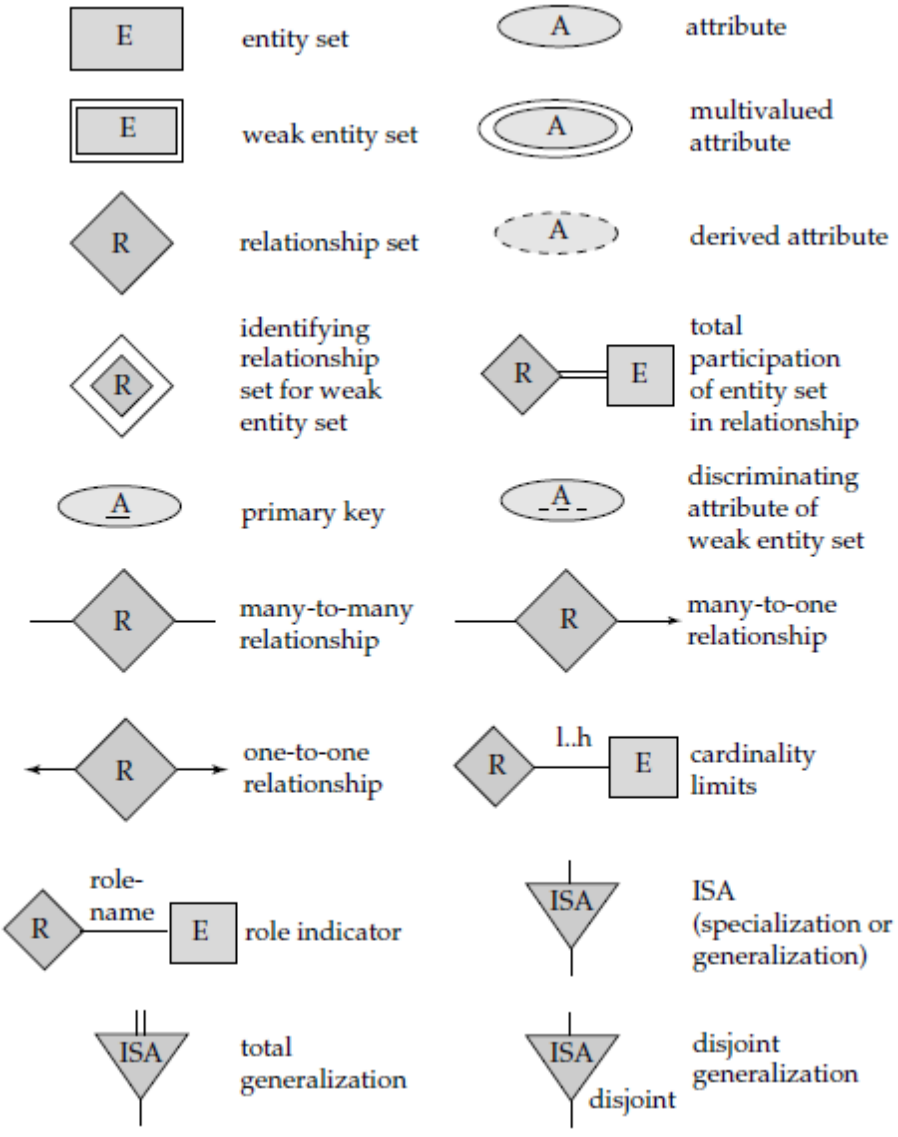


Fig: Symbols used in E-R notations

Appendix: Some Solved example of ER-Diagram

Q. Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

Answer:

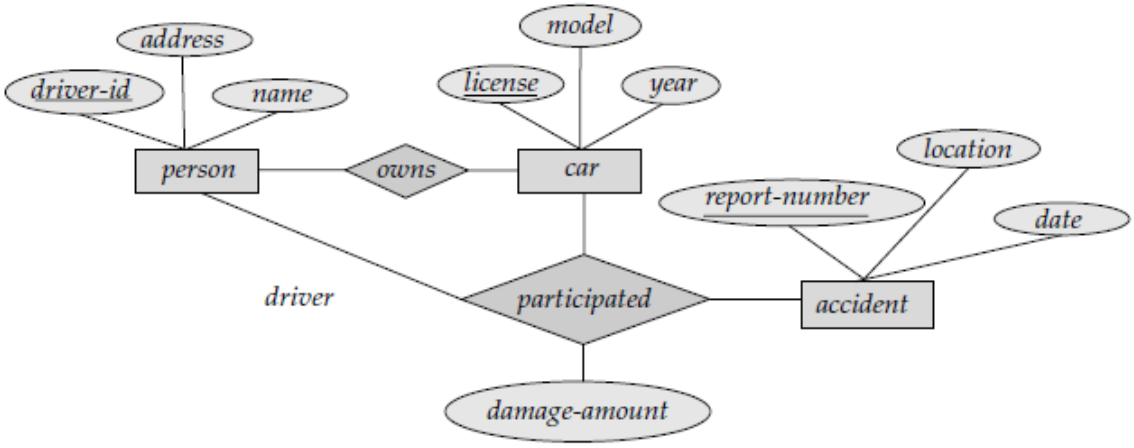


Fig: ER diagram for a car-insurance company

Q. A university registrar's office maintains data about the following entities: (a) courses, including number, title, credits, syllabus, and prerequisites; (b) course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom; (c) students, including student-id, name, and program; and (d) instructors, including identification number, name, department, and title. Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled.

Construct an E-R diagram for the registrar's office. Document all assumptions that you make about the mapping constraints.

Answer: Here, the main entity sets are *student, course, course-offering*, and *instructor*. The entity set *course-offering* is a weak entity set dependent on *course*. The assumptions made are :

a. a class meets only at one particular place and time. This E-R diagram cannot model a class meeting at different places at different times.

b. There is no guarantee that the database does not have two classes meeting at the same place and time.

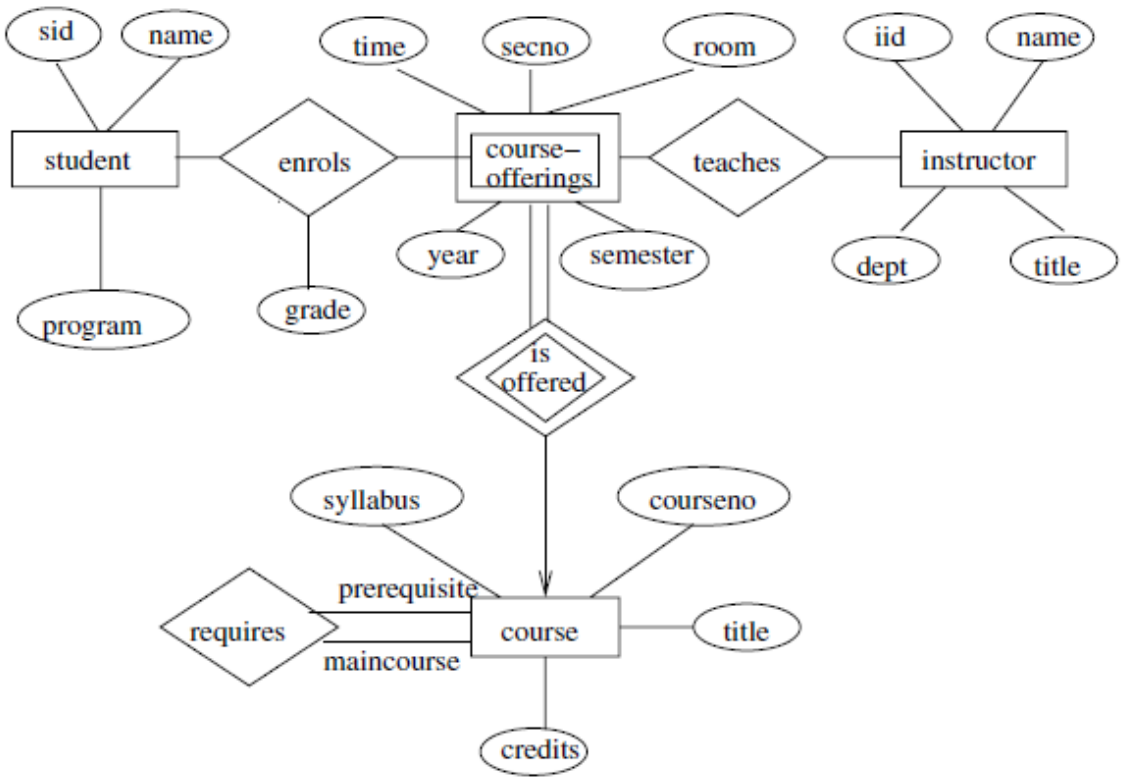


Fig: ER diagram for university

Keys in Relational Model:

Student:

Stud_id	Name	Phone_no	Address
1	Ram	9899967000	Ktm
2	Ram	9796582000	Ktm
3	suresh	9432516782	pokhara

Student_course:

Stud_id	Course_no	Course_name
1	C1	DBMS
2	C2	Computer Network
1	C1	Computer Network

Super Key: The set of one or more attributes which can uniquely identify a tuple is known as Super Key. For Example, STUD_NO, (STUD_NO, STUD_NAME) etc.

- Adding zero or more attributes to candidate key generates super key.
- A candidate key is a super key but vice versa is not true

Candidate Key: The minimal set of attributes which can uniquely identify a tuple is known as candidate key. OR A super key with no redundant attribute is known as candidate key. For Example, STUD_NO in STUDENT relation.

- The value of Candidate Key is unique and non-null for every tuple.

- There can be more than one candidate key in a relation. For Example, STUD_NO as well as STUD_PHONE both are candidate keys for relation STUDENT.
- The candidate key can be simple (having only one attribute) or composite as well. For Example, {STUD_NO, COURSE_NO} is a composite candidate key for relation STUDENT_COURSE.

Primary Key: There can be more than one candidate key in a relation out of which one can be chosen as primary key. For Example, STUD_NO as well as STUD_PHONE both are candidate keys for relation STUDENT but STUD_NO can be chosen as primary key (only one out of many candidate keys).

Foreign Key: If an attribute can only take the values which are present as values of some other attribute, it will be foreign key to the attribute to which it refers. The relation which is being referenced is called referenced relation and corresponding attribute is called referenced attribute and the relation which refers to referenced relation is called referencing relation and corresponding attribute is called referencing attribute. Referenced attribute of referencing attribute should be primary key. For Example, STUD_NO in STUDENT_COURSE is a foreign key to STUD_NO in STUDENT relation.

Schema Diagram:

A database schema, along with primary key and foreign key dependencies, can be depicted pictorially by **schema diagram**s. Figure 3.9 shows the schema diagram for our banking enterprise. Each relation appears as a box, with the attributes listed inside it and the relation's name above it. If there are primary key attributes, a horizontal line crosses the box, with the primary key attributes listed above the line. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

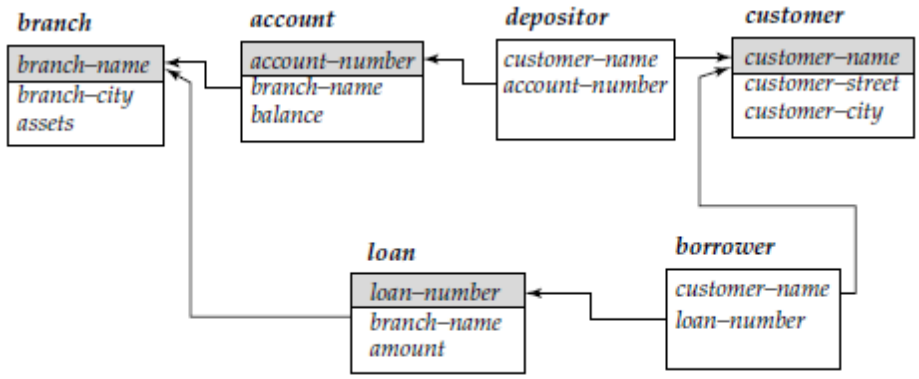


Figure 3.9 Schema diagram for the banking enterprise.

Reducing ER diagrams to Relational Model:

Mapping Entity:

- Create table for each entity.
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key.

Mapping Relationship

- Create table for a relationship.
- Add the primary keys of all participating Entities as fields of table with their respective data types.
- If relationship has any attribute, add each attribute as field of table.
- Declare a primary key composing all the primary keys of participating entities.

- Declare all foreign key constraints.

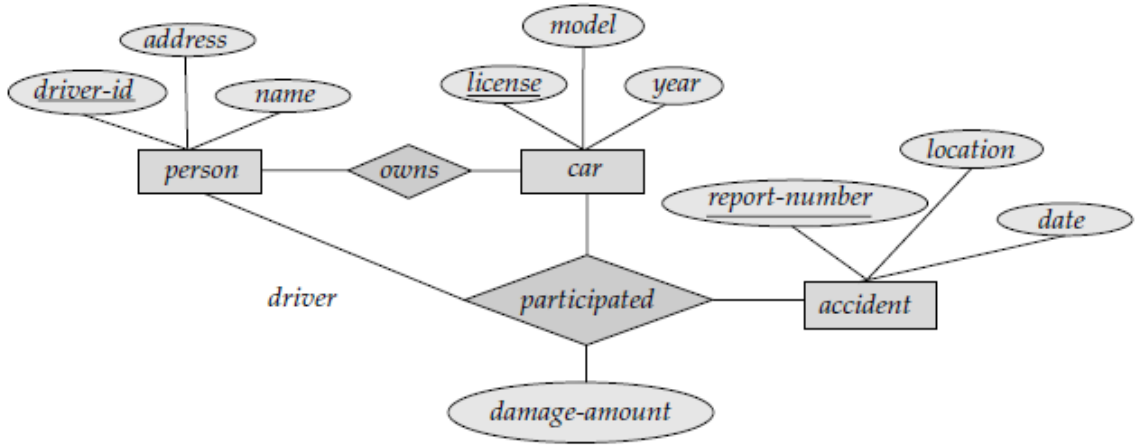
Mapping Weak Entity Sets

- Create table for weak entity set.
- Add all its attributes to table as field.
- Add the primary key of identifying entity set.
- Declare all foreign key constraints.

Mapping Hierarchical Entities

- Create tables for all higher-level entities.
- Create tables for lower-level entities.
- Add primary keys of higher-level entities in the table of lower-level entities.
- In lower-level tables, add all other attributes of lower-level entities.
- Declare primary key of higher-level table and the primary key for lower-level table.
- Declare foreign key constraints.

Example: Design a relational database corresponding to the E-R diagram below.



Answer: The relational database schema is given below.

person (driver-id, name, address)
car (license, year, model)
accident (report-number, location, date)
owns (driver-id, license)
participated (report-number driver-id, license, damage-amount)

The Relational Algebra

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are select, project, union, set difference, Cartesian product, and rename. In addition to the fundamental operations, there are several other operations—namely, set intersection, natural join, division, and assignment.

3.2.1 Fundamental Operations

The select, project, and rename operations are called *unary* operations, because they operate on one relation. The other three operations operate on pairs of relations and are, therefore, called *binary* operations.

3.2.1.1 The Select Operation

The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. Thus, to select those tuples of the *loan* relation where the branch is "Perryridge," we write

$$\sigma_{\text{branch-name} = \text{“Perryridge”}} (\text{loan})$$

If the *loan* relation is as shown in Figure 3.6, then the relation that results from the preceding query is as shown in Figure 3.10.

We can find all tuples in which the amount lent is more than $1200 by writing

$$\sigma_{\text{amount} > 1200} (\text{loan})$$

In general, we allow comparisons using =, _=, <, ≤, >, ≥ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). Thus, to find those tuples pertaining to loans of more than $1200 made by the Perryridge branch, we write

$$\sigma_{\textit{branch-name} = \text{“Perryridge”} \wedge \textit{amount} > 1200} (\textit{loan})$$

loan-number	*branch-name*	*amount*
L-15	Perryridge	1500
L-16	Perryridge	1300

Figure 3.10 Result of $\sigma_{\textit{branch-name}} =$"Perryridge" (*loan*).

3.2.1.2 The Project Operation

Suppose we want to list all loan numbers and the amount of the loans, but do not care about the branch name. The **project** operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter pi (Π).We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses. Thus, we write the query to list all loan numbers and the amount of the loan as

$$\Pi_{\textit{loan-number}, \textit{ amount}} (\textit{loan})$$

3.2.1.3 Composition of Relational Operations

The fact that the result of a relational operation is itself a relation is important. Consider the more complicated query "Find those customers who live in Harrison." We write:

$$\Pi_{\textit{customer-name}} (\sigma_{\textit{customer-city}} = \text{“Harrison”} (\textit{customer}))$$

3.2.1.4 The Union Operation

Consider a query to find the names of all bank customers who have either an account or a loan or both. Note that the *customer* relation does not contain the information, since a customer does not need to have either an account or a loan at the bank. To answer this query, we need the information in the *depositor* relation (Figure 3.5) and in the *borrower* relation (Figure 3.7).We know how to find the names of all customers with a loan in the bank:

$$\Pi_{\textit{customer-name}} (\textit{borrower})$$

We also know how to find the names of all customers with an account in the bank:

$$\Pi_{\textit{customer-name}} (\textit{depositor})$$

To answer the query, we need the **union** of these two sets; that is, we need all customer names that appear in either or both of the two relations.We find these data by the binary operation union, denoted, as in set theory, by $\cup$. So the expression needed is

$\Pi_{customer\text{-}name}(borrower) \cup \Pi_{customer\text{-}name}(depositor)$

The result relation for this query appears in Figure 3.12. Notice that there are 10 tuples in the result, even though there are seven distinct borrowers and six depositors. This apparent discrepancy occurs because Smith, Jones, and Hayes are borrowers as well as depositors. Since relations are sets, duplicate values are eliminated.

customer-name
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

Figure 3.12 Names of all customers who have either a loan or an account.

Therefore, for a union operation $r \cup s$ to be valid, we require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.

2. The domains of the ith attribute of r and the ith attribute of s must be the same, for all i.

Note that r and s can be, in general, temporary relations that are the result of relational-algebra expressions.

3.2.1.5 The Set Difference Operation

The set-difference operation, denoted by −, allows us to find tuples that are in one relation but are not in another. The expression r − s produces a relation containing those tuples in r but not in s.

We can find all customers of the bank who have an account but not a loan by writing

$\Pi_{customer\text{-}name}(depositor) - \Pi_{customer\text{-}name}(borrower)$

The result relation for this query appears in Figure 3.13.

As with the union operation, we must ensure that set differences are taken between *compatible* relations. Therefore, for a set difference operation $r - s$ to be valid, we require that the relations r and s be of the same arity, and that the domains of the ith attribute of r and the ith attribute of s be the same.

3.2.1.6 The Cartesian-Product Operation

The **Cartesian-product** operation, denoted by a cross ($\times$), allows us to combine information from any two relations.We write the Cartesian product of relations r_1 and r_2 as $r_1 \times r_2$.

For example, the relation schema for $\boldsymbol{r = borrower \times loan}$ is

(*borrower.customer-name*, *borrower.loan-number*, *loan.loan-number*, *loan.branch-name*, *loan.amount*)

With this schema, we can distinguish *borrower.loan-number* from *loan.loan-number*. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for r as

(*customer-name*, *borrower.loan-number*, *loan.loan-number*, *branch-name*, *amount*)

Now that we know the relation schema for $r = borrower \times loan$, what tuples appear in r? As you may suspect, we construct a tuple of r out of each possible pair of tuples: one from the *borrower* relation and

one from the *loan* relation. Thus, *r* is a large relation, as you can see from Figure 3.14, which includes only a portion of the tuples that make up *r*.

Assume that we have n_1 tuples in *borrower* and n_2 tuples in *loan*. Then, there are $n_1 * n_2$ ways of choosing a pair of tuples—one tuple from each relation; so there are $n_1 * n_2$ tuples in *r*. In particular, note that for some tuples *t* in *r*, it may be that ***t*[*borrower.loan-number*] _= *t*[*loan.loan-number*].**

Suppose that we want to find the names of all customers who have a loan at the Perryridge branch.We need the information in both the *loan* relation and the *borrower* relation to do so. If we write

$$\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan)$$

then the result is the relation in Figure 3.15. We have a relation that pertains to only the Perryridge branch. However, the *customer-name* column may contain customers who do not have a loan at the Perryridge branch. (If you do not see why that is true, recall that the Cartesian product takes all possible pairings of one tuple from *borrower* with one tuple of *loan*.)

Since the Cartesian-product operation associates *every* tuple of *loan* with every tuple of *borrower*, we know that, if a customer has a loan in the Perryridge branch, then there is some tuple in *borrower* ×*loan* that contains his name, and *borrower.loan-number= loan.loan-number*. So, if we write

$$\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan))$$

we get only those tuples of *borrower* × *loan* that pertain to customers who have a loan at the Perryridge branch.

Finally, since we want only *customer-name*, we do a projection:

$$\Pi_{customer\text{-}name}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{“Perryridge”}}(borrower \times loan)))$$

customer-name	loan-number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.7 The *borrower* relation.

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

Figure 3.6 The *loan* relation.

customer-name	account-number
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

Figure 3.5 The *depositor* relation.

customer-name
Johnson
Lindsay
Turner

Figure 3.13 Customers with an account but no loan.

The result of this expression, shown in Figure 3.16, is the correct answer to our query

customer-name	borrower. loan-number	loan. loan-number	branch-name	amount
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

Figure 3.14 Result of *borrower* × *loan*.

customer-name	borrower. loan-number	loan. loan-number	branch-name	amount
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

Figure 3.15 Result of $\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrower \times loan)$.

customer-name
Adams
Hayes

Figure 3.16 Result of $\Pi_{customer\text{-}name}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(\sigma_{branch\text{-}name = \text{"Perryridge"}}(borrower \times loan)))$

3.2.1.7 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the **rename** operator, denoted by the lowercase Greek letter rho (ρ), lets us do this. Given a relational-algebra expression E, the expression

$$\rho_x(E)$$

returns the result of expression E under the name x.

3.2.3 Additional Operations

3.2.3.1 The Set-Intersection Operation

The first additional-relational algebra operation that we shall define is **set intersection** ($\cap$). Suppose that we wish to find all customers who have both a loan and an account. Using set intersection, we can write

Π*customer-name* (*borrower*) ∩ Π*customer-name* (*depositor*)

Note that we can rewrite any relational algebra expression that uses set intersection by replacing the intersection operation with a pair of set-difference operations as:

$r \cap s = r - (r - s)$

3.2.3.2 The Natural-Join Operation

It is often desirable to simplify certain queries that require a Cartesian product. Usually, a query that involves a Cartesian product includes a selection operation on the result of the Cartesian product. Consider the query "Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount." We first form the Cartesian product of the *borrower* and *loan* relations. Then, we select those tuples that pertain to only the same *loan-number*, followed by the projection of the resulting *customer-name*, *loan-number*, and *amount*:

$$\Pi_{customer\text{-}name,\ loan.loan\text{-}number,\ amount}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan))$$

The *natural join* is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the "join" symbol . The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.

As an illustration, consider again the example "Find the names of all customers who have a loan at the bank, and find the amount of the loan."We express this query by using the natural join as follows:

$$\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (borrower \bowtie\ loan)$$

Since the schemas for *borrower* and *loan* (that is, *Borrower-schema* and *Loan-schema*) have the attribute *loan-number* in common, the natural-join operation considers only pairs of tuples that have the same value on *loan-number*. It combines each such pair of tuples into a single tuple on the union of the two schemas (that is, *customer-name,branch-name, loan-number, amount*). After performing the projection, we obtain the relation in Figure 3.21.

customer-name	*loan-number*	*amount*
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-23	2000
Smith	L-11	900
Williams	L-17	1000

Figure 3.21 Result of $\Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (borrower \bowtie\ loan)$.

We are now ready for a formal definition of the natural join. Consider two relations $r(R)$ and $s(S)$. The **natural join** of r and s, denoted by $r\ s$, is a relation on schema $R \cup S$ formally defined as follows:

$$r \bowtie\ s\ =\ \Pi_{R \cup S}\ (\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}\ r\ \times\ s)$$

where $R \cap S = \{A_1, A_2, \ldots, A_n\}$.

• Find the names of all branches with customers who have an account in the bank and who live in Harrison.

$$\Pi_{branch\text{-}name}(\sigma_{customer\text{-}city = \text{"Harrison"}}(customer \bowtie account \bowtie depositor))$$

Find all customers who have *both* a loan and an account at the bank.

$$\Pi_{customer\text{-}name}(borrower \bowtie depositor)$$

3.2.3.3 The Division Operation

The **division** operation, denoted by ÷, is suited to queries that include the phrase "for all." Suppose that we wish to find all customers who have an account at *all* the branches located in Brooklyn. We can obtain all branches in Brooklyn by the expression

$r1 = \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch))$

We can find all (*customer-name, branch-name*) pairs for which the customer has an account at a branch by writing

$$r_2 = \Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account)$$

Figure 3.24 shows the result relation for this expression.Now, we need to find customers who appear in r_2 with *every* branch name in r_1. The operation that provides exactly those customers is the divide operation. We formulate the query by writing

$$\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account) \div \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch))$$

A tuple t is in $r \div s$ if and only if both of two conditions hold:

1. t is in $\Pi_{R-S}(r)$

2. For every tuple t_s in s, there is a tuple t_r in r satisfying both of the following:

a. $t_r[S] = t_s[S]$

b. $t_r[R - S] = t$

3.3.2 Aggregate Functions

Aggregate functions take a collection of values and return a single value as a result. To illustrate the concept of aggregation, we shall use the *pt-works* relation in Figure 3.27, for part-time employees. Suppose that we want to find out the total sum of salaries of all the part-time employees in the bank. The relational-algebra expression for this query is:

$\mathcal{G}$ **sum(*salary*)(*pt-works*)**

The symbol G is the letter G in calligraphic font; read it as "calligraphic G." The relational-algebra operation G signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied. If we do want to eliminate duplicates, we use the same function names as before, with the addition of the hyphenated string "distinct" appended to the end of the function name (for example, count-distinct). An example arises in the query "Find the number of branches appearing in the pt-works relation."

Gcount-distinct($branch\text{-}name$)($pt\text{-}works$)

Suppose we want to find the total salary sum of all part-time employees at each branch of the bank separately, rather than the sum for the entire bank. To do so, we need to partition the relation *pt-works* into groups based on the branch, and to apply the aggregate function on each group.

$branch\text{-}name\ G$sum($salary$)($pt\text{-}works$)

Going back to our earlier example, if we want to find the maximum salary for part-time employees at each branch, in addition to the sum of the salaries, we write the expression

$branch\text{-}name\ G$sum($salary$),max($salary$)($pt\text{-}works$)

3.4 Modification of the Database

3.4.1 Deletion

We express a delete request in much the same way as a query. However, instead of displaying tuples to the user, we remove the selected tuples from the database. We can delete only whole tuples; we cannot delete values on only particular attributes.

In relational algebra a deletion is expressed by

$$r \leftarrow r - E$$

where r is a relation and E is a relational-algebra query. Here are several examples of relational-algebra delete requests:

• Delete all of Smith's account records.

$depositor \leftarrow depositor - \sigma_{customer\text{-}name} =$"Smith" ($depositor$)

• Delete all loans with amount in the range 0 to 50.

$loan \leftarrow loan - \sigma_{amount \geq 0 \text{ and } amount \leq 50}$ ($loan$)

• Delete all accounts at branches located in Needham.

$r1 \leftarrow \sigma_{branch\text{-}city} =$"Needham" ($account$ ⋈ $branch$)

$r2 \leftarrow \Pi_{branch\text{-}name,\ account\text{-}number,\ balance}$ ($r1$)

$account \leftarrow account - r2$

Note that, in the final example, we simplified our expression by using assignment to temporary relations (r_1 and r_2).

3.4.2 Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the attribute's domain. Similarly, tuples inserted must be of the correct arity. The relational algebra expresses an insertion by $r \leftarrow r \cup E$

where r is a relation and E is a relational-algebra expression.

Suppose that we wish to insert the fact that Smith has $1200 in account A-973 at the Perryridge branch.We write

$account \leftarrow account \cup \{$("A-973", "Perryridge", 1200)$\}$

$depositor \leftarrow depositor \cup \{$("Smith", "A-973")$\}$

3.4.3 Updating

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple.We can use the generalized-projection operator to do this task:

$$r \leftarrow \Pi_{F1,F2,\ldots,Fn}(r)$$

where each F_i is either the ith attribute of r, if the ith attribute is not updated, or, if the attribute is to be updated, F_i is an expression, involving only constants and the attributes of r, that gives the new value for the attribute.If we want to select some tuples from r and to update only them, we can use the following expression; here, P denotes the selection condition that chooses which tuples to update:

$$r \leftarrow \Pi F1, F2, \ldots, Fn(\sigma P(r)) \cup (r - \sigma P(r))$$

To illustrate the use of the update operation, suppose that interest payments are being made, and that all balances are to be increased by 5 percent.We write

$$account \leftarrow \Pi account\text{-}number,\ branch\text{-}name,\ balance * 1.05\ (account)$$

Now suppose that accounts with balances over $10,000 receive 6 percent interest, whereas all others receive 5 percent.We write

$$account \leftarrow \Pi_{AN, BN, balance * 1.06\ (\sigma balance > 10000\ (account))} \cup \Pi_{AN, BN\ balance * 1.05\ (\sigma balance \leq 10000\ (account))}$$

where the abbreviations *AN* and *BN* stand for *account-number* and *branch-name*, respectively