
Databases and Database Users Introduction, An Example, Characteristics of the database approach, Actors on the scene, Workers behind the scenes.

Database System Concepts and Architecture Data models, Schemas and Instances, Three-Schema Architecture and data Independence, Database languages and interfaces.

Data Modeling Using the Entity–Relationship (ER) Model Entity Types, Entity Sets, Attributes and Keys, Relationship types, Relationship Sets, Roles and Structural Constraints, Weak Entity Types, Refining the ER Design for the COMPANY Database, ER Diagrams, Naming Conventions, and Design Issues.

The Relational Data Model and Relational Database Constraints Relational Model Concepts, Relational Model Constraints and Relational Database Schemas, Update Operations, transactions, and dealing with constraint violations.

The Relational Algebra and Relational Calculus Unary Relational Operations: SELECT and PROJECT, Relational Algebra Operations from Set Theory. Binary Relational Operations: JOIN and DIVISION, Additional Relational Operations; Examples of Queries in Relational Algebra.

Introduction

A **database** is a collection of related data. By **data**, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book or you may have stored it on a hard drive, using a personal computer and software such as Microsoft Access or Excel. This collection of related data with an implicit meaning is a database.

A database has the following implicit properties:

- A database represents some aspect of the real world, sometimes called the miniworld. Changes to the miniworld are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

In other words, a database has some source from which data is derived, some degree of interaction with events in the real world, and an audience that is actively interested in its contents.

A **database management system (DBMS)** is a collection of programs that enables users to create and maintain a database. The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, manipulating, and sharing databases among various users and applications. Defining a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary; it is called **meta-data**.

Constructing the database is the process of storing the data on some storage medium that is controlled by the DBMS. Manipulating a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data. **Sharing** a database allows multiple users and programs to access the database simultaneously.

An **application program** accesses the database by sending queries or requests for data to the DBMS. Other important functions provided by the DBMS include protecting the database and maintaining it over a long period of time. **Protection** includes system

protection against hardware or software malfunction (or crashes) and security protection against unauthorized or malicious access. We will call the database and DBMS software together a **database system**.

An Example

A UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. Figure 1 shows the database structure and a few sample data for such a database. The database is organized as five files, each of which stores data records of the same type. The STUDENT file stores data on each student, the COURSE file stores data on each course, the SECTION file stores data on each section of a course, the GRADE_REPORT file stores the grades that students receive in the various sections they have completed, and the PREREQUISITE file stores the prerequisites of each course.

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

***Figure 1**: Database that stores Student and Course information*

To define this database, we must specify the structure of the records of each file by specifying the different types of data elements to be stored in each record. In Figure 1, each STUDENT record includes data to represent the student's Name, Student_number, Class (such as freshman or '1', sophomore or '2', and so forth), and Major (such as mathematics or 'MATH' and computer science or 'CS'); each COURSE record includes data to represent the Course_name, Course_number, Credit_hours and Department (the department that offers the course); and so on.

We must also specify a data type for each data element within a record. For example, we can specify that Name of STUDENT is a string of alphabetic characters, Student_number of STUDENT is an integer, and Grade of GRADE_REPORT is a single character from the set {'A', 'B', 'C', 'D', 'F', 'I'}. We may also use a coding scheme to represent the values of a data item. For example, in Figure 1 we represent the Class of a STUDENT as 1 for freshman, 2 for sophomore, 3 for junior, 4 for senior, and 5 for graduate student.

Notice that records in the various files may be related. For example, the record for Smith in the STUDENT file is related to two records in the GRADE_REPORT file that specify Smith 's grades in two sections. Similarly, each record in the PREREQUISITE file relates two course records: one representing the course and the other representing the prerequisite.

Characteristics of the Database Approach

In traditional file processing, each user defines and implements the files needed for a specific application as part of programming the application. For example, one user, the grade reporting office, may keep files on students and their grades. Programs to print a student's transcript and to enter new grades are implemented as part of the application. A second user, the accounting office, may keep track of students' fees and their payments. Although both users are interested in data about students, each user maintains separate files—and programs to manipulate these files—because each requires some data not available from the other user's files. This redundancy in defining and storing data results in wasted storage space and in redundant efforts to maintain common up-to-date data.

In the database approach, a single repository maintains data that is defined once and then accessed by various users. In file systems, each application is free to name data elements independently. In contrast, in a database, the names or labels of data are defined once, and used repeatedly by queries, transactions, and applications.

The main characteristics of the database approach versus the file-processing approach are the following:

- Self-describing nature of a database system
- Insulation between programs and data, and data abstraction
- Support of multiple views of the data
- Sharing of data and multiuser transaction processing

Self-Describing Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database structure and constraints. This definition is stored in the DBMS catalog, which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called meta-data.

The catalog is used by the DBMS software and also by database users who need

information about the database structure. A general-purpose DBMS software package is not written for a specific database application. Therefore, it must refer to the catalog to know the structure of the files in a specific database. The DBMS software must work equally well with any number of database applications—for example, a university database, a banking database, or a company database—as long as the database definition is stored in the catalog.

In traditional file processing, data definition is typically part of the application programs themselves. Hence, these programs are constrained to work with only one specific database, whose structure is declared in the application programs.

For the example shown in Figure 1, the DBMS catalog will store the definitions of all the files shown. Figure 2 shows some sample entries in a database catalog. These definitions are specified by the database designer prior to creating the actual database and are stored in the catalog. Whenever a request is made to access, say, the Name of a STUDENT record, the DBMS software refers to the catalog to determine the structure of the STUDENT file and the position and size of the Name data item within a STUDENT record.

RELATIONS

Relation_name	No_of_columns
STUDENT	4
COURSE	4
SECTION	5
GRADE_REPORT	3
PREREQUISITE	2

COLUMNS

Column_name	Data_type	Belongs_to_relation
Name	Character (30)	STUDENT
Student_number	Character (4)	STUDENT
Class	Integer (1)	STUDENT
Major	Major_type	STUDENT
Course_name	Character (10)	COURSE
Course_number	XXXXNNNN	COURSE
....	....	.....
....	....	.....
....	....	.....
Prerequisite_number	XXXXNNNN	PREREQUISITE

***Figure 2**: An Example of a database catalog for the database in Figure I*

Insulation between Programs and Data, and Data Abstraction

In traditional file processing, the structure of data files is embedded in the application programs, so any changes to the structure of a file may require changing all programs that access that file. By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property **program-data independence**.

For example, a file access program may be written in such a way that it can access only STUDENT records of the structure shown in Figure 3. If we want to add another piece

of data to each STUDENT record, say the Birth_date , such a program will no longer work and must be changed. By contrast, in a DBMS environment, we only need to change the description of STUDENT records in the catalog to reflect the inclusion of the new data item Birth_date ; no programs are changed.

Data Item Name	Starting Position in Record	Length in Characters (bytes)
Name	1	30
Student_number	31	4
Class	35	1
Major	36	4

***Figure 3**: Internal storage format for a STUDENT record, based on thedatabase catalog in Figure II.*

In some types of database systems, such as object-oriented and object-relational systems, users can define operations on data as part of the database definitions. An operation (also called a function or method) is specified in two parts. The interface (or signature) of an operation includes the operation name and the data types of its arguments (or parameters). The implementation (or method) of the operation is specified separately and can be changed without affecting the interface. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence**.

The characteristic that allows program-data independence and program-operation independence is called **data abstraction**. A DBMS provides users with a **conceptual representation** of data that does not include many of the details of how the data is stored or how the operations are implemented. Informally, a **data model** is a type of data abstraction that is used to provide this conceptual representation.

The internal implementation of a file may be defined by its record length—the number of characters (bytes) in each record—and each data item may be specified by its starting byte within a record and its length in bytes. The STUDENT record would thus be represented as shown in Figure 3. But a typical database user is not concerned with the location of each data item within a record or its length; rather, the user is concerned that when a reference is made to Name of STUDENT , the correct value is returned. A conceptual representation of the STUDENT records is shown in Figure 1.

In the database approach, the detailed structure and organization of each file are stored in the catalog. Database users and application programs refer to the conceptual representation of the files, and the DBMS extracts the details of file storage from the catalog when these are needed by the DBMS file access modules.

Support of Multiple Views of the Data

A database typically has many users, each of whom may require a different perspective or **view** of the database. A view may be a subset of the database or it may contain **virtual data** that is derived from the database files but is not explicitly stored. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views. For example, one user of the database of Figure 1 may be interested only in accessing and printing the transcript of each student; the view for this user is shown in Figure 4(a). A second user, who is interested only in checking that students have taken all the prerequisites of each course for which they register, may require the view shown in Figure 4(b).

(a)

TRANSCRIPT

Student_name	Student_transcript				
	Course_number	Grade	Semester	Year	Section_id
Smith	CS1310	C	Fall	08	119
	MATH2410	B	Fall	08	112
Brown	MATH2410	A	Fall	07	85
	CS1310	A	Fall	07	92
	CS3320	B	Spring	08	102
	CS3380	A	Fall	08	135

(b)

COURSE_PREREQUISITES

Course_name	Course_number	Prerequisites
Database	CS3380	CS3320
		MATH2410
Data Structures	CS3320	CS1310

Figure 4*: Two views derived from the database in Figure I. (a) The TRANSCRIPT view.(b) The COURSE_PREREQUISITES view.*

Sharing of Data and Multiuser Transaction Processing

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include **concurrency control** software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. For example, when several reservation agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time for assignment to a passenger.

A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly and efficiently. The concept of a **transaction** has become central to many database applications. A transaction is an executing program or process that includes one or more database accesses, such as reading or updating of database records. The DBMS must enforce several transaction properties. The **isolation** property ensures that each transaction appears to execute in isolation from other transactions, even though hundreds of transactions may be executing concurrently. The **atomicity** property ensures that either all the database operations in a transaction are executed or none are.

Actors on the Scene

In large organizations, many people are involved in the design, use, and maintenance of a large database with hundreds of users. In this section we identify the people whose jobs involve the day-to-day use of a large database; we call them the actors on the scene.

Database Administrators

In a database environment, the primary resource is the database itself, and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the **database administrator (DBA)**. The DBA is responsible for authorizing access to the database, coordinating and monitoring its use, and acquiring

software and hardware resources as needed. The DBA is accountable for problems such as security breaches and poor system response time.

Database Designers

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. These tasks are mostly undertaken before the database is actually implemented and populated with data. Database designers typically interact with each potential group of users and develop views of the database that meet the data and processing requirements of these groups. Each view is then analyzed and integrated with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

End Users

End users are the people whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use. There are several categories of end users:

- **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests.
- **Naive or parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called **canned transactions** —that have been carefully programmed and tested. The tasks that such users perform are varied: Bank tellers check account balances and post withdrawals and deposits. Reservation agents for airlines, hotels, and car rental companies check availability for a given request and make reservations.
- **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS in order to implement their own applications to meet their complex requirements.
- **Standalone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces.

System Analysts and Application Programmers (Software Engineers)

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for standard canned transactions that meet these requirements. **Application programmers** implement these specifications as programs; then they test, debug, document, and maintain these canned transactions. Such analysts and programmers—commonly referred to as **software developers** or **software engineers**—should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

Workers behind the Scene

In addition to those who design, use, and administer a database, others are associated with the design, development, and operation of the DBMS software and system environment. These persons are typically not interested in the database content itself. We call them the workers behind the scene, and they include the following categories:

- **DBMS system designers and implementers** design and implement the DBMS modules and interfaces as a software package. A DBMS is a very complex software system that consists of many components, or modules, including modules for implementing the catalog, query language processing, interface processing, accessing and buffering data, controlling concurrency, and handling data recovery and security.
- **Tool developers** design and implement tools—the software packages that facilitate

database modeling and design, database system design, and improved performance.

- **Operators and maintenance personnel** (system administration personnel) are responsible for the actual running and maintenance of the hardware and software environment for the database system.

Advantages of Using the DBMS Approach

Controlling Redundancy

In the database approach, the views of different user groups are integrated during database design. Ideally, we should have a database design that stores each logical data item—such as a student's name or birth date—in only one place in the database. This is known as **data normalization**, and it ensures consistency and saves storage space. However, in practice, it is sometimes necessary to use controlled redundancy to improve the performance of queries. For example, we may store Student_name and Course_number redundantly in a GRADE_REPORT file because whenever we retrieve a GRADE_REPORT record, we want to retrieve the student name and course number along with the grade, student number, and section identifier. By placing all the data together, we do not have to search multiple files to collect this data. This is known as **denormalization**.

In such cases, the DBMS should have the capability to control this redundancy in order to prohibit inconsistencies among the files. This may be done by automatically checking that the Student_name–Student_number values in any GRADE_REPORT record in Figure 5(a) match one of the Name–Student_number values of a STUDENT record (Figure 1). Similarly, the Section_identifier–Course_number values in GRADE_REPORT can be checked against SECTION records. Such checks can be specified to the DBMS during database design and automatically enforced by the DBMS whenever the GRADE_REPORT file is updated. Figure 5(b) shows a GRADE_REPORT record that is inconsistent with the STUDENT file in Figure 1.

(a)

GRADE_REPORT

Student_number	Student_name	Section_identifier	Course_number	Grade
17	Smith	112	MATH2410	B
17	Smith	119	CS1310	C
8	Brown	85	MATH2410	A
8	Brown	92	CS1310	A
8	Brown	102	CS3320	B
8	Brown	135	CS3380	A

(b)

GRADE_REPORT

Student_number	Student_name	Section_identifier	Course_number	Grade
17	Brown	112	MATH2410	B

***Figure 5**: Redundant storageof Student_name and Course_name in GRADE_REPORT.(a) Consistent data.(b) Inconsistent record.*

Restricting Unauthorized Access

When multiple users share a large database, it is likely that most users will not be

authorized to access all information in the database. For example, financial data is often considered confidential, and only authorized persons are allowed to access such data. In addition, some users may only be permitted to retrieve data, whereas others are allowed to retrieve and update. Hence, the type of access operation— retrieval or update—must also be controlled.

Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database. A DBMS should provide a security and authorization subsystem, which the DBA uses to create accounts and to specify account restrictions.

Providing Persistent Storage for Program Objects

Databases can be used to provide persistent storage for program objects and data structures. The values of program variables or objects are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. When the need arises to read this data once more, the programmer must convert from the file format to the program variable or object structure.

A complex object in C++ can be stored permanently in an object-oriented DBMS. Such an object is said to be persistent, since it survives the termination of program execution and can later be directly retrieved by another C++ program.

Providing Storage Structures and Search Techniques for Efficient Query Processing

Database systems must provide capabilities for efficiently executing queries and updates. Because the database is typically stored on disk, the DBMS must provide specialized data structures and search techniques to speed up disk search for the desired records.

In order to process the database records needed by a particular query, those records must be copied from disk to main memory. Therefore, the DBMS often has a **buffering** or **caching** module that maintains parts of the database in main memory buffers. In general, the operating system is responsible for disk-to-memory buffering. However, because data buffering is crucial to the DBMS performance, most DBMSs do their own data buffering.

Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The backup and recovery subsystem of the DBMS is responsible for recovery. For example, if the computer system fails in the middle of a complex update transaction, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the transaction started executing. Disk backup is also necessary in case of a catastrophic disk failure.

Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include query languages for casual users, programming language interfaces for application programmers, forms and command codes for parametric users, and menu-driven interfaces and natural language interfaces for standalone users.

Representing Complex Relationships among Data

A database may include numerous varieties of data that are interrelated in many ways. Consider the example shown in Figure 1. The record for ‘Brown’ in the STUDENT file is related to four records in the GRADE_REPORT file. Similarly, each section record is related to one course record and to a number of GRADE_REPORT records—one for each student who completed that section. A DBMS must have the capability to represent a variety of complex relationships among the data, to define new relationships as they arise, and to

retrieve and update related data easily and efficiently.

Enforcing Integrity Constraints

Most database applications have certain integrity constraints that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints. The simplest type of integrity constraint involves specifying a data type for each data item.

A more complex type of constraint that frequently occurs involves specifying that a record in one file must be related to records in other files. For example, in Figure 1, we can specify that every section record must be related to a course record. This is known as a referential integrity constraint.

Another type of constraint specifies uniqueness on data item values, such as every course record must have a unique value for Course_number . This is known as a key or uniqueness constraint. It is the responsibility of the database designers to identify integrity constraints during database design.

Additional Implications of Using the Database Approach

- Potential for Enforcing Standards. The database approach permits the DBA to define and enforce standards among database users in a large organization. This facilitates communication and cooperation among various departments, projects, and users within the organization. Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on.
- Reduced Application Development Time. A prime selling feature of the database approach is that developing a new application—such as the retrieval of certain data from the database for printing a new report—takes very little time. However, once a database is up and running, substantially less time is generally required to create new applications using DBMS facilities.
- Flexibility. It may be necessary to change the structure of a database as requirements change. For example, a new user group may emerge that needs information not currently in the database. In response, it may be necessary to add a file to the database or to extend the data elements in an existing file. Modern DBMSs allow certain types of evolutionary changes to the structure of the database without affecting the stored data and the existing application programs.
- Availability of Up-to-Date Information. A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can immediately see this update. This availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases, and it is made possible by the concurrency control and recovery subsystems of a DBMS.

Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction. Data abstraction generally refers to the suppression of details of data organization and storage, and the highlighting of the essential features for an improved understanding of data. One of the main characteristics of the database approach is to support data abstraction so that different users can perceive data at their preferred level of detail. A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction. By structure of a database we mean the data types, relationships, and constraints that apply to the data.

Categories of Data Models

Many data models have been proposed, which we can categorize according to the types of

concepts they use to describe the database structure. **High-level** or **conceptual data models** provide concepts that are close to the way many users perceive data, whereas **low-level** or **physical data models** provide concepts that describe the details of how data is stored on the computer storage media, typically magnetic disks.

Between these two extremes is a class of **representational (or implementation) data models**, which provide concepts that may be easily understood by end users but that are not too far removed from the way data is organized in computer storage. Representational data models hide many details of data storage on disk but can be implemented on a computer system directly.

Conceptual data models use concepts such as entities, attributes, and relationships. An **entity** represents a real-world object or concept, such as an employee or a project from the miniworld that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A relationship among two or more entities represents an association among the entities, for example, a works-on relationship between an employee and a project.

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs. These include the widely used relational data model, as well as the so-called legacy data models—the network and hierarchical models—that have been widely used in the past. Representational data models represent data by using record structures and hence are sometimes called record-based data models.

Physical data models describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths. An **access path** is a structure that makes the search for particular database records efficient. An index is an example of an access path that allows direct access to data using an **index** term or a keyword.

Schemas, Instances, and Database State

In any data model, it is important to distinguish between the description of the database and the database itself. The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently. A displayed schema is called a **schema diagram**. Figure 6 shows a schema diagram for the database shown in Figure 1; the diagram displays the structure of each record type but not the actual instances of records. We call each object in the schema—such as STUDENT or COURSE —a **schema construct**.

A schema diagram displays only some aspects of a schema, such as the names of record types and data items, and some types of constraints. Other aspects are not specified in the schema diagram; for example, Figure 6 shows neither the data type of each data item, nor the relationships among the various files. The actual data in a database may change quite frequently. For example, the database shown in Figure 1 changes every time we add a new student or enter a new grade. The data in the database at a particular moment in time is called a **database state** or **snapshot**.

In a given database state, each schema construct has its own current set of instances; for example, the STUDENT construct will contain the set of individual student entities (records) as its instances. Every time we insert or delete a record or change the value of a data item in a record, we change one state of the database into another state.

When we define a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the empty state with no data. We get the initial state of the database when the database is first populated or loaded with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a current state.

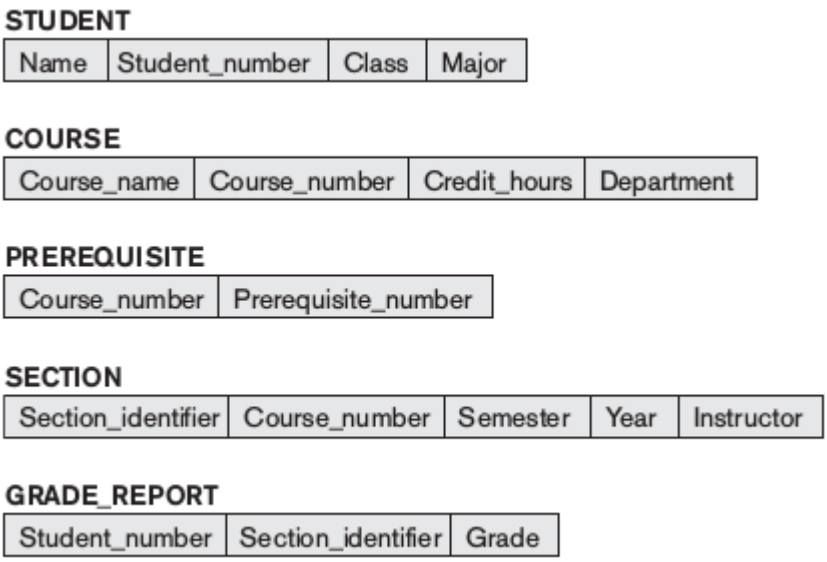


***Figure 6**: Schema diagram for the database in Figure I.*

The DBMS stores the descriptions of the schema constructs and constraints—also called the meta-data—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

Although, as mentioned earlier, the schema is not supposed to change frequently, it is not uncommon that changes occasionally need to be applied to the schema as the application requirements change. For example, we may decide that another data item needs to be stored for each record in a file, such as adding the Date_of_birth to the STUDENT schema in Figure 6. This is known as schema evolution.

Three-Schema Architecture and Data Independence

Three of the four important characteristics of the database approach

1. use of a catalog to store the database description (schema) so as to make it self-describing,
2. insulation of programs and data (program-data and program-operation independence)
3. support of multiple user views.

The three-schema architecture, was proposed to help achieve and visualize these characteristics. The goal of the three-schema architecture is to separate the user applications from the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented.
3. The **external** or **view level** includes a number of **external schemas** or **user views**. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. As in the

previous level, each external schema is typically implemented using a representational data model.

The three schemas are only descriptions of data; the stored data that actually exists is at the physical level only. In a DBMS based on the three-schema architecture, each user group refers to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called mappings.

These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

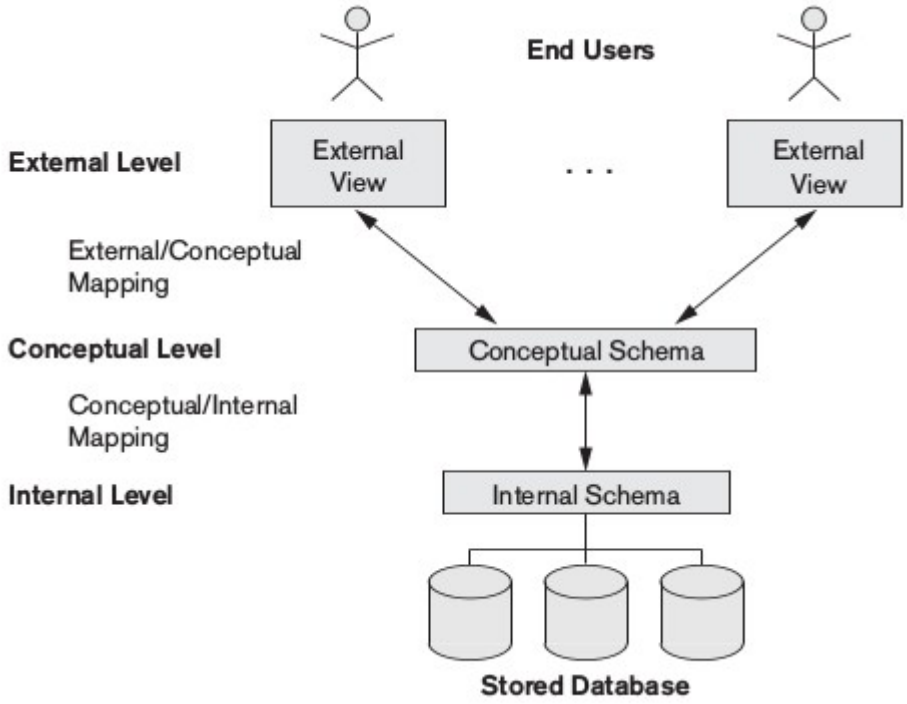


***Figure 7**: The three-schema architecture*

Data Independence

The three-schema architecture can be used to further explain the concept of data independence, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). Only the view definition and the mappings need to be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before.

2. Physical data independence is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures —to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema.

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged; only the mapping between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed. Three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS.

Database Languages and Interfaces

DBMS Languages

Once the design of a database is completed and a DBMS is chosen to implement the database, the first step is to specify conceptual and internal schemas for the database and any mappings between the two. In many DBMSs where no strict separation of levels is maintained, one language, called the **data definition language (DDL)**, is used by the DBA and by database designers to define both schemas.

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the **storage definition language (SDL)**, is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages.

For a true three-schema architecture, we would need a third language, the **view definition language (VDL)**, to specify user views and their mappings to the conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a set of operations or a language called the **data manipulation language (DML)** for these purposes.

There are two main types of DMLs. A **high-level or nonprocedural DML** can be used on its own to specify complex database operations concisely. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. A **low-level or procedural DML** must be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately.

Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**. On the other hand, a high-level DML used in a standalone interactive manner is called a **query language**. Casual end users typically use a high-level query language to specify their requests, whereas programmers use the DML in its embedded form.

DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following:

Menu-Based Interfaces for Web Clients or Browsing. These interfaces present the user with lists of options (called menus) that lead the user through the formulation of a request. The query is composed step-by- step by picking options from a menu that is displayed by the system.

Forms-Based Interfaces. A forms-based interface displays a form to each user. Users can fill out all of the form entries to insert new data, or they can fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions.

Graphical User Interfaces. A GUI typically displays a schema to the user in diagrammatic form. The user then can specify a query by manipulating the diagram.

Natural Language Interfaces. These interfaces accept requests written in English or some other language and attempt to understand them. A natural language interface usually has its own schema, which is similar to the database conceptual schema, as well as a dictionary of important words. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request.

Speech Input and Output. The speech input is detected using a library of predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech takes place.

Interfaces for Parametric Users. Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. For example, a teller is able to use single function keys to invoke routine and repetitive transactions such as account deposits or withdrawals, or balance inquiries. This allows the parametric user to proceed with a minimal number of keystrokes.

Interfaces for the DBA. Most database systems contain privileged commands that can be used only by the DBA staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

A Sample Database Application

In this section we describe a sample database application, called COMPANY , which serves to illustrate the basic ER model concepts and their use in schema design.

The COMPANY database keeps track of a company's employees, departments, and projects. Suppose that after the requirements collection and analysis phase, the database designers provide the following description of the miniworld—the part of the company that will be represented in the database.

- The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
- A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
- We store each employee's name, Social Security number, address, salary, sex (gender), and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. We keep track

of the current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).

- We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relationship to the employee.

Figure 14 shows how the schema for this database application can be displayed by means of the graphical notation known as ER diagrams.

Entity Types, Entity Sets, Attributes, and Keys

The ER model describes data as entities, relationships, and attributes.

Entities and Attributes

The basic object that the ER model represents is an entity, which is a thing in the real world with an independent existence. 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. Figure 15 shows two entities and the values of their attributes. The EMPLOYEE entity e_1 has four attributes: Name, Address, Age and Home_phone ; their values are 'John Smith,' '2311 Kirby, Houston, Texas 77001', '55', and '713-749-2630', respectively. The COMPANY entity c_1 has three attributes: Name , Headquarters , and President ; their values are 'Sunco Oil', 'Houston', and 'John Smith', respectively. Several types of attributes occur in the ER model: simple versus composite, single-valued versus multivalued, and stored versus derived.

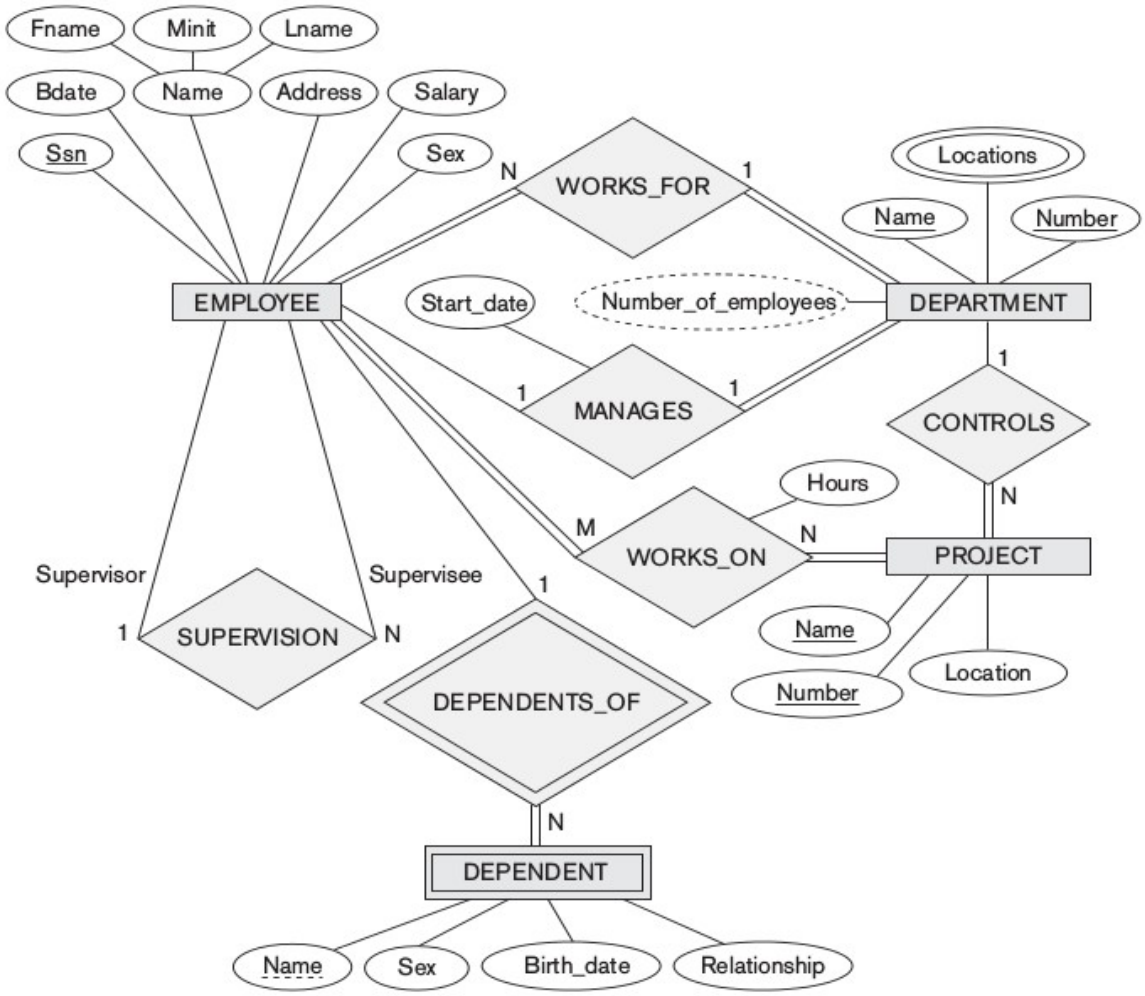


***Figure 14**: An ER schema diagram for the COMPANY database.*

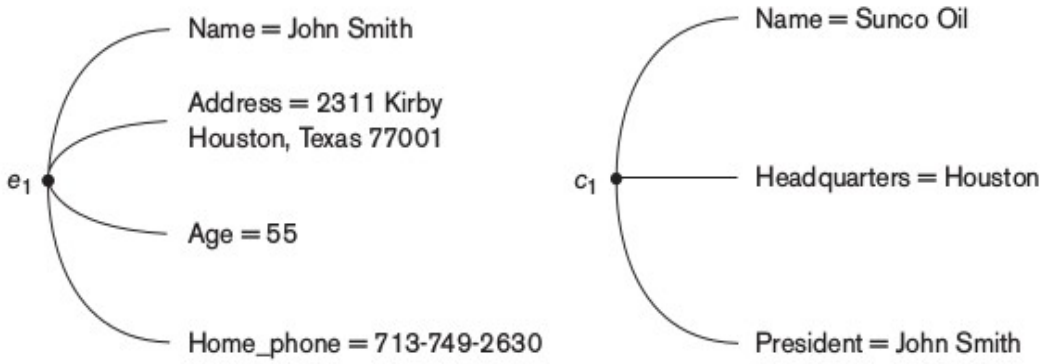


***Figure 15**: Two entities, EMPLOYEE e1 and COMPANY c1 and their attributes.*

Composite versus Simple (Atomic) Attributes. Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity shown in Figure 15 can be subdivided into Street_address, City, State and Zip with the values '2311 Kirby', 'Houston', 'Texas', and '77001.' Attributes that are not divisible are called simple or atomic attributes. Composite attributes can form a hierarchy; for example, Street_address can be further subdivided into three simple component attributes: Numbe, Street and Apartment_number as shown in Figure 16. The value of a composite attribute is the concatenation of the values of its component simple attributes.

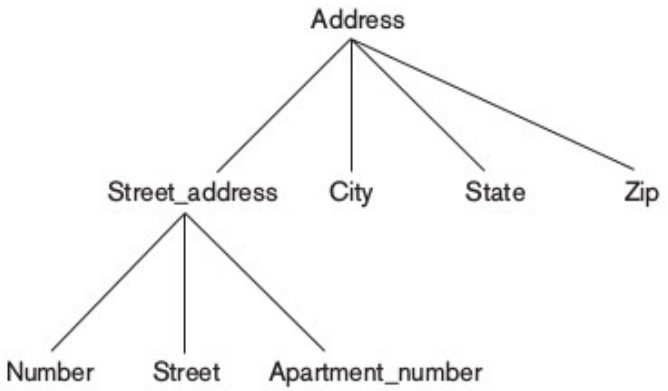


***Figure 16**: A hierarchy of composite attributes.*

Single-Valued versus Multivalued Attributes. Most attributes have a single value for a particular entity; such attributes are called single-valued. For example, Age is a single valued attribute of a person. In some cases an attribute can have a set of values for the same entity—for instance, a Colors attribute for a car, or a College_degrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two color values. Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees; therefore, different people can have different numbers of values for the College_degrees attribute. Such attributes are called multivalued.

Stored versus Derived Attributes. In some cases, two (or more) attribute values are related—for example, the Age and Birth_date attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth_date . The Age attribute is hence called a derived attribute and is said to be derivable from the Birth_date attribute, which is called a stored attribute.

NULL Values. In some cases, a particular entity may not have an applicable value for an attribute. For example, the Apartment_number attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as

single-family homes. Similarly, a College_degrees attribute applies only to people with college degrees. For such situations, a special value called NULL is created. An address of a single-family home would have NULL for its Apartment_number attribute, and a person with no college degree would have NULL for College_degrees. NULL can also be used if we do not know the value of an attribute for a particular entity—for example, if we do not know the home phone number of 'John Smith' in Figure 15. The unknown category of NULL can be further classified into two cases. The first case arises when it is known that the attribute value exists but is missing—for instance, if the Height attribute of a person is listed as NULL . The second case arises when it is not known whether the attribute value exists—for example, if the Home_phone attribute of a person is NULL.

Entity Types, Entity Sets, Keys, and Value Sets

Entity Types and Entity Sets. A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has its own value(s) for each attribute. An entity type defines a collection (or set) of entities that have the same attributes. Each entity type in the database is described by its name and attributes. Figure 17 shows two entity types: EMPLOYEE and COMPANY and a list of some of the attributes for each. A few individual entities of each type are also illustrated, along with the values of their attributes. The collection of all entities of a particular entity type in the database at any point in time is called an **entity set**; the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a type of entity as well as the current set of all employee entities in the database.

An entity type is represented in ER diagrams as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals.

An entity type describes the **schema or intension** for a set of entities that share the same structure. The collection of entities of a particular entity type is grouped into an entity set, which is also called the **extension** of the entity type.

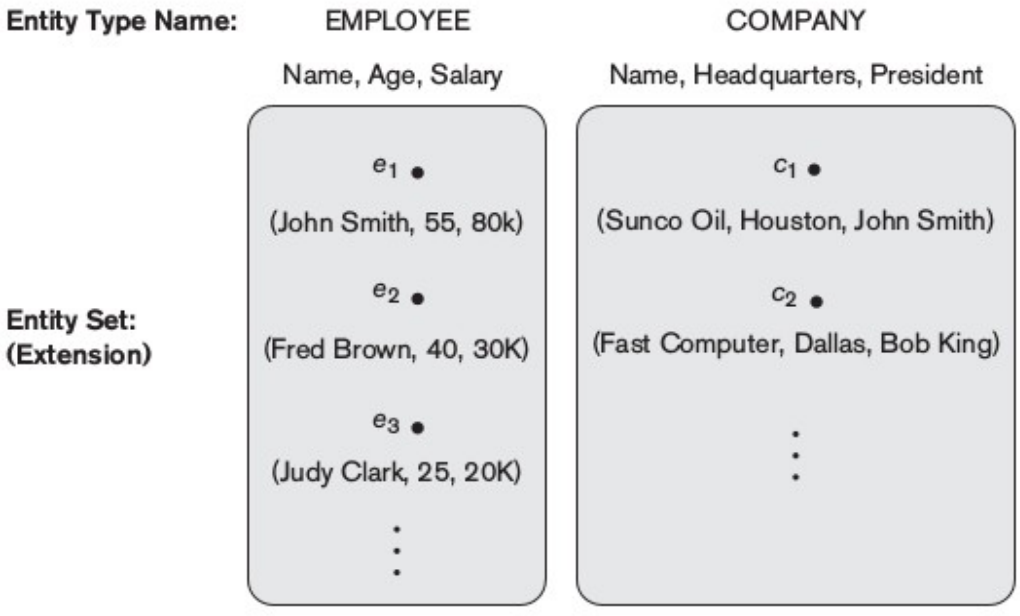


Figure 17: Two entity types EMPLOYEE and COMPANY and some member entities of each.

Key Attributes of an Entity Type. An important constraint on the entities of an entity type is the key or uniqueness constraint on attributes. An entity type usually has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a key attribute, and its values can be used to identify each entity uniquely. For example, the Name attribute is a key of the COMPANY entity type in Figure 19 because no two companies are allowed to have the same name. Sometimes several attributes together form a key, meaning that the combination of the attribute values must be distinct for each entity.

Some entity types have more than one key attribute. For example, each of the Vehicle_id and Registration attributes of the entity type CAR (Figure 19) is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, State and Number , neither of which is a key on its own.

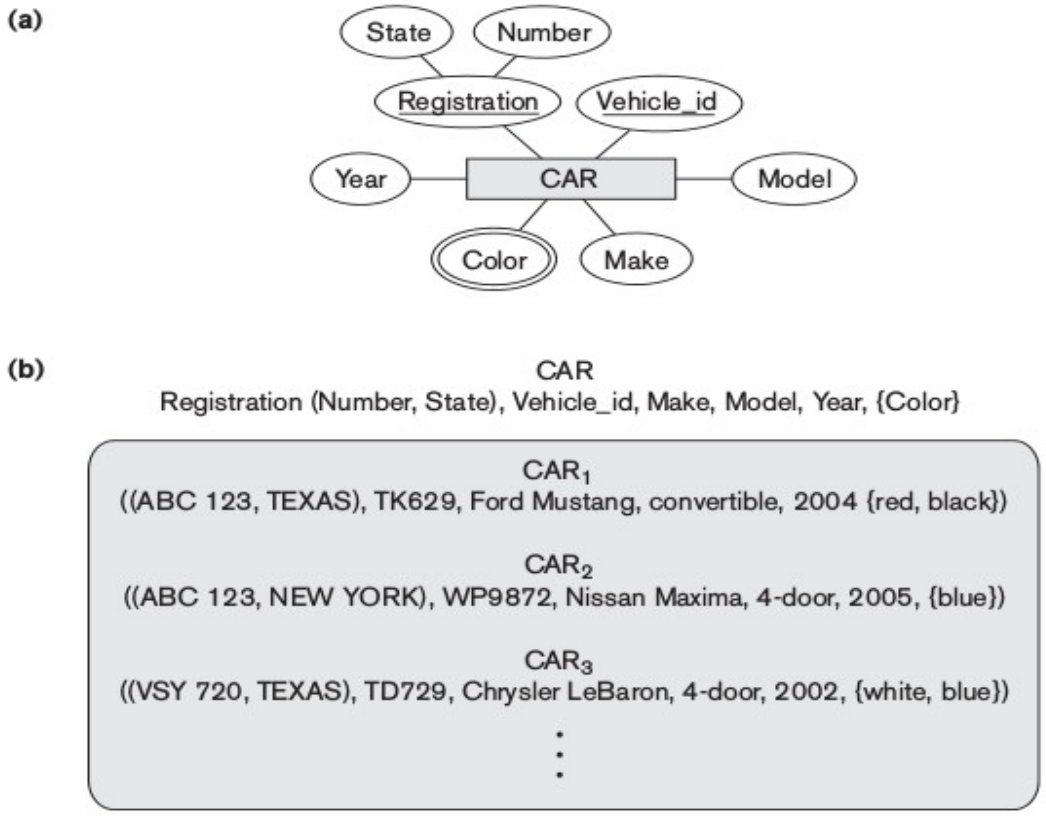


Figure 19 *The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.*

Value Sets (Domains) of Attributes. Each simple attribute of an entity type is associated with a value set (or domain of values), which specifies the set of values that may be assigned to that attribute for each individual entity. Value sets are not displayed in ER diagrams, and are typically specified using the basic data types available in most programming languages. Mathematically, an attribute A of entity set E whose value set is V can be defined as a function from E to the power set P(V) of V:

$A : E \rightarrow P(V)$

We refer to the value of attribute A for entity e as A(e). The previous definition covers both single-valued and multivalued attributes, as well as NULL s.

For a composite attribute A, the value set V is the power set of the Cartesian product of $P(V_1)$, $P(V_2)$, ...,$P(V_n)$, where V_1 , V_2 , ..., V_n are the value sets of the simple component attributes that form A:

$V = P (P(V_1) \times P(V_2) \times ... \times P(V_n))$

Initial Conceptual Design of the COMPANY Database

We can now define the entity types for the COMPANY database based on the requirements described.

1. An entity type DEPARTMENT with attributes Name, Number, Locations, Manager and Manager_start_date. Locations is the only multivalued attribute. We can specify that both Name and Number are (separate) key attributes because each was specified to be unique.
2. An entity type PROJECT with attributes Name, Number, Location and Controlling_department. Both Name and Number are (separate) key attributes.
3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth_date, Department and Supervisor.
4. An entity type DEPENDENT with attributes Employee, Dependent_name, Sex, Birth_date and Relationship (to the employee).

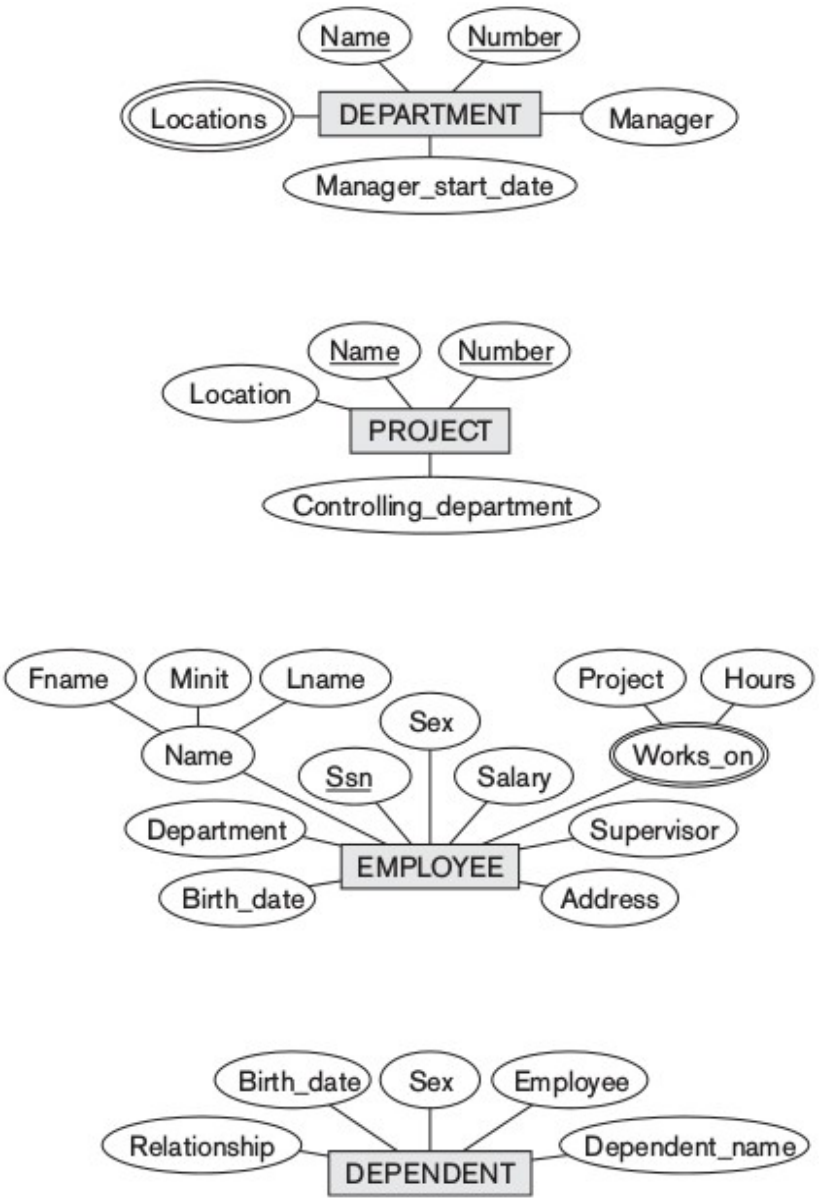


***Figure 20**: Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.*

Relationship Type, Relationship Set, Role and Structural Constraints

In Figure 20 there are several implicit relationships among the various entity types. In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists. For example, the attribute Manager of DEPARTMENT refers to an employee who manages the department; the attribute Controlling_department of PROJECT refers to the department that controls the project; the attribute Supervisor of EMPLOYEE refers to another employee (the one who supervises this employee); the attribute Department of EMPLOYEE refers to the department for which the employee works; and so on.

In the ER model, these references should not be represented as attributes but as relationships. In the initial design of entity types, relationships are typically captured in the form of attributes. As the design is refined, these attributes get converted into relationships between entity types.

Relationship Types, Sets, and Instances

A relationship type R among n entity types E_1, E_2, ..., E_n defines a set of associations—or a relationship set—among entities from these entity types. Mathematically, the relationship set R is a set of relationship instances r_i , where each r_i associates n individual entities (e_1, e_2, ..., e_n), and each entity e_j in r_i is a member of entity set E_j.

Informally, each relationship instance r_i in R is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance r_i represents the fact that the entities participating in r_i are related in some way in the corresponding miniworld situation. For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works in the corresponding entity set. Each relationship instance in the relationship set WORKS_FOR associates one EMPLOYEE entity and one DEPARTMENT entity.

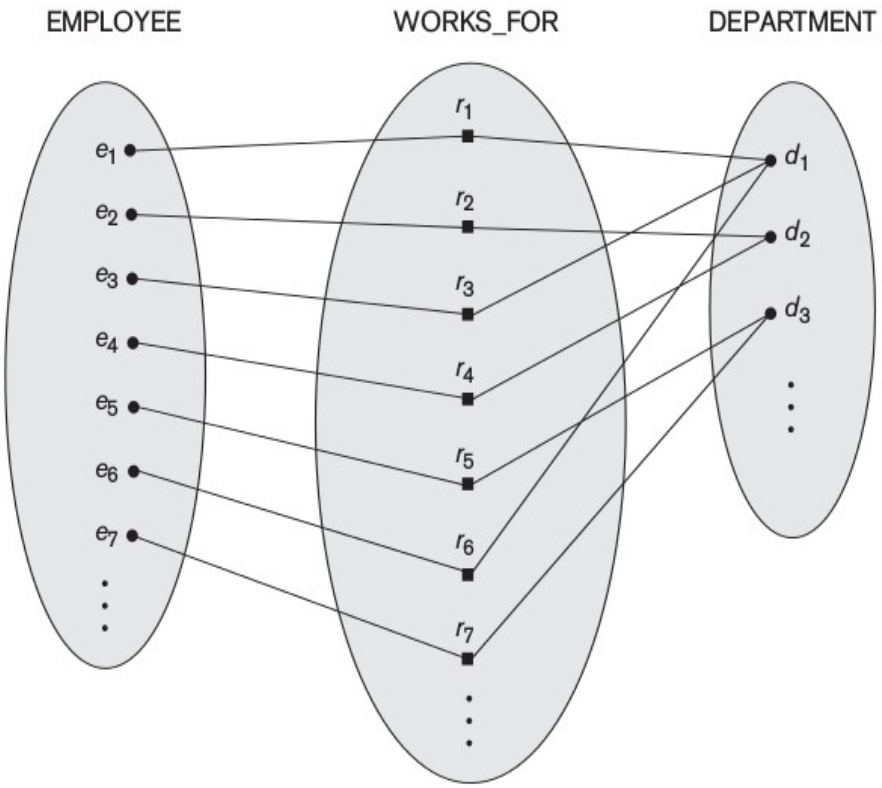


***Figure 21**: Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.*

Figure 21 illustrates this example, where each relationship instance r_i is shown connected to the EMPLOYEE and DEPARTMENT entities that participate in r_i. Employees e_1, e_3 and e_6 work for department d_1 ; employees e_2 and e_4 work for department d_2; and employees e_5 and e_7 work for department d_3.

In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box (see Figure 14).

Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type. The degree of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called **binary**, and one of degree three is called **ternary**. An example of a ternary relationship is SUPPLY, shown in Figure 22, where each relationship instance r_i associates three entities—a supplier s, a part p, and a project j—whenever s supplies part p to project j.

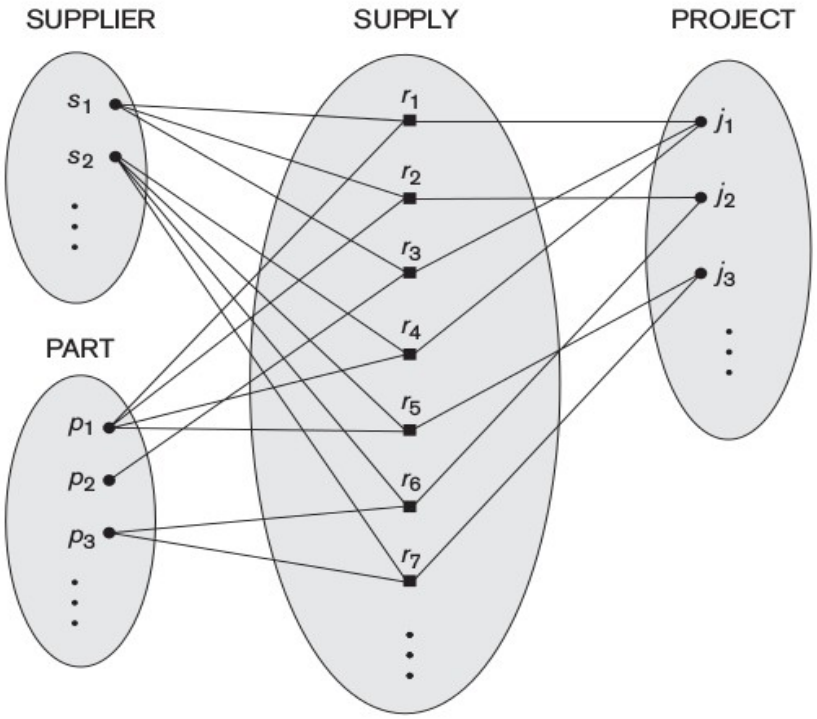


Figure 22: *Some relationship instances inthe SUPPLY ternary relationshipset.*

Relationships as Attributes. It is sometimes convenient to think of a binary relationship type in terms of attributes. Consider the WORKS_FOR relationship type in Figure 21. One can think of an attribute called Department of the EMPLOYEE entity type, where the value of Department for each EMPLOYEE entity is (a reference to) the DEPARTMENT entity for which that employee works. Hence, the value set for this Department attribute is the set of all DEPARTMENT entities, which is the DEPARTMENT entity set.

However, when we think of a binary relationship as an attribute, we always have two options. In this example, the alternative is to think of a multivalued attribute Employee of the entity type DEPARTMENT whose values for each DEPARTMENT entity is the set of EMPLOYEE entities who work for that department. The value set of this Employee attribute is the power set of the EMPLOYEE entity set.

Role Names and Recursive Relationships. Each entity type that participates in a relationship type plays a particular role in the relationship. The role name signifies the role that a participating entity from the entity type plays in each relationship instance. For example, in the WORKS_FOR relationship type, EMPLOYEE plays the role of employee or worker and

DEPARTMENT plays the role of department or employer.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each participating entity type name can be used as the role name. However, in some cases the same entity type participates more than once in a relationship type in different roles. In such cases the role name becomes essential for distinguishing the meaning of the role that each participating entity plays.Such relationship types are called **recursive relationships**.

The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. Hence, the EMPLOYEE entity type participates twice in SUPERVISION: once in the role of supervisor (or boss), and once in the role of supervisee (or subordinate). Each relationship instance r_i in SUPERVISION associates two employee entities e_j and e_k, one of which plays the role of supervisor and the other the role of supervisee.

In Figure 23, the lines marked '1' represent the supervisor role, and those marked '2' represent the supervisee role; hence, e_1 supervises e_2 and e_3, e_4 supervises e_6 and e_7, and e_5 supervises e_1 and e_4.

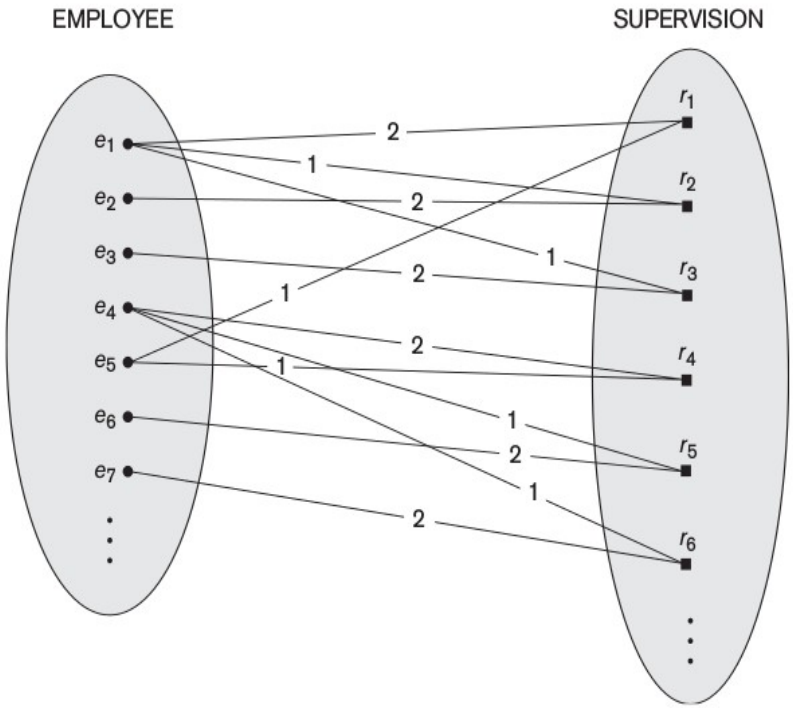


Figure 23: *A recursive relationship SUPERVISION between EMPLOYEE in the supervisor role (1) andEMPLOYEE in the subordinate role (2).*

Constraints on Binary Relationship Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. For example, in Figure 21, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of binary relationship constraints: **cardinality ratio** and **participation**.

Cardinality Ratios for Binary Relationships. The cardinality ratio for a binary relationship specifies the maximum number of relationship instances that an entity can participate in. For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to any number of employees, but an employee can be related to (work for) only one department. The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N.

An example of a 1:1 binary relationship is MANAGES (Figure 24), which relates a department entity to the employee who manages that department. This represents the miniworld constraints that—at any point in time—an employee can manage one department only and a department can have one manager only. The relationship type WORKS_ON (Figure 25) is of cardinality ratio M:N, because the mini-world rule is that an employee can

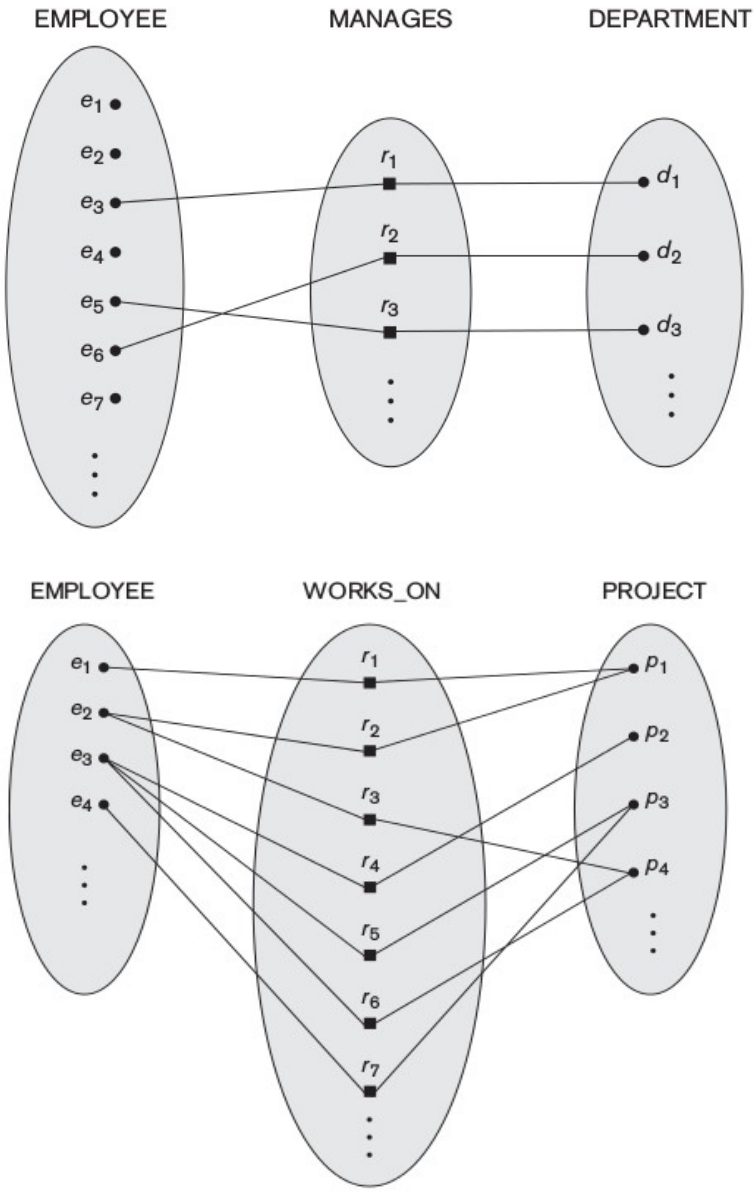


Figure 25: An M:N relationship,WORKS_ON

work on several projects and a project can have several employees. Notice that in this notation, we can either specify no maximum (N) or a maximum of one (1) on participation.

Participation Constraints and Existence Dependencies. The participation constraint specifies whether the existence of an entity depends on its being related to another entity via the relationship type. This constraint specifies the minimum number of relationship instances that each entity can participate in, and is sometimes called the minimum cardinality constraint. There are two types of participation constraints—**total and partial**—that we illustrate by example.

If a company policy states that every employee must work for a department, then an employee entity can exist only if it participates in at least one WORKS_FOR relationship instance (Figure 21). Thus, the participation of EMPLOYEE in WORKS_FOR is called total participation, meaning that every entity in the total set of employee entities must be related

to a department entity via WORKS_FOR . Total participation is also called existence dependency.

In Figure 24 we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is partial, meaning that some or part of the set of employee entities are related to some department entity via MANAGES , but not necessarily all. We will refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type. In ER diagrams, total participation (or existence dependency) is displayed as a double line connecting the participating entity type to the relation, whereas partial participation is represented by a single line(Figure 14).

Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that an employee works on a particular project, we can include an attribute Hours for the WORKS_ON relationship type. Another example is to include the date on which a manager started managing a department via an attribute Start_date for the MANAGES relationship type. Notice that attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types.

For example, the Start_date attribute for the MANAGES relationship can be an attribute of either EMPLOYEE or DEPARTMENT, although conceptually it belongs to MANAGES . This is because MANAGES is a 1:1 relationship, so every department or employee entity participates in at most one relationship instance. Hence, the value of the Start_date attribute can be determined separately, either by the participating department entity or by the participating employee (manager) entity.

For a 1:N relationship type, a relationship attribute can be migrated only to the entity type on the N-side of the relationship. For example, in Figure 21, if the WORKS_FOR relationship also has an attribute Start_date that indicates when an employee started working for a department, this attribute can be included as an attribute of EMPLOYEE. This is because each employee works for only one department, and hence participates in at most one relationship instance in WORKS_FOR.

For M:N relationship types, some attributes may be determined by the combination of participating entities in a relationship instance, not by any single entity. Such attributes must be specified as relationship attributes. An example is the Hours attribute of the M:N relationship WORKS_ON; the number of hours per week an employee currently works on a project is determined by an employee-project combination and not separately by either entity.

Weak Entity Types

Entity types that do not have key attributes of their own are called weak entity types. In contrast, regular entity types that do have a key attribute—which include all the examples discussed so far—are called strong entity types. Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. We call this other entity type the **identifying** or **owner entity type**, and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type. A weak entity type always has a total participation constraint (existence dependency) with respect to its identifying relationship because a weak entity cannot be identified without an owner entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship (Figure 14). In our example, the attributes of DEPENDENT are Name (the first name of the dependent), Birth_date, Sex

and Relationship(to the employee). Two dependents of two distinct employees may, by chance, have the same values for Name, Birth_date, Sex and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the particular employee entity to which each dependent is related. Each employee entity is said to own the dependent entities that are related to it. A weak entity type normally has a **partial key**, which is the attribute that can uniquely identify weak entities that are related to the same owner entity. In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key.

In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with double lines (see Figure 14). The partial key attribute is underlined with a dashed or dotted line.

Refining the ER Design for the COMPANY Database

We can now refine the database design in Figure 20 by changing the attributes that represent relationships into relationship types. The cardinality ratio and participation constraint of each relationship type are determined from the requirements listed. In our example, we specify the following relationship types:

- The MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is total. The attribute Start_date is assigned to this relationship type.
- WORKS_FOR, a 1:N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.
- CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT . The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users indicates that some departments may control no projects.
- SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.
- WORKS_ON, determined to be an M:N relationship type with attribute Hours , after the users indicate that a project can have several employees working on it. Both participations are determined to be total.
- DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

ER Diagrams, Naming Conventions, and Design Issues

Summary of Notation for ER Diagrams

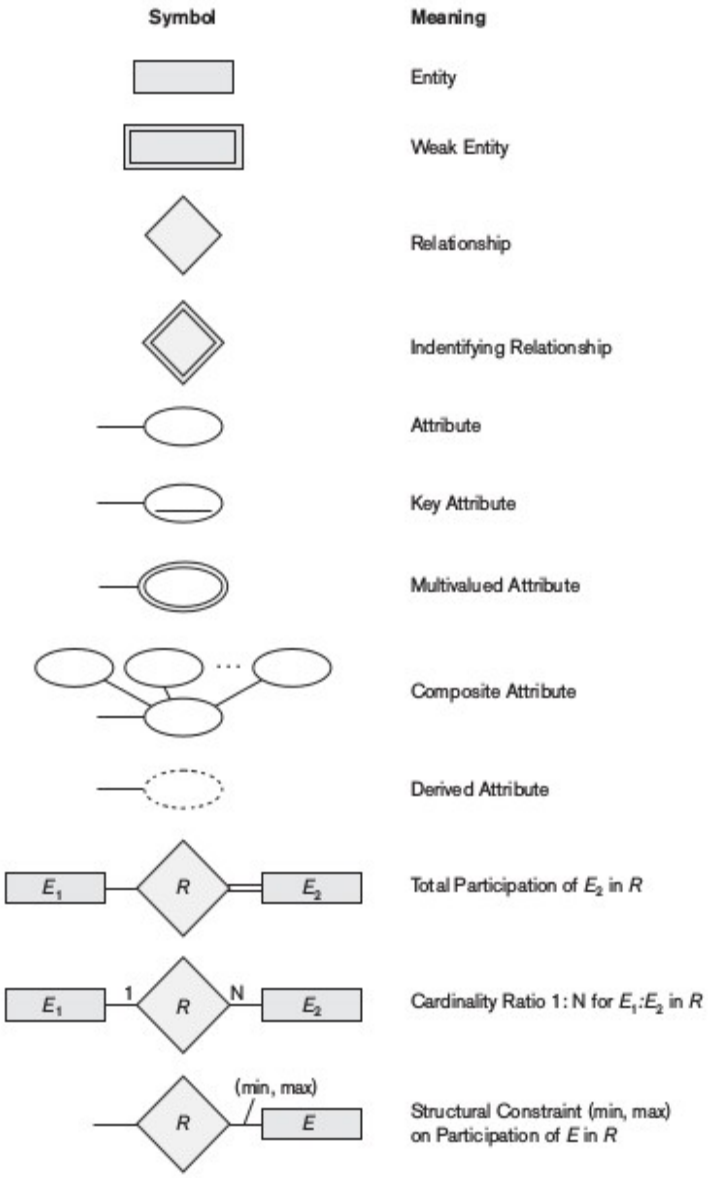


Figure 26:*Summary of the notation for ER diagrams.*

Figure 14 displays the COMPANY ER database schema as an ER diagram. We now review the full ER diagram notation. Entity types such as EMPLOYEE, DEPARTMENT and PROJECT are shown in rectangular boxes. Relationship types such as WORKS_FOR, MANAGES, CONTROLS, and WORKS_ON are shown in diamond-shaped boxes attached to the participating entity types with straight lines. Attributes are shown in ovals, and each attribute is attached by a straight line to its entity type or relationship type. Component attributes of a composite attribute are attached to the oval representing the composite attribute, as illustrated by the Name attribute of EMPLOYEE. Multivalued attributes are shown in double ovals, as illustrated by the Locations attribute of DEPARTMENT. Key attributes have their names underlined. Derived attributes are shown in dotted ovals, as illustrated by the Number_of_employees attribute of DEPARTMENT. Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds, as illustrated by the DEPENDENT entity type and the DEPENDENTS_OF identifying relationship type. The partial key of the weak entity type

is underlined with a dotted line.

In Figure 14 the cardinality ratio of each binary relationship type is specified by attaching a 1, M, or N on each participating edge. The cardinality ratio of DEPARTMENT:EMPLOYEE in MANAGES is 1:1, whereas it is 1:N for DEPARTMENT: EMPLOYEE in WORKS_FOR , and M:N for WORKS_ON . The participation constraint is specified by a single line for partial participation and by double lines for total participation (existence dependency).

In Figure 14 we show the role names for the SUPERVISION relationship type because the same EMPLOYEE entity type plays two distinct roles in that relationship. Notice that the cardinality ratio is 1:N from supervisor to supervisee because each employee in the role of supervisee has at most one direct supervisor, whereas an employee in the role of supervisor can supervise zero or more employees.

Proper Naming of Schema Constructs

When designing a database schema, the choice of names for entity types, attributes, relationship types, and (particularly) roles is not always straightforward. One should choose names that convey, as much as possible, the meanings attached to the different constructs in the schema. We choose to use singular names for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type.

Another naming consideration involves choosing binary relationship names to make the ER diagram of the schema readable from left to right and from top to bottom. We have generally followed this guideline in Figure 14. To explain this naming convention further, we have one exception to the convention in Figure 14—the DEPENDENTS_OF relationship type, which reads from bottom to top. When we describe this relationship, we can say that the DEPENDENT entities (bottom entity type) are DEPENDENTS_OF (relationship name) an EMPLOYEE (top entity type). To change this to read from top to bottom, we could rename the relationship type to HAS_DEPENDENTS, which would then read as follows: An EMPLOYEE entity (top entity type) HAS_DEPENDENTS (relationship name) of type DEPENDENT (bottom entity type).

Design Choices for ER Conceptual Design

It is occasionally difficult to decide whether a particular concept in the miniworld should be modeled as an entity type, an attribute, or a relationship type. In general, the schema design process should be considered an iterative refinement process, where an initial design is created and then iteratively refined until the most suitable design is reached.

Some of the refinements that are often used include the following:

- A concept may be first modeled as an attribute and then refined into a relationship because it is determined that the attribute is a reference to another entity type. It is important to note that in our notation, once an attribute is replaced by a relationship, the attribute itself should be removed from the entity type to avoid duplication and redundancy.
- Similarly, an attribute that exists in several entity types may be elevated or promoted to an independent entity type. For example, suppose that several entity types in a UNIVERSITY database, such as STUDENT, INSTRUCTOR, and COURSE, each has an attribute Department in the initial design; the designer may then choose to create an entity type DEPARTMENT with a single attribute Dept_name and relate it to the three entity types (STUDENT, INSTRUCTOR, and COURSE) via appropriate relationships.
- An inverse refinement to the previous case may be applied—for example, if an entity

type DEPARTMENT exists in the initial design with a single attribute Dept_name and is related to only one other entity type, STUDENT. In this case, DEPARTMENT may be reduced or demoted to an attribute of STUDENT.

Alternative Notations for ER Diagrams

There are many alternative diagrammatic notations for displaying ER diagrams. In this section, we describe one alternative ER notation for specifying structural constraints on relationships, which replaces the cardinality ratio (1:1, 1:N, M:N) and single/double line notation for participation constraints. This notation involves associating a pair of integer numbers (min,max) with each participation of an entity type E in a relationship type R, where $0 \leq \min \leq \max$ and $\max \geq 1$. The numbers mean that for each entity e in E, e must participate in at least min and at most max relationship instances in R at any point in time. In this method, min = 0 implies partial participation, whereas min > 0 implies total participation. Figure 27 displays the COMPANY database schema using the (min,max) notation.

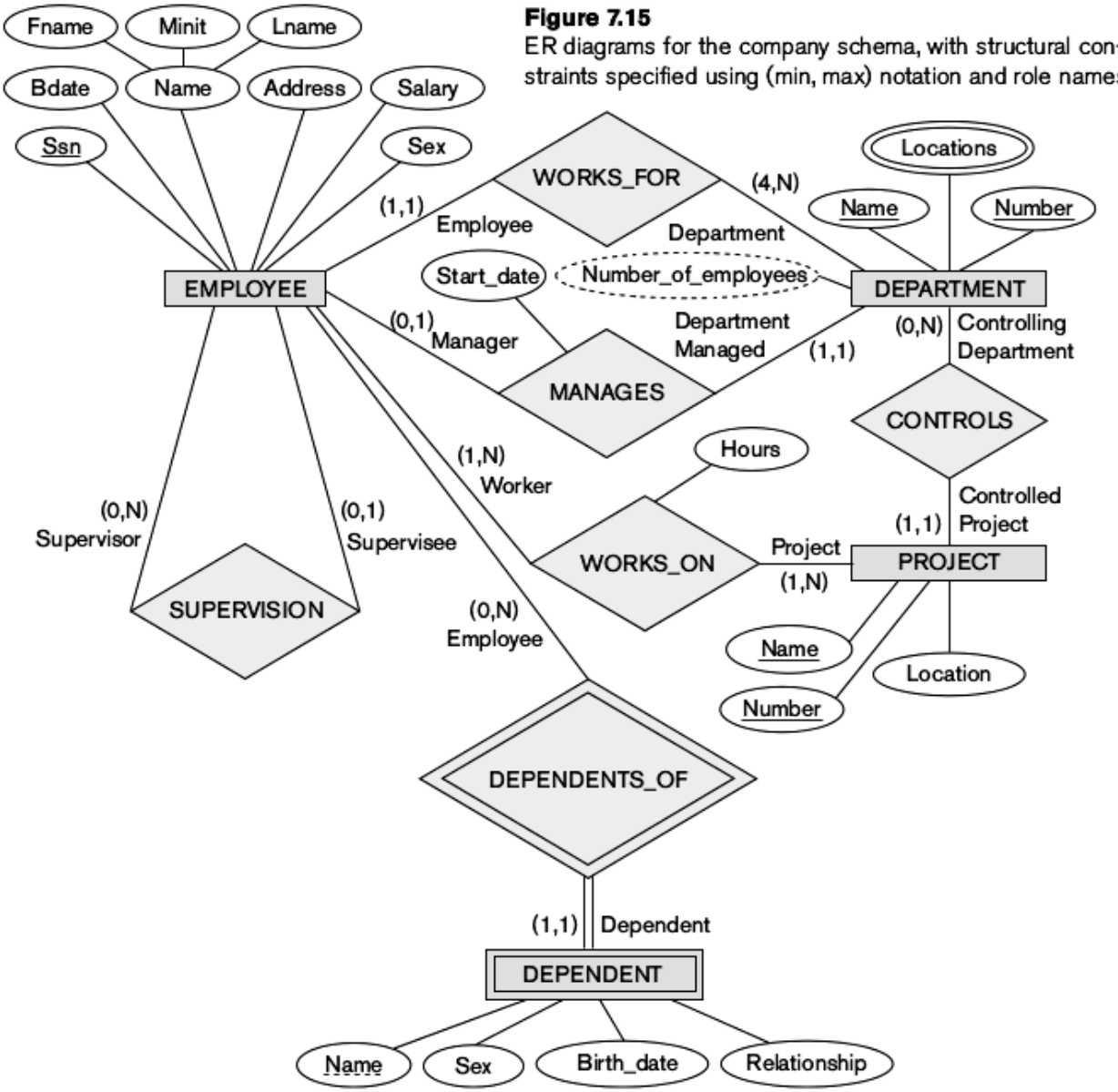


***Figure 27**: ER diagrams for the company schema, with structural constraints specified using (min, max) notation and role names.*

Relational Model Concepts

The relational model represents the database as a collection of relations. Informally, each relation resembles a table of values or, to some extent, a flat file of records. It is called a flat file because each record has a simple linear or flat structure. When a relation is thought of as a table of values, each row in the table represents a collection of related data values. A row represents a fact that typically corresponds to a real-world entity or relationship. The table name and column names are used to help to interpret the meaning of the values in each row.

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

Domains, Attributes, Tuples, and Relations

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 follow:

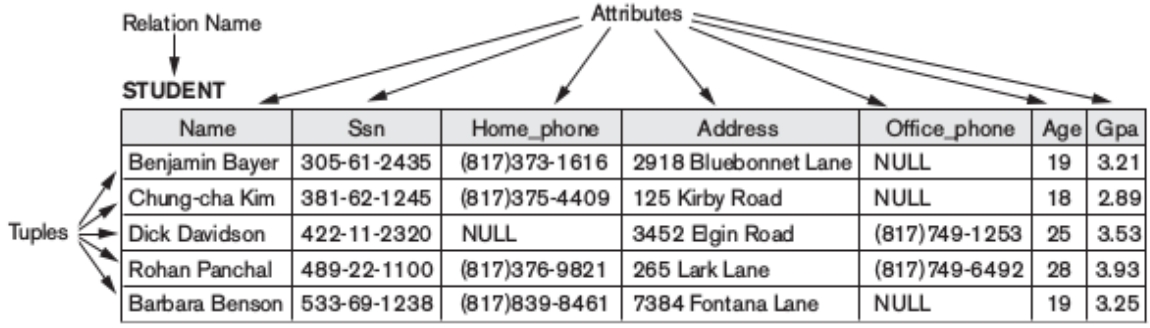
- Social_security_numbers . The set of valid nine-digit Social Security numbers. (This is a unique identifier assigned to each person in the United States for employment, tax, and benefits purposes.)
- Names : The set of character strings that represent names of persons.
- Academic_department_names . The set of academic department names in a university, such as Computer Science, Economics, and Physics.
- Academic_department_codes . The set of academic department codes, such as 'CS', 'ECON', and 'PHYS'.

The preceding are called logical definitions of domains. A data type or format is also specified for each domain. For Academic_department_names , the data type is the set of all character strings that represent valid department names. A domain is thus given a name, data type, and format.

A **relation schema** R, denoted by $R(A_1, A_2, ..., A_n)$, is made up of a relation name R and a list of attributes, $A_1, A_2, ..., A_n$. Each attribute A_i is the name of a role played by some domain D in the relation schema R. D is called the **domain** of A_i and is denoted by $dom(A_i)$. A relation schema is used to describe a relation; R is called the name of this relation. The **degree** (or arity) of a relation is the number of attributes n of its relation schema. A relation of degree seven, which stores information about university students, would contain seven attributes describing each student. as follows:

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

For this relation schema, STUDENT is the name of the relation, which has seven attributes. It is possible to refer to attributes of a relation schema by their position within the relation; thus, the second attribute of the STUDENT relation is Ssn , whereas the fourth attribute is Address. A **relation (or relation state)** of the relation schema $R(A_1, A_2, ..., A_n)$, also denoted by r(R), is a set of n-tuples $r = \{t_1, t_2, ..., t_m\}$. Each n-tuple t is an ordered list of n values. The i^{th} value in tuple t, which corresponds to the attribute A_i, is referred to as $t[A_i]$ or $t.A_i$ (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

Figure 1 *The attributes and tuples of a relation STUDENT.*

Figure 1 shows an example of a STUDENT relation, which corresponds to the STUDENT schema just specified. Each tuple in the relation represents a particular student entity (or object). We display the relation as a table, where each tuple is shown as a row and each attribute corresponds to a column header indicating a role or interpretation of the values in that column. NULL values represent attributes whose values are unknown or do not exist for some individual STUDENT tuple.

The earlier definition of 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 $dom(A_1)$, $dom(A_2)$, ..., $dom(A_n)$, which is a subset of the Cartesian product (denoted by ×) of the domains that define R:

$$r(R) \subseteq (dom(A_1) \times dom(A_2) \times ... \times dom(A_n))$$

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 $|D|$, the total number of tuples in the Cartesian product is

$$|dom(A_1)| \times |dom(A_2)| \times ... \times |dom(A_n)|$$

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). Of all these possible combinations, a relation state at a given time—the current relation state—reflects only the valid tuples that represent a particular state of the real world. I

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. Similarly, when we display a relation as a table, the rows are displayed in a certain order. Tuple ordering is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level. Many tuple orders can be specified on the same relation. For example, tuples in the STUDENT relation in Figure 1 could be ordered by values of Name , Ssn , Age , or some other attribute. Hence, the relation displayed in Figure 2 is considered identical to the one shown in Figure 1. When a relation is implemented as a file or displayed as a table, a particular ordering may be specified on the records of the file or the rows of the table.

Ordering of Values within a Tuple and an Alternative Definition of a Relation. According to the preceding definition of a relation, an n-tuple is an ordered list of n values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important. However, at a more abstract level, the order of attributes and their values is not that important as long as the correspondence between attributes and values is maintained.

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

***Figure 2**: The relation STUDENT from Figure 1 with a different order of tuples.*

According to this, a tuple can be considered as a set of (<attribute>, <value>) pairs, where each pair gives the value of the mapping from an attribute A_i to a value v_i from dom(A_i). The ordering of attributes is not important, because the attribute name appears with its value. By this definition, the two tuples shown in Figure 3 are identical.

$t = <$ (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)>

***Figure 3**: Two identical tuples when the order of attributes and values is not part of relation definition.*

Values and NULLs in the Tuples. Each value in a tuple is an atomic value; that is, it is not divisible into components within the framework of the basic relational model. This model is sometimes called the flat relational model. Hence, multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes in the basic relational model.

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL , is used in these cases. For example, in Figure 1, some STUDENT tuples have NULL for their office phones because they do not have an office (that is, office phone does not apply to these students). Another student has a NULL for home phone, presumably because either he does not have a home phone or he has one but we do not know it (value is unknown).

In general, we can have several meanings for NULL values, such as value unknown, value exists but is not available, or attribute does not apply to this tuple (also known as value undefined). An example of the last type of NULL will occur if we add an attribute Visa_status to the STUDENT relation that applies only to tuples representing foreign students.

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 of Figure 1 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. For example, the first tuple in Figure 1 asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on. Notice that some relations may represent facts about entities, whereas other relations may represent facts about relationships.

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 of Figure 1. These tuples represent five different propositions or facts in the real world.

Relational Model Notation

We will use the following notation in our presentation:

- A relation schema R of degree n is denoted by $R(A_1, A_2, ..., A_n)$.
- 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.
- An n-tuple t in a relation r(R) is denoted by $t = <v_1, v_2, ..., v_n>$, where v_i is the value corresponding to attribute A_i. The following notation refers to component values of tuples: Both $t[A_i]$ and $t.A_i$ (and sometimes t[i]) refer to the value v_i in t for attribute A_i.

Relational Model Constraints and Relational Database Schemas

So far, we have discussed the characteristics of single relations. In a relational database, there will typically be many relations, and the tuples in those relations are usually related in various ways. The state of the whole database will correspond to the states of all its relations at a particular point in time. There are generally many restrictions or constraints on the actual values in a database state. In this section, we discuss the various restrictions on data that can be specified on a relational database in the form of constraints. Constraints on databases can generally be divided into three main categories:

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

The constraints we discuss in this section are of the second category, namely, constraints that can be expressed in the schema of the relational model via the DDL.

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 money, or other special data types.

Key Constraints and Constraints on NULL Values

In the formal relational model, a relation is defined as a set of tuples. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes. Usually, there are other subsets of attributes of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two distinct tuples t_1 and t_2 in a relation state r of R, we have the constraint that:

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

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

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

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

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

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

***Figure 4**: The CAR relation, with two candidate keys:License_number and Engine_serial_number*

Consider the STUDENT relation of Figure 1. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn. Any set of attributes that includes Ssn —for example, { Ssn, Name, Age }—is a superkey. However, the superkey { Ssn, Name, Age } is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey. In general, any superkey formed from a single attribute is also a key.

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key**. For example, the CAR relation in Figure 4 has two candidate keys: License_number and Engine_serial_number. It is common to designate one of the candidate keys as the primary key of the relation. This is the candidate key whose values are used to identify tuples in the relation.

Relational Databases and Relational Database Schemas

A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section we define a relational database and a relational database schema. A relational database schema S is a set of relation schemas S = {R_1, R_2, ..., R_m} and a set of integrity constraints IC. A relational database state DB of S is a set of relation states DB = {r_1, r_{12}, ..., r_m} such that each r_i is a state of R_i and such that the r_i relation states satisfy the integrity constraints specified in IC.

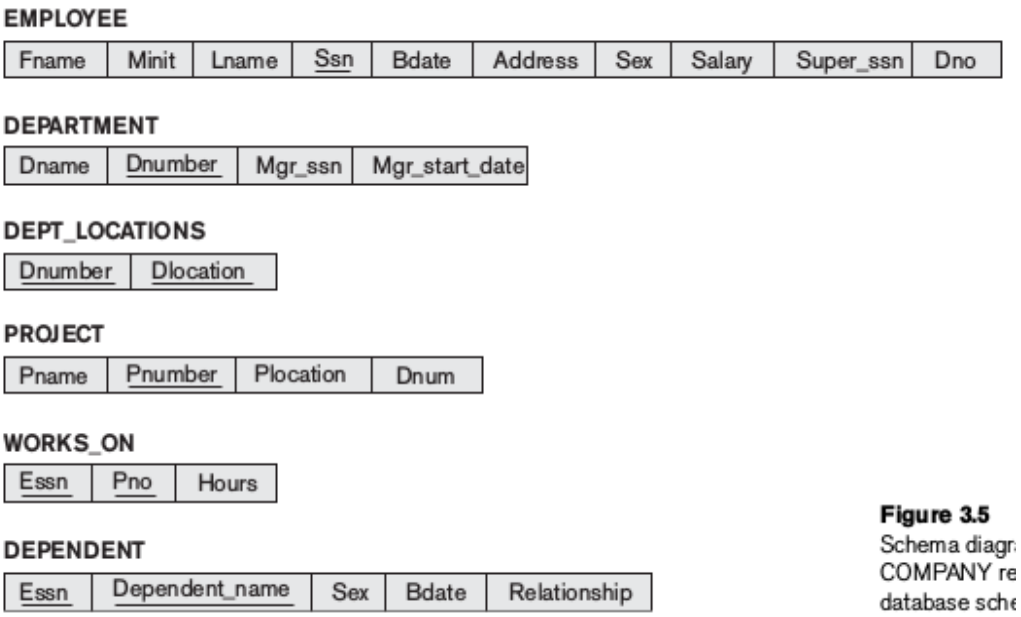


Figure 3.5
Schema diagr
COMPANY re
database sch

Figure 5*: Schema diagram for the COMPANY relational database schema.*

Figure 5 shows a relational database schema that we call COMPANY= {EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. The underlined attributes represent primary keys. Figure 6 shows a relational database state corresponding to the COMPANY schema.

When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is called an **invalid state**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state**.

In Figure 5, the Dnumber attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called Dno in EMPLOYEE and Dnum in PROJECT. Attributes that represent the same real-world concept may or may not have identical names in different relations. Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name Name for both Pname of PROJECT and Dname of DEPARTMENT ; in this case, we would have two attributes that share the same name but represent different realworld concepts—project names and department names.

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

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

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

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

***Figure 6**: One possible database state for the COMPANY relational database schema.*

Integrity, Referential Integrity, and Foreign Keys

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

Key constraints and entity integrity constraints are specified on individual relations. The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation.

For example, in Figure 6, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must

match the Dnumber value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, first we define the concept of a foreign key. The conditions for a **foreign key**, given below, specify a referential integrity constraint between the two relation schemas R_1 and R_2. A set of attributes FK in relation schema R_1 is a foreign key of R_1 that references relation R_2 if it satisfies the following rules:

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

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.

Referential integrity constraints typically arise from the relationships among the entities represented by the relation schemas. For example, consider the database shown in Figure 6. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of EMPLOYEE referencing the DEPARTMENT relation. This means that a value of Dno in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of DEPARTMENT —the Dnumber attribute—in some tuple t_2 of the DEPARTMENT relation, or the value of Dno can be NULL if the employee does not belong to a department or will be assigned to a department later.

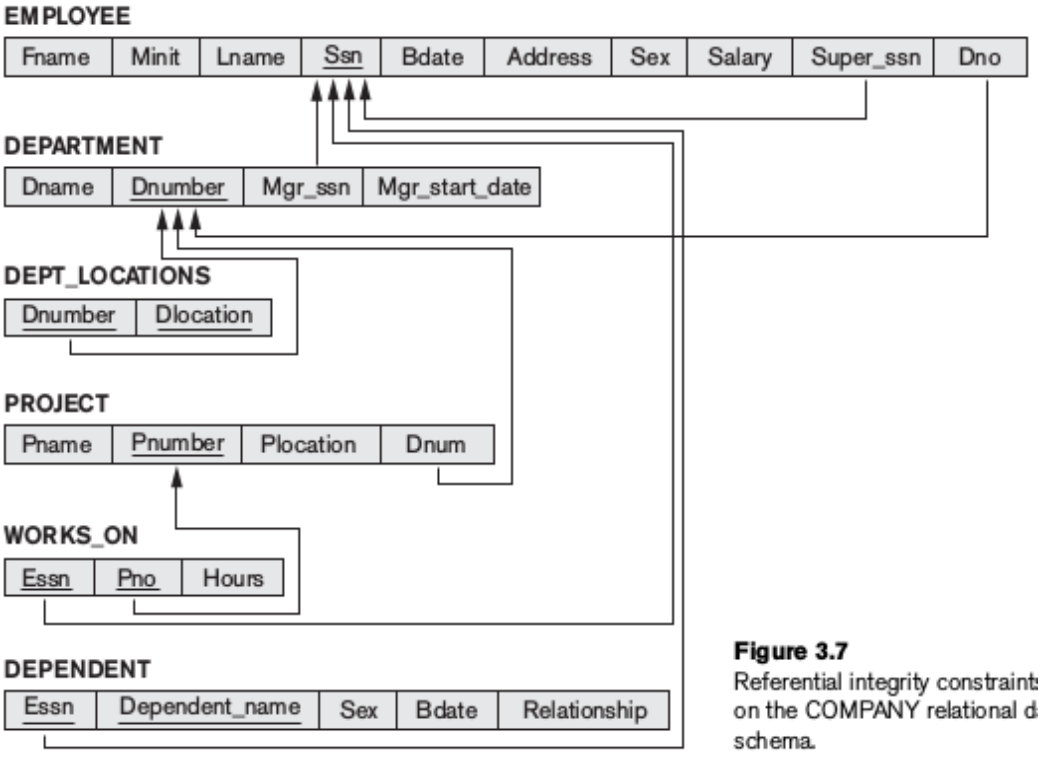


Figure 3.7
Referential integrity constraints on the COMPANY relational d schema.

***Figure 7**: Referential integrity constraints displayed on the COMPANY relational database schema.*

For example, in Figure 6 the tuple for employee 'John Smith' references the tuple for the 'Research' department, indicating that 'John Smith' works for this department. All integrity constraints should be specified on the relational database schema (i.e., defined as part of its definition) if we want to enforce these constraints on the database states.

We can diagrammatically display referential integrity constraints by drawing a

directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. Figure 7 shows the schema in Figure 5 with the referential integrity constraints displayed in this manner.

Other Types of Constraints

The preceding integrity constraints are included in the data definition language because they occur in most database applications. However, they do not include a large class of general constraints, sometimes called semantic integrity constraints, which may have to be specified and enforced on a relational database. Examples of such constraints are the salary of an employee should not exceed the salary of the employee's supervisor and the maximum number of hours an employee can work on all projects per week is 56. 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.

Another type of constraint is the functional dependency constraint, which establishes a functional relationship among two sets of attributes X and Y. This constraint specifies that the value of X determines a unique value of Y in all states of a relation; it is denoted as a functional dependency $X \rightarrow Y$.

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.

Update Operations, Transactions, and Dealing with Constraint Violations

In this section, we concentrate on the database modification or update operations. There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records. 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. Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this section we discuss the types of constraints that may be violated by each of these operations and the types of actions that may be taken if an operation causes a violation.

The Insert Operation

The Insert operation provides a list of attribute values for a new tuple t that is to be inserted into a relation R. Insert can violate any of the four types of constraints discussed in the previous section. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type. Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation r(R). Entity integrity can be violated if any part of the primary key of the new tuple t is NULL. Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation. Here are some examples to illustrate this discussion.

- 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.

- 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.
- 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.
- 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.

If an insertion violates one or more constraints, the default option is to reject the insertion. In this case, it would be useful if the DBMS could provide a reason to the user as to why the insertion was rejected. Another option is to attempt to correct the reason for rejecting the insertion. In the first operation, the DBMS could ask the user to provide a value for Ssn, and could then accept the insertion if a valid Ssn value is provided. In operation 3, the DBMS could either ask the user to change the value of Dno to some valid value (or set it to NULL), or it could ask the user to insert a DEPARTMENT tuple with Dnumber = 7 and could accept the original insertion only after such an operation was accepted.

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. Here are some examples.

- Delete the WORKS_ON tuple with Essn = ‘999887777’ and Pno =10
  - Result: This deletion is acceptable and deletes exactly one tuple.
- 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.
- 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. For example, in operation 2, the DBMS could automatically delete the offending tuples from WORKS_ON with Essn = ‘999887777’. 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. Notice that if a referencing attribute that causes a violation is part of the primary key, it cannot be set to NULL ; otherwise, it would violate entity integrity.

Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, the DBMS may automatically delete all tuples from WORKS_ON and DEPENDENT with Essn = ‘333445555’. Tuples in EMPLOYEE with Super_ssn = ‘333445555’ and the tuple in DEPARTMENT with Mgr_ssn = ‘333445555’ can

have their Super_ssn and Mgr_ssn values changed to other valid values or to NULL.

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.

Here are some examples.

- Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000.
  - Result: Acceptable.
- Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1.
  - Result: Acceptable.
- Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7
  - Result: Unacceptable, because it violates referential integrity.
- 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 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.

Unary Relational Operations: SELECT and PROJECT

The SELECT Operation

The SELECT operation is used to choose a subset of the tuples from a relation that satisfies a selection condition. One can consider the SELECT operation to be a filter that keeps only those tuples that satisfy a qualifying condition. Alternatively, we can consider the SELECT operation to restrict the tuples in a relation to only those tuples that satisfy the 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 discarded.

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

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

In general, the SELECT operation is denoted by

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

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. Notice that R is generally a relational algebra expression whose result is a relation—the simplest such expression is just the name of a database relation. The relation resulting from the SELECT operation has the same attributes as R. The Boolean expression specified in <selection condition> is made up of a number of clauses of the form

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

or

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

where <attribute name> is the name of an attribute of R, <comparison op> is normally one of the operators {=, <, ≤, >, ≥, ≠}, and <constant value> is a constant value from the attribute domain. Clauses can be connected by the standard Boolean operators and, or, and not to form a general selection condition. For example, to select the tuples for all employees who either work in department 4 and make over $25,000 per year, or work in department 5 and make over $30,000, we can specify the following SELECT operation:

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

The result is shown in Figure 8(a). Notice that all the comparison operators in the set {=, <, ≤, >, ≥, ≠} can apply to attributes whose domains are ordered values, such as numeric or date domains. The <selection condition> is applied independently to each individual tuple t in R. This is done by substituting each occurrence of an attribute A_i in the selection condition with its value in the tuple $t[A_i]$. If the condition evaluates to TRUE , then tuple t is selected. All the selected tuples appear in the result of the SELECT operation. The Boolean conditions AND, OR, and NOT have their normal interpretation, as follows:

- (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. The number of tuples in the resulting relation is always less than or equal to the number of tuples in R. Notice that the SELECT operation is commutative; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

Hence, a sequence of SELECTs can be applied in any order. In addition, we can always combine a cascade (or sequence) of SELECT operations into a single SELECT operation with a conjunctive (AND) condition; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\ldots)) = \sigma_{<cond1>\ AND\ <cond2>\ AND\ \ldots\ AND\ <condn>}(R)$$

In SQL, the SELECT condition is typically specified in the WHERE clause of a query. For example, the following operation:

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

would correspond to the following SQL query:

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

Figure 8

Results of SELECT and PROJECT operations. (a) $\sigma_{(Dno=4\ \textbf{AND}\ Salary>25000)\ OR\ (Dno=5\ \textbf{AND}\ Salary>30000)}$ (EMPLOYEE). (b) $\pi_{Lname,\ Fname,\ Salary}$(EMPLOYEE). (c) $\pi_{Sex,\ Salary}$(EMPLOYEE).

(a)

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

(b)

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

(c)

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

The PROJECT Operation

If we think of a relation as a table, the SELECT operation chooses some of the rows from the table while discarding other rows. The PROJECT operation, on the other hand, selects certain columns from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to project the relation over these attributes only. Therefore, the result of the PROJECT operation can be visualized as a vertical partition of the relation into two relations: one has the needed columns (attributes) and contains the result of the operation, and the other contains the discarded columns.

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

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

The resulting relation is shown in Figure 8(b). The general form of the PROJECT operation is

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

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

If the attribute list includes only nonkey attributes of R, duplicate tuples are likely to occur. The PROJECT operation removes any duplicate tuples, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as duplicate elimination. For example, consider the following PROJECT operation:

$$\Pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$$

The result is shown in Figure 8(c). Notice that the tuple <'F', 25000> appears only once in Figure 8(c), even though this combination of values appears twice in the EMPLOYEE relation.

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. Moreover,

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

as long as <list2> contains the attributes in <list1>; otherwise, the left-hand side is an incorrect expression. It is also noteworthy that commutativity does not hold on PROJECT.

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

$$\Pi_{\text{Sex, Salary}}(\text{EMPLOYEE})$$

would correspond to the following SQL query:

```
SELECT DISTINCT Sex , Salary
FROM EMPLOYEE ;
```

Notice that if we remove the keyword DISTINCT from this SQL query, then duplicates will not be eliminated. This option is not available in the formal relational algebra.

Sequences of Operations and the RENAME Operation

The relations shown in Figure 8 that depict operation results do not have any names. In general, for most queries, we need to apply several relational algebra operations one after the other. Either we can write the operations as a single relational algebra expression by nesting the operations, or we can apply one operation at a time and create intermediate result relations. In the latter case, we must give names to the relations that hold the intermediate results.

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

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

Figure 9(a) shows the result of this in-line relational algebra expression. Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

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

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

$$\text{TEMP} \leftarrow \sigma_{\text{Dno=5}}(\text{EMPLOYEE})$$
$$\text{R}(\text{First\_name, Last\_name, Salary}) \leftarrow \Pi_{\text{Fname, Lname, Salary}}(\text{TEMP})$$

These two operations are illustrated in Figure 9(b). If no renaming is applied, the names of the attributes in the resulting relation of a SELECT operation are the same as those in the original relation and in the same order. For a PROJECT operation with no renaming, the resulting relation has the same attribute names as those in the projection list and in the same order in which they appear in the list.

Figure 9

(a)

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

Figure 6.2
Results of a sequence of operations. (a) $\pi_{Fname, Lname, Salary}(\sigma_{Dno=5}(EMPLOYEE))$. (b) Using intermediate relations and renaming of attributes.

(b)

TEMP

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

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:

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

where the symbol ρ(rho) is used to denote the RENAME operator, S is the new relation name, and B_1, B_2, ..., B_n are the new attribute names. The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of R are (A_1, A_2, ..., A_n) in that order, then each A_i is renamed as B_i. 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;
```

Relational Algebra Operations from Set Theory

The UNION, INTERSECTION, and MINUS Operations

The next 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:

$$DEP5\_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$
$$RESULT1 \leftarrow \Pi_{Ssn}(DEP5\_EMPS)$$
$$RESULT2(Ssn) \leftarrow \Pi_{Super\_ssn}(DEP5\_EMPS)$$
$$RESULT \leftarrow RESULT1 \cup RESULT2$$

The relation RESULT1 has the Ssn of all employees who work in department 5, whereas RESULT2 has the Ssn of all employees who directly supervise an employee who works in

department 5. The UNION operation produces the tuples that are in either RESULT1 or RESULT2 or both (see Figure 10), while eliminating any duplicates. Thus, the Ssn value '333445555' appears only once in the result.

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

Figure 6.3
Result of the UNION operation RESULT ← RESULT1 ∪ RESULT2.

Figure 10

Several set theoretic operations are used to merge the elements of two sets in various ways, including UNION , INTERSECTION , and SET DIFFERENCE (also called MINUS or EXCEPT). These are binary operations; that is, each is applied to two sets (of tuples). When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same type of tuples; this condition has been called **union compatibility** or **type compatibility**. Two relations $R(A_1 , A_2 , ..., A_n)$ and $S(B_1 , B_2 , ..., B_n)$ are said to be union compatible (or type compatible) if they have the same degree n and if $dom(A_i) = dom(B_i)$. This means that the two relations have the same number of attributes and each corresponding pair of attributes has the same domain.

We can define the three operations UNION, INTERSECTION, and SET DIFFERENCE

on two union-compatible relations R and S as follows:

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

Figure 11 illustrates the three operations. The relations STUDENT and INSTRUCTOR in Figure 11(a) are union compatible and their tuples represent the names of students and the names of instructors, respectively. The result of the UNION operation in Figure 11(b) shows the names of all students and instructors. Note that duplicate tuples appear only once in the result. The result of the INTERSECTION operation (Figure 11(c)) includes only those who are both students and instructors. 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,

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

Figure 11(d) shows the names of students who are not instructors, and Figure 11(e) shows the names of instructors who are not students. Note that INTERSECTION can be expressed in terms of union and set difference as follows:

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

Figure 11

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

(a) STUDENT

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

INSTRUCTOR

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

(b)

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

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

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

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

In SQL, there are three operations— UNION, INTERSECT, and EXCEPT —that correspond to the set operations described here. In addition, there are multiset operations (UNION ALL, INTERSECT ALL, and EXCEPT ALL) that do not eliminate duplicates.

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

FEMALE_EMPS

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

EMPNAMES

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

EMP_DEPENDENTS

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

ACTUAL_DEPENDENTS

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

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

Figure 12*: The Cartesian Product (Cross Product) operation.*

Next, we discuss the CARTESIAN PRODUCT operation—also known as CROSS PRODUCT or CROSS JOIN— which is denoted by ×. This is also a binary set operation, but the relations on which it is applied do not have to be union compatible. In its binary form, this set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set). In general, the result of $R(A_1, A_2, ..., A_n) \times S(B_1, B_2, ..., B_m)$ is a relation Q with degree n + m attributes $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$, in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S. Hence, if R has n_R tuples and S has n_S tuples, then R × S will have $n_R * n_S$ tuples.

In general, the CARTESIAN PRODUCT operation applied by itself is generally meaningless. It is mostly useful when followed by a selection that matches values of attributes coming from the component relations. For example, suppose that we want to retrieve a list of names of each female employee's dependents. We can do this as follows:

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

The resulting relations from this sequence of operations are shown in Figure 12. The EMP_DEPENDENTS relation is the result of applying the CARTESIAN PRODUCT operation to EMPNAMES from Figure 12 with DEPENDENT from Figure 6. In EMP_DEPENDENTS , every tuple from EMPNAMES is combined with every tuple from DEPENDENT , giving a result that is not very meaningful (every dependent is combined with every female employee). We want to combine a female employee tuple only with her particular dependents—namely, the DEPENDENT tuples whose Essn value match the Ssn value of the EMPLOYEE tuple. The ACTUAL_DEPENDENTS relation accomplishes this. The EMP_DEPENDENTS relation is a good example of the case where relational algebra can be correctly applied to yield results that make no sense at all.

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, as we did in the preceding example. Because this 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.

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.

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

$$\text{DEPT\_MNGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE} \quad 、$$
$$\text{RESULT} \leftarrow \Pi_{\text{Dname, Lname, Fname}}(\text{DEPT\_MNGR})$$

The first operation is illustrated in Figure 13. Note that 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. Consider the earlier example illustrating CARTESIAN PRODUCT, which included the following sequence of operations:

$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES X DEPENDENT}$$
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP\_DEPENDENTS})$$

These two operations can be replaced with a single JOIN operation as follows:

$$\text{ACTUAL\_DEPENDENTS} \leftarrow \text{EMPNAMES} \bowtie_{\text{Ssn=Essn}} \text{DEPENDENT}$$

The general form of a JOIN operation on two relations $R(A_1, A_2, ..., A_n)$ and $S(B_1, B_2, ..., B_m)$ is $R \bowtie_{<\text{join condition}>} S$. The result of the JOIN is a relation Q with n + m attributes $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$ in that order; Q has one tuple for each combination of tuples—one from R and one from S—whenever the combination satisfies the join condition. This is the main difference between CARTESIAN PRODUCT and JOIN . In JOIN , only combinations of tuples satisfying the join condition appear in the result, whereas in the CARTESIAN PRODUCT all combinations of tuples are included in the result.

A general join condition is of the form <condition> AND <condition> AND...AND <condition> where each <condition> is of the form $A_i \theta B_j$, A_i is an attribute of R, B_j is an attribute of S, A_i and B_j have the same domain, and θ(theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠}. A JOIN operation with such a general join condition is called a THETA JOIN . Tuples whose join attributes are NULL or for which the join condition is FALSE do not appear in the result.

DEPT_MGR

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

Figure 6.6
Result of the JOIN operation DEPT_MGR ← DEPARTMENT $\bowtie_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE.

Figure 13

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. Notice that in the result of an EQUIJOIN we always have one or more pairs of attributes that have identical values in every tuple. For example, in Figure 13, the values of the attributes Mgr_ssn and Ssn are identical in every tuple of DEPT_MGR (the EQUIJOIN result) because the equality join condition specified on these two attributes requires the values to be identical in every tuple in the result.

The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first. Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project. In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum —so that it has the same name as the Dnum attribute in PROJECT —and then we apply NATURAL JOIN:

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

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

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

The attribute Dnum is called the join attribute for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations. The resulting relation is illustrated in Figure 14(a). In the PROJ_DEPT relation, each tuple combines a PROJECT tuple with the DEPARTMENT tuple for the department that controls the project, but only one join attribute value is kept. If the attributes on which the natural join is specified already have the same names in both relations, renaming is unnecessary. For example, to apply a natural join on the Dnumber attributes of DEPARTMENT and DEPT_LOCATIONS , it is sufficient to write

$$\text{DEPT\_LOCS} \leftarrow \text{DEPARTMENT} * \text{DEPT\_LOCATIONS}$$

The resulting relation is shown in Figure 14(b), which combines each department with its locations and has one tuple for each location.

Notice that if no combination of tuples satisfies the join condition, the result of a JOIN is an empty relation with zero tuples. In general, if R has n_R tuples and S has n_S tuples, the result of a JOIN operation $R \bowtie_{\text{<join condition>}} S$ will have between zero and $n_R * n_S$ tuples. The expected size of the join result divided by the maximum size $n_R * n_S$ leads to a ratio called **join selectivity**, which is a property of each join condition.

Informally, an **inner join** is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. 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:

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

This combines each project tuple with its controlling department tuple into a single tuple, and then combines that tuple with an employee tuple that is the department manager. The net result is a consolidated relation in which each tuple contains this project-department-manager combined information.

(a)

PROJ_DEPT

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

(b)

DEPT_LOCS

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

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

Figure 14

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

Figure 15

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

(a)

SSN_PNOS

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

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

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

S

A
a1
a2
a3

T

B
b1
b4

The DIVISION operation, denoted by ÷, is useful for a special kind of query that sometimes occurs in database applications. An example is Retrieve the names of employees who work on all the projects that 'John Smith' works on. To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that 'John Smith' works on in the intermediate relation SMITH_PNOS:

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

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

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

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

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

The preceding operations are shown in Figure 15(a).

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 t_R appear in R with $t_R[Y] = t$, and with $t_R[X] = t_S[X]$ for every tuple t_S in S. This means that, for a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with every tuple in S.

Figure 15(b) illustrates a 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.

Notation for Query Trees

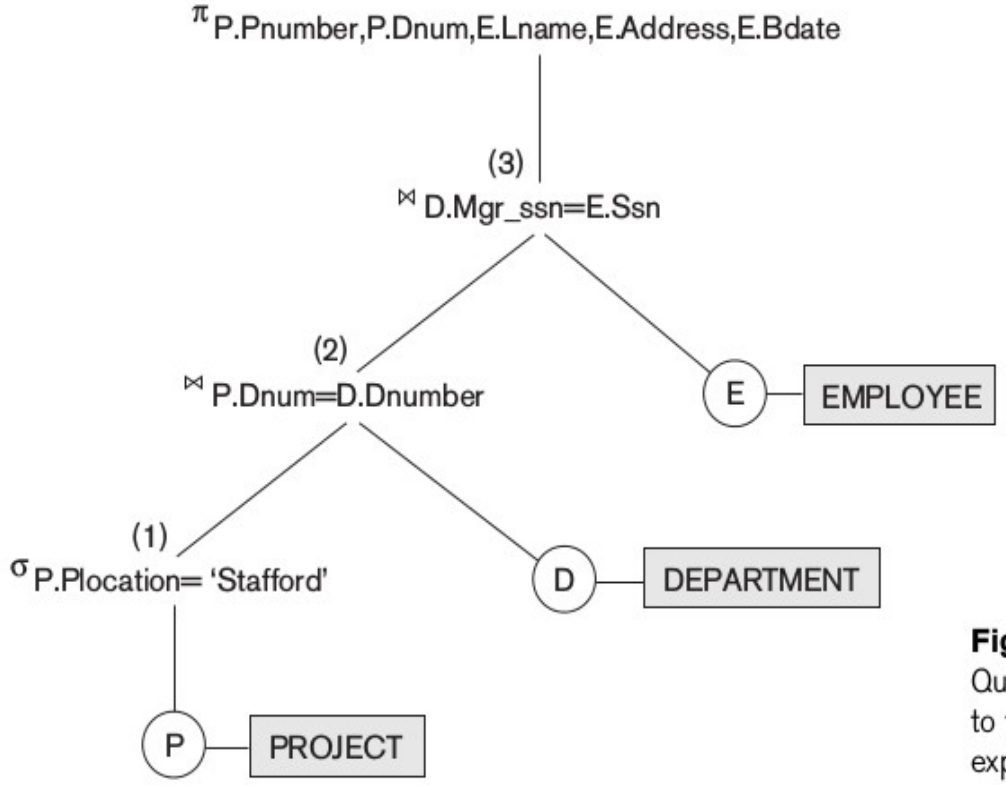


Figure 6.9
Query tree corresponding to the relational algebra expression for Q2.

Figure 16

In this section we describe a notation typically used in relational systems to represent queries internally. The notation is called a query tree or sometimes it is known as a query evaluation tree or query execution tree.

A query tree is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands (represented by its child nodes) are available, and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Figure 16 shows a query tree for Query: 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.

$$\Pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation='Stanford'}}(\text{PROJECT}))\bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT}))\bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$

In Figure 16, the three leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE. The relational algebra operations in the expression are represented by internal tree nodes. In order to execute the Query, the node marked (1) in Figure 16 must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin to execute operation (2). Similarly, node (2) must begin to execute and produce results before node (3) can start execution, and so on.

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 F_1, F_2, ..., F_n 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:

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

Aggregate Functions and Grouping

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

Another common type of request involves grouping the tuples in a relation by the value of some of their attributes and then applying an aggregate function independently to each group. An example would be to group EMPLOYEE tuples by Dno, so that each group includes the tuples for employees working in the same department. We can then list each Dno value along with, say, the average salary of employees within the department, or the number of employees who work in the department.

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

$$\text{<grouping attributes>}\ \Im\ \text{<function list>}\ (R)$$

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

For 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:

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

The result of this operation on the EMPLOYEE relation is shown in Figure 17(a). In the above example, we specified a list of attribute names—between parentheses in the RENAME operation—for the resulting relation R. If no renaming is applied, then the attributes of the resulting relation that correspond to the function list will each be the concatenation of the function name with the attribute name in the form <function>_<attribute>. For example, Figure 17(b) shows the result of the following operation:

$${}_{\text{Dno}}\Im_{\text{Count Ssn,Average Salary}}(\text{EMPLOYEE})$$

If no grouping attributes are specified, the functions are applied to all the tuples in the relation, so the resulting relation has a single tuple only. For example, Figure 17(c) shows the result of the following operation:

$$\Im_{\text{Count Ssn,Average Salary}}(\text{EMPLOYEE})$$

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.

Figure 17

The aggregate function operation.

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

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

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

R

(a)

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

(b)

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

(c)

Count_ssn	Average_salary
8	35125

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.

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. It is relatively straightforward in the relational algebra to specify all employees supervised by e at a specific level by joining the table with itself one or more times.

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:

$$\text{BORG\_SSN} \leftarrow \Pi_{\text{Ssn}}(\sigma_{\text{Fname}='\text{James}'\text{ AND Lname}='\text{Borg}'}(\text{EMPLOYEE}))$$
$$\text{SUPERVISION}(\text{Ssn1}, \text{Ssn2}) \leftarrow \Pi_{\text{Ssn, Super\_ssn}}(\text{EMPLOYEE})$$
$$\text{RESULT1}(\text{Ssn}) \leftarrow \Pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2}=\text{Ssn}} \text{BORG\_SSN})$$

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:

$$\text{RESULT2}(\text{Ssn}) \leftarrow \Pi_{\text{Ssn1}}(\text{SUPERVISION} \bowtie_{\text{Ssn2}=\text{Ssn}} \text{RESULT1})$$

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:

$$\text{RESULT} \leftarrow \text{RESULT1} \cup \text{RESULT2}$$

The results of these queries are illustrated in Figure 18.

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)

Figure 18 *A two-level recursive query.*

OUTER JOIN Operations

The JOIN operations described earlier match tuples that satisfy the join condition. For example, for a NATURAL JOIN operation R * S, only tuples from R that have matching tuples in S—and vice versa—appear in the result. Hence, tuples without a matching (or related) tuple are eliminated from the JOIN result. Tuples with NULL 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**. The join operations we described earlier are all inner joins.

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. For 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; if they do not manage

one, we can indicate it with a NULL value. We can apply an operation LEFT OUTER JOIN , denoted by ⟕ , to retrieve the result as follows:

$$\text{TEMP} \leftarrow (\text{EMPLOYEE} \text{ ⟕}_{\text{Ssn}=\text{Mgr\_ssn}} \text{DEPARTMENT})$$
$$\text{RESULT} \leftarrow \Pi_{\text{Fname, Minit, Lname, Dname}}(\text{TEMP})$$

The LEFT OUTER JOIN operation keeps every tuple in the first, or left, relation R in R ⟕ S; if no matching tuple is found in S, then the attributes of S in the join result are filled or padded with NULL values. The result of these operations is shown in Figure 19.

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.

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

***Figure 19**: The result of a LEFTOUTER JOIN*

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 t_1 in R and t_2 in S are said to match if $t_1[X]=t_2[X]$. 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.

For example, an OUTER UNION can be applied to two relations whose schemas are STUDENT(Name, Ssn, Department, Advisor) and INSTRUCTOR(Name, Ssn, Department, Rank). Tuples from the two relations are matched based on having the same combination of values of the shared attributes— Name, Ssn, Department. The resulting relation, STUDENT_OR_INSTRUCTOR , will have the following attributes:

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

All the tuples from both relations are included in the result, but tuples with the same (Name, Ssn, Department) combination will appear only once in the result. Tuples appearing only in STUDENT will have a NULL for the Rank attribute, whereas tuples appearing only in INSTRUCTOR will have a NULL for the Advisor attribute. A tuple that exists in both relations, which represent a student who is also an instructor, will have values for all its attributes.

Table 6.1 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 $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

Examples of Queries in Relational Algebra

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

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

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

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

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

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

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.

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

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.

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

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.

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

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

$$\text{MGRS}(\text{Ssn}) \leftarrow \Pi_{\text{Mgr\_ssn}}(\text{DEPARTMENT})$$
$$\text{EMPS\_WITH\_DEPS}(\text{Ssn}) \leftarrow \Pi_{\text{Essn}}(\text{DEPENDENT})$$
$$\text{MGRS\_WITH\_DEPS} \leftarrow \text{MGRS} \cap \text{EMPS\_WITH\_DEPS}$$
$$\text{RESULT} \leftarrow \Pi_{\text{Lname, Fname}}(\text{MGRS\_WITH\_DEPS} * \text{EMPLOYEE})$$