Relational Keys

There are six keys those play important role in relational database. Super key, candidate key, and surrogate key are used in designing phase, but primary key, composite key, and foreign key are implementation concept.

Doctor ID	Patient ID	Appointment Date	Time	Room
D001	P001	23 May 07	1800	KW 00

Table 2.1 Appointment Table

Supper Key

A supper key is a column or set of columns that can uniquely identify the occurrences of each entity. In table 1, the supper key is (Doctor ID, Patient ID, Appointment Date, Time, and room). Supper keys are selected in a pessimistic way.

Candidate Key

A candidate key is a set of a supper key that has minimum number of attributes necessary for unique identification of each entity occurrences. The candidate depends on business rules of an organization. For example,
Business Rule 1: Patient can see doctor more than one in a day
Candidate Key (Doctor ID, Patient ID, Appointment Date, Time, and room).
Business Rule 2: Patient can see doctor more than once in any day but in different room.
Candidate Key (Doctor ID, Patient ID, Appointment Date, and Time).
Business Rule 3: Patient can see doctor more than once but in different room and day.
Candidate Key (Doctor ID, Patient ID and Time).

Surrogate Key

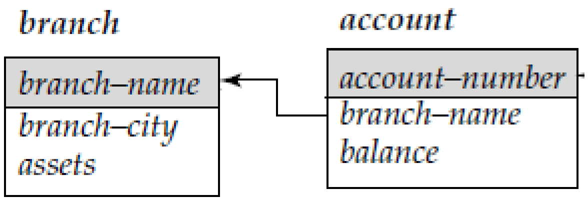
A surrogate key is used to uniquely identify of each entity occurrences when a candidate key is very long. The key does not relate with the context. The surrogate key for the appointment table is appointment ID.

Primary Key

A primary is an attribute that uniquely identifies each row in a table. A PRIMARY KEY constraint creates a primary key for the table. Only one primary key can be created for an each table. The PRIMARY KEY constraint is a column or set of columns that uniquely identifies each row in a table. This constraint enforces uniqueness of the column or column combination and ensures that no column that is part of the primary key can contain a null value. A composite is a primary key that consists of two or more attributes.

Foreign Key

Foreign keys are used to represent One-to-Many relationship, is an attributes of one table that serves as a primary key of another table in the same database. The FOREIGN KEY, or referential integrity constraint, designates a column or combination of columns as a foreign key and establishes a relationship between a primary key or a unique key in the same table or a different table. A foreign key value must match an existing value in the parent table or be NULL. Foreign keys are based on data values and are purely logical, not physical, pointers.



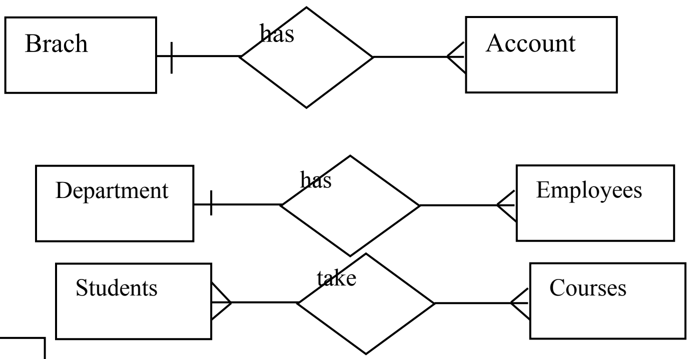
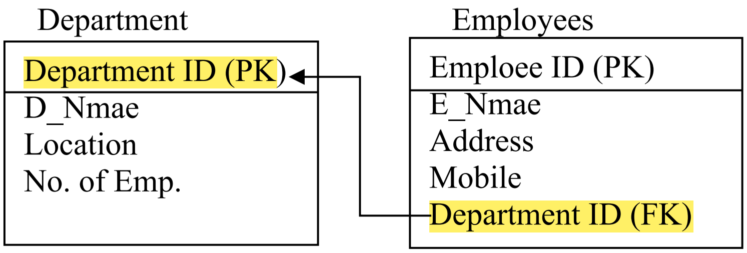


Figure: ER Diagram



Department Table

Department ID	D_Nmae	Location	No. of Emp
001	IT	Dhaka	02
002	HR	Feni	05
003	Sales	Chittagong	19

Employee Table

Employee ID	E_Nmae	Phone	Department ID
0001	Joy	01766853636	001
0002	Zehan	01820862923	001
0003	Mehdi	01666553321	002

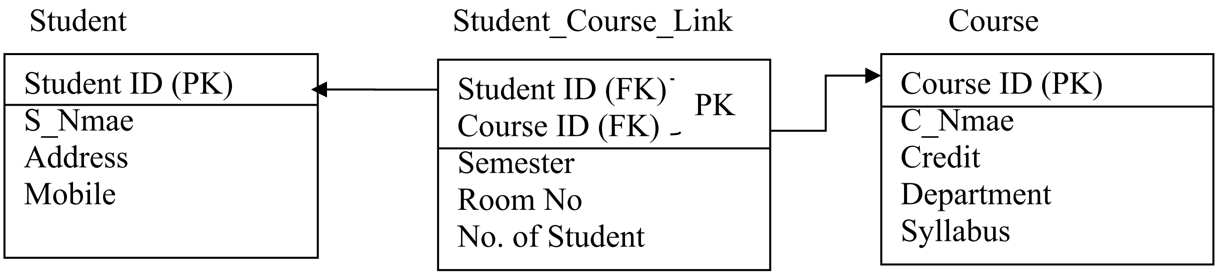


Figure: Relational Schema

Relational Model

The relational model is today the primary data model for commercial data-processing applications. It has attained its primary position because of its simplicity, which eases the job of the programmer, as compared to earlier data models such as the network model or the hierarchical model. In this chapter, we first study the fundamentals of the relational model, which provides a very simple yet powerful way of representing data. In addition, we cover the formal query language, relational algebra, in great detail. The relational algebra forms the basis of the widely used SQL query language.

Structure of Relational Databases

A relational database consists of a collection of **tables**, each of which is assigned a unique name. Each table has a structure similar to that presented in Chapter 7, where we represented E-R databases by tables. A row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In what follows, we introduce the concept of relation.

Basic Structure

Consider the *account* table of Figure 3.1. It has three column headers: *account-number*, *branch-name*, and *balance*. Following the terminology of the relational model, we refer to these headers as **attributes**. For each attribute, there is a set of permitted values, called the **domain** of that attribute. For the attribute *branch-name*, for example, the domain is the set of all branch names. Let $D1$ denote the set of all account numbers, $D2$ the set of all branch names, and $D3$ the set of all balances. As we saw in Chapter 2, any row of *account* must consist of a 3-tuple ($v1, v2, v3$), where $v1$ is an account number (that is, $v1$ is in domain $D1$), $v2$ is a branch name (that is, $v2$ is in domain $D2$), and $v3$ is a balance (that is, $v3$ is in domain $D3$). In general, *account* will contain only a subset of the set of all possible rows. Therefore, *account* is a subset of

$$D1 \times D2 \times D3$$

In general, a **table** of n attributes must be a subset of

$$D1 \times D2 \times \cdots \times Dn-1 \times Dn$$

Mathematicians define a **relation** to be a subset of a Cartesian product of a list of domains. This definition corresponds almost exactly with our definition of *table*. The only difference is that we have assigned names to attributes, whereas mathematicians rely on numeric "names," using the integer 1 to denote the attribute whose domain appears first in the list of domains, 2 for the attribute whose domain appears second, and so on. Because tables are essentially relations, we shall use the mathematical

account-number	***branch-name***	***Balance***
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

Figure 3.1 The *account* relation.

account-number	***branch-name***	***Balance***
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Brighton	900
A-222	Redwood	700
A-217	Brighton	750

Figure 3.2 The *account* relation with unordered tuples.

terms **relation** and **tuple** in place of the terms **table** and **row**. A **tuple variable** is a variable that stands for a tuple; in other words, a tuple variable is a variable whose domain is the set of all tuples.

In the *account* relation of Figure 3.1, there are seven tuples. Let the tuple variable *t* refer to the first tuple of the relation.We use the notation *t*[*account-number*] to denote the value of *t* on the *account-number* attribute. Thus, *t*[*account-number*] = "A-101," and *t*[*branch-name*] = "Downtown". Alternatively, we may write *t*[1] to denote the value of tuple *t* on the first attribute (*account-number*), *t*[2] to denote *branch-name*, and so on.

Since a relation is a set of tuples, we use the mathematical notation of *t* *r* to denote that tuple *t* is in relation *r*. The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 3.1, or are unsorted, as in Figure 3.2, does not matter; the relations in the two figures above are the same, since both contain the same set of tuples. We require that, for all relations *r*, the domains of all attributes of *r* be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units. For example, the set of integers is an atomic domain, but the set of all sets of integers is a nonatomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts—namely, the integers composing the set. The important issue is not what the domain itself is, but rather how we use domain elements in our database. The domain of all integers would be nonatomic if we considered each integer to be an ordered list of digits. In all our examples, we shall assume atomic domains. In Chapter 9, we shall discuss extensions to the relational data model to permit nonatomic domains.It is possible for several attributes to have the same domain. For example, suppose that we have a relation *customer* that has the three attributes *customer-name*, *customer-street*, and *customer-city*, and a relation *employee* that includes the attribute *employee-name*. It is possible that the attributes *customer-name* and *employee-name* will have the same domain: the set of all person names, which at the physical level is the set of all character strings. The domains of *balance* and *branch-name*, on the other hand, certainly ought to be distinct. It is perhaps less clear whether *customer-name* and *branch-name* should have the same domain. At the physical level, both customer names and branch names are character strings. However, at the logical level, we may want *customer-name* and *branch-name* to have distinct domains.

One domain value that is a member of any possible domain is the **null** value, which signifies that the value is unknown or does not exist. For example, suppose that we include the attribute *telephone-number* in the *customer* relation. It may be that a customer does not have a telephone number, or that the telephone number is unlisted. We would then have to resort to null values to signify that the value is unknown or does not exist. We describe the effect of nulls on different operations.

- ✓ It is possible for tuples to have a null value, denoted by *null*, for some of their attributes

- ✓ *null* signifies an unknown value or that a value does not exist.
- ✓ The result of any arithmetic expression involving *null* is *null.*
- ✓ Aggregate functions simply ignore null values (as in SQL)

There is often more than one possible way of dealing with null values, and as a result our definitions can sometimes be arbitrary. Operations and comparisons on null values should therefore be avoided, where possible Since the special value *null* indicates "value unknown or nonexistent," any arithmetic operations (such as *+,−,**, */*) involving null values must return a null result. Similarly, any comparisons (such as *<,<=,>,>=,* _=) involving a null value evaluate to special value **unknown**; we cannot say for sure whether the result of the comparison is true or false, so we say that the result is the new truth value *unknown*. Comparisons involving nulls may occur inside Boolean expressions involving the and, or, and not operations. We must therefore define how the three Boolean operations deal with the truth value *unknown*.

• **and**: (*true* **and** *unknown*) = *unknown*; (*false* **and** *unknown*) = *false*; (*unknown* **and** *unknown*) = *unknown*.

• **or**: (*true* **or** *unknown*) = *true*; (*false* **or** *unknown*) = *unknown*; (*unknown* **or** *unknown*) = *unknown*.

• **not**: (**not** *unknown*) = *unknown*.

Result of select predicate is treated as *false* if it evaluates to *unknown*

Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and a **database instance**, which is a snapshot of the data in the database at a given instant in time. The concept of a relation corresponds to the programming-language notion of a variable. The concept of a **relation schema** corresponds to the programming-language notion of type definition. It is convenient to give a name to a relation schema, just as we give names to type definitions in programming languages. We adopt the convention of using lowercase names for relations, and names beginning with an uppercase letter for relation schemas. Following this notation, we use *Account-schema* to denote the relation schema for relation *account*. Thus,

Account-schema = (*account-number*, *branch-name*, *balance*)

We denote the fact that *account* is a relation on *Account-schema* by *account*(*Account-schema*)

In general, a relation schema consists of a list of attributes and their corresponding domains

The concept of a **relation instance** corresponds to the programming language notion of a value of a variable. The value of a given variable may change with time; similarly the contents of a relation instance may change with time as the relation is updated. However, we often simply say "relation" when we actually mean "relation instance."

As an example of a relation instance, consider the *branch* relation of Figure 3.3. The schema for that relation is

Branch-schema = (*branch-name*, *branch-city*, *assets*)

Note that the attribute *branch-name* appears in both *Branch-schema* and *Account schema*.

This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations. For example, suppose we wish to find the information about all of the accounts maintained in branches

branch-name	***branch-city***	***assets***
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000

Mianus	Horseneck	400000
North	Town Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Figure 3.3 The *branch* relation.

located in Brooklyn. We look first at the *branch* relation to find the names of all the branches located in Brooklyn. Then, for each such branch, we would look in the *account* relation to find the information about the accounts maintained at that branch. This is not surprising—recall that the primary key attributes of a strong entity set appear in the table created to represent the entity set, as well as in the tables created to represent relationships that the entity set participates in. Let us continue our banking example. We need a relation to describe information about customers. The relation schema is

Customer-schema = (*customer-name*, *customer-street*, *customer-city*)

Figure 3.4 shows a sample relation *customer* (*Customer-schema*). Note that we have omitted the *customer-id* attribute, which we used Chapter 2, because now we want to have smaller relation schemas in our running example of a bank database. We assume that the customer name uniquely identifies a customer—obviously this may not be true in the real world, but the assumption makes our examples much easier to read.

customer-name	***customer-street***	***customer-city***
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

Figure 3.4 The *customer* relation.

In a real-world database, the *customer-id* (which could be a *social-security* number, or an identifier generated by the bank) would serve to uniquely identify customers. We also need a relation to describe the association between customers and accounts. The relation schema to describe this association is

Depositor -schema = (*customer-name*, *account-number*)

Figure 3.5 shows a sample relation *depositor* (*Depositor-schema*). It would appear that, for our banking example, we could have just one relation schema, rather than several. That is, it may be easier for a user to think in terms of one relation schema, rather than in terms of several. Suppose that we used only one relation for our example, with schema

(*branch-name*, *branch-city*, *assets*, *customer-name*, *customer-street*
customer-city, *account-number*, *balance*)

Observe that, if a customer has several accounts, we must list her address once for each account. That is, we must repeat certain information several times. This repetition is wasteful and is avoided by the use of several relations, as in our example. In addition, if a branch has no accounts (a newly created branch, say, that has no customers yet), we cannot construct a complete tuple on the preceding single relation, because no data concerning *customer* and *account* are available yet. To represent incomplete tuples, we must use *null* values that signify that the value is unknown or does not exist. Thus, in our example, the values for *customer-name*, *customer-street*, and so on must be null. By using several relations, we can represent the branch information for a bank with no customers without using null values. We simply use a tuple on *Branch-schema* to represent the information about the branch, and create tuples on the other schemas only when the appropriate information becomes available. In Chapter 7, we shall study criteria to help us decide when one set of relation schemas is more appropriate than another, in terms of information repetition and the existence of null values. For now, we shall assume that the relation schemas are given.

We include two additional relations to describe data about loans maintained in the various branches in the bank:

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

Figure 3.5 The *depositor* relation.

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

Figure 3.6 The *loan* relation.

Loan-schema = (*loan-number*, *branch-name*, *amount*)
Borrower-schema = (*customer-name, loan-number*)

Figures 3.6 and 3.7, respectively, show the sample relations *loan* (*Loan-schema*) and *borrower* (*Borrower-schema*).

The E-R diagram in Figure 3.8 depicts the banking enterprise that we have just described. The relation schemas correspond to the set of tables that we might generate by the method outlined in Section 2.9. Note that the tables for *account-branch* and *loan-branch* have been combined into the tables for *account* and *loan* respectively. Such combining is possible since the relationships

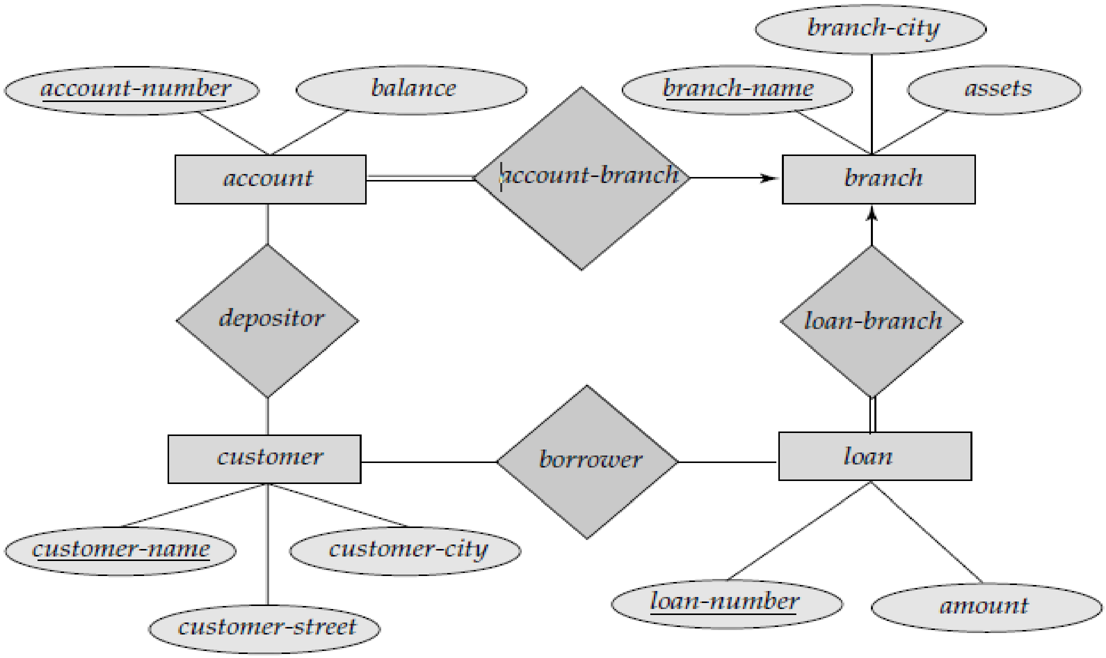


Figure 3.8 E-R diagram for the banking enterprise.

are many to one from *account* and *loan*, respectively, to *branch*, and, further, the participation of *account* and *loan* in the corresponding relationships is total, as the double lines in the figure indicate. Finally, we note that the *customer* relation may contain information about customers who have neither an account nor a loan at the bank.

The banking enterprise described here will serve as our primary example in this chapter and in subsequent ones. On occasion, we shall need to introduce additional relation schemas to illustrate particular points.

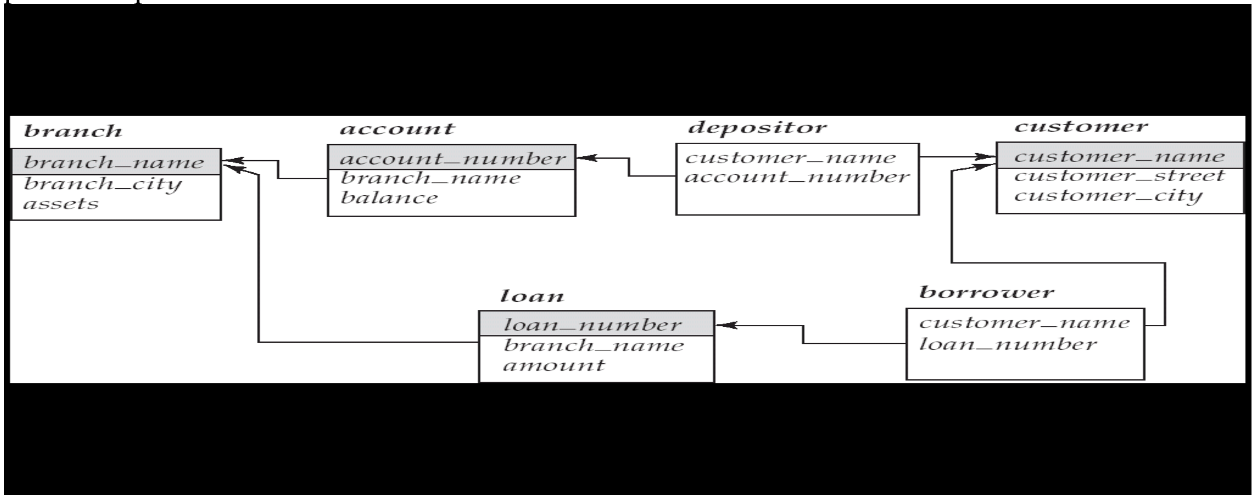


Figure: Relational Data Model for banking enterprise