Underlying the structure of a database is the concept of a data model. Modeling is done at three levels of abstraction: physical, logical, and view. The physical level describes how data are stored at a low level and captures implementation details that are not covered in this chapter. The logical level describes the real-world entities to be modeled in the database and the relationships among these entities. The view level describes only a part of the entire database and serves the needs of users who do not require access to the entire database. Typically in a database, the same data model is used at both the logical and the view levels. Data models differ in the primitives available for describing data and in the amount of semantic detail that can be expressed.

Database system interfaces used by application programs are based on the logical data model; databases hide the underlying implementation details from applications. This conceptual separation of physical organization from logical structure allows the implementation of database applications to be independent of the underlying implementation of the data structures representing the data. The low-level implementations affect system performance and there has been considerable effort to design physical organizations that take advantage of how certain applications access data and how the underlying computer architectures store and manipulate data.

In this chapter, we focus on logical data models. In Section 8.1 we cover the relational data model. In Section 8.2 we cover object-based data models including the E-R model (in Section 8.2.1), the object-oriented model (in Section 8.2.2), and the object-relational model (in Section 8.2.3). The E-R model is a representative of a class of models called semantic data models, which provide richer data structuring and modeling capabilities than the record-based format of the relational model.

In Section 8.3, we describe two models for representing nested structures, namely, XML (in Section 8.3.1) and JSON (in Section 8.3.2). These models are widely used for data exchange between applications. In Section 8.5 we outline models for streaming data.
Our coverage of data models is intended as an introduction to the area. In Section 8.6 we provide references for further reading on the topics covered in this chapter, as well as on related topics such as data modeling issues related to knowledge representation.

### 8.1 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.

A relational database consists of a collection of tables, each of which is assigned a unique name. An instance of a table storing customer information is shown in Table 8.1. The table has several rows, one for each customer, and several columns, each storing some information about the customer. The values in the customer-id column of the customer table serve to uniquely identify customers, while other columns store information such as the name, street address, and city of the customer.

The information stored in a database is broken up into multiple tables, each storing a particular kind of information. For example, information about accounts and loans at a bank would be stored in separate tables. Table 8.2 shows an instance of the loan table, which stores information about loans taken from the bank.

In addition to information about “entities” such as customers or loans, there is also a need to store information about “relationships” between such entities. For example, the bank needs to track the relationship between customers and loans. Table 8.3 shows the borrower table, which stores information indicating which customers have taken which loans. If several people have jointly taken a loan, the same loan number would appear several times in the table with different customer-ids (e.g., loan number L-17). Similarly, if a particular customer has taken multiple loans, there would be several rows in the table with the customer-id of that customer (e.g., 019-28-3746), with different loan numbers.

#### Table 8.1 The Customer Table

<table>
<thead>
<tr>
<th>Customer-Id</th>
<th>Customer-Name</th>
<th>Customer-Street</th>
<th>Customer-City</th>
</tr>
</thead>
<tbody>
<tr>
<td>019-28-3746</td>
<td>Smith</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>182-73-6091</td>
<td>Turner</td>
<td>Putnam</td>
<td>Stamford</td>
</tr>
<tr>
<td>192-83-7465</td>
<td>Johnson</td>
<td>Alma</td>
<td>Palo Alto</td>
</tr>
<tr>
<td>244-66-8800</td>
<td>Curry</td>
<td>North</td>
<td>Rye</td>
</tr>
<tr>
<td>321-12-3123</td>
<td>Jones</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>335-57-7991</td>
<td>Adams</td>
<td>Spring</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>336-66-9999</td>
<td>Lindsay</td>
<td>Park</td>
<td>Pittsfield</td>
</tr>
<tr>
<td>677-89-9011</td>
<td>Hayes</td>
<td>Main</td>
<td>Harrison</td>
</tr>
<tr>
<td>963-96-3963</td>
<td>Williams</td>
<td>Nassau</td>
<td>Princeton</td>
</tr>
</tbody>
</table>

#### Table 8.2 The Loan Table

<table>
<thead>
<tr>
<th>Loan-Number</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-11</td>
<td>900</td>
</tr>
<tr>
<td>L-14</td>
<td>1500</td>
</tr>
<tr>
<td>L-15</td>
<td>1500</td>
</tr>
<tr>
<td>L-16</td>
<td>1300</td>
</tr>
<tr>
<td>L-17</td>
<td>1000</td>
</tr>
<tr>
<td>L-23</td>
<td>2000</td>
</tr>
<tr>
<td>L-93</td>
<td>500</td>
</tr>
</tbody>
</table>
8.1.1 Formal Basis

The power of the relational data model lies in its rigorous mathematical foundations and a simple user-level paradigm for representing data. Mathematically speaking, a relation is a subset of the Cartesian product of an ordered list of domains. For example, let $E$ be the set of all employee identification numbers, $D$ the set of all department names, and $S$ the set of all salaries. An employment relation is a set of 3-tuples $(e, d, s)$ where $e \in E$, $d \in D$, and $s \in S$. A tuple $(e, d, s)$ represents the fact that employee $e$ works in department $d$ and earns salary $s$.

At the user level, a relation is represented as a table. The table has one column for each domain and one row for each tuple. Each column has a name, which serves as a column header, and is called an attribute of the relation. The list of attributes for a relation is called the relation schema. The terms table and relation are used synonymously, as are row and tuple, as also column and attribute.

Data models also permit the definition of constraints on the data stored in the database. For instance, key constraints are defined as follows. If a set of attributes $L$ is specified to be a super-key for relation $r$, in any consistent (“legal”) database, the set of attributes $L$ would uniquely identify a tuple in $r$; that is, no two tuples in $r$ can have the same values for all attributes in $L$. For instance, customer-id would form a super-key for relation customer. A relation can have more than one super-key, and usually one of the super-keys is chosen as a primary key; this key must be a minimal set, that is, dropping any attribute from the set would make it cease to be a super-key.

Another form of constraint is the foreign key constraint, which specifies that for each tuple in one relation, there must exist a matching tuple in another relation. For example, a foreign key constraint from borrower referencing customer specifies that for each tuple in borrower, there must be a tuple in customer with a matching customer-id value.

Users of a database system may query the data, insert new data, delete old data, or update the data in the database. Of these tasks, the task of querying the data is usually the most complicated. In the case of the relational data model, since data are stored as tables, a user may query these tables, insert new tuples, delete tuples, and update (modify) tuples. There are several languages for expressing these operations.

The tuple relational calculus and the domain relational calculus are nonprocedural languages that represent the basic power required in a relational query language. Both of these languages are based on statements written in mathematical logic. We omit details of these languages.

The relational algebra is a procedural query language that defines several operations, each of which takes one or more relations as input, and returns a relation as output. For example:

- The selection operation is used to get a subset of tuples from a relation, by specifying a predicate. The selection operation $\sigma_P(r)$ returns the set of tuples of $r$ that satisfy the predicate $P$.
- The projection operation, $\Pi_i(r)$, is used to return a relation containing a specified set of attributes $L$ of a relation $r$, removing the other attributes of $r$.
- The union, operation $r \cup s$ returns the union of the tuples in $r$ and $s$. The intersection and difference operations are similarly defined.

<table>
<thead>
<tr>
<th>Customer-Id</th>
<th>Loan-Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>019-28-3746</td>
<td>L-11</td>
</tr>
<tr>
<td>019-28-3746</td>
<td>L-23</td>
</tr>
<tr>
<td>244-66-8800</td>
<td>L-93</td>
</tr>
<tr>
<td>321-12-3123</td>
<td>L-17</td>
</tr>
<tr>
<td>335-57-7991</td>
<td>L-16</td>
</tr>
<tr>
<td>555-55-5555</td>
<td>L-14</td>
</tr>
<tr>
<td>677-89-9011</td>
<td>L-15</td>
</tr>
<tr>
<td>963-96-3963</td>
<td>L-17</td>
</tr>
</tbody>
</table>
The natural join operation $\bowtie$ is used to combine information from two relations. For example, the natural join of the relations loan and borrower, denoted $\text{loan} \bowtie \text{borrower}$, would be the relation defined as follows. First, match each tuple in loan with each tuple in borrower that has the same values for the shared attribute loan-number; for each pair of matching tuples, the join operation creates a tuple containing all attributes from both tuples; the join result relation is the set of all such tuples.

For instance, the natural join of the loan and borrower tables in Tables 8.2 and 8.3 contains tuples (L-17, 1000, 321-12-3123) and (L-17, 1000, 963-96-3963), since the tuple with loan number L-17 in the loan table matches two different tuples with loan number L-17 in the borrower table.

The relational algebra has other operations as well, for instance, operations that can aggregate values from multiple tuples, for example, by summing them up, or finding their average. Since the result of a relational algebra operation is itself a relation, it can be used in further operations. As a result, complex expressions with multiple operations can be defined in the relational algebra.

Among the reasons for the success of the relational model are its basic simplicity, representing all data using just a single notion of tables, as well as its formal foundations in mathematical logic and algebra. The relational algebra and the relational calculi are terse, formal languages that are inappropriate for casual users of a database system. Commercial database systems have, therefore, used languages with more “syntactic sugar.” Queries in these languages can be translated into queries in relational algebra.

### 8.1.2 SQL

The SQL language has clearly established itself as the standard relational database language. The SQL language has a data definition component for specifying schemas, and a data manipulation component for querying data as well as for inserting, deleting, and updating data.

We illustrate some examples of queries and updates in SQL. The following query finds the name of the customer whose customer-id is 192-83-7465:

```sql
SELECT customer.customer-name
FROM customer
WHERE customer.customer-id = '192-83-7465'
```

Queries may involve information from more than one table. For instance, the following query finds the amount of all loans owned by the customer with customer-id 019-28-3746:

```sql
SELECT loan.loan-number, loan.amount
FROM borrower, loan
WHERE borrower.customer-id = '019-28-3746' AND borrower.loan-number = loan.loan-number
```

If the previous query were run on the tables shown earlier, the system would find that the loans L-11 and L-23 are owned by customer 019-28-3746, and would print out the amounts of the two loans, namely, 900 and 2000.

The following SQL statement adds an interest of 5% to the loan amount of all loans with amount greater than 1000.

```sql
UPDATE loan
SET amount = amount * 1.05
WHERE amount > 10000
```

Over the years, there have been several revisions of the SQL standard. The most recent is SQL:2011.
QBE and Quel are two other significant query languages. Of these, Quel is no longer in wide-
spread use. Although QBE itself is no longer in use, it greatly influenced the QBE-style language in
Microsoft Access.

8.1.3 Relational Database Design

The process of designing a conceptual-level schema for a relational database involves the selection of
a set of relation schemas. There are several approaches to relational database design. One approach,
which we describe later, in Section 8.2.1, is to create a model of the enterprise using a higher-level data
model, such as the entity-relationship model, and then translate the higher-level model into a relational
database design.

Another approach is to directly create a design, consisting of a set of tables and a set of attributes for
each table. There are often many possible choices that the database designer might make. A proper bal-
ance must be struck among three criteria for a good design:

1. Minimization of redundant data
2. Ability to represent all relevant relationships among data items
3. Ability to test efficiently data dependencies that require certain attributes to be unique identifiers

To illustrate these criteria for a good design, consider a database of employees, departments, and man-
gers. Let us assume that a department has only one manager, but a manager may manage one or more
departments. If we use a single relation emp-info1(employee, department, manager), then we must repeat
the manager of a department once for each employee. Thus we have redundant data.

We can avoid redundancy by decomposing (breaking up) the above relation into two relations
emp-mgr(employee, manager) and emp-dept(manager, department). However, consider a manager,
Martin, who manages both the sales and the service departments. If Clark works for Martin, we
cannot represent the fact that Clark works in the service department but not the sales department.
Thus we cannot represent all relevant relationships among data items using the decomposed rela-
tions, such a decomposition is called a lossy-join decomposition. If instead, we chose the two relations
emp-dept(employee, department) and dept-mgr(department, manager), we would avoid this difficulty,
and at the same time avoid redundancy. With this decomposition, joining the information in the two
relations would give back the information in emp-info1, such a decomposition is called a lossless-join
decomposition.

There are several types of data dependencies. The most important of these are functional depen-
dencies. A functional dependency is a constraint that the value of a tuple on one attribute or set of
attributes determines its value on another. For example, the constraint that a department has only
one manager could be stated as “department functionally determines manager.” Because func-
tional dependencies represent facts about the enterprise being modeled, it is important that the
system check newly inserted data to ensure no functional dependency is violated (as in the case of a
second manager being inserted for some department). Such checks ensure that the update does not
make the information in the database inconsistent. The cost of this check depends on the design
of the database.

There is a formal theory of relational database design that allows us to construct designs that have
minimal redundancy, consistent with meeting the requirements of representing all relevant relation-
ships, and allowing efficient testing of functional dependencies. This theory specifies certain properties
that a schema must satisfy, based on functional dependencies. For example, a database design is said to
be in a Boyce–Codd normal form if it satisfies a certain specified set of properties; there are alternative
specifications, for instance, the third normal form. The process of ensuring that a schema design is in a
desired normal form is called normalization.

More details can be found in standard textbooks on databases; Abiteboul et al. [AHV95] provide a
detailed coverage of database design theory.
8.1.4 History

The relational model was developed in the late 1960s and early 1970s by E. F. Codd. For this work, Codd won the most prestigious award in computing, the ACM A. M. Turing Award. The 1970s saw the development of several experimental database systems based on the relational model and the emergence of a formal theory to support the design of relational databases. The commercial application of relational databases began in the late 1970s, but was limited by the poor performance of early relational systems. During the 1980s, numerous commercial relational systems with good performance became available. The relational model has since established itself as the primary data model for commercial data processing applications.

Earlier generation database systems were based on the network data model or the hierarchical data model. Those two older models are tied closely to the data structures underlying the implementation of the database. We omit details of these models since they are now of historical interest only.

8.2 Object-Based Models

The relational model is the most widely used data model at the implementation level; most databases in use around the world are relational databases. However, the relational view of data is often too detailed for conceptual modeling. Data modelers need to work at a higher level of abstraction.

Object-based logical models are used in describing data at the conceptual level. The object-based models use the concepts of entities or objects and relationships among them rather than the implementation-based concepts of the record-based models. They provide flexible structuring capabilities and allow data constraints to be specified explicitly. Several object-based models are in use; some of the more widely known ones are

- The entity-relationship model
- The object-oriented model
- The object-relational model

The entity-relationship model has gained acceptance in database design and is widely used in practice. The object-oriented model includes many of the concepts of the entity-relationship model, but represents executable code as well as data. The object-relational data model combines features of the object-oriented data model with the relational data model.

8.2.1 Entity-Relationship Model

The E-R data model represents the information in an enterprise as a set of basic objects called entities, and relationships among these objects. It facilitates database design by allowing the specification of an enterprise schema, which represents the overall logical structure of a database. The E-R data model is one of several semantic data models; that is, it attempts to represent the meaning of the data. In addition to providing an easily understood conceptual view of a database design, an E-R design can be converted algorithmically to a relational design that often requires little further modification.

8.2.1.1 Basics

There are three basic notions that the E-R data model employs: entity sets, relationship sets, and attributes. An entity is a “thing” or “object” in the real world that is distinguishable from all other objects. For example, each person in the universe is an entity.

Each entity is described by a collection of features, called attributes. For example, the attributes account-number and balance may describe one particular account in a bank, and they form attributes of the account entity set. Similarly, attributes customer-name, customer-street address, and customer-city may describe a customer entity.
The values for some attributes may uniquely identify an entity. For example, the attribute `customer-id` may be used to uniquely identify customers (since it may be possible to have two customers with the same name, street address, and city). A unique customer identifier must be assigned to each customer.

An entity may be concrete, such as a person or a book, or it may be abstract, such as a bank account, or a holiday, or a concept.

An *entity set* is a set of entities of the same type that share the same properties (attributes). The set of all persons working at a bank, for example, can be defined as the entity set `employee`, and the entity John Smith may be a member of the `employee` entity set. Similarly, the entity set `account` might represent the set of all accounts in a particular bank. An E-R design thus includes a collection of entity sets, each of which contains any number of entities of the same type.

Attributes are descriptive properties possessed by all members of an entity set. The designation of attributes expresses that the database stores similar information concerning each entity in an entity set; however, each entity has its own value for each attribute. Possible attributes of the `employee` entity set are `employee-name`, `employee-id`, and `employee-address`. Possible attributes of the `account` entity set are `account-number` and `account-balance`. For each attribute, there is a set of permitted values called the *domain* (or *value set*) of that attribute. The domain of the attribute `employee-name` might be the set of all text strings of a certain length. Similarly, the domain of attribute `account-number` might be the set of all positive integers.

Entities in an entity set are distinguished based on their attribute values. A set of attributes that suffices to distinguish all entities in an entity set is chosen, and called a primary key of the entity set. For the `employee` entity set, `employee-id` could serve as a primary key; the enterprise must ensure that no two people in the enterprise can have the same employee identifier.

A *relationship* is an association among several entities. Thus, an `employee` entity might be related by an `emp-dept` relationship to a `department` entity where that employee entity works. For example, there would be an `emp-dept` relationship between John Smith and the bank’s credit department if John Smith worked in that department. Just as all `employee` entities are grouped into an `employee` entity set, all `emp-dept` relationship instances are grouped into an `emp-dept` relationship set. A relationship set may also have descriptive attributes. For example, consider a relationship set `depositor` between the `customer` and the `account` entity sets. We could associate an attribute `last-access` to specify the date of the most recent access to the account. The relationship sets `emp-dept` and `depositor` are examples of a binary relationship set, that is, one that involves two entity sets. Most of the relationship sets in a database system are binary.

The overall logical structure of a database can be expressed graphically by an E-R diagram. There are multiple different notations used for E-R diagrams; we describe in the following one of the widely used notations, based on the one proposed by Chen [Che76]. Such a diagram consists of the following major components:

- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationship sets
- **Lines**, which link entity sets to relationship sets, and link attributes to both entity sets and relationship sets

An entity-relationship diagram for a portion of our simple banking example is shown in Figure 8.1. The primary key attributes (if any) of an entity set are shown underlined.

Composite attributes are attributes that can be divided into subparts (that is, other attributes). For example, an attribute `name` could be structured as a composite attribute consisting of `first-name`, `middle-initial`, and `last-name`. Using composite attributes in a design schema is a good choice if a user wishes to refer to an entire attribute on some occasions, and to only a component of the attribute on other occasions.

The attributes in our examples so far all have a single value for a particular entity. For instance, the `loan-number` attribute for a specific loan entity refers to only one loan number. Such attributes are said
to be *single valued*. There may be instances where an attribute has a set of values for a specific entity. Consider an *employee* entity set with the attribute *phone-number*. An employee may have zero, one, or several phone numbers, and hence this type of attribute is said to be *multivalued*.

Suppose that the *customer* entity set has an attribute *age* that indicates the customer’s age. If the *customer* entity set also has an attribute *date-of-birth*, we can calculate *age* from *date-of-birth* and the current date. Thus, *age* is a *derived attribute*. The value of a derived attribute is not stored, but is computed when required.

Figure 8.2 shows how composite, multivalued, and derived attributes can be represented in the E-R notation. Ellipses are used to represent composite attributes as well as their subparts, with lines connecting the ellipse representing the attribute to the ellipse representing its subparts. Multivalued attributes are represented using a double ellipse, while derived attributes are represented using a dashed ellipse.

Figure 8.3 shows an alternative E-R notation, based on the widely used UML class diagram notation, which allows entity sets and attributes to be represented in a more concise manner. The figure shows the *customer* entity set with the same attributes as described earlier, the *account* entity set with attributes *account-number* and *balance*, and the *depositor* relationship set with attribute *access-date*. For more details on this alternative E-R notation, see [SKS11].

Most of the relationship sets in a database system are *binary*, that is, they involve only two entity sets. Occasionally, however, relationship sets involve more than two entity sets. As an example, consider the entity sets *employee*, *branch*, and *job*. Examples of *job* entities could include manager, teller, auditor, and so on. Job entities may have the attributes *title* and *level*. The relationship set *works-on* among
A ternary relationship among Jones, Perryridge, and manager indicates that Jones acts as a manager at the Perryridge branch. Jones could also act as auditor at the Downtown branch, which would be represented by another relationship. Yet another relationship could be among Smith, Downtown, and teller, indicating Smith acts as a teller at the Downtown branch.

Consider, for example, a relationship set *works-for* relating the entity set *employee* with itself. Each employee entity is related to the entity representing the manager of the employee. One employee takes on the role of *worker*, whereas the second takes the role of *manager*. Roles can be depicted in E-R diagrams as shown in Figure 8.4.

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. Commonly used extended E-R features include specialization, generalization, higher- and lower-level entity sets, attribute inheritance, and aggregation. The notion of specialization and generalization are covered in the context of object-oriented data models in Section 8.2.2. A full explanation of the other features is beyond the scope of this chapter; we refer readers to [SKS11] for additional information.

### 8.2.1.2 Representing Data Constraints

In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is *mapping cardinalities*, which express the number of entities to which another entity can be associated via a relationship set. Therefore, relationships...
can be classified as many-to-many, many-to-one, or one-to-one. A many-to-many \textit{works-for} relationship between employee and department exists if a department may have one or more employees and an employee may work for one or more departments. A many-to-one \textit{works-for} relationship between employee and department exists if a department may have one or more employees but an employee must work for only department. A one-to-one \textit{works-for} relationship exists if a department were required to have exactly one employee, and an employee required to work for exactly one department. In an E-R diagram, an arrow is used to indicate the type of relationship as shown in Figure 8.5. The arrow indicates “exactly one.”

E-R diagrams also provide a way to indicate more complex constraints on the number of times each entity participates in relationships in a relationship set. An edge between an entity set and a binary relationship set can have an associated minimum and maximum cardinality, shown in the form $l..h$, where $l$ is the minimum and $h$ the maximum cardinality. A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value * indicates no limit.

For example, consider Figure 8.6. The edge between \textit{loan} and \textit{borrower} has a cardinality constraint of 1..1, meaning the minimum and the maximum cardinality are both 1. That is, each loan must have exactly one associated customer. The limit 0..* on the edge from \textit{customer} to \textit{borrower} indicates that a customer can have zero or more loans. Thus, the relationship \textit{borrower} is one to many from \textit{customer} to \textit{loan}.

It is easy to misinterpret the 0..* on the edge between \textit{customer} and \textit{borrower}, and think that the relationship \textit{borrower} is many to one from \textit{customer} to \textit{loan}—this is exactly the reverse of the correct interpretation. In this case, the earlier arrow notation is easier to understand.

If both edges from a binary relationship have a maximum value of 1, the relationship is one-to-one. If we had specified a cardinality limit of 1..* on the edge between \textit{customer} and \textit{borrower}, we would be saying that each customer must have at least one loan.

### 8.2.1.3 Use of E-R Model in Database Design

A high-level data model, such as the E-R model, serves the database designer by providing a conceptual framework in which to specify, in a systematic fashion, the data requirements of the database users and...
how the database will be structured to fulfill these requirements. The initial phase of database design, then, is to characterize fully the data needs of prospective database users. The outcome of this phase will be a specification of user requirements. The initial specification of user requirements may be based on interviews with the database users and the designer’s own analysis of the enterprise. The description that arises from this design phase serves as the basis for specifying the logical structure of the database.

By applying the concepts of the E-R model, the user requirements are translated into a conceptual schema of the database. The schema developed at this conceptual design phase provides a detailed overview of the enterprise. Stated in terms of the E-R model, the conceptual schema specifies all entity sets, relationship sets, attributes, and mapping constraints. The schema can be reviewed to confirm that all data requirements are indeed satisfied and are not in conflict with each other. The design can also be examined to remove any redundant features. The focus at this point is on describing the data and their relationships rather than on physical storage details.

A fully developed conceptual schema also indicates the functional requirements of the enterprise. In a specification of functional requirements, users describe the kinds of operations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. A review of the schema for meeting functional requirements can be made at the conceptual design stage.

The process of moving from a conceptual schema to the actual implementation of the database involves two final design phases. In the logical design phase, the high-level conceptual schema is mapped onto the implementation data model of the database management system (DBMS). In the next section, we discuss how a relational schema can be derived from an E-R design. The resulting DBMS-specific database schema is then used in the subsequent physical design phase, in which the physical features of the database are specified. These features include the form of file organization and the internal storage structures.

Because the E-R model is extremely useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema, a number of database design tools draw on E-R concepts. Further, the relative simplicity and pictorial clarity of the E-R diagramming technique may well account in large part for the widespread use of the E-R model.

### 8.2.1.4 Deriving a Relational Database Design from the E-R Model

A database that conforms to an E-R diagram can be represented by a collection of relational tables. For each entity set and each relationship set in the database, there is a unique table that is assigned the name of the corresponding entity set or relationship set. Each table has a number of columns that, again, have unique names. The conversion of database representation from an E-R diagram to a table format is the basis for deriving a relational database design.

The column headers of a table representing an entity set correspond to the attributes of the entity, and the primary key of the entity becomes the primary key of the relation. The column headers of a table representing a relationship set correspond to the primary key attributes of the participating entity sets, and the attributes of the relationship set. Rows in the table can be uniquely identified by the combined primary keys of the participating entity sets. For such a table, the primary keys of the participating entity sets are foreign keys of the table. The rows of the tables correspond to individual members of the entity or relationship set.

Tables 8.1 through 8.3 show instances of tables that correspond, respectively, to the customer and loan entity sets, the borrower relationship set, of Figure 8.6.

The full set of steps to convert an E-R design to a well-designed set of tables is presented in [SKS11].

### 8.2.2 Object-Oriented Model

The object-oriented data model is an adaptation of the object-oriented programming paradigm to database systems. The object-oriented approach to programming was first introduced by the language Simula 67, which was designed for programming simulations and advanced further by Smalltalk.
Today, C++ and Java have become the most widely known object-oriented programming languages, and most application development is done in these or other object-oriented languages. The model is based on the concept of encapsulating data, and code that operates on those data, in an object.

In this section, we present an overview of the concepts of object-oriented application development and how they are used in database systems.

8.2.2.1 Basics

Like the E-R model, the object-oriented model is based on a collection of objects. Entities, in the sense of the E-R model, are represented as objects with attribute values represented by instance variables within the object. The value stored in an instance variable may itself be an object. Objects can contain objects to an arbitrarily deep level of nesting. At the bottom of this hierarchy are objects such as integers, character strings, and other data types that are built into the object-oriented language and serve as the foundation of the object-oriented model. The set of built-in object types varies among languages.

Usually, the data within an object are private to that object and manipulated only by procedures (usually called methods) within that object. Therefore, the internal representation of an object’s data need not influence the implementation of any other object; all access to data within an object is via publicly accessible methods for that object. This encapsulation of code and data has proven useful in developing higher modular systems.

Unless the developer decides to make some parts of the object structure public, the only way in which one object can access the data of another object is by invoking a method of that other object. Thus, the call interface of the methods of an object defines its externally visible part. The internal parts of the object—the instance variables and method code—are not visible externally. The result is two levels of data abstraction.

To illustrate the concept, consider an object representing a bank account. Such an object contains instance variables account-number and account-balance, representing the account number and account balance. It contains a method pay-interest, which adds interest to the balance. Assume that the bank had been paying 4% interest on all accounts but now is changing its policy to pay 3% if the balance is less than $1000 or 4% if the balance is $1000 or greater. Under most data models, this would involve changing code in one or more application programs. Under the object-oriented model, the only change is made within the pay-interest method. The external interface to the object remains unchanged.

8.2.2.2 Classes

Objects that contain the same types of values and the same methods are grouped together into classes. A class may be viewed as a type definition for objects. This combination of data and code into a type definition is similar to the programming language concept of abstract data types. Thus, all employee objects may be grouped into an employee class. Classes themselves can be grouped into a hierarchy of classes; for example, the employee class and the customer classes may be grouped into a person class. The class person is a superclass of the employee and customer classes, since all objects of the employee and customer classes also belong to the person class. Superclasses are also called generalizations. Correspondingly, the employee and customer classes are subclasses of person; subclasses are also called specializations.

The hierarchy of classes allows sharing of common methods. It also allows several distinct views of objects: an employee, for example, may be viewed either in the role of person or employee, whichever is more appropriate.

8.2.2.3 Object-Oriented Database Programming Languages

There are two approaches to creating an object-oriented database language: the concepts of object orientation can be added to existing database languages, or existing object-oriented languages can be extended to deal with databases by adding concepts such as persistence and collections. Object-relational database systems take the former approach. Persistent programming languages follow the latter approach.
Any persistent programming language must have a mechanism for making objects persist; those not designated persistent are transient and vanish when the program terminates. Among the approaches to persistence are:

- **Persistence by class.** A class may be declared persistent, making all objects of such classes persistent. Objects of all other classes are transient. This approach is simple but lacks flexibility.
- **Persistence by creation.** At the time of object creation, the program may choose to declare the object persistent; otherwise it is transient.
- **Persistence by marking.** All objects are transient at the time they are created. However, at any point prior to program termination, an object can be marked as persistent. This approach improves on persistence by creation in that the decision on persistence can be deferred beyond creation time. It has the disadvantage of making it less clear to the developer which objects are in fact persistent.
- **Persistence by reachability.** Objects may be declared to be persistent. Not only are those objects persistent but also those objects reachable from those objects by a sequence of object references. This makes it easy to create a persistent data structure simply by making its root persistent, but it makes it more expensive for the system to determine which objects are persistent.

The Object Data Management Group (ODMG) developed standards for integrating persistence support into several programming languages such as C++ and Java. The Java Database Objects (JDO) standard provided a similar functionality, with tighter integration with Java. However, both these standards are no longer extensively used.

Instead, a new approach called **Object-Relational Mapping (ORM)** has found widespread acceptance. In this approach, a schema designer provides an object-oriented data model, along with a specification of how the data are to be mapped to a relational model. Library functions provided by ORM systems convert data from the relational representation used for storage to an object-oriented representation stored in memory. Updates to the in-memory object-oriented representation are translated back into updates on the underlying relational model. ORM systems provide an API to fetch objects based on a primary key; in addition, when the program traverses a reference from a previously fetched object, the referenced object is transparently fetched from the database in case it had not been fetched earlier. These features are ideally suited for online transaction processing systems. ORM systems also typically provide a query language for bulk data processing, with queries expressed in terms of the object-oriented data model. The Hibernate ORM system is currently the most widely used ORM implementation.

The alternative to persistent programming languages is to leave persistence to the database system and provide ways for standard programming language applications to access the database. The most popular of these are ODBC for C and C++ programs and JDBC for Java.

### 8.2.3 Object-Relational Data Models

Object-relational data models extend the relational data model by providing a richer type system that allows representation of instance variables and methods. In many implementations, stored procedures serve in place of methods.

The rigid structure of relations arose historically from the use of fixed length records, first on punched cards and later in early file systems. A more general model of relations would allow set-valued and relation-valued attributes. Such concepts were proposed in the 1980s [RK87:Design], but have only more recently appeared in practice. SQL has implemented this concept through **structured types**. Constructs are added to relational query languages such as SQL to deal with the added data types. The extended type systems allow attributes of tuples to have complex types including non-atomic values such as nested relations. Such extensions attempt to preserve the relational foundations, in particular the declarative access to data, while extending the modeling power.
As an example, consider a person entity whose attributes include name, which consists of first- and last-name) and address, which consists of street, city, and postal code. This can be expressed as

```sql
create type Name as
(first name varchar(20),
last name varchar(20))
final;
create type Address as
(street varchar(20),
city varchar(20),
zipcode varchar(9))
not final;
create table Person
(name Name,
address Address,
dateOf Birth date);
```

Observe that the Person table has two attributes, Name and Address that themselves have substructures. Such data types were specifically disallowed in early relation systems and, indeed, first normal form was defined to specify that relations have only atomic, indivisible values. Clearly, then, relations such as Person are non-first-normal-form relations.

In addition to allowing tuples within tuples as in our example, SQL allows attributes to hold arrays and multisets (sets in which values may occur multiple times). These features now give SQL the full modeling capability of earlier nested relational model proposals.

Data types, like Address in our example, that have the not final specification can have subtypes defined under them, allowing the definition of subclasses. Finally, data types may have methods defined on them that, such as methods in an object-oriented language, operate on an instance (object) of these types. In addition to object-oriented data-modeling features, SQL supports an imperative extension of the basic SQL query language, providing features such as for and while loops, if-then-else statements, procedures, and functions. SQL also supports references to objects; references must be to objects of a particular type, which are stored as tuples of a particular relation. Objects, however, do not have an independent existence, they correspond to tuples in a relation.

Relations are allowed to form an inheritance hierarchy; each tuple in a lower level relation must correspond to a unique tuple in a higher-level relation that represents information about the same object. Inheritance of relations provides a convenient way of modeling roles, where an object can acquire and relinquish roles over a period of time.

SQL support this via table inheritance; if \( r \) is a subtable of \( s \), the type of tuples of \( r \) must be a subtype of the type of tuples of \( s \). Every tuple present in \( r \) is implicitly (automatically) present in \( s \) as well. A query on \( s \) would find all tuples inserted directly to \( s \) as well as tuples inserted into \( r \); however, only the attributes of table \( s \) would be accessible, even for the \( r \) tuples. Thus subtables can be used to represent specialization/generalization hierarchies. However, while subtables in SQL can be used to represent disjoint specializations, where an object cannot belong to two different subclasses of a particular class, they cannot be used to represent the general case of overlapping specialization.

The combination of these features gives SQL a rich set of object-oriented features. In practice, however, the exact syntax of these constructs varies from the published standard. The reason for this is that, prior to object-relational features being added to the SQL standard, many commercial database systems incorporated their own proprietary object-relational extensions into their SQL implementations. While these all offer similar computational and modeling capabilities, the syntactic distinctions create problems when porting database designs across database systems. As a result, object-relational extensions of SQL are not widely used in practice.
Object-relational database systems (that is, database systems based on the object-relation model) provide a convenient migration path for users of relational databases who wish to use object-oriented features. Complex types such as nested relations are useful to model complex data in many applications. Object-relational systems combine complex data based on an extended relational model with object-oriented concepts such as object identity and inheritance.

8.3 Modeling Nested Structures

We have already seen how SQL has been extended over the years to allow representation of nested relational structures. There are other models in wide use for representing nested structures, including XML and JSON. These models are particularly useful when several applications need to share data, but do not all use the same database. In the section, we show how these models can be used in conjunction with a database system, but we do not go into the full feature set of these models.

8.3.1 XML

Extensible Markup Language (XML) was not originally conceived as a database technology. In fact, like the Hyper-Text Markup Language (HTML) on which the world wide web is based, XML has its roots in document management. However, unlike HTML, XML can represent database data, as well as many other kinds of structured data used in business applications.

The term *markup* in the context of documents refers to anything in a document that is not intended to be part of the printed output. For the family of markup languages that includes HTML and XML, the markup takes the form of *tags* enclosed in angle-brackets, <>. Tags are used in pairs, with <tag> and </tag> delimiting the beginning and the end of the portion of the document to which the tag refers.

Unlike HTML, XML does not prescribe the set of tags allowed, and the set may be specialized as needed. This feature is the key to XML’s major role in data representation and exchange, whereas HTML is used primarily for document formatting.

For example, in our running banking application, account and customer information can be represented as part of an XML document as in Figure 8.7. Observe the use of tags such as `account` and `account-number`. These tags provide context for each value and allow the semantics of the value to be identified. The contents between a start tag and its corresponding end tag is called an *element*.

Compared to storage of data in a database, the XML representation is inefficient, since tag names are repeated throughout the document. However, in spite of this disadvantage, an XML representation has significant advantages when it is used to exchange data:

- First, the presence of the tags makes the message self-documenting; that is, a schema need not be consulted to understand the meaning of the text.
- Second, the format of the document is not rigid. For example, if some sender adds additional information, such as a tag `last-accessed` noting the last date on which an account was accessed, the recipient of the XML data may simply ignore the tag. The ability to recognize and ignore unexpected tags allows the format of the data to evolve over time, without invalidating existing applications.
- Third, elements can be nested inside other elements, to any level of nesting. Figure 8.8 shows a representation of the bank information from Figure 8.7, but with `account` elements nested within `customer` elements.
- Finally, since the XML format is widely accepted, a wide variety of tools are available to assist in its processing, including browser software and database tools.

Just as SQL is the dominant language for querying relational data, XML is becoming the dominant format for data exchange.
In addition to elements, XML specifies the notion of an attribute. For example, the type of an account is represented in the following as an attribute named acct-type.

```
<account acct-type="checking">
  <account-number> A-101 </account-number>
  <branch-name> Downtown </branch-name>
  <balance> 500 </balance>
</account>
```

The attributes of an element appear as name = value pairs before the closing “>” of a tag. Attributes are strings, and do not contain markup. Furthermore, an attribute name can appear only once in a given tag, unlike subelements, which may be repeated.

Note that in a document construction context, the distinction between subelement and attribute is important—an attribute is implicitly text that does not appear in the printed or displayed document.
However, in database and data exchange applications of XML, this distinction is less relevant, and the choice of representing data as an attribute or a subelement is often arbitrary.

The document type definition (DTD) is an optional part of an XML document. The main purpose of a DTD is much like that of a schema: to constrain and type the information present in the document. However, the DTD does not in fact constrain types in the sense of basic types like integer or string. Instead, it only constrains the appearance of subelements and attributes within an element. The DTD is primarily a list of rules for what pattern of subelements appears within an element.

The XMLSchema language plays the same role as DTDs, but is more powerful in terms of the types and constraints it can specify. The XPath and XQuery languages are used to query XML data. The XQuery language can be thought of as an extension of SQL to handle data with nested structure, although its syntax is different from that of SQL.

Many database systems store XML data by mapping them to relations. Unlike in the case of E-R diagram to relation mappings, the XML to relation mappings are more complex and done transparently. Users can write queries directly in terms of the XML structure, using XML query languages.

### 8.3.2 JSON

JavaScript Object Notation (JSON), such as XML, is a text-based format for data exchange. Whereas XML is based on a pairing of tags and values, JSON is based on a nested collection of pairings of names and values. A simple name–value pair takes the form

\[
\text{n: v}
\]

where

- \text{n} is a name
- \text{v} is a value

FIGURE 8.8 Nested XML representation of bank information.

\[
\begin{align*}
\text{<bank-1>}
\text{<customer>}
\text{<customer-name> Johnson </customer-name>}
\text{<customer-street> Alma </customer-street>}
\text{<customer-city> Palo Alto </customer-city>}
\text{<account>}
\text{<account-number> A-101 </account-number>}
\text{<branch-name> Downtown </branch-name>}
\text{<balance> 500 </balance>}
\text{</account>}
\text{<account>}
\text{<account-number> A-201 </account-number>}
\text{<branch-name> Brighton </branch-name>}
\text{<balance> 900 </balance>}
\text{</account>}
\text{</customer>}
\text{<customer>}
\text{<customer-name> Hayes </customer-name>}
\text{<customer-street> Main </customer-street>}
\text{<customer-city> Harrison </customer-city>}
\text{<account>}
\text{<account-number> A-102 </account-number>}
\text{<branch-name> Perryridge </branch-name>}
\text{<balance> 400 </balance>}
\text{</account>}
\text{</customer>}
\text{</bank-1>}
\end{align*}
\]
The name \( n \) is a quoted string (in double quotes), and the value \( v \) is one of the JSON basic types (number, string, Boolean), or a composite type (array elements listed in square brackets separated by commas), or object (set of name–value pairs, listed within set brackets and separated by commas). The bank information we listed in XML in Figure 8.7 appears in JSON format in Figure 8.9. It is easy to note the similarity between the JSON and the XML representations of our sample bank data. JSON's representation is somewhat more compact.

Because JSON syntax is compatible with the popular scripting language JavaScript, JSON data are easily manipulated in JavaScript. JSON Schema permits specification of a schema for JSON data. Tools are available to test whether input JSON data are compatible with a given schema.

8.4 Temporal Data

Suppose we retain data in our bank showing not only the address of each customer but also all former addresses of which the bank is aware. We may then ask queries such as “Find all customers who lived in Princeton in 1981.” In this case, we may have multiple addresses for customers. Each address has an associated start and end date, indicating when the customer was resident at that address. A special value for the end date, e.g., null, or a value well into the future such as 9999-12-31, can be used to indicate that the customer is still resident at that address.

In general, temporal data are data that have an associated time interval during which they are valid. (There are other models of temporal data that distinguish between valid time and transaction time, the latter recording when a fact was recorded in the database. We ignore such details for simplicity.) We use the term snapshot of data to mean the value of the data at a particular point in time. Thus a snapshot of customer data gives the values of all attributes, such as address, of customers at a particular point in time.

Modeling temporal data is a challenging problem for several reasons. For example, suppose we have a customer entity with which we wish to associate a time-varying address. To add temporal information to an address, we would then have to create a multivalued attribute, each of whose values...
Data Models

is a composite value containing an address and a time interval. In addition to time-varying attribute values, entities may themselves have an associated valid time. For example, an account entity may have a valid time from the date it is opened to the date it is closed. Relationships too may have associated valid times. For example, the depositor relationship between a customer and an account may record when the customer became an owner of the account. We would thus have to add valid time intervals to attribute values, entities, and relationships. Adding such detail to an E-R diagram makes it very difficult to create and to comprehend. There have been several proposals to extend the E-R notation to specify in a simple manner that an attribute or relationship is time-varying, but there are no accepted standards.

Consider the constraint that a department has only one manager, which could be stated as “department functionally determines manager.” Such a constraint is expected to hold at a point in time, but surely cannot be expected to hold across time, since a department may have different managers at different points in time. Functional dependencies that hold at a particular point in time are called temporal functional dependencies; data in every snapshot of the database must satisfy the functional dependency.

8.5 Stream Data

The discussion of data models to this point is based on the assumption that there is a collection of data to model and store. Another view of data is that of a stream, in which data arrive continuously and in a sufficiently large volume that all of the data cannot be stored in the system. When new data arrive in the stream, the system uses that input to update a set of pre-defined computations. These could be simple aggregates (such as sum) or more complicated queries expressed in a SQL-like language designed for streams. These are sometimes referred to as continuous queries. A continuous query has a set of associated tuples at any point in time; and whenever the set of tuples changes, the system outputs either a tuple insertion or tuple deletion event. Thus, the output of a continuous query on a data stream is itself a data stream.

Because all prior data entering from the stream cannot be stored in the system, any computation done by the system is usually restricted to a limited amount of stream data. That restricted amount of data is called a window and stream query languages permit users to specify the maximum size of a window and how window boundaries are computed.

Stream data are becoming increasing common. Examples include telecommunication billing systems in which each billable event (call, text message, etc.) requires a price calculation and bill update, sensor networks in which a large volume of deployed sensors transmit data to a collection site, and ground stations for satellite data that need to collect and process data in real time as they arrive from the satellite.

The data in a stream could then be modeled, for example, using the relational data model, and one could view handling of data streams as a lower-level physical design issue. However, designers of streaming data systems have found it beneficial to model streaming data using a stream data model, which is an extension of the relational model, and to provide operators that convert data between the stream data model and the relational model.

The stream data model extends the relational model by associating a sequence number, typically a timestamp, with each tuple in the data stream. A window operation operates on a data stream, and creates multiple windows, with data in each window being modeled as a relation. There are also operators that take the result of a relational query using a window relation, and create an output stream whose timestamp/sequence number is defined by a timestamp/sequence number associated with the window. More recently, there have been proposals to model streaming data as having a beginning and an end timestamp; the timestamps of a particular tuple corresponding to the period in which the tuple is considered valid.
Further Readings

1. **The Relational model**

   The relational model was proposed by E. F. Codd of the IBM San Jose Research Laboratory in the late 1960s [Cod70]. Following Codd’s original paper, several research projects were formed with the goal of constructing practical relational database systems, including System R at the IBM San Jose Research Laboratory (Chamberlin et al. [CAB’81]), Ingres at the University of California at Berkeley (Stonebraker [Sto86b]), and Query-by-Example at the IBM T. J. Watson Research Center (Zloof [Zlo77]).

   General discussion of the relational data model appears in most database texts, including Date [Dat00], Ullman [Ull88], ElMasri and Navathe [EN11], Ramakrishnan and Gehrke [RG02], and Silberschatz et al. [SKS11]. Textbook descriptions of the SQL-92 language include Date and Darwen [DD97] and Melton and Simon [MS93].

   Textbook descriptions of the network and hierarchical models, which predated the relational model, can be found on the website [http://www.db-book.com](http://www.db-book.com) (this is the website of the text by Silberschatz et al. [SKS11]).

2. **The object-based models**

   a. **The entity-relationship model**: The entity-relationship data model was introduced by Chen [Che76]. Basic textbook discussions are offered by ElMasri and Navathe [EN11], Ramakrishnan and Gehrke [RG02], and Silberschatz et al. [SKS11]. Various data manipulation languages for the E-R model have been proposed, though none is in widespread commercial use. The concepts of generalization, specialization, and aggregation were introduced by Smith and Smith [SS77].

   b. **Object-oriented models**: Numerous object-oriented database systems were implemented as either products or research prototypes. Some of the commercial products include ObjectStore, Ontos, Orion, and Versant. More information on these may be found in overviews of object-oriented database research, such as Kim and Lochovsky [KL89], Zdonik and Maier [ZM90], and Dogac et al. [DOBS94]. The ODMG standard is described by Cattell [Cat00]. Information about the Hibernate object-relational mapping system can be found at [http://www.hibernate.org](http://www.hibernate.org).

   c. **Object-relational models**: The nested relational model was introduced in [Mak77] and [JS82]. Design and normalization issues are discussed in [OY87], [RK87], and [MNE96]. POSTGRES ([SR86] and [Sto86a]) was an early implementation of an object-relational system. Commercial databases such as IBM DB2, Informix, and Oracle support various object-relational features of SQL:1999. Refer to the user manuals of these systems for more details.

   Melton et al. [MSG01] and Melton [Mel02] provide descriptions of SQL:1999; [Mel02] emphasizes advanced features, such as the object-relational features, of SQL:1999. Date and Darwen [DD00] describes future directions for data models and database systems.

3. **Nested structures**

   The World Wide Web Consortium (W3C) acts as the standards body for web-related standards, including basic XML and all the XML-related languages such as XPath, XSLT, and XQuery. A large number of technical reports defining the XML-related standards are available at [http://www.w3c.org](http://www.w3c.org).

   A large number of books on XML are available in the market. These include [CSK01], [CRZ03], and [E+00].


4. **Stream data**

   Several systems have been built to handle data streams, including Aurora, StreamBase, Stanford STREAMS, and Oracle CEP. The Aurora data stream management system is described in [ACc’03].
[MWA’03] provides an overview of many issues in managing data streams, in the context of the Stanford STREAMS system. The CQL query language, described in [ABW06], was developed as part of the Stanford STREAMS project and is used in the Oracle Complex Event Processing (CEP) system. The StreamBase system (http://www.streambase.com) implements an SQL-based query language, which has some similarities to the CQL language, but differs in several respects. [JMS’08] is a step toward a standard for SQL on data streams.

5. Knowledge representation models

The area of knowledge representation, which has been widely studied in the AI community, aims at modeling not only data but also knowledge in the form of rules. A variety of knowledge representation models and languages have been proposed; Brachman and Levesque [BL04] provide detailed coverage of the area of knowledge representation and reasoning. We note that one of the knowledge representation models, called Resource Description Framework, or RDF for short, has been widely used in recent years for modeling semistructured data. RDF is an example of a graph-based data model, where nodes represent concepts, entities, and values, while labeled edges specify relationships between the concepts/entities/values. See http://www.w3.org/RDF for more information on RDF.

Glossary

Attribute: (1) A descriptive feature of an entity or relationship in the entity-relationship model; (2) the name of a column header in a table, or, in relational-model terminology, the name of a domain used to define a relation.

Class: A set of objects in the object-oriented model that contain the same types of values and the same methods; also, a type definition for objects.

Data model: A data model is a collection of conceptual tools for describing the real-world entities to be modeled in the database and the relationships among these entities.

Element: The contents between a start tag and its corresponding end tag in an XML document.

Entity: A distinguishable item in the real-world enterprise being modeled by a database schema.

Foreign key: A set of attributes in a relation schema whose value identifies a unique tuple in another relational schema.

Functional dependency: A rule stating that given values for some set of attributes, the value for some other set of attributes is uniquely determined. X functionally determines Y if whenever two tuples in a relation have the same value on X, they must also have the same value on Y.

Generalization: A super-class; an entity set that contains all the members one or more specialized entity sets.

Instance variable: Attribute values within objects.

Key: (1) A set of attributes in the entity relationship model that serves as a unique identifier for entities. Also known as super-key. (2) a set of attributes in a relation schema that functionally determines the entire schema. (3) candidate key—a minimal key; that is, a super-key for which no proper subset is a super-key. (4) primary key—a candidate key chosen as the primary means of identifying/accessing an entity set, relationship set, or relation.

Message: The means by which an object invokes a method in another object.

Method: Procedures within an object that operate on the instance variables of the object and/or send messages to other objects.

Normal form: A set of desirable properties of a schema. Examples include the Boyce–Codd normal form and the third normal form.

Object: Data and behavior (methods) representing an entity.

Persistence: The ability of information to survive (persist) despite failures of all kinds, including crashes of programs, operating systems, networks, and hardware.

Relation: (1) A subset of a Cartesian product of domains. (2) informally, a table.
**Relation schema:** A type definition for relations consisting of attribute names and a specification of the corresponding domains.

**Relational algebra:** An algebra on relations; consists of a set of operations, each of which takes as input one or more relations and returns a relation, and a set of rules for combining operations to create expressions.

**Relationship:** An association among several entities.

**Subclass:** A class that lies below some other class (a superclass) in a class inheritance hierarchy; a class that contains a subset of the objects in a superclass.

**Subtable:** A table such that (a) its tuples are of a type that is a subtype of the type of tuples of another table (the supertable), and (b) each tuple in the subtable has a corresponding tuple in the supertable.

**Specialization:** A subclass; an entity set that contains a subset of entities of another entity set.

### References


Data Models


