Database System Concepts

By Alfred V. Aho et all

First Edition

CONTENTS

CHAPTER 1

INTRODUCTION

A **database-management system** (**DBMS**) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise.

1 Database-System Applications

Broadly speaking, there are two modes in which databases are used.

- The first mode is to support **online transaction processing**, where a large number of users use the database, with each user retrieving relatively small amounts of data, and performing small updates.
- The second mode is to support **data analytics**, that is, the processing of data to draw conclusions, and infer rules or decision procedures, which are then userd to drive business decisions.

2 Purpose of Database Systems

One way to keep the information on a computer is to store it in operating-system files.

This typical **file-processing system** is supported by a conventional operating system.

Keeping organizational information in a file-processing system has a number of major disadvantages:

- **Data redundancy and inconsistency**. Since different programmers create the files and application programs over a long period, the various files are likely to have different structures, and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). In addition, it may lead to **data inconsistency**; that is, the various copies of the same data may no longer agree.
- **Difficulty in accessing data**.
- **Data isolation**.
- **Integrity problems**. The data values stored in the database must satisfy certain types of **consistency constraints**.

- **Atomicity problems**.
- **Concurrent-access anomalies**.
- **Security problems**.

3 View of Data

3.1 Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

The data models can be classified into four different categories:

- **Relational Model**. The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as **relations**.
- **Entity-Relationship Model**.
- **Semi-structured Data Model**.
- **Object-Based Data Model**.

3.2 Data Abstraction

Since many database-system users are not computer trained, developers hide the complexity from users through several levels of **data abstraction**, to simplify users' interactions with the system:

- **Physical level**.
- **Logical level**. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as **physical data independence**.
- **View level**.

3.3 Instances and Schemas

The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**.

The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

4 Database Languages

A database system provides a **data-definition langauge** (**DDL**) to specify the database schema and a **data-manipulation language** (**DML**) to express database queries and updates.

4.1 Data-Definition Language

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language.

In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, database systems implement only those integrity constraints that can be tested with minimal overhead:

- **Domain Constraints**.
- **Referential Integrity**.
- **Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data.

The output of the DDL is placed in the **data dictionary**, which contains **metadata** – that is, data about data.

4.2 Data-Manipulation Language

A **data-manipulation language** (**DML**) is a language that enables users to access or manipulate as organized by the appropriate data model. There are basically two types of data-manipulation language:

- **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.
- **Declarative DMLs** (also referred to as **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**.

4.3 Database Access from Application Programs

Non-procedural query languages are not as powerful as a universal Turing machine; that is, there are some computations that are possible using a general-purpose programming language but are not possible using SQL. SQL also does not support actions such as input from users, output to displays, or communication over the network. Such computations and actions must be written in a *host* language. **Application programs** are programs that are used to interact with the database in this fashion.

5 Database Design

A high-level data model provides the database designer with a conceptual framework in which to specify 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 the prospective database users.

Next, the deisgner chooses a data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase provides a detailed overview of the enterprise.

In terms of the relational model, the conceptual-design process involves decisions on *what* attributes we want to capture in the database and *how to group* these attributes to form the various tables. The "how" part is mainly a computer-science problem. There are principally two ways to tackle the problem. The first one is to use the entity-relationship model; the other is to employ a set of algorithms (collectively known as **normalization**) that takes as input the set of all attributes and generates a set of tables.

In a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data.

In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified.

6 Database Engine

The functional components of a database system can be broadly divided into the storage manager, the **query processor** components, and the transaction management component.

6.1 Storage Manager

The **storage manager** is the component of a database system that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

The storage manager components include:

- **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
- **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicts.
- **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory.

The storage manager implements several data structures as part of the physical system implementation:

- **Data files**, which store the database itself.
- **Data dictionary**, which stores metadata about the structure of the database, in particular the schema of the database.
- **Indices**, which can provide fast accesss to data items.

6.2 The Query Processor

The query processor components include:

- **DDL interpreter**, which interprets DDL statements and records the definitions in the data dictionary.
- **DML compiler**, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query-evaluation engine understands.

  A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**; that is, it picks the lowest cost evaluation plan from among the alternatives.
- **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.

6.3 Transaction Management

Often, several operations on the database form a single logical unit of work. An example is a funds transfer in which one account A is debited and another account B is credited. Clearly, it is essential that either both the credit and debit occur, or that neither occur. This all-or-none requirement is called **atomicity**. In addition, it is essential that the execution of the funds transfer preserves the consistency of the database. This correctness requirement is called **consistency**. Finally, after the successful execution of a funds transfer, the new values of the balances of accounts A and B must persist, despite the possibility of system failure. This persistence requirement is called **durability**.

A **transaction** is a collection of operations that performs a single logical function in a database application.

Ensuring the atomicity and durability properties is the responsibility of the database system itself – specifically, of the **recovery manager**. If we are to ensure the atomicity property, a failed transaction must have no effect on the state of the database. Thus, the database must be restored to the state in which it was before the transaction in question started executing. The database system must therefore perform **failure recovery**, that is, it must detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

It is the responsibility of the **concurrency-control manager** to control the interaction among the concurrent transactions, to ensure the consistency of the database. The **transaction manager** consists of the concurrency-control manager and the recovery manager.

7 Database and Application Architecture

Earlier-generation database applications used a **two-tier architecture**, where the application resides at the client machine, and invokes database system functionality at the server machine through query language statements.

In contrast, modern database applications use a **three-tier architecture**, where the client machine acts as merely a front end and does not contain any direct database calls; web browsers and mobile applications are the most commonly used application clients today. The front end communicates with an **application server**. The application server, in turn, communicates with a database system to access data. The **business logic** of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients.

8 Database Users and Administrators

8.1 Database Administrators

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrators** (**DBA**).

9 History of Database Systems

Techniques for data storage and processing have evolved over the years:

- **1950s and early 1960s**: Magnetic tapes were developed for data storage.
- **Late 1960s and early 1970s**: Widespread use of hard disks in the late 1960s changed the scenario for data processing greatly, since hard disks allowed direct access to data.
- **Late 1970s and 1980s**: Although academically interesting, the relational model was not used in practice initially because of its perceived performance disadvantages; relational databases could not match the performance of existing network and hierarchical databases.
- **1990s**: The SQL language was designed primarily for decision support applications, which are query-intensive, yet the mainstay of databases in the 1980s was transaction-processing applications, which are update-intensive.
- **2000s**: In the latter part of the decade, the use of data analytics and **data mining** in enterprises became ubiquitous.
- **2010s**: The limitations of NoSQL systems were found acceptable by many applications, in return for the benefits they provided.

CHAPTER 2

INTRODUCTION TO THE RELATIONAL MODEL

1 Structure of Relational Databases

A relational database consists of a collection of **tables**, each of which is assigned a unique name.

In mathematical terminology, a *tuple* is simply a sequence (or list) of values. A relationship between n values is represented mathematically by an *n-tuple* of values, that is, a tuple with n values, which corresponds to a row in a table.

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.

We use the term **relation instance** to refer to a specific instance of a relation, that is, containing a specific set of rows.

For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute.

A domain is **atomic** if elements of the domain are considered to be indivisible units.

The **null value** is a special value that signifies that the value is unknown or does not exist.

2 Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **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, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

3 Keys

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation.

We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal meansof identifying tuples within a relation.

The designation of a key represents a constraint in the real-world enterprise being modeled. Thus, primary keys are also referred to as **primary key constraints**.

A **foreign-key constraint** from attribute(s) A of relation r_1 to the primary-key B of relation r_2 states that on any database instance, the value of A for each tuple in r_1 must also be the value of B for some tuple in r_2. Attribute set A is called a **foreign key** from r_1, referencing r_2. The relation r_1 is also called the **referencing relation** of the foreign-key constraint, and r_2 is called the **referenced relation**.

In general, a **referential integrity constraint** requires that the values appearing in specified attributes of any typle in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

4 Schema Diagrams

A database schema, along with primary key and foreign-key constraints, can be depicted by **schema diagrams**.

5 Relational Query Languages

A **query language** is a language in which a user requests information from the database. In an **imperative query language**, the user instructs the system to perform a specific sequence of operations on the database to compute the desired result; such languages usually have a notion of state variables, which are updated in the course of the computation.

In a **functional query language**, the computation is expressed as the evaluation of functions that may operate on data in the database or on the results of other functions; functions are side-effect free, and they do not update the program state.[1] In a **declarative query language**, the user describes the desired information without giving a specific sequence of steps or function calls for obtaining that information; the desired information is typically described using some form of mathematical logic.

There are a number of "pure" query languages.

- The *relational algebra* is a functional query language.
- The tuple relational calculus and domain relational calculus are declarative.

6 The Relational Algebra

The relational algebra consists of a set of operations that take one or two relations as input and produce a new relation as their result.

Some of these operations are called *unary* operations because they operate on one relation. The other operations operate on pairs of relations and are, therefore, called *binary* operations.

[1]The term *procedure language* include functional languages; however, the term is also widely used to refer to imperative languages.

6.1 The Select Operation

The **select** operation selects tuples that satisfy a given predicate.

6.2 The Project Operation

Suppose we want to list all instructors' *ID*, *name*, and *salary*, but we do not care about the *dept_name*. The **project** operation allows us to produce this relation.

6.3 Composition of Relational Operations

In general, since the result of a relational-algebra operation is of the same type (relation) as its inputs, relational-algebra operations can be composed together into a **relational-algebra expression**.

6.4 The Cartesian-Product Operation

The **Cartesian-product** operation, denoted by a cross ($\times$), allows us to combine information from any two relations.

6.5 The Join Operation

Consider relations $r(R)$ and $s(S)$, and let θ be a predicate on attributes in the schema $R \cup S$. The **join** operation $r \bowtie_\theta s$ is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

6.6 Set Operations

In general, for a union operation to make sense:

1. We must ensure that the input relations to the union operation have the same number of attributes; the number of attributes of a relation is referred to as its **arity**.
2. When the attributes have associated types, the types of the ith attributes of both input relations must be the same, for each i.

Such relations are referred to as **compatible** relations.

The **intersection** operation, denote by $\cap$, allows us to find tuples that are in both the input relations.

The **set-difference** operation, denoted by $-$, allows us to find tuples that are in one relation but are not in another.

6.7 The Assignment Operation

The **assignment** operation, denoted by $\leftarrow$, works like assignment in a programming language.

6.8 The Rename Operation

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

CHAPTER 3

INTRODUCTION TO SQL

1 Overview of the SQL Query Language

The SQL language has several parts:

- **Data-definition language** (**DDL**).
- **Data-manipulation language** (**DML**).
- **Integrity**.
- **View definition**.
- **Transaction control**.
- **Embedded SQL** and **dynamic SQL**.
- **Authorization**.

2 SQL Data Definition

2.1 Basic Types

The SQL standard supports a variety of built-in types, including:

- **char**(n): A fixed length character string with user-specified length n.
- **varchar**(n): A variable-length character string with user-specified maximum length n.
- **int**: An integer (a finite subset of the integers that is machine dependent).
- **smallint**: A small integer (a machine-dependent subset of the integer type).
- **numeric**(p, d): A fixed-point number with user-specified precision.
- **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.
- **float**(n): A floating-point number with precision of at least n digits.

Each type may include a special value called the **null** value.

2.2 Basic Schema Definition

SQL supports a number of different integrity constraints. In this section, we discuss only a few of them:

- **primary key** $(A_{j_1}, A_{j_2}, \ldots, A_{j_m})$: The **primary-key** specification says that attributes $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ form the primary key for the relation.
- **foreign key** $(A_{k_1}, A_{k_2}, \ldots, A_{k_n})$ **references** s: The **foreign key** specification says that the values of attributes $(A_{k_1}, A_{k_2}, \ldots, A_{k_n})$ for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation s.
- **not null**: The **not null** constraint on an attribute specifies that the null value is not allowed for that attribute; in other wrods, the constraint excludes the null value from the domain of that attribute.

3 Basic Structure of SQL Queries

The basic structure of an SQL query consists of three clauses: **select**, **from**, and **where**.

4 Additional Basic Operations

4.1 The Rename Operation

Consider the query:

select *name*, *course_id*
from *instructor*, *teaches*
where *instructor.ID* = *teaches.ID*

The result of this query is a relation with the following attributes:

name, *course_id*

The names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause.

We cannot, however, always derive names in this way, for several reasons: First, two relations in the **from** clause may have attributes with the same name, in which case an attribute name is duplicated in the result. Second, if we use an arithmetic expression in the **select** clause, the resultant attribute does not have a name. Third, even if an attribute name can be derived from the base relations, we may want to change the attribute name in the result. Hence, SQL provides a way of renaming the attributes of a result relation. It uses the **as clause**, taking the form:

old-name **as** *new-name*

The **as** clause can appear in both the **select** and **from** clauses.[1]

[1] Early versions of SQL did not include the keyword **as**. As a result, some implementations of SQL do not permit the keyword **as** in the **from** clause.

SQL AND MULTISET RELATIONAL ALGEBRA - PART 1

The SQL standard defines how many copies of each tuple are there in the output of a query, which depends, in turn, on how many copies of tuples are present in the input relations.

To model this behavior of SQL, a version of relational algebra, called the **multiset relational algebra**, is defined to work on multisets: sets that may contain duplicates.

An identifier that is used to rename a relation is referred to as a **correlation name** in the SQL standard, but it is also commonly referred to as a **table alias**, or a **correlation variable**, or a **tuple variable**.

4.2 Ordering the Display of Tuples

The **order by** clause causes the tuples in the result of a query to appear in sorted order.

5 Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the mathematical set operations $\cup$, $\cap$, and $-$.

6 Null Values

Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.

7 Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five standard built-in aggregate functions:

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

7.1 Aggregation with Grouping

Tuples with the same value on all attributes in the **group by** clause are placed in one group.

7.2 The Having Clause

At times, it is useful to state a condition that applies to groups rather than to tuples. To express such a query, we use the **having** clause of SQL.

8 Nested Subqueries

8.1 Set Comparison

The phrase "greater than at least one" is represented in SQL by > **some**.

The construct > **all** corresponds to the phrase "greater than all."

8.2 Test for Empty Relations

The **exists** construct returns the value **true** if the argument subquery is nonempty.

A subquery that uses a correlation name from an outer query is called a **correlated subquery**.

8.3 Test for the Absence of Duplicate Tuples

The **unique** construct returns the value **true** if the argument subquery contains no duplicate tuples.

8.4 Subqueries in the From Clause

We note that nested subqueries in the **from** clause cannot use correlation variables from other relations in the same **from** clause. However, the SQL standard, starting with SQL:2003, allows a subquery in the **from** clause that is prefixed by the **lateral** keyword to access attributes of preceding tables or subqueries in the same **from** clause.

8.5 The With Clause

The **with** clause provides a way of defining a temporary relation whose definition is available only to teh query in which the **with** clause occurs.

8.6 Scalar Subqueries

SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called **scalar subqueries**.

9 Modification of the Database

9.1 Deletion

A **delete** request is expressed in much the same way as a query.

9.2 Insertion

The simplest **insert** statement is a request to insert one tuple.

9.3 Updates

In general, the **where** clause of the **update** statement may contain any construct legal in the **where** clause of the **select** statement (including nested **select**s).

CHAPTER 4

INTERMEDIATE SQL

1 Join Expressions

1.1 The Natural Join

SQL supports an operation called the *natural join.* In fact, SQL supports several other ways in which information from two or more relations can be **joined** together.

The **natural join** operation operates on two relations and produces a relation as the result.

1.2 Outer Joins

The **outer-join** operation works in a manner similar to the join operations we have already studied, but it preserves those tuples that would be lost in a join by creating tuples in the result containing null values.

There are three forms of outer join:

- The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation.
- The **right outer join** preserves tuples only in the relation named after (to the right of) the **right outer join** operation.
- The **full outer join** preserves tuples in both relations.

In contrast, the join operations we studied earlier that do not preserve nonmatched tuples are called **inner-join** operations, to distinguish them from the outer-join operations.

2 Views

The **with** clause allows us to assign a name to a subquery for use as often as desired. but in one particular query only. Here, we present a way to extend this concept beyond a single query by defining a **view**.

2.1 Materialized Views

Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views**.

The process of keeping the materialized view up-to-date is called **materialized view maintenance** (or often, just **view maintenance**).

2.2 Update of a View

In general, an SQL view is said to be **updatable** (i.e., inserts, updates, or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:

- The **from** clause has only one database relation.
- The **select** clause contains only attribute names of the relation and does not have any expressions, aggregates, or **distinct** specification.
- Any attribute not listed in the **select** clause can be set to *null*; that is, it does not have a **not null** constraint and is not part of a primary key.
- The query does not have a **group by** or **having** clause.

3 Transactions

A **transaction** consists of a sequence of query and/or update statements. One of the following SQL statements must end the transaction:

- **Commit work** commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database.
- **Rollback work** causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction.

By either committing the actions of a transaction after all its steps are completed, or rolling back all its actions in case the transaction could not complete all its actions successfully, the database provides an abstraction of a transaction as being **atomic**, that is, indivisible.

4 Integrity Constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency.

4.1 Not Null Constraint

The **not null** constraint prohibits the insertion of a null value for the attribute, and is an example of a **domain constraint**.

4.2 Unique Constraint

SQL also supports an integrity constraint:

$$\textbf{unique}(A_{j_1}, A_{j_2}, \ldots, A_{j_m})$$

The **unique** specification says that attribute $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ form a superkey; that is, no two tuples in the relation can be equal on all the listed attributes.

4.3 The Check Clause

A common use of the **check** clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system.

4.4 Referential Integrity

Consider this definition of an integrity constraint on the relation *course*:

create table *course*
(...
foreign key (*dept_name*) **references** *department*
on delete cascade
on update cascade,
...);

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "**cascades**" to the *course* relation, deleting the tuple that refers to the department that was deleted.

4.5 Complex Check Conditions and Assertions

An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.

5 SQL Data Types and Schemas

5.1 Data and Time Types in SQL

In addition to the basic data types we introduced in Section 3.2, the SQL standard supports several data types relating to dates and times:

- **date**: A calendar data containing a (four-digit) year, month, and day of the month.
- **time**: The time of day, in hours, minutes, and seconds.
- **timestamp**: A combination of **date** and **time**.

5.2 Type Conversion and Formatting Functions

Although systems perform some data type **conversions** automatically, others need to be requested explicitly.

5.3 Default Values

SQL allows a **default** value to be specified for an attribute as illustrated by following **create table** statement:

```
create table student
    (ID             varchar(5),
    name            varchar(20) not null,
    dept_name       varchar(20),
    tot_cred        numeric(3,0) default 0,
    primary key (ID));
```

The following **insert** statement illustrates how an insertion can omit the value for the *tot_cred* attribute.

```
insert into student(ID, name, dept_name)
    values('12789','Newman','Comp. Sci.');
```

5.4 Large-Object Types

Many database applications need to store attributes whose domain consists of large data items. SQL, therefore, provides **large-object data types** for character data (**clob**) and binary data (**blob**).

5.5 User-Defined Types

SQL supports two forms of **user-defined data types**. The first form, which we cover here, is called **distinct types**. The other form, called **structured data types**, allows the creation of complex data types with nested record structures, arrays, and multisets.

Even before user-defined types were added to SQL (in SQL:1999), SQL had a similar but subtly different notion of **domain** (introduced in SQL-92), which can add integrity constraints to an underlying type.