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 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**.

6.3 The Cartesian-Product Operation

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

6.4 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:

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

6.5 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.

6.6 The Assignment Operation

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

6.7 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful 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

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 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 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**).

5.4 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.

5.5 Schemas, Catalogs, and Environments

Contemporary database systems provide a three-level hierarchy for naming relations. The top level of the hierarchy consists of **catalogs**, each of which can contain **schemas**.

6 Index Definition in SQL

An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.

7 Authorization

Authorizations on data include:

- Authorization to read data.
- Authorization to insert new data.
- Authorization to update data.
- Authorization to delete data.

Each of these types of authorizations is called a **privilege**.

7.1 Granting and Revoking of Privileges

The SQL standard includes the **privileges** **select**, **insert**, **update**, and **delete**.

The **grant** statement is used to confer authorization.

To revoke an authorization, we use the **revoke** statement.

7.2 Roles

Consider the real-world roles of various people in a university. Each instructor must have the same types of authorizations on the same set of relations. Whenever a new instructor is appointed, she will have to be given all these authorizations individually.

A better approach would be to specify the authorizations that every instructor is to be given, and to identify separately which database users are instructors.

The notion of **roles** captures this concept.

7.3 Authorization on Views

The **execute** privilege can be granted on a function or procedure, enabling a user to execute the function or procedure.

7.4 Transfer of Privileges

The passing of a specific authorization from one user to another can be represented by an **authorization graph**.

7.5 Row-Level Authorization

VPD provides authorization at the level of specific tuples, or rows, of a relation, and is therefore said to be a **row-level authorization** mechanism.

CHAPTER 5

ADVANCED SQL

1 Accessing SQL from a Programming Language

1.1 JDBC

The **JDBC** standard defines an **application program interface (API)** that Java programs can use to connect to database servers.

Prepared Statements

A technique called **SQL injection** can be used by malicious hackers to steal data or damage the database.

Other Features

JDBC provides a number of other features, such as **updatable result sets**.

1.2 ODBC

The **Open Database Connectivity** (**ODBC**) standard defines an API that applications can use to open a connection with a database, send queries and updates, and get back results.

1.3 Embedded SQL

EMBEDDED DATABASES

Some applications use a database that exists entirely within the application. In such cases, one may use an **embedded database** and use one of several packages that implement an SQL database accessible from within a programming language.

2 Functions and Procedures

2.1 Declaring and Invoking SQL Functions and Procedures

The SQL standard supports functions that can retrun tables as results; such functions are called **table functions**.

2.2 Language Constructs for Procedures and Functions

SQL supports constructs that give it almost all the power of a general-purpose programming language. The part of the SQL standard that deals with these constructs is called the **Persistent Storage Module (PSM)**.

The SQL procedure language also supports the signaling of **exception conditions** and declaring of **handlers** that can handle the exception.

2.3 External Language Routines

Functions defined in a programming language and compiled outside the database system may be loaded and executed with the database-system code. Database systems that are concerned about security may execute such code as part of a separate process, communicate the parameter values to it, and fetch results back via interprocess communication.

If the code is written in a "safe" language, there is another possibility: executing the code in a **sandbox** within the database query execution process itself.

3 Triggers

A **trigger** is a statement that the system executes automatically as a side effect of a modification to the database.

4 Recursive Queries

There are numerous applications that require computation of similar transitive closures on **hierarchies**.

4.1 Transitive Closure Using Iteration

Note that SQL allows the creation of temporary tables using the command **create temporary table**; such tables are available only within the transaction executing the query and are dropped when the transaction finishes.

4.2 Recursion in SQL

Any recursive view must be defined as the union of two subqueries: a **base query** that is nonrecursive and a **recursive query** that uses the recursive view.

There are some restrictions on the recursive query in a recursive view; specifically, the query must be **monotonic**, that is, its result on a view relation instance V_1 must be a superset of its result on a view relation instance V_2 if V_1 is a superset of V_2.

5 Advanced Aggregation Features

5.1 Pivoting

In general, a cross-tab is a table derived from a relation (say, R), where values for some attribute of relation R (say, A) become attribute names in the result; the attribute A is the **pivot** attribute.

CHAPTER 6

DATABASE DESIGN USING THE E-R MODEL

1 Overview of the Design Process

1.1 Design Phases

A high-level data 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 a database structure that fulfills these requirements.

- The initial phase of database design is to characterize fully the data needs of the prospective database users.
- Next, the designer 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 a **specification of functional requirements**, users describe the kinds of operations (or transactions) that will be performed on the data.
- The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.
  - 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.
  - Finally, 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.

2 The Entity-Relationship Model

The **entity-relationship** (**E-R**) **data model** was developed to facilitate database design by allowing specification of an *enterprise schema* that represents the overall logical structure of a database.

An **E-R diagram** can express the overall logical structure of a database graphically.

2.1 Entity Sets

An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects.

An **entity set** is a set of entities of the same type that share the same properties, or attributes.

We use the term **extension** of the entity set to refer to the actual collection of entities belonging to the entity set.

An entity is represented by a set of **attributes**.

Each entity has a **value** for each of its attributes.

2.2 Relationship Sets

A **relationship** is an association among several entities. A **relationship set** is a set of relationships of the same type.

A **relationship instance** in an E-R schema represents an association between the named entities in the real-world enterprise that is being modeled.

Formally, a **relationship set** is a mathematical relation on $n \geq 2$ (possibly nondistinct) entity sets. If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set R is a subset of

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

where $(e_1, e_2, \ldots, e_n)$ is a relationship instance.

The association between entity sets is referred to as participation; i.e., the entity sets $E_1, E_2, \ldots, E_n$ **participate** in relationship set R.

The function that an entity plays in a relationship is called that entity's **role**. Since entity sets participating in a relationship set are generally distinct, roles are implicit and are not usually specified. However, they are useful when the meaning of a relationship needs clarification. Such is the case when the entity sets of a relationship set are not distinct; that is, the same entity set participates in a relationship set more than once, in different roles. In this type of relationship set, sometimes called a **recursive** relationship set, explicit role names are necessary to specify how an entity participates in a relationship instance.

A relationship may also have attributes called **descriptive attributes**.

The number of entity sets that participate in a relationship set is the **degree of the relationship set**. A binary relationship set is of degree 2; a **ternary relationship set** is of degree 3.

3 Complex Attributes

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

An attribute, as used in the E-R model, can be characterized by the following attribute types.

- **Simple** and **composite** attributes.
- **Single-valued** and **multivalued** attributes. There may be instances where an attribute has a set of values for a specific entity. This type of attribute is said to be **multivalued**.

- **Derived attributes**.

An attribute takes a **null** value when an entity does not have a value for it.

4 Mapping Cardinalities

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

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

- **One-to-one**.
- **One-to-many**.
- **Many-to-one**.
- **Many-to-many**.

The participation of an entity set E in a relationship set R is said to be **total** if every entity in E must participate in at least one relationship in R. If it is possible that some entities in E do not participate in relationships in R, the participation of entity set E in relationship R is said to be **partial**.

5 Primary Key

5.1 Weak Entity Sets

A **weak entity set** is one whose existence is dependent on another entity set, called its **identifying entity set**; instead of associating a primary key with a weak entity, we use the primary key of the identifying entity, along with extra attributes, called **discriminator attributes** to uniquely identify a weak entity. An entity set that is not a weak entity set is termed a **strong entity set**.

Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.

6 Extended E-R Features

6.1 Specialization

The process of designating subgroupings within an entity set is called **specialization**.

The way we depict specialization in an E-R diagram depends on whether an entity may belong to multiple specialized entity sets or if it must belong to at most one specialized entity set. The former case (multiple sets permitted) is called **overlapping specialization**, while the latter case (at most one permitted) is called **disjoint specialization**. The specialization relationship may also be referred to as a **superclass-subclass** relationship.

6.2 Generalization

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

Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively.

6.3 Attribute Inheritance

A crucial property of the higher- and lower-level entities created by specialization and generalization is **attribute inheritance**. The attributes of the higher-level entity sets are said to be **inherited** by the lower-level entity sets.

If an entity set is a lower-level entity set in more than one ISA relationship, then the entity set has **multiple inheritance**, and the resulting structure is said to be a *lattice*.

6.4 Constraints on Specializations

One type of constraint on specialization specifies whether a specialization is disjoint or overlapping. Another type of constraint on a specialization/generalization is a **completeness constraint**, which specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/specialization. This constraint may be one of the following:

- **Total specialization** or **generalization**.
- **Partial specialization** or **generalization**.

6.5 Aggregation

Aggregation is an abstraction through which relationships are treated as higher-level entities.

6.6 The Unified Modeling Language UML

The **Unified Modeling Language** (**UML**) is a standard developed under the auspices of the **Object Management Group** (**OMG**) for creating specifications of various components of a software system.

In UML terminology, relationship sets are referred to as **associations**; we shall refer to them as relationship sets for consistency with E-R terminology.

CHAPTER 7

RELATIONAL DATABASE DESIGN

This chapter will be filled later.

CHAPTER 8

COMPLEX DATA TYPES

This chapter will be filled later.

CHAPTER 9

APPLICATION DEVELOPMENT

This chapter will be filled later.

CHAPTER 10

BIG DATA

This chapter will be filled later.

CHAPTER 11

DATA ANALYTICS

This chapter will be filled later.

CHAPTER 12

PHYSICAL STORAGE SYSTEMS

1 Overview of Physical Storage Media

Among the media typically available are these:

- **Cache**.
- **Main memory**. Main memory may contain tens of gigabytes of data on a personal computer, and even hundreds to thousands of gigabytes of data in large server systems. It is generally too small (or too expensive) for storing the entire database for every large databases, but many enterprise databases can fit in main memory. However, the contents of main memory are lost in the event of power failure or system crash; main memory is therefore said to be **volatile**.
- **Flash memory**. Flash memory differs from main memory in that stored data are retained even if power is turned off (or fails) – that is, it is **non-volatile**.

  A **solid-state drive** (**SSD**) uses flash memory internally to store data but provides an interface similar to a magnetic disk, allowing data to be stored or retrieved in units of a block; such an interface is called a *block-oriented interface*.
- **Magnetic-disk storage**. The primary medium for the long-term online storage of data is the magnetic disk drive, which is also referred to as the **hard disk drive** (**HDD**).
- **Optical storage**.

  Optical disk **jukebox** systems contain a few drives and numerous disks that can be loaded into one of the drives automatically (by a robot arm) on demand.
- **Tape storage**. Magnetic tape is cheaper than disks and can safely store data for many years. However, access to data is much slower because the tape must be accesse sequentially from the beginning of the tape; tapes can be very long, requiring tens to hundreds of seconds to access data. For this reason, tape storage is referred to as **sequential-access** storage. In contrast, magnetic disk and SSD storage are referred to as **direct-access** storage because it is possible to read data from any location on disk.

The various storage media can be organized in a hierarchy (Figure 12.1) according to their speed and their cost.

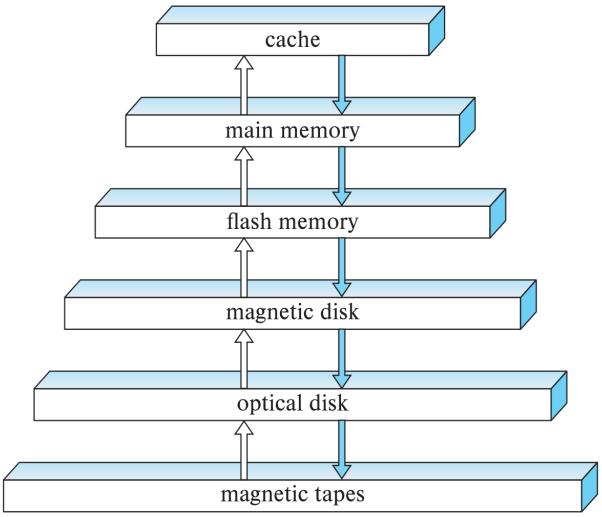

Figure 12.1: Storage device hierarchy.

The fastest storage media are referred to as **primary storage**. The media in the next level in the hierarchy are referred to as **secondary storage**, or **online storage**. The media in the lowest level in the hierarchy are referred to as **tertiary storage**, or **offline storage**.

2 Storage Interfaces

Disks typically support either the **Serial ATA** (**SATA**) interface, or the **Serial Attached SCSI** (**SAS**) interface; the SAS interface is typically used only in servers. The **Non-Volatile Memory Express** (**NVMe**) interface is a logical interface standard developed to better support SSDs and is typically used with the PCIe interface (the PCIe interface provides high-speed data transfer internal to computer systems).

In the **storage area network** (**SAN**) architecture, large numbers of disks are connected by a high-speed network to a number of server computers.

Network attached storage (**NAS**) is an alternative to SAN. Recent years have also seen the growth of **cloud storage**, where data are stored in the cloud and accessed via an API.

3 Magnetic Disks

3.1 Physical Characteristics of Disks

Each disk **platter** has a flat, circular shape. The disk surface is logically divided into **tracks**, which are subdivided into **sectors**. A **sector** is the smallest unit of information that can be read from or written to the disk.

The **read-write head** stores information on a sector magnetically as reversals of the direction of magnetization of the magnetic material.

A disk typically contains many platters, and the read-write heads of all the tracks are mounted on a single assembly called a **disk arm** and move together. The disk platters

mounted on a spindle and the heads mounted on a disk arm are together known as **head-disk assemblies**. Since the heads on all the platters move together, when the head on one platter is on the ith track, the heads on all other platters are also on the ith track of their respective platters. Hence, the ith tracks of all the platters together are called the ith **cylinder**.

A **disk controller** interfaces between the computer system and the actual hardware of the disk drive; in modern disk systems, the disk controller is implemented within the disk drive unit. A disk controller accepts high-level commands to read or write a sector, and initiates actions. Disk controllers also attach **checksums** to each sector that is written; the checksum is computed from the data written to the sector.

Another interesting task that disk controllers perform is **remapping of bad sectors**.

3.2 Performance Measures of Disks

Access time is the time from when a read or write request is issued to when data transfer begins. The time for repositioning the arm is called the **seek time**, and it increases with the distance that the arm must move.

The **average seek time** is the average of the seek times, measured over a sequence of (uniformly distributed) random requests.

Once the head has reached the desired track, the time spent waiting for the sector to be accessed to appear under the head is called the **rotational latency time**. On an average, one-half of a rotation of the disk is required for the beginning of the desired sector to appear under the head. Thus, the **average latency time** of the disk is one-half the time for a full rotation of the disk.

The **data-transfer rate** is the rate at which data can be retrieved from or stored to the disk.

A **disk block** is a logical unit of storage allocation and retrieval, and block sizes today typically range from 4 to 16 kilobytes. The term **page** is often used to refer to blocks, although in a few contexts they refer to different things.

In a **sequential access** pattern, successive requests are for successive block numbers, which are on the same track, or on adjacent tracks.

In contrast, in a **random access** pattern, successive requests are for blocks that are randomly located on disk. The number of **I/O operations per second** (**IOPS**), that is, the number random block accesses that can be satisfied by a disk in a second, depends on the access time, and the block size, and the data transfer rate of the disk.

The final commonly used measure of a disk is the **mean time to failure** (**MTTF**),[1] which is a measure of the reliability of the disk.

4 Flash Memory

A write to a page of flash memory typically takes about 100 microseconds. However, once written, a page of flash memory cannot be directly overwritten. Instead, it has to be erased and rewritten subsequently. The erase operation must be performed on a group of pages, called an **erase block**, erasing all the pages in the block, and takes about 2 to 5 milliseconds.

[1] The term **mean time between failures** (**MTBF**) is often used to refer to MTTF in context of disk drives, although technically MTBF should only be used in the context of systems that can be repaired after failure, and may fail again; MTBF would then be the sum of MTTF and the mean time to repair.

The logical-to-physical page mapping is replicated in an in-memory **translation table** for quick access.

Since each physical page can be updated only a fixed number of times, physical pages that have been erased many times are assigned "cold data," that is, data that are rarely updated, while pages that have not been erased many times are used to store "hot data," that is, data that are updated frequently. This principle of evenly distributing erase operations across physical blocks is called **wear leveling** and is usually performed transparently by flash-memory controllers.

All the above actions are carried out by a layer of software called the **flash translation layer**; above this layer, flash storage looks identical to magnetic disk storage, providing the same page/sector-oriented interface, except that flash storage is much faster.

STORAGE CLASS MEMORY

Although flash is the most widely used type of non-volatile memory, there have been a number of alternative non-volatile memory technologies developed over the years. Several of these technologies allow direct read and write access to individual bytes or words, avoiding the need to read or write in units of pages (and also avoiding the erase overhead of NAND flash). Such types of non-volatile memory are referred to as **storage class memory**, since they can be treated as a large non-volatile block of memory.

Hybrid disk drives are hard-disk systems that combine magnetic storage with a smaller amount of flash memory, which is used as a cache for frequently accessed data.

5 RAID

A variety of disk-organization techniques, collectively called **redundant arrays of independent disks** (**RAID**), have been proposed to achieve improved performance and reliability.

5.1 Improvement of Reliability via Redundancy

The solution to the problem of reliability is to introduce **redundancy**; that is, we store extra information that is not needed normally but that can be used in the event of failure of a disk to rebuild the lost information.

The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring** (or, sometimes, *shadowing*).

The mean time to failure (where failure is the loss of data) of a mirrored disk depends on the mean time to failure of the individual disks, as well as on the **mean time to repair**, which is the time it takes (on an average) to replace a failed disk and to restore the data on it.

5.2 Improvement in Performance via Parallelism

With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as

is almost always the case). The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.

With multiple disks, we can improve the transfer rate as well (or instead) by **striping data** across multiple disks. In its simplest form, data striping consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**.

Block-level striping stripes blocks across multiple disks.

5.3 RAID Levels

For a given set of blocks, a **parity block** can be computed and stored on disk; the ith bits of the parity block is computed as the "exclusive or" (XOR) of the ith bits of the all blocks in the set.

Whenever a block is written, the parity block for its set must be recomputed and written to disk. The new value of the parity block can be computed by either (i) reading all the other blocks in the set from disk and computing the new parity block, or (ii) by computing the XOR of the old value of the parity block with the old and new value of the updated block. These schemes have different cost-performance trade-offs. The schemes are classified into **RAID levels**. Figure 12.2 illustrates the four levels that are used in practice. For all levels, the figure depicts four disks worth of data, and the extra disks depicted are used to store redundant information for failure recovery.

- **RAID level 0** refers to disk arrays with striping at the level of blocks, but without any redundancy.
- **RAID level 1** refers to disk mirroring with block striping.

  Note that some vendors use the term **RAID level 1+0** or **RAID level 10** to refer to mirroring with striping, and they use the term RAID level 1 to refer to mirroring without striping.
- **RAID level 5** refers to block-interleaved distributed parity.
- **RAID level 6**, the $P + Q$ redundancy scheme, is much like RAID level 5, but it stores extra redundant information to guard against multiple disk failures.

5.4 Hardware Issues

RAID can be implemented with no change at the hardware level, using only software modification. Such RAID implementations are called **software RAID**. However, there are significant benefits to be had by building special-purpose hardware to support RAID; systems with special hardware support are called **hardware RAID** systems.

Reasons for loss of data on individual sectors could range from manufacturing defects to data corruption on a track when an adjacent track is written repeatedly. To minimize the chance of such data loss, good RAID controllers perform **scrubbing**; that is, during periods when disks are idle, every sector of every disk is read, and if any sector is found to be unreadable, the data are recovered from the remaining disks in the RAID organization, and the sector is written back.

Server hardware is often designed to permit **hot swapping**; that is, faulty disks can be removed and replaced by new ones without turning power off.

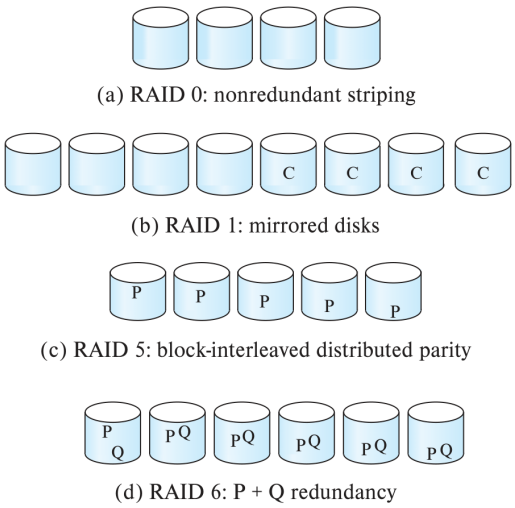

Figure 12.4 RAID levels.

Figure 12.2: RAID levels

5.5 Choice of RAID Level

The **rebuild performance** of a RAID system may be an important factor if continuous availability of data is required, as it is in high-performance database systems.

6 Disk-Block Access

Reducing the number of random accesses is very important for data stored on magnetic disks; SSDs support much faster random access than do magnetic disks, so the impact of random access is less with SSDs, but data access from SSDs can still benefit from some of the techniques described below.

- **Buffering**.
- **Read-ahead**. When a disk block is accessed, consecutive blocks from the same track are read into an in-memory buffer even if there is no pending request for the blocks. In the case of sequential access, such **read-ahead** ensures that many blocks are already in memory when they are requested, and it minimizes the time wasted in disk seeks and rotational latency per block read.
- **Scheduling**. **Disk-arm-scheduling** algorithms attempt to order accesses to tracks in a fashion that increases the number of accesses that can be processed. A commonly used algorithm is the **elevator algorithm**, which works in the same way many elevators do.
- **File organization**.

  Storing a large file in a single long sequence of consecutive blocks poses challenges to disk block allocation; instead, operating systems allocate some number of consecutive blocks (an **extent**) at a time to a file.

Over time, a sequential file that has multiple small appends may become **fragmented**; that is, its blocks become scattered all over the disk.

- **Non-volatile write buffers**.

  We can use *non-volatile random-access memory* (NVRAM) to speed up disk writes. The idea is that, when the database system (or the operating system) requests that a block be written to disk, the disk controller writes the block to a **non-volatile write buffer** and immediately notifies the operating system that the write completed successfully.

CHAPTER 13

DATA STORAGE STRUCTURES

1 Database Storage Architecture

Databases that store the entire database in memory and optimize in-memory data structures as well as query processing and other algorithms used by the database to exploit the memory residency of data are called **main-memory databases**.

2 File Organization

A **file** is organized logically as a sequence of records.

Each file is also logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer.

2.1 Fixed-Length Records

At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted. We use this first record to store the address of the second available record, and so on. The deleted records thus form a linked list, which is often referred to as a **free list**.

2.2 Variable-Length Records

Variable-length records arise in database systems due to several reasons.

The **slotted-page structure** is commonly used for organizing records within a block.

2.3 Storing Large Objects

Large objects may be stored either as files in file system area managed by the database, or as file structures stored in and managed by the database.

3 Organization of Records in Files

Several of the possible ways of organizing records in files are:

- **Heap file organization**.
- **Sequential file organization**.
- **Multitable clustering file organization**: Generally, a separate file or set of files is used to store the records of each relation.
- **B+-tree file organization**.
- **Hashing file organization**.

3.1 Heap File Organization

Most databases use a space-efficient data structure called a **free-space map** to track which blocks have free space to store records.

3.2 Sequential File Organization

A **sequential file** is designed for efficient processing of records in sorted order based on some search key. A **search key** is any attribute or set attributes; it need not be the primary key, or even a superkey.

We can manage deletion by using pointer chains. For insertion, we apply the following two rules:

1. Locate the record in the file that comes before the record to be inserted in search-key order.
2. If there is a free record (i.e., space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an *overflow block*. In either case, adjust the pointers so as to chain together the records in search-key order.

If relatively few records need to be stored in overflow blocks, this approach works well. Eventually, however, the correspondence between search-key order and physical order may be totally lost over a period of time, in which case sequential processing will become much less efficient. At this point, the file should be **reorganized** so that it is once again physically in sequential order.

3.3 Multitable Clustering File Organization

A **multitable clustering file organization** is a file organization that stores related records of two or more relations in each block. The **cluster key** is the attribute that defines which records are stored together.

3.4 Partitioning

Many databases allow the records in a relation to be partitioned into smaller relations that are stored separately. Such **table partitioning** is typically done on the basis of an attribute value.

4 Data-Dictionary Storage

A relational database system needs to maintain data *about* the relations. In general, such "data about data" are referred to as **metadata**.

Relational schemas and other metadata about relations are stored in a structure called the **data dictionary** or **system catalog**.

5 Database Buffer

The **buffer** is that part of main memory available for storage of copies of disk blocks. The subsystem responsible for the allocation of buffer space is called the **buffer manager**.

5.1 Buffer Manager

Buffer replacement strategy

When there is no room left in the buffer, a block must be **evicted**, that is, removed, from the buffer before a new one can be read in. Most operating systems use a **least recently used** (**LRU**) scheme, in which the block that was referenced least recently is written back to disk and is removed from the buffer.

Pinned blocks

Once a block has been brought into the buffer, a database process can read the contents of the block from the buffer memory.

It is therefore important that before a process reads data from a buffer block, it ensures that the block will not get evicted. To do so, the process executes a **pin** operation on the block; the buffer manager never evicts a pinned block. When it has finished reading data, the process should execute an **unpin** operation, allowing the block to be evicted when required.

Multiple processes can read data from a block that is in the buffer. The block cannot be evicted until all processes that have executed a pin have then executed an unpin operation. A simple way to ensure this property is to keep a **pin count** for each buffer block.

Forced output of blocks

There are situations in which it is necessary to write a block to disk, to ensure that certain data on disk are in a consistent state. Such a write is called a **forced output** of a block.

6 Buffer-Replacement Strategies

The assumption generally made is that blocks that have been referenced recently are likely to be referenced again. Therefore, if a block must be replaced, the least recently referenced block is replaced. This approach is called the **least recently used** (**LRU**) block-replacement scheme.

6.1 Reordering of Writes and Recovery

If a non-volatile write buffer were available, it could be used to perform the writes in order to non-volatile RAM and later reorder the writes when writing them to disk.

However, most disks do not come with a non-volatile write buffer; instead, modern file systems assign a disk for storing a log of the writes in the order that they are performed. Such a disk is called a **log disk**.

File systems that support log disks as above are called **journaling file systems**.

7 Column-Oriented Storage

Databases traditionally store all attributes of a tuple together in a record, and tuples are stored in a file. Such a storage layout is referred to as a *row-oriented storage*.

In contrast, in **column-oriented storage**, also called a **columnar storage**, each attribute of a relation is stored separately with values of the attribute from successive tuples stored at successive positions in the file.

If a query needs to access the entire contents of the ith row of a table, the values at the ith position in each of the columns are retrieved and used to reconstruct the row. Column-oriented storage thus has the drawback that fetching multiple attributes of a single tuple requires multiple I/O operations. Thus, it is not suitable for queries that fetch multiple attributes from a few rows of a relation.

However, column-oriented storage is well suited for data analysis queries, which process many rows of a relation, but often only access some of the attributes. The reasons are as follows:

- **Reduced I/O**.
- **Improved CPU cache performance**.
- **Improved compression**.
- **vector processing**. Many modern CPU architectures support **vector processing**, which allows a CPU operation to be applied in parallel on a number of elements of an array.

Databases that use column-oriented storage are referred to as **column stores**, while databases that use row-oriented storage are referred to as **row stores**.

In ORC, a row-oriented representation is converted to column-oriented representation as follows: A sequence of tuples occupying several hundred megabytes is broken up into a columnar representation called a **stripe**.

Some databases support two underlying storage systems, one a row-oriented one designed for transaction processing, and the second a column-oriented one, designed for data analysis. Such systems are called **hybrid row/column stores**.

8 Storage Organization in Main-Memory Databases

A **main-memory database** is a database where all data reside in memory; main-memory database systems are typically designed to optimize performance by making use of this fact.

CHAPTER 14

INDEXING

1 Basic Concepts

We shall consider several techniques for ordered indexing. Each technique must be evaluated on the basis of these factors:

- **Access types**: The types of access that are supported efficiently.
- **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time**: The time it takes to insert a new data item.
- **Deletion time**: The time it takes to delete a date item.
- **Space overhead**: The additional space occupied by an index structure.

An attribute or set of attributes used to look up records in a file is called a **search key**.

2 Ordered Indices

An **ordered index** stores the values of the search keys in sorted order and associates with each search key the records that contain it.

If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file. Clustering indices are also called **primary indices**; the term *primary index* may appear to denote an index on a primary key, but such indices can in fact be built on any search key. Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices**, or **secondary indices**.

Such files, with a clustering index on the search key, are called **index-sequential files**.

2.1 Dense and Sparse Indices

An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value.

There are two types of ordered indices that we can use:

- **Dense index**: In a dense index, an index entry appears for every search-key value in the file.
- **Sparse index**: In a sparse index, an index entry appears for only some of the search-key values.

2.2 Multilevel Indices

Indices with two or more levels are called **multilevel indices**.

2.3 Secondary Indices

If a relation can have more than one record containing the same search key value (that is, two or more records can have the same values for the indexed attributes), the search key is said to be a **nonunique search key**.

2.4 Indices on Multiple Keys

A search key containing more than one attribute is referred to as a **composite search key**.

3 B^+-Tree Index Files

The **B^+-tree index** structure is most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B^+-tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length.

3.1 Structure of a B^+-Tree

We consider first the structure of the **leaf nodes**.

The **nonleaf nodes** of the B^+-tree form a multilevel (sparse) index on the leaf nodes. Nonleaf nodes are also referred to as **internal nodes**.

3.2 Queries on B^+-Trees

Let us consider how we process queries on a B^+-tree.

B^+-tree can also be used to find all records with search key values in a specified range $[lb, ub]$. Such queries are called **range queries**.

3.3 Updates on B^+-Trees

Insertion and deletion are more complicated than lookup, since it may be necessary to **split** a node that becomes too large as the result of an insertion, or to **coalesce** nodes (i.e., combine nodes) if a node becomes too small (fewer than $\lceil n/2 \rceil$ pointers).

3.4 Nonunique Search Keys

We have assumed so far that search keys are unique. Recall also how to make search keys unique by creating a composite search key containing the original search key and extra attributes, that together are unique across all records.

The extra attribute is called a **uniquifier** attribute.

4 B$^+$-Tree Extensions

4.1 B$^+$-Tree File Organization

In a **B$^+$-tree file organization**, the leaf nodes of the tree store records, instead of storing pointers to records.

4.2 Indexing Strings

The fanout of nodes can be increased by using a technique called **prefix compression**.

4.3 Bulk Loading of B$^+$-Tree Indices

Insertion of a large number of entries at a time into an index is referred to as **bulk loading** of the index.

In **bottom-up B$^+$-tree construction**, after sorting the entries, we break up the sorted entries into blocks, keeping as many entries in a block as can fit in the block; the resulting blocks form the leaf level of the B$^+$-tree.

4.4 B-Tree Index Files

B-tree indices are similar to B$^+$-tree indices.

5 Hash Indices

In our description of hashing, we shall use the term **bucket** to denote a unit of storage that can store one or more records. In a **hash file organization**, instead of record pointers, buckets store the actual records; such structures only make sense with disk-resident data.

Formally, let K denote the set of all search-key values, and let B denote the set of all bucket addresses. A **hash function** h is a function from K to B.

To insert a record with search key K_i, we compute $h(K_i)$, which gives the address of the bucket for that record. We add the index entry for the record to the list at offset i. Note that there are other variants of hash indices that handle the case of multiple records in a bucket differently; the form described here is the most widely used variant and is called **overflow chaining**.

Hash indexing using overflow chaining is also called **closed addressing** (or, less commonly, **closed hashing**).

In a disk-based hash index, when we insert a record, we locate the bucket by using hashing on the search key, as described earlier. Assume for now that there is space in the bucket to store the record. Then, the record is stored in that bucket. If the bucket does

not have enough space, a **bucket overflow** is said to occur. We handle bucket overflow by using **overflow buckets**.

Bucket overflow can occur if there are insufficient buckets for the given number of records. Bucket overflow can also occur if some buckets are assigned more records than are others, resulting in one bucket overflowing even when other buckets still have a lot of free space.

Such **skew** in the distribution of records can occur if multiple records may have the same search key.

Hash indexing where the number of buckets is fixed when the index is created, is called **static hashing**. One of the problems with static hashing is that we need to know how many records are going to be stored in the index.

To handle this problem, the hash index can be rebuilt with an increased number of buckets. For example, if the number of records becomes twice the number of buckets, the index can be rebuilt with twice as many buckets as before. However, rebuilding the index has the drawback that it can take a long time if the relations are large, causing disruption of normal processing. Several schemes have been proposed that allow the number of buckets to be increased in a more incremental fashion. Such schemes are called **dynamic hashing** techniques; the *linear hashing* technique and the *extendable hashing* technique are two such schemes.

6 Multiple-Key Access

6.1 Covering Indices

Covering indices are indices that store the values of some attributes (other than the search-key attributes) along with the pointers to the record.

7 Write-Optimized Index Structures

The **log-structured merge tree** or LSM tree and its variants are write-optimized index structures that have seen very significant adoption.

7.1 LSM Trees

An LSM tree consists of several B^+-trees, starting with an in-memory tree, called L_0, and on-disk trees $L_1, L_2, \ldots, L_k$ for some k, where k is called the level.

Note that all entries in the leaf level of the old L_1 tree, including those in leaf nodes that do not have any updates, are copied to the new tree instead of performing updates on the existing L_1 tree node. This gives the following benefits:

1. The leaves of the new tree are sequentially located, avoiding random I/O during subsequent merges.
2. The leaves are full, avoiding the overhead of partially occupied leaves that can occur with page splits.

There is, however, a cost to using the LSM structure as described above: the entire contents of the tree are copied each time a set of entries from L_0 are copied into L_1. One of two techniques is used to reduce this cost:

- Multiple levels are used, with level L_{i+1} trees having a maximum size that is k times the maximum size of level L_i trees.
- Each level (other than L_0) can have up to some number b of trees, instead of just 1 tree.

  This variant of the LSM tree is called a **stepped-merge index**.

Instead of directly finding an index entry and deleting it, deletion results in insertion of a new **deletion entry** that indicates which index entry is to be deleted.

7.2 Buffer Tree

The key idea behind the **buffer tree** is to associate a buffer with each internal node of a B^+-tree, including the root node.

Buffer trees have been implemented as part of the **Generalized Search Tree** (**GiST**) index structure in PostgreSQL.

8 Bitmap Indices

A **bitmap** is simply an array of bits. In it simplest form, a **bitmap index** on the attribute A of relation r consists of one bitmap for each value that A can take.

9 Indexing of Spatial and Temporal Data

9.1 Indexing of Spatial Data

A tree structure called a **k-d tree** was one of the early structures used for indexing in multiple dimensions.

The **k-d-B tree** extends the k-d tree to allow multiple child nodes for each internal node, just as a B-tree extends a binary tree, to reduce the height of the tree.

Instead of dividing the data one dimension at a time, **quadtrees** divide up a two-dimensional space into four quadrants at each node of the tree.

A storage structure called an **R-tree** is useful for indexing of objects spanning regions of space in addition to points. An R-tree is a balanced tree structure with the indexed objects stored in leaf nodes. However, instead of a range of values, a rectangular **bounding box** is associated with each tree node.

9.2 Indexing Temporal Data

A **time interval** has a start time and an end time. Further a time interval indicates whether the interval starts at the start time, or just after the start time, that is, whether the interval is **closed** or **open** at the start time.

CHAPTER 15

QUERY PROCESSING

Query processing refers to the range of activities involved in extracting data from a database.

1 Overview

A relational-algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive**. A sequence of primitive operations that can be used to evaluate a query is a **query-execution plan** or **query-evaluation plan**. The **query-execution engine** takes a query-evaluation plan, executes that plan, and returns the answers to the query.

2 Measures of Query Cost

The costs of all the algorithms that we consider depend on the size of the buffer in main memory.

In addition, although we assume that data must be read from disk initially, it is possible that a block that is accessed is already present in the in-memory buffer.

The **response time** for a query-evaluation plan (that is, the wall-clock time required to execute the plan), assuming no other activity is going on in the computer, would account for all these costs, and could be used as a measure of the cost of the plan.

Interestingly, a plan may get a better response time at the cost of extra resource consumption.

As a result, instead of trying to minimize the response time, optimizers generally try to minimize the total **resource consumption** of a query plan.

3 Selection Operation

In query processing, the **file scan** is the lowest-level operator to access data.

3.1 Selections Using Files Scans and Indices

The most straightforward way of performing a selection is as follows:

- **A1** (**linear search**).

Index structures are referrd to as **access paths**, since they provide a path through which data can be located and accessed.

Search algorithms that use an index are referred to as **index scans**.

3.2 Implementation of Complex Selections

So far, we have considered only simple selection conditions of the form A *op* B, where *op* is an equality or comparison operation. We now consider more complex selection predicates.

- **Conjunction**: A **conjuntive selection** is a selection of the form:

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(r)$$

- **Disjunction**: A **disjuntive selection** is a selection of the form:

$$\sigma_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n}(r)$$

- **Negation**: The result of a selection $\sigma_{\neg\theta}(r)$ is the set of tuples of r for which the condition θ evaluates to false.

We can implement a selection operation involving either a conjunction or a disjunction of simple conditions by using one of the following algorithms:

- **A7 (conjunctive selection using one index)**.
- **A8 (conjunctive selection using composite index)**. An appropriate **composite index** (that is, an index on multiple attributes) may be available for some conjunctive selections.
- **A9 (conjunctive selection by intersection of identifiers)**.
- **A10 (disjunctive selection by union of identifiers)**.

4 Sorting

4.1 External Sort-Merge Algorithm

Sorting of relations that do not fit in memory is called **external sorting**. The most commonly used technique for external sorting is the **external sort-merge** algorithm.

Let M denote the number of blocks in the main memory buffer available for sorting, that is, the number of disk blocks whose contents can be buffered in available main memory.

1. In the first stage, a number of sorted **runs** are created; each run is sorted but contains only some of the records of the relation.
2. In the second stage, the runs are *merged*.

The preceding merge operation is a generalization of the two-way merge used by the standard in-memory sortmerge algorithm; it merges N runs, so it is called an **N-way merge**.

5 Join Operation

We use the term **equi-join** to refer to a join of the form $r \bowtie_{r.A=s.B} s$, where A and B are attributes or sets of attributes of relations r and s, respectively.

5.1 Nested-Loop Join

Figure 15.1 shows a simple algorithm to compute the theta join, $r \bowtie_\theta s$, of two relations r and s. This algorithm is called the **nested-loop join** algorithm, since it basically consists of a pair of nested **for** loops. Relation r is called the **outer relation** and relation s the **inner relation** of the join, since the loop for r encloses the loop for s.

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add t_r · t_s to the result;
    end
end
```

Figure 15.1: Nested-loop join.

5.2 Block Nested-Loop Join

Figure 15.2 shows **block nested-loop join**, which is a variant of the nested-loop join where every block of the inner relation is paired with every block of the outer relation.

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                test pair (t_r, t_s) to see if they satisfy the join condition
                if they do, add t_r · t_s to the result;
            end
        end
    end
end
```

Figure 15.2: Block nested-loop join.

5.3 Indexed Nested-Loop Join

In a nested-loop join, if an index is available on the inner loops join attribute, index lookups can replace file scans.

This join method is called an **indexed nested-loop join**; it can be used with existing indices, as well as with temporary indices created for the sole purpose of evaluating the join.

5.4 Merge Join

The **merge-join** algorithm (also called the **sort-merge-join** algorithm) can be used to compute natural joins and equi-joins.

Hybrid Merge Join

The **hybrid merge-join algorithm** merges the sorted relation with the leaf entries of the secondary B^+-tree index.

5.5 Hash Join

Basics

```
/* Partition s */
for each tuple t_s in s do begin
    i := h(t_s[JoinAttrs]);
    H_{s_i} := H_{s_i} ∪ {t_s};
end
/* Partition r */
for each tuple t_r in r do begin
    i := h(t_r[JoinAttrs]);
    H_{r_i} := H_{r_i} ∪ {t_r};
end
/* Perform join on each partition */
for i := 0 to n_h do begin
    read H_{s_i} and build an in-memory hash index on it;
    for each tuple t_r in H_{r_i} do begin
        probe the hash index on H_{s_i} to locate all tuples t_s
            such that t_s[JoinAttrs] = t_r[JoinAttrs];
        for each matching tuple t_s in H_{s_i} do begin
            add t_r ⋈ t_s to the result;
        end
    end
end
```

Figure 15.3: Hash join.

Figure 15.3 shows the details of the **hash-join** algorithm to compute the natural join of relations r and s. After the partitioning of the relations, the rest of the hash-join code performs a separate indexed nested-loop join on each of the partition pairs i, for $i = 0, \ldots, n_h$. To do so, it first **builds** a hash index on each s_i, and then **probes** (that is, looks up s_i) with tuples from r_i. The relation s is the **build input**, and r is the **probe input**.

6 Evaluation of Expressions