**Read through and take notes. Red underlined areas are very important but surrounding information could also be tested on as well so it's important to at least have an understanding of each area.**

**1.1 DATABASE BASICS**

### Data

***Data*** is numeric, textual, visual, or audio information that describes real-world systems. Data is collected and processed to aid in a variety of tasks, such as forecasting weather, analyzing financial investments, and tracking the global spread of pandemics.

Data can vary in several important ways:

* **Scope.** The amount of data produced and collected can vary. Ex: A small business might track an inventory of a few thousand items, but a large commerce website might track billions of items.
* **Format.** Data may be produced as numbers, text, image, audio, or video. Ex: A phone's proximity sensor generates raw numbers, and a satellite captures images.
* **Access.** Some data sources are private while others are made publicly available. Ex: A retail company may use private customer data to discover purchasing behavior patterns, but a government may be required by law to share certain data sets.

### Databases

A ***database*** is a collection of data in a structured format. In principle, databases can be stored on paper or even clay tablets. In practice, however, modern databases are invariably stored on computers. The database structure ensures that similar data is stored in a standardized manner.

* A ***database system***, also known as a ***database management system*** or ***DBMS***, is software that reads and writes data in a database. Database systems ensure data is secure, internally consistent, and available at all times. These functions are challenging for large databases with many users, so database systems are complex.
* A ***query*** is a request to retrieve or change data in a database. A ***query language*** is a specialized programming language, designed specifically for database systems. Query languages read and write data efficiently
* A ***database application*** is software that helps business users interact with database systems. Many databases are complex, and most users are not familiar with query languages. Consequently, direct database access is usually not feasible. Instead, programmers write applications to simplify the user experience and ensure data access is efficient and secure.

### Database roles

People interact with databases in a variety of roles:

* A ***database administrator*** is responsible for securing the database system against unauthorized users. A database administrator enforces procedures for user access and database system availability.
* A ***database designer*** determines the format of each data element and the overall database structure. Database designers must balance several priorities, including storage, response time, and support for rules that govern the data. Since these priorities often conflict, database design is technically challenging.
* A ***database programmer*** develops computer programs that utilize a database. Database programmers write applications that combine database query languages and general-purpose programming languages. Query languages and general-purpose languages have significant differences, so database programming is a specialized challenge.
* A ***database user*** is a consumer of data in a database. Database users request, update, or use stored data to generate reports or information. Database users usually access the database via applications but can also submit queries directly to the database system.

**1.2 DATABASE SYSTEMS**

### File systems and database systems

Small databases that are shared by one or two users can be managed in a text file or spreadsheet. Text files and spreadsheets are inadequate, however, as databases grow in size, complexity, and use. Large, complex databases that are shared by many users have special requirements:

* **Performance**. When many users and applications simultaneously access large databases, query response time degrades rapidly. Database systems maintain fast response times by structuring data properly on storage media and processing queries efficiently.
* **Authorization**. Many database users should have limited access to specific tables, columns, or rows of a database. Database systems authorize individual users to access specific data.
* **Security**. Database systems ensure authorized users only access permissible data. Database systems also protect against hackers by encrypting data and restricting access.
* **Rules**. Database systems ensure data is consistent with structural and business rules. Ex: When multiple copies of data are stored in different locations, copies must be synchronized as data is updated. Ex: When a course number appears in a student registration record, the course must exist in the course catalog.
* **Recovery**. Computers, database systems, and individual transactions occasionally fail. Database systems must recover from failures and restore the database to a consistent state without loss of data.

A ***transaction*** is a group of queries that must be either completed or rejected as a whole. Execution of some, but not all, queries results in inconsistent or incorrect data. Ex: A debit-credit transaction transfers funds from one bank account to another. The first query removes $100 from one account and the second query deposits $100 in another account. If the first query succeeds but the second fails, $100 is mysteriously lost. The transaction must process either both queries or neither query.

When processing transactions, database systems must:

* **Ensure transactions are processed completely or not at all**. A computer or application might fail while processing a transaction. When failing to process a transaction, the database system must reverse partial results and restore the database to the values prior to the transaction.
* **Prevent conflicts between concurrent transactions**. When multiple transactions access the same data at the same time, a conflict may occur. Ex: Sam selects a seat on a flight. Maria purchases the same seat in a separate transaction before Sam completes his transaction. When Sam clicks the 'purchase' button, his seat is suddenly unavailable.
* **Ensure transaction results are never lost**. Once a transaction completes, transaction results must always be saved on storage media, regardless of application or computer failures.

### Architecture

The ***architecture*** of a database system describes the internal components and the relationships between components. At a high level, the components of most database systems are similar:

* The ***query processor*** interprets queries, creates a plan to modify the database or retrieve data, and returns query results to the application. The query processor performs ***query optimization*** to ensure the most efficient instructions are executed on the data.
* The ***storage manager*** translates the query processor instructions into low-level file-system commands that modify or retrieve data. Database sizes range from megabytes to many terabytes, so the storage manager uses ***indexes*** to quickly locate data.
* The ***transaction manager*** ensures transactions are properly executed. The transaction manager prevents conflicts between concurrent transactions. The transaction manager also restores the database to a consistent state in the event of a transaction or system failure.
* The ***log*** is a file containing a complete record of all inserts, updates, and deletes processed by the database. The transaction manager writes log records before applying changes to the database. In the event of a failure, the transaction manager uses log records to restore the database.
* The ***catalog***, also known as a ***data dictionary***, is a directory of tables, columns, indexes, and other database objects. Other components use catalog information to process and execute queries.

**Metadata** is data about the database, such as column names and the number of rows in each table.

### Products

Most leading database systems are relational. A ***relational database*** stores data in tables, columns, and rows, similar to a spreadsheet. All data in a column has the same format. All data in a row represents a single object, such as a person, place, product, or activity.

All relational database systems support the SQL query language. ***SQL*** stands for Structured Query Language and includes statements that read and write data, create and delete tables, and administer the database system.

Relational systems are ideal for databases that require an accurate record of every transaction, such as banking, airline reservation systems, and student records. The growth of the internet in the 1990s generated massive volumes of online data, called ***big data***, often with poorly structured or missing information. Relational systems were not initially designed for big data and, as a result, many non-relational systems have appeared since 2000. The newer non-relational systems are called ***NoSQL***, for 'not only SQL', and are optimized for big data.

Prior to 2000, most database systems were commercial products, developed by for-profit companies and licensed for a fee. Since 2000, an alternative licensing model, called open source, has become popular. ***Open source*** software is software that anyone can inspect, copy, and modify with no licensing fee.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Product | Sponsor | Type | License | DB-Engines rank (May 2020) |
| Oracle Database | Oracle | Relational | Commercial | 1 |
| MySQL | Oracle | Relational | Open source | 2 |
| SQL Server | Microsoft | Relational | Commercial | 3 |
| PostgreSQL | PostgreSQL Global Development Group | Relational | Open source | 4 |
| MongoDB | MongoDB | NoSQL | Open source | 5 |

### **1.3 QUERY LANGUAGES**

### Common queries

A database system responds to queries written in a query language. A ***query*** is a command for a database that typically inserts new data, retrieves data, updates data, or deletes data from a database. A ***query language*** is a computer programming language for writing database queries.

*The four common queries are sometimes referred to as* ***CRUD*** *operations, an acronym for Create, Read, Update, and Delete data.*

*The term* ***NoSQL*** *refers to a new generation of non-relational databases. NoSQL originally meant 'does not support SQL'. However, many NoSQL databases have added support for SQL, and 'NoSQL' has come to mean 'not only SQL'.*

An SQL ***statement*** is a database command, such as a query that inserts, selects, updates, or deletes data:

* ***INSERT*** inserts rows into a table.
* ***SELECT*** retrieves data from a table.
* ***UPDATE*** modifies data in a table.
* ***DELETE*** deletes rows from a table.

The SQL ***CREATE TABLE*** statement creates a new table by specifying the table and column names. Each column is assigned a ***data type*** that indicates the format of column values. Data types can be numeric, textual, or complex. Ex:

* INT stores integer values.
* DECIMAL stores fractional numeric values.
* VARCHAR stores textual values.
* DATE stores year, month, and day.

Some data types are followed by one or two numbers in parentheses, indicating the size of the data type. Ex: VARCHAR(10) indicates ten characters. DECIMAL(10, 3) indicates ten significant digits, including three after the decimal point.

**1.4 DATABASE DESIGN AND PROGRAMMING**

### Analysis

A ***database design*** is a specification of database objects such as tables, columns, data types, and indexes. Database design also refers to the process used to develop the specification.

For small, simple databases, the database design process can be informal and unstructured. For large, complex databases, the process has three phases:

1. Analysis
2. Logical design
3. Physical design

The ***analysis*** phase specifies database requirements without regard to a specific database system. Requirements are represented as entities, relationships, and attributes. An entity is a person, place, activity, or thing. A relationship is a link between entities, and an attribute is a descriptive property of an entity.

***Analysis*** *has many alternative names, such as conceptual design, entity-relationship modeling, and requirements definition.*

Entities, relationships, and attributes are depicted in ***ER diagrams***:

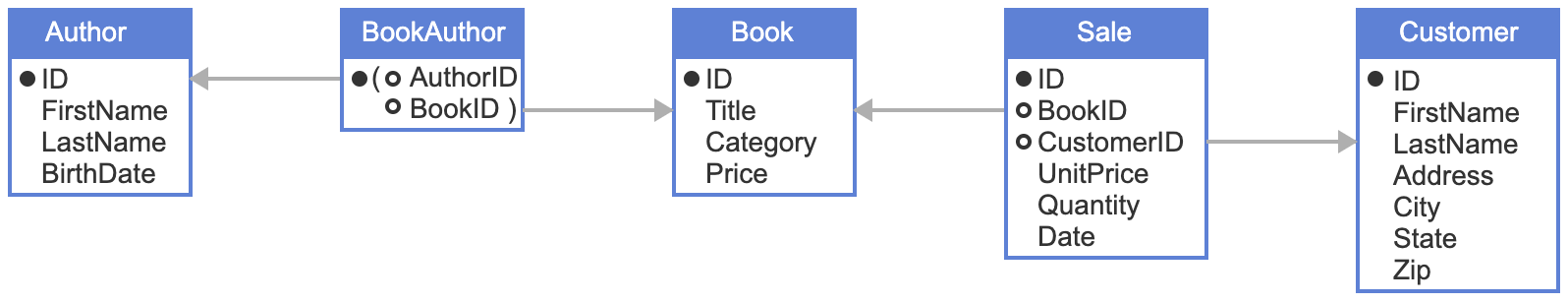
* Rectangles represent entities. Entity names appear at the top of rectangles.
* Lines between rectangles represent relationships.
* Text inside rectangles and below entity names represent attributes.

ER diagrams are usually supplemented by textual descriptions of entities, relationships, and attributes.

### Logical design

The ***logical design*** phase implements database requirements in a specific database system. For relational database systems, logical design converts entities, relationships, and attributes into tables, keys, and columns. A ***key*** is a column used to identify individual rows of a table. Tables, keys, and columns are specified in SQL with CREATE TABLE statements.

The logical design is depicted in a ***table diagram***. Table diagrams are similar to ER diagrams but more detailed.



### Physical design

The ***physical design*** phase adds indexes and specifies how tables are organized on storage media. Ex: Rows of a table may be sorted on the values of a column and stored in sort order. Physical design is specified with SQL statements such as CREATE INDEX and, like logical design, is specific to a database system.

Physical design can be depicted in diagrams. However, logical design is more important for database users and programmers, so physical design diagrams are not commonly used.

In relational databases, logical and physical design affect queries differently. Logical design affects the query result. Physical design affects query processing speed but never affects the query result. The principle that physical design never affects query results is called ***data independence***.

Data independence allows database designers to tune query performance without changes to application programs. When database designers modify indexes or row order, applications run faster or slower but always generate the same results.

The term **information independence** is occasionally used instead of data independence. These terms are synonymous.

### Programming

Because of data independence, relational database applications can be programmed before the physical design is in place. Applications may run slowly but will generate correct results.

SQL is the standard relational query language but lacks important programming features. Ex: Most SQL implementations are not object-oriented. To write a database program, SQL is usually combined with a general-purpose programming language such as C++, Java, or Python.

To simplify the use of SQL with a general-purpose language, database programs typically use an application programming interface. An ***application programming interface***, or ***API***, is a library of procedures or classes that links a host programming language to a database. The host language calls library procedures, which handle details such as connecting to the database, executing queries, and returning results. Ex: JDBC is a library of Java classes that access relational databases.

**1.5 MYSQL**

### MySQL Command-Line Client

The ***MySQL Command-Line Client*** is a text interface included in the MySQL Server download. The Command-Line Client allows developers to connect to the database server, perform administrative functions, and execute SQL statements.

The animation above shows the user typing SQL commands that use the 'world' database, a database that is usually installed with MySQL. The world database contains three tables: city, country, and countrylanguage. Users can practice entering SQL statements that work with and manipulate the world database. Some installations do not include the world database, so users must download and install the world database from MySQL.com separately.

MySQL Server returns an ***error code*** and description when an SQL statement is syntactically incorrect or the database cannot execute the statement.

### MySQL Workbench

Some developers prefer to interact with MySQL Server via a graphical user interface. ***MySQL Workbench*** is installed with MySQL Server and allows developers to execute SQL commands using an editor. When MySQL Workbench is started, the user can connect to MySQL Server running on the local machine or on the network.

**2.1 RELATIONAL MODEL**

### Database models

A ***database model*** is a conceptual framework for database systems, with three parts:

* **Data structures** that prescribe how data is organized.
* **Operations** that manipulate data structures.
* **Rules** that govern valid data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Primary data structure | Initial product releases | Example database system | Strengths |
| **Hierarchical** | Tree | 1960s | IMS | Fast queries Efficient storage |
| **Network** | Linked list | 1970s | IDMS | Fast queries Efficient storage |
| **Relational** | Table | 1980s | Oracle Database | Productivity and simplicity Transactional applications |
| **Object** | Class | 1990s | ObjectStore | Integration with object-oriented programming languages |
| **Graph** | Vertex and edge | 2000s | Neo4j | Flexible schema Evolving business requirements |
| **Document** | XML JSON | 2010s | MongoDB | Flexible schema Unstructured and semi-structured data |

### Relational data structure

The relational data structure is based on set theory. A ***set*** is an unordered collection of elements enclosed in braces. Ex: **{a, b, c}** and **{c, b, a}** are the same, since sets are not ordered. A ***tuple*** is an ordered collection of elements enclosed in parentheses. Ex: **(a, b, c)** and **(c, b, a)** are different, since tuples are ordered.

The data structure organizes data in tables:

* A ***table*** has a name, a fixed tuple of columns, and a varying set of rows.
* A ***column*** has a name and a data type.
* A ***row*** is an unnamed tuple of values. Each value corresponds to a column and belongs to the column's data type.
* A ***data type*** is a named set of values, from which column values are drawn.

Since a table is a set of rows, the rows have no inherent order.

Synonyms:

Table, File, Relation

Row, Record, Tuple

Column, Field, Attribute

### Relational operations

Like the relational data structure, relational operations are based on set theory. Each operation generates a result table from one or two input tables:

* **Select** selects a subset of rows of a table.
* **Project** eliminates one or more columns of a table.
* **Product** lists all combinations of rows of two tables.
* **Join** combines two tables by comparing related columns.
* **Union** selects all rows of two tables.
* **Intersect** selects rows common to two tables.
* **Difference** selects rows that appear in one table but not another.
* **Rename** changes a table name.
* **Aggregate** computes functions over multiple table rows, such as sum and count.

### Relational rules

Rules are logical constraints that ensure data is valid.

***Relational rules*** are part of the relational model and govern data in every relational database. Ex:

* **Unique primary key**. All tables have a primary key column, or group of columns, in which values may not repeat.
* **Unique column names**. Different columns of the same table have different names.
* **No duplicate rows**. No two rows of the same table have identical values in all columns.

***Business rules*** are based on business policy and specific to a particular database. Ex: All rows of the Employee table must have a valid entry in the DepartCode column. Ex: PassportNumber values may not repeat in different Employee rows.

Relational rules are implemented as SQL ***constraints*** and enforced by the database system. Business rules are discovered during database design and, like relational rules, often implemented as SQL constraints. However, some complex business rules must be enforced by applications running on the database.

**2.2 STRUCTURED QUERY LANGUAGE**

***Structured Query Language*** (***SQL***) is a high-level computer language for storing, manipulating, and retrieving data. SQL is the standard language for relational databases, and is commonly supported in non-relational databases. SQL is pronounced either 'S-Q-L' or 'seekwəl', but the preferred pronunciation is 'S-Q-L'.

### SQL syntax

An SQL ***statement*** is a complete command composed of one or more clauses. A ***clause*** groups SQL keywords like SELECT, FROM, and WHERE with table names like City, column names like Name, and conditions like Population > 100000.

SELECT Name

FROM City

WHERE Population > 1000

|  |  |  |
| --- | --- | --- |
| Type | Description | Examples |
| Literals | Explicit values that are string, numeric, or binary. Strings must be surrounded by single quotes or double quotes. Binary values are represented with x'0' where the 0 is any hex value. | 'String' "String" 123 x'0fa2' |
| Keywords | Words with special meaning. | SELECT, FROM, WHERE |
| Identifiers | Objects from the database like tables, columns, etc. | City, Name, Population |
| Comments | Statement intended only for humans and ignored by the database when parsing an SQL statement. | *-- single line comment* */\* multi-line*   *Comment \*/* |

### SQL sublanguages

The SQL language is divided into five sublanguages:

* ***Data Definition Language*** (DDL) defines the structure of the database.
* ***Data Query Language*** (DQL) retrieves data from the database.
* ***Data Manipulation Language*** (DML) manipulates data stored in a database.
* ***Data Control Language*** (DCL) controls database user access.
* ***Data Transaction Language*** (DTL) manages database transactions.

**2.3 MANAGING DATABASES**

### CREATE DATABASE and DROP DATABASE statements

A ***database system instance*** is a single executing copy of a database system. Personal computers usually run just one instance of a database system. Shared computers, such as computers used for cloud services, usually run multiple instances of a database system. Each instance usually contains multiple system and user databases.

Several SQL statements help database administrators, designers, and users manage the databases on an instance. ***CREATE DATABASE DatabaseName*** creates a new database. ***DROP DATABASE DatabaseName*** deletes a database, including all tables in the database.

### USE and SHOW statements

***USE DatabaseName*** selects a default database for use in subsequent SQL statements.

Several SHOW statements provide information about databases, tables, and columns:

* ***SHOW DATABASES*** lists all databases in the database system instance.
* ***SHOW TABLES*** lists all tables in the default database.
* ***SHOW COLUMNS FROM TableName*** lists all columns in the TableName table of the default database.
* ***SHOW CREATE TABLE TableName*** shows the CREATE TABLE statement for the TableName table of the default database.

**2.4 TABLES**

### Tables, columns, and rows

All data in a relational database is structured in tables:

* A ***table*** has a name, a fixed sequence of columns, and a varying set of rows.
* A ***column*** has a name and a data type.
* A ***row*** is an unnamed sequence of values. Each value corresponds to a column and belongs to the column's data type.
* A ***cell*** is a single column of a single row.

A table must have at least one column but any number of rows. A table without rows is called an ***empty table***.

### Rules governing tables

Tables must obey relational rules, including:

1. **Exactly one value per cell**. A cell may not contain multiple values. Unknown data is represented with a special NULL value.
2. **No duplicate column names**. Duplicate column names are allowed in different tables, but not in the same table.
3. **No duplicate rows**. No two rows may have identical values in all columns.
4. **No row order**. Rows are not ordered. The organization of rows on a storage device, such as a disk drive, never affects query results.

Rules 6 and 7 follow directly from the definition of a table. A table is a set of rows. Since a set's elements may not repeat and are not ordered, the same is true of a table's rows.

Rule 7 is called ***data independence***. Data independence allows database administrators to improve query performance by changing the organization of data on storage devices, without affecting query results.

### CREATE TABLE and DROP TABLE statements

The ***CREATE TABLE*** statement creates a new table by specifying the table name, column names, and column data types. Example data types are:

* **INT or INTEGER** — integer values
* **VARCHAR(N)** — values with 0 to N characters
* **DATE** — date values
* **DECIMAL(M, D)** — numeric values with M digits, of which D digits follow the decimal point

The ***DROP TABLE*** statement deletes a table, along with all the table's rows, from a database.

### ALTER TABLE statement

The ***ALTER TABLE*** statement adds, deletes, or modifies columns on an existing table. The ALTER TABLE statement specifies the table name followed by a clause that indicates what should be altered. The table below summarizes the three ALTER TABLE clauses.

|  |  |  |
| --- | --- | --- |
| ALTER TABLE clause | Description | Syntax |
| ADD | Adds a column | ALTER TABLE TableName  ADD ColumnName DataType; |
| CHANGE | Modifies a column | ALTER TABLE TableName  CHANGE CurrentColumnName NewColumnName NewDataType; |
| DROP | Deletes a column | ALTER TABLE TableName  DROP ColumnName; |

**2.5 DATA TYPES**

### Data type categories

A ***data type*** is a named set of values from which column values are drawn. In relational databases, most data types fall into one of the following categories:

* ***Integer*** data types represent positive and negative integers. Several integer data types exist, varying by the number of bytes allocated for each value. Common integer data types include INT, implemented as 4 bytes of storage, and SMALLINT, implemented as 2 bytes.
* ***Decimal*** data types represent numbers with fractional values. Decimal data types vary by number of digits after the decimal point and maximum size. Common decimal data types include FLOAT and DECIMAL.
* ***Character*** data types represent textual characters. Common character data types include CHAR, a fixed string of characters, and VARCHAR, a string of variable length up to a specified maximum size.
* ***Date and time*** data types represent date, time, or both. Some date and time data types include a time zone or specify a time interval. Some date and time data types represent an interval rather than a point in time. Common date and time data types include DATE, TIME, DATETIME, and TIMESTAMP.
* ***Binary*** data types store data exactly as the data appears in memory or computer files, bit for bit. The database manages binary data as a series of zeros and ones. Common binary data types include BLOB, BINARY, VARBINARY, and IMAGE.
* ***Spatial*** data types store geometric information, such as lines, polygons, and map coordinates. Examples include POLYGON, POINT, and GEOMETRY. Spatial data types are relatively new and consequently vary greatly across database systems.
* ***Document*** data types contain textual data in a structured format such as XML or JSON.

|  |  |  |
| --- | --- | --- |
| Category | Data type | Value |
| Integer | INT | -9281344 |
| Decimal | FLOAT | 3.1415 |
| Character | VARCHAR | Chicago |
| Date and time | DATETIME | 12/25/2020 10:35:00 |
| Binary | BLOB | 1001011101 . . . |
| Spatial | POINT | (2.5, 33.44) |
| Document | XML | <menu>  <selection>  <name>Greek salad</name>  <price>$13.90</price>  <text>Cucumbers, tomatoes, onions, and feta cheese</text>  </selection>  <selection>  <name>Turkey sandwich</name>  <price>$9.00</price>  <text>Turkey, lettuce, tomato on choice of bread</text>  </selection> </menu> |

### MySQL data types

All relational databases support integer, decimal, date and time, and character data types. Most databases allow integer and decimal numbers to be signed or unsigned. A ***signed*** number may be negative. An ***unsigned*** number cannot be negative.

Data types vary in storage requirements. Ex:

* Character data types use one or two bytes per character.
* Integer data types use a fixed number of bytes per number.
* Unsigned data types can store larger numbers than the signed version of the same data type.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | Example | Data type | Storage | Notes |
| Integer | 34 and -739448 | TINYINT | 1 byte | Signed range: -128 to 127 Unsigned range: 0 to 255 |
| SMALLINT | 2 bytes | Signed range: -32,768 to 32,767 Unsigned range: 0 to 65,535 |
| MEDIUMINT | 3 bytes | Signed range: -8,388,608 to 8,388,607 Unsigned range: 0 to 16,777,215 |
| INTEGER or INT | 4 bytes | Signed range: -2,147,483,648 to 2,147,483,647 Unsigned range: 0 to 4,294,967,295 |
| BIGINT | 8 bytes | Signed range: -263 to 263 -1 Unsigned range: 0 to 264 -1 |
| Decimal | 123.4 and -54.29685 | DECIMAL(M,D) | Varies depending on *M* and *D* | Exact decimal number where *M* = number of significant digits, *D* = number of digits after decimal point |
| FLOAT | 4 bytes | Approximate decimal numbers with range: -3.4E+38 to 3.4E+38 |
| DOUBLE | 8 bytes | Approximate decimal numbers with range: -1.8E+308 to 1.8E+308 |
| Date and time | '1776-07-04 13:45:22' | DATE | 3 bytes | Format: YYYY-MM-DD. Range: '1000-01-01' to '9999-12-31' |
| TIME | 3 bytes | Format: hh:mm:ss |
| DATETIME | 5 bytes | Format: YYYY-MM-DD hh:mm:ss. Range: '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. |
| Character | 'string' | CHAR(N) | *N* bytes | Fixed-length string of length *N*; 0 ≤ *N* ≤ 255 |
| VARCHAR(N) | Length of characters + 1 bytes | Variable-length string with maximum *N* characters; 0 ≤ *N* ≤ 65,535 |
| TEXT | Length of characters + 2 bytes | Variable-length string with maximum 65,535 characters |

**2.6 SELECTING ROWS**

### Operators

An ***operator*** is a symbol that computes a value from one or more other values, called ***operands***:

* Arithmetic operators compute numeric values from numeric operands.
* Comparison operators compute logical values TRUE or FALSE. Operands may be numeric, character, and other data types.
* Logical operators compute logical values from logical operands.

A ***unary*** operator has one operand. A ***binary*** operator has two operands. Most operators are binary. The logical operator NOT is unary. The arithmetic operator - is either unary or binary.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Operator | Description | Example | Value |
| Arithmetic | + | Adds two numeric values | 4 + 3 | 7 |
| - (unary) | Reverses the sign of one numeric value | -(-2) | 2 |
| - (binary) | Subtracts one numeric value from another | 11 - 5 | 6 |
| \* | Multiplies two numeric values | 3 \* 5 | 15 |
| / | Divides one numeric value by another | 4 / 2 | 2 |
| % (modulo) | Divides one numeric value by another and returns the integer remainder | 5 % 2 | 1 |
| ^ | Raises one numeric value to the power of another | 5^2 | 25 |
| Comparison | = | Compares two values for equality | 1 = 2 | FALSE |
| != | Compares two values for inequality | 1 != 2 | TRUE |
| < | Compares two values with < | 2 < 2 | FALSE |
| <= | Compares two values with ≤ | 2 <= 2 | TRUE |
| > | Compares two values with > | '2019-08-13' > '2021-08-13' | FALSE |
| >= | Compares two values with ≥ | 'apple' >= 'banana' | FALSE |
| Logical | AND | Returns TRUE only when both values are TRUE | TRUE AND FALSE | FALSE |
| OR | Returns FALSE only when both values are FALSE | TRUE OR FALSE | TRUE |
| NOT | Reverses a logical value | NOT FALSE | TRUE |

### Expressions

An ***expression*** is a string of operators, operands, and parentheses that evaluates to a single value. Operands may be column names or fixed values. The value of an expression may be any data type. Ex: Salary > 34000 AND Department = 'Marketing' is an expression with a logical value.

A simple expression may consist of a single column name or a fixed value. Ex: The column EmployeeName and the fixed value 'Maria' are expressions with a character data type.

When an expression is evaluated, column names are replaced with column values for a specific row. Consequently, an expression containing column names may have different values for different rows.

The order of operator evaluation may affect the value of an expression. Operators in an expression are evaluated in the order of ***operator precedence***, shown in the table below. Operators of the same precedence are evaluated from left to right. Regardless of operator precedence, expressions enclosed in parentheses are evaluated before any operators outside the parentheses are applied.

### SELECT statement

The SELECT statement selects rows from a table. The statement has a ***SELECT*** clause and a ***FROM*** clause. The FROM clause specifies the table from which rows are selected. The SELECT clause specifies one or more expressions, separated by commas, that determine what values are returned for each row.

The SELECT statement returns a set of rows, called the ***result table***.

### WHERE clause

An expression may return a value of any data type. A ***condition*** is an expression that evaluates to a logical value.

A SELECT statement has an optional ***WHERE*** clause that specifies a condition for selecting rows. A row is selected when the condition is TRUE for the row values. A row is omitted when the condition is either FALSE or NULL.

The WHERE clause follows the FROM clause. When a SELECT statement has no WHERE clause, all rows are selected.

**2.7 NULL VALUES**

### NULL

***NULL*** is a special value that represents either unknown or inapplicable data. NULL is not the same as zero for numeric data types or blanks for character data types. Ex: A zero bonus indicates an employee can, but has not, earned a bonus. A zero bonus is known and applicable, and should not be represented as NULL.

### NOT NULL constraint

The ***NOT NULL*** constraint prevents a column from having a NULL value. Statements that insert NULL, or update a value to NULL, are automatically rejected. NOT NULL follows the column name and data type in a CREATE TABLE statement.

**2.8 INSERTING, UPDATING, AND DELETING ROWS**

### INSERT statement

The ***INSERT*** statement adds rows to a table. The INSERT statement has two clauses:

* The ***INSERT INTO*** clause names the table and columns where data is to be added. The keyword INTO is optional.
* The ***VALUES*** clause specifies the column values to be added.

The VALUES clause may list any number of rows in parentheses to insert multiple rows.

INSERT [INTO] TableName (Column1, Column2, ...)  
VALUES (Value1, Value2, ...);

### DEFAULT values

The optional ***DEFAULT*** keyword and default value follow the column name and data type in a CREATE TABLE statement. The column is assigned the default value, rather than NULL, when omitted from an INSERT statement.

### UPDATE statement

The ***UPDATE*** statement modifies existing rows in a table. The UPDATE statement uses the ***SET*** clause to specify the new column values. An optional WHERE clause specifies which rows are updated. Omitting the WHERE clause results in all rows being updated.

### DELETE statement

The ***DELETE*** statement deletes existing rows in a table. The ***FROM*** keyword is followed by the table name whose rows are to be deleted. An optional WHERE clause specifies which rows should be deleted. Omitting the WHERE clause results in all rows in the table being deleted.

*The* ***TRUNCATE*** *statement deletes all rows from a table. TRUNCATE is nearly identical to a DELETE statement with no WHERE clause except for minor differences that depend on the database system.*

The **MERGE** statement selects data from one table, called the source, and inserts the data to another table, called the target.

**2.9 PRIMARY KEYS**

### Primary keys

A ***primary key*** is a column, or group of columns, used to identify a row. The primary key is usually the table's first column and appears on the left of table diagrams, but the position is not significant to the database.

Often, primary key values are used in the WHERE clause to select a specific row.

The primary key is specified in SQL when the table is created. If a table contains several unique columns, any unique column, or group of columns, may be specified.

Primary keys must be:

* **Unique**. This rule ensures that each value identifies at most one row.
* **Not NULL**. This rule ensures that each value identifies at least one row.

Together, the two rules ensure that each primary key value identifies exactly one row.

### Composite primary keys

Sometimes multiple columns are necessary to identify a row. A ***simple primary key*** consists of a single column. A ***composite primary key*** consists of multiple columns. Composite primary keys are denoted with parentheses. Ex: (ColumnA, ColumnB).

Composite primary keys must be:

* **Unique**. Values of primary key columns, when grouped together, must be unique. No group of values may repeat in multiple rows.
* **Not NULL**. No column of a composite primary key may contain a NULL value.
* ***Minimal***. All primary key columns are necessary for uniqueness. When any column is removed, the resulting simple or composite column is no longer unique.

### PRIMARY KEY constraint

The ***PRIMARY KEY*** constraint in a CREATE TABLE statement names the table's primary key. The PRIMARY KEY constraint ensures that a column or group of columns is always unique and non-null.

### Auto-increment columns

An ***auto-increment column*** is a numeric column that is assigned an automatically incrementing value when a new row is inserted. The ***AUTO\_INCREMENT*** keyword defines an auto-increment column. AUTO\_INCREMENT follows the column's data type in a CREATE TABLE statement.

Database users occasionally make the following errors when inserting primary keys:

* Inserting values for auto-increment primary keys.
* Omitting values for primary keys that are not auto-increment columns.

MySQL allows insertion of a specific value to an auto-increment column. However, overriding auto-increment for a primary key is usually a mistake.

**2.10 FOREIGN KEYS**

### Foreign keys

A ***foreign key*** is a column, or group of columns, that refer to a primary key. The data types of the foreign and primary keys must be the same, but the names may be different.

Foreign keys do not obey the same rules as primary keys. Foreign key values may be repeated and may be NULL.

Foreign keys obey a relational rule called referential integrity. ***Referential integrity*** requires foreign key values must either be NULL or match some value of the referenced primary key.

### Special cases

Multiple foreign keys may refer to the same primary key.

A foreign key may refer to a primary key in the same table.

### FOREIGN KEY constraint

A foreign key constraint is added to a CREATE TABLE statement with the ***FOREIGN KEY*** and ***REFERENCES*** keywords. When a foreign key constraint is specified, the database rejects insert, update, and delete statements that violate referential integrity.

**2.11 REFERENTIAL INTEGRITY**

### Referential integrity rule

A ***fully NULL*** foreign key is a simple or composite foreign key in which all columns are NULL. ***Referential integrity*** is a relational rule that requires foreign key values are either fully NULL or match some primary key value.

In a relational database, foreign keys must obey referential integrity at all times. Occasionally, data entry errors or incomplete data result in referential integrity violations. Violations must be corrected before data is stored in the database.

### Referential integrity violations

Referential integrity can be violated in four ways:

1. A primary key is updated.
2. A foreign key is updated.
3. A row containing a primary key is deleted.
4. A row containing a foreign key is inserted.

### Referential integrity actions

An insert, update, or delete that violates referential integrity can be corrected manually. However, manual corrections are time-consuming and error-prone. Instead, databases automatically correct referential integrity violations with any of four actions, specified as SQL constraints:

* ***RESTRICT*** rejects an insert, update, or delete that violates referential integrity.
* ***SET NULL*** sets invalid foreign keys to NULL.
* ***SET DEFAULT*** sets invalid foreign keys to the foreign key default value.
* ***CASCADE*** propagates primary key changes to foreign keys.

CASCADE behaves differently for primary key updates and deletes. If a primary key is deleted, rows containing matching foreign keys are deleted. If a primary key is updated, matching foreign keys are updated to the same value.

RESTRICT, SET NULL, and SET DEFAULT apply to primary key update and delete, and foreign key insert and update. CASCADE applies to primary key update and delete only.

### ON UPDATE and ON DELETE clauses

For foreign key inserts and updates, MySQL supports only RESTRICT. Foreign key inserts and updates that violate referential integrity are automatically rejected.

For primary key updates and deletes, MySQL supports all four actions. Actions are specified in the optional ***ON UPDATE*** and ***ON DELETE*** clauses of the FOREIGN KEY constraint. ON UPDATE and ON DELETE are followed by either RESTRICT, SET NULL, SET DEFAULT, or CASCADE.

ON UPDATE and ON DELETE determine what happens to the foreign key when the referenced primary key is updated or deleted. When several foreign keys refer to the same primary key, different actions can be specified for each foreign key.

MySQL has several limitations on primary key updates and deletes:

* RESTRICT is applied when the ON UPDATE or ON DELETE clause is omitted.
* SET NULL cannot be used when a foreign key is not allowed NULL values.
* SET DEFAULT is not supported in some MySQL configurations.

**2.12 CONSTRAINTS**

### Column and table constraints

A ***constraint*** is a rule that governs allowable values in a database. Constraints are based on relational and business rules, and implemented with special keywords in a CREATE TABLE statement. The database automatically rejects insert, update, and delete statements that violate a constraint.

The following constraints are described elsewhere in this material:

* NOT NULL
* DEFAULT
* PRIMARY KEY
* FOREIGN KEY

A ***column constraint*** appears after the column name and data type in a CREATE TABLE statement. Column constraints govern values in a single column. Ex: NOT NULL is a column constraint.

A ***table constraint*** appears in a separate clause of a CREATE TABLE statement and governs values in one or more columns. Ex: FOREIGN KEY is a table constraint.

Some constraint types can be defined as either column or table constraints.

### UNIQUE constraint

The ***UNIQUE*** constraint ensures that values in a column, or group of columns, are unique. When applied to a single column, UNIQUE may appear either in the column declaration or a separate clause. When applied to a group of columns, UNIQUE is a table constraint and must appear in a separate clause.

### CHECK constraint

The ***CHECK*** constraint specifies an expression on one or more columns of a table. The constraint is violated when the expression is FALSE and satisfied when the expression is either TRUE or NULL.

### Adding and dropping constraints

Constraints are added and dropped with the ALTER TABLE TableName followed by an ADD, DROP, or CHANGE clause.

Unnamed constraints such as NOT NULL and DEFAULT are added or dropped with a CHANGE clause:

* CHANGE CurrentColumnName NewColumnName NewDataType [ConstraintDeclaration]

Named constraints are added with an ADD clause:

* ADD [CONSTRAINT ConstraintName] PRIMARY KEY (Column1, Column2 ...)
* ADD [CONSTRAINT ConstraintName] FOREIGN KEY (Column1, Column2 ...) REFERENCES TableName (Column)
* ADD [CONSTRAINT ConstraintName] UNIQUE (Column1, Column2 ...)
* ADD [CONSTRAINT ConstraintName] CHECK (expression)

Adding a constraint fails when the table contains data that violates the constraint.

Named constraints are dropped with a DROP clause:

* DROP PRIMARY KEY
* DROP FOREIGN KEY ConstraintName
* DROP INDEX ConstraintName (drops UNIQUE constraints)
* DROP CHECK ConstraintName
* DROP CONSTRAINT ConstraintName (drops any named constraint)

Dropping a table fails when a foreign key constraint refers to the table's primary key. Before dropping the table, either the foreign key constraint or the foreign key table must be dropped.

**3.1 SPECIAL OPERATORS AND CLAUSES**

### IN operator

The ***IN*** operator is used in a WHERE clause to determine if a value matches one of several values.

### BETWEEN operator

The ***BETWEEN*** operator provides an alternative way to determine if a value is between two other values. The operator is written value BETWEEN minValue AND maxValue and is equivalent to value >= minValue AND value <= maxValue.

### LIKE operator

The ***LIKE*** operator, when used in a WHERE clause, matches text against a pattern using the two wildcard characters % and \_.

* % matches any number of characters. Ex: LIKE 'L%t' matches "Lt", "Lot", "Lift", and "Lol cat".
* \_ matches exactly one character. Ex: LIKE 'L\_t' matches "Lot" and "Lit" but not "Lt" and "Loot".

The LIKE operator performs case-insensitive pattern matching by default or case-sensitive pattern matching if followed by the ***BINARY*** keyword. Ex: LIKE BINARY 'L%t' matches 'Left' but not 'left'.

To search for the wildcard characters % or \_, a backslash (\) must precede % or \_. Ex: LIKE 'a\%' matches "a%".

### DISTINCT clause

The ***DISTINCT*** clause is used with a SELECT statement to return only unique or 'distinct' values.

### ORDER BY clause

A SELECT statement selects rows from a table with no guarantee the data will come back in a certain order. The ***ORDER BY*** clause orders selected rows by one or more columns in ascending (alphabetic or increasing) order. The ***DESC*** keyword with the ORDER BY clause orders rows in descending order.

**3.2 SIMPLE FUNCTIONS**

### Numeric functions

A ***function*** operates on an expression enclosed in parentheses, called an ***argument***, and returns a value. Usually, the argument is a simple expression, such as a column name or fixed value. Some functions have several arguments, separated by commas, and a few have no arguments at all.

Each function operates on, and evaluates to, specific data types.

|  |  |  |
| --- | --- | --- |
| Function | Description | Example |
| ***ABS(n)*** | Returns the absolute value of *n* | SELECT ABS(-5);  returns 5 |
| ***LOG(n)*** | Returns the natural logarithm of *n* | SELECT LOG(10);  returns 2.302585092994046 |
| ***POW(x, y)*** | Returns *x* to the power of *y* | SELECT POW(2, 3);  returns 8 |
| ***RAND()*** | Returns a random number between 0 (inclusive) and 1 (exclusive) | SELECT RAND();  returns 0.11831825703225868 |
| ***ROUND(n, d)*** | Returns *n* rounded to *d* decimal places | SELECT ROUND(16.25, 1);  returns 16.3 |
| ***SQRT(n)*** | Returns the square root of *n* | SELECT SQRT(25);  returns 5 |

### String functions

String functions manipulate string values. SQL string functions are similar to string functions in programming languages like Java and Python.

|  |  |  |
| --- | --- | --- |
| Function | Description | Example |
| ***CONCAT(s1, s2, ...)*** | Returns the string that results from concatenating the string arguments | SELECT CONCAT('Dis', 'en', 'gage');  returns 'Disengage' |
| ***LOWER(s)*** | Returns the lowercase *s* | SELECT LOWER('MySQL');  returns 'mysql' |
| ***REPLACE(s, from, to)*** | Returns the string *s* with all occurrences of *from* replaced with *to* | SELECT REPLACE('This and that', 'and', 'or');  returns 'This or that' |
| ***SUBSTRING(s, pos, len)*** | Returns the substring from *s* that starts at position *pos* and has length *len* | SELECT SUBSTRING('Boomerang', 1, 4);  returns 'Boom' |
| ***TRIM(s)*** | Returns the string *s* without leading and trailing spaces | SELECT TRIM(' test ');  returns 'test' |
| ***UPPER(s)*** | Returns the uppercase *s* | SELECT UPPER('mysql');  returns 'MYSQL' |

### Date and time functions

Date and time functions operate on DATE, TIME, and DATETIME data types.

|  |  |  |
| --- | --- | --- |
| Function | Description | Example |
| ***CURDATE()*** ***CURTIME()*** ***NOW()*** | Returns the current date, time, or date and time in 'YYYY-MM-DD', 'HH:MM:SS', or 'YYYY-MM-DD HH:MM:SS' format | SELECT CURDATE();  returns '2019-01-25'  SELECT CURTIME();  returns '21:05:44'  SELECT NOW();  returns '2019-01-25 21:05:44' |
| ***DATE(expr)*** ***TIME(expr)*** | Extracts the date or time from a date or datetime expression *expr* | SELECT DATE('2013-03-25 22:11:45');  returns '2013-03-25'  SELECT TIME('2013-03-25 22:11:45');  returns '22:11:45' |
| ***DAY(d)*** ***MONTH(d)*** ***YEAR(d)*** | Returns the day, month, or year from date *d* | SELECT DAY('2016-10-25');  returns 25  SELECT MONTH('2016-10-25');  returns 10  SELECT YEAR('2016-10-25');  returns 2016 |
| ***HOUR(t)*** ***MINUTE(t)*** ***SECOND(t)*** | Returns the hour, minute, or second from time *t* | SELECT HOUR('22:11:45');  returns 22  SELECT MINUTE('22:11:45');  returns 11  SELECT SECOND('22:11:45');  returns 45 |
| ***DATEDIFF(expr1, expr2)*** ***TIMEDIFF(expr1, expr2)*** | Returns *expr1 - expr2* in number of days or time values, given *expr1* and *expr2* are date, time, or datetime values | SELECT DATEDIFF('2013-03-10', '2013-03-04');  returns 6  SELECT TIMEDIFF('10:00:00', '09:45:30');  returns 00:14:30 |

**3.3 AGGREGATE FUNCTIONS**

### Aggregate functions

An ***aggregate function*** processes values from a set of rows and returns a summary value. Common aggregate functions are:

* ***COUNT()*** counts the number of rows in the set.
* ***MIN()*** finds the minimum value in the set.
* ***MAX()*** finds the maximum value in the set.
* ***SUM()*** sums all the values in the set.
* ***AVG()*** computes the arithmetic mean of all the values in the set.

Aggregate functions appear in a SELECT clause and process all rows that satisfy the WHERE clause condition. If a SELECT statement has no WHERE clause, the aggregate function processes all rows.

### GROUP BY clause

Aggregate functions are commonly used with the GROUP BY clause.

The ***GROUP BY*** clause consists of the GROUP BY keyword and one or more columns. Each simple or composite value of the column(s) becomes a group. The query computes the aggregate function separately, and returns one row, for each group.

### HAVING clause

The *HAVING* clause is used with the GROUP BY clause to filter group results. The optional HAVING clause follows the GROUP BY clause and precedes the optional ORDER BY clause.

**3.4 JOIN QUERIES**

### Joins

In relational databases, reports are commonly generated from data in multiple tables. Multi-table reports are written with join statements.

A ***join*** is a SELECT statement that combines data from two tables, known as the ***left table*** and ***right table***, into a single result. The tables are combined by comparing columns from the left and right tables, usually with the = operator. The columns must have comparable data types.

Usually, a join compares a foreign key of one table to the primary key of another. However, a join can compare any columns with comparable data types.

### Prefixes and aliases

Occasionally, join tables contain columns with the same name. When duplicate column names appear in a query, the names must be distinguished with a prefix. The prefix is the table name followed by a period.

Use of a prefix makes column names more complex. To simplify queries or result tables, a column name can be replaced with an alias. The alias follows the column name, separated by an optional ***AS*** keyword.

### Inner and full joins

A ***join clause*** determines how a join query handles unmatched rows. Two common join clauses are:

* ***INNER JOIN*** selects only matching left and right table rows.
* ***FULL JOIN*** selects all left and right table rows, regardless of match.

In a FULL JOIN result table, unmatched left table rows appear with NULL values in right table columns, and vice versa.

The join clause appears between a FROM clause and an ON clause:

* The FROM clause specifies the left table.
* The INNER JOIN or FULL JOIN clause specifies the right table.
* The ***ON*** clause specifies the join columns.

An optional WHERE clause follows the ON clause.

### Left and right joins

In some cases, the database user wants to see unmatched rows from either the left or right table, but not both. To enable these cases, relational databases support left and right joins:

* ***LEFT JOIN*** selects all left table rows, but only matching right table rows.
* ***RIGHT JOIN*** selects all right table rows, but only matching left table rows.

An ***outer join*** is any join that selects unmatched rows, including left, right, and full joins.

The ***UNION*** keyword combines the two results into one table.

**3.5 EQUIJOINS, SELF-JOINS, AND CROSS-JOINS**

### Equijoins

An ***equijoin*** compares columns of two tables with the = operator. Most joins are equijoins. A ***non-equijoin*** compares columns with an operator other than =, such as < and >.

### Self-joins

A ***self-join*** joins a table to itself. A self-join can compare any columns of a table, as long as the columns have comparable data types. If a foreign key and the referenced primary key are in the same table, a self-join commonly compares those key columns. In a self-join, aliases are necessary to distinguish left and right tables.

### Cross-joins

A ***cross-join*** combines two tables without comparing columns. A cross-join uses a ***CROSS JOIN*** clause without an ON clause. As a result, all possible combinations of rows from both tables appear in the result.

**3.6 SUBQUERIES**

### Subqueries

A ***subquery***, sometimes called a ***nested query*** or ***inner query***, is a query within another SQL query. The subquery is typically used in a SELECT statement's WHERE clause to return data to the outer query and restrict the selected results. The subquery is placed inside parentheses ().

### Correlated subqueries

A subquery is ***correlated*** when the subquery's WHERE clause references a column from the outer query. In a correlated subquery, the rows selected depend on what row is currently being examined by the outer query.

An alias can also help differentiate the columns. An ***alias*** is a temporary name assigned to a column or table. The ***AS*** keyword follows a column or table name to create an alias.

### EXISTS operator

Correlated subqueries commonly use the ***EXISTS*** operator, which returns TRUE if a subquery selects at least one row and FALSE if no rows are selected. The ***NOT EXISTS*** operator returns TRUE if a subquery selects no rows and FALSE if at least one row is selected.

### Flattening subqueries

Many subqueries can be rewritten as a join. Most databases optimize a subquery and outer query separately, whereas joins are optimized in one pass. So joins are usually faster and preferred when performance is a concern.

Replacing a subquery with an equivalent join is called ***flattening*** a query.

**3.7 COMPLEX QUERY EXAMPLE**

**3.8 VIEW TABLES**

### Creating views

Table design is optimized for a variety of reasons, such as minimal storage, fast query execution, and support for relational and business rules. Occasionally, the design is not ideal for database users and programmers.

View tables solve this problem. Views restructure table columns and data types without changes to the underlying database design.

A ***view table*** is a table name associated with a SELECT statement, called the ***view query***. The ***CREATE VIEW*** statement creates a view table and specifies the view name, query, and, optionally, column names. If column names are not specified, column names are the same as in the view query result table.

### Querying views

A table specified in the view query's FROM clause is called a ***base table***. Unlike base table data, view table data is not normally stored. Instead, when a view table appears in an SQL statement, the view query is merged with the SQL query. The database executes the merged query against base tables.

In some databases, view data can be stored. A ***materialized view*** is a view for which data is stored at all times. Whenever a base table changes, the corresponding view tables can also change, so materialized views must be refreshed.

### Advantages of views

View tables have several advantages:

* **Protect sensitive data**. A table may contain sensitive data. A view can exclude sensitive columns but include all other columns. Authorizing users and programmers access to the view but not the underlying table protects the sensitive data.
* **Save complex queries**. Complex SELECT statements can be saved as a view. Database users can reference the view without writing the SELECT statement.
* **Save optimized queries**. Often, the same result table can be generated with equivalent SELECT statements. Although the results of equivalent statements are the same, performance may vary. To ensure fast execution, the optimal statement can be saved as a view and distributed to database users.

For the above reasons, views are supported in all relational databases and are frequently created by database administrators. Database users need not be aware of the difference between view and base tables.

### Inserting, updating, and deleting views

View tables are commonly used in SELECT statements. Using views in INSERT, UPDATE, and DELETE statements is problematic:

* **Primary keys**. If a base table primary key does not appear in a view, an insert to the view generates a NULL primary key value. Since primary keys may not be NULL, the insert is not allowed.
* **Aggregate values**. A view query may contain aggregate functions such as AVG() or SUM(). One aggregate value corresponds to many base table values. An update or insert to the view may create a new aggregate value, which must be converted to many base table values. The conversion is undefined, so the insert or update is not allowed.
* **Join views**. In a join view, foreign keys of one base table may match primary keys of another. A delete from a view might delete foreign key rows only, or primary key rows only, or both the primary and foreign key rows. The effect of the join view delete is undefined and therefore not allowed.

### WITH CHECK OPTION clause

Databases that allow view updates face one particularly bothersome behavior. A view insert or update may create a row that does not satisfy the view query WHERE clause. In this case, the inserted or updated row does not appear in the view table. From the perspective of the database user, the insert or update appears to fail even though the base tables have changed.

To prevent inserts or updates that appear to fail, databases that support view updates have an optional WITH CHECK OPTION clause. When ***WITH CHECK OPTION*** is specified, the database rejects inserts and updates that do not satisfy the view query WHERE clause. Instead, the database generates an error message that explains the violation.

**3.9 RELATIONAL ALGEBRA**

Relational algebra has nine operations. Each operation is denoted with a special symbol, often a letter of the Greek alphabet. Operation symbols can be combined with tables in expressions, just as + - × / can be combined with numbers in arithmetic expressions. Each relational algebra expression is equivalent to an SQL query and defines a single result table.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Symbol | Greek letter | Derivation |
| Select | *�* | sigma | corresponds to Latin letter s, for Select |
| Project | *Π* | Pi | corresponds to Latin letter P, for Project |
| Product | *×* |  | multiplication symbol |
| Join | *⋈* |  | multiplication symbol with vertical bars |
| Union | *∪* |  | set theory |
| Intersect | *∩* |  | set theory |
| Difference | *−* |  | set theory |
| Rename | *�* | rho | corresponds to Latin letter r, for Rename |
| Aggregate | *�* | gamma | corresponds to Latin letter g, for group |

**4.1 ENTITIES, RELATIONSHIPS, AND ATTRIBUTES**

An ***entity-relationship model*** is a high-level representation of data requirements, ignoring implementation details. An entity-relationship model guides implementation in a particular database system, such as MySQL.

An entity-relationship model includes three kinds of objects:

* An ***entity*** is a person, place, product, concept, or activity.
* A ***relationship*** is a statement about two entities.
* An ***attribute*** is a descript­ive property of an entity.

A relationship is usually a statement about two different entities, but the two entities may be the same. A ***reflexive relationship*** relates an entity to itself.

When the model is implemented in SQL, entities typically become tables. Relationships and attributes typically become foreign keys and columns, respectively.

### Entity-relationship diagram and glossary

An ***entity-relationship diagram***, commonly called an ***ER diagram***, is a schematic picture of entities, relationships, and attributes. Entities are drawn as rectangles. Relationships are drawn as lines connecting rectangles. Attributes appear as additional text within an entity rectangle, under the entity name.

Entities and relationships always appear in ER diagrams. Attributes are optional and only appear when additional detail is needed.

A ***glossary***, also known as a ***data dictionary*** or ***repository***, documents additional detail in text format. A glossary includes names, synonyms, and descriptions of entities, relationships, and attributes.

### Types and instances

In entity-relationship modeling, a type is a set:

* An ***entity type*** is a set of things. Ex: All employees in a company.
* A ***relationship type*** is a set of related things. Ex: Employee-Manages-Department is a set of (employee, department) pairs, where the employee manages the department.
* An ***attribute type*** is a set of values. Ex: All employee salaries.

Entity, relationship, and attribute types usually become tables, foreign keys, and columns, respectively.

An instance is an element of a set:

* An ***entity instance*** is an individual thing. Ex: The employee Sam Snead.
* A ***relationship instance*** is a statement about entity instances. Ex: "Maria Rodriguez manages Sales."
* An ***attribute instance*** is an individual value. Ex: The salary $35,000.

Entity, relationship, and attribute instances usually become rows, foreign key values, and column values, respectively

### Database design

Complex databases are developed in three phases:

1. ***Analysis*** develops an entity-relationship model, capturing data requirements while ignoring implementation details.
2. ***Logical design*** converts the entity-relationship model into tables, columns, and keys for a particular database system.
3. ***Physical design*** adds indexes and specifies how tables are organized on storage media.

Analysis is particularly important for complex databases with many users when documenting requirements is challenging. For small databases with just a few tables and users, analysis is less important and often omitted.

Analysis and logical design steps are summarized in the table below. Although these steps are presented in sequence, in practice execution is not always sequential. Often an early step is revisited after a later step is completed.

Analysis steps

|  |  |
| --- | --- |
| Step | Name |
| 1 | Discover entities, relationships, and attributes |
| 2 | Determine cardinality |
| 3 | Distinguish strong and weak entities |
| 4 | Create supertype and subtype entities |

Logical design steps

|  |  |
| --- | --- |
| Step | Name |
| 5 | Implement entities |
| 6 | Implement relationships |
| 7 | Implement attributes |
| 8 | Apply normal form |

**4.2 DISCOVERY**

Entities, relationships, and attributes are discovered in interviews with database users and managers. Users and managers are usually familiar with data requirements from an old database, or perhaps a manual process with paper records. When users are difficult to reach, a database designer may communicate with surrogates. Ex: A sales representative might communicate on behalf of prospective customers.

In addition to interviews, written documents are a good source of data requirements. Ex: The user manual for an older version of the database is a good source of requirements.

In interviews and documents, entities, relationships, and attributes surface as nouns and verbs:

* Entities usually appear as nouns, but not all nouns are entities. Designers should ignore nouns that denote specific data or are not relevant to the database.
* Relationships are often expressed as verbs. Designers should ignore statements that are not about entities, not relevant to the database, or redundant to other relationships. Designers should look for relationships that are not explicitly stated, since users may overlook important information.
* Attributes are usually nouns that denote specific data, such as names, dates, quantities, and monetary values.

### Names

Entity names are a singular noun. Ex: Employee rather than Employees. The best names are commonly used and easily understood by database users.

Relationships names have the form Entity-Verb-Entity, such as Division-Contains-Department. When the related entities are obvious, in ER diagrams or informal conversation, Verb is sufficient and entity names can be omitted. The verb should be active rather than passive. Ex: Manages rather than IsManagedBy. Occasionally, the same verb relates different entity pairs. Ex: Order-Contains-LineItem and Division-Contains-Department.

Attribute names have the form EntityQualifierType, such as EmployeeFirstName:

* Entity is the name of the entity that the attribute describes. When the entity is obvious, in ER diagrams or informal conversation, QualifierType is sufficient and the entity name can be omitted.
* Qualifier describes the meaning of the attribute. Ex: First, Last, and Alternate. Sometimes a qualifier is unnecessary and can be omitted. Ex: StudentNumber.
* Type is chosen from a list of standard attribute types such as Name, Number, and Count. Attribute types are not identical to SQL data types. Ex: "Amount" might be an attribute type representing monetary values, implemented as the MONEY data type in SQL. "Count" might be an attribute type representing quantity, implemented as NUMBER in SQL.

### Synonyms and descriptions

Often, entity, relationship, and attribute names have synonyms. Ex: Representative may be a synonym for SalesAgent. Synonyms are common in informal communications. To avoid confusion, one official name is selected for each entity, relationship, and attribute. Other names are documented in the glossary as synonyms.

The glossary also contains complete descriptions of entities, relationships, and attributes. The description states the meaning of each entity, relationship, or attribute in complete sentences. The description begins with the name and includes examples and counterexamples to illustrate usage.

### Database design

The first step of the analysis phase is discovery of entities, relationships, and attributes in interviews and document review. As discovery proceeds, the designer draws an ER diagram, determines standard attributed types, and documents names, synonyms, and descriptions in the glossary.

Although the step numbers suggest a sequence, database designers commonly move back and forth between steps. As names, synonyms, and descriptions are documented, additional entities, relationships, and attributes are discovered. The ER diagram and glossary are usually developed in parallel.

**4.3 CARDINALITY**

### Relationship maximum

In entity-relationship modeling, *cardinality* refers to maxima and minima of relationships and attributes.

***Relationship maximum*** is the greatest number of instances of one entity that can relate to a single instance of another entity. A relationship has two maxima, one for each of the related entities. Maxima are usually specified as one or many. A related entity is ***singular*** when the maximum is one and ***plural*** when the maximum is many.

### Relationship minimum

***Relationship minimum*** is the least number of instances of one entity that can relate to a single instance of another entity. A relationship has two minima, one for each of the related entities. Minima are usually specified as zero or one. A related entity is ***optional*** when the minimum is zero and ***required*** when the minimum is one.

### Attribute maximum and minimum

***Attribute maximum*** is the greatest number of attribute values that can describe each entity instance. Attribute maximum is usually specified as one (singular) or many (plural).

***Attribute minimum*** is the least number of attribute values that can describe each entity instance. Attribute minimum is usually specified as zero (optional) or one (required).

In ER diagrams, attribute maximum and minimum follow the attribute name. The minimum appears in parentheses.

### Unique attributes

Each value of a ***unique attribute*** describes at most one entity instance. A unique attribute is not the same as a singular attribute:

* A unique attribute has at most one entity instance for each attribute value.
* A singular attribute has at most one attribute value for each entity instance.

In ER diagrams, 1 indicates a unique attribute and M indicates a non-unique attribute. The 1 or M appears before the attribute maximum and minimum.

**4.4 STRONG AND WEAK ENTITIES**

### Strong entities

An ***identifying attribute*** is unique, singular, and required. Identifying attribute values correspond one-to-one to, or ***identify***, entity instances.

A ***strong entity*** has one or more identifying attributes. When a strong entity is implemented as a table, one of the identifying attributes may become the primary key.

### Weak entities

A ***weak entity*** does not have an identifying attribute. Instead, a weak entity usually has a relationship, called an ***identifying relationship***, to another entity, called an ***identifying entity***. Cardinality of the identifying entity is 1(1).

In an ER diagram, an identifying relationship has a diamond next to the identifying entity. Cardinality of the identifying entity is always 1(1), so the diamond replaces the cardinality symbol.

### Identifying entities

A weak entity is usually identified by a strong entity. However, a weak entity can be identified by another weak entity or by several entities.

**4.5 SUPERTYPE AND SUBTYPE ENTITIES**

### Supertype and subtype entities

An entity type is a set of entity instances. A ***subtype entity*** is a subset of another entity type, called the ***supertype entity***. Ex: Managers are a subset of employees, so Manager is a subtype entity of the Employee supertype entity. On ER diagrams, subtype entities are drawn within the supertype.

A supertype entity usually has several subtypes. Attributes of the supertype apply to all subtypes. Attributes of a subtype do not apply to other subtypes or the supertype.

A supertype entity identifies its subtype entities. The identifying relationship is called an ***IsA relationship***. Ex: Manager-IsAn-Employee relates each manager instance to the corresponding employee instance. Since a supertype entity always identifies its subtypes, the IsA relationship is assumed and can be omitted from the ER diagram.

### Similar entities and optional attributes

Supertype and subtype entities are often created from similar entities and optional attributes.

***Similar entities*** are entities that have many common attributes and relationships. Similar entities become subtypes of a new supertype entity,

### Partitions

A ***partition*** of a supertype entity is a group of mutually exclusive subtype entities. A supertype entity can have several partitions. Subtype entities within each partition are disjoint and do not share instances. Subtype entities in different partitions overlap and do share instances.

In diagrams, subtype entities within each partition are vertically aligned. Subtype entities in different partitions are horizontally aligned.

Each partition corresponds to an optional ***partition attribute*** of the supertype entity. The partition attribute indicates which subtype entity is associated with each supertype instance.

### Database design

After entities, relationships, attributes, cardinality, and strong and weak entities are determined, the database designer looks for supertype and subtype entities. Similar entities and optional attributes suggest new supertype and subtype entities and warrant special attention. Mutually exclusive subtype entities are grouped into partitions. For each partition, a partition attribute is added to the supertype entity.

Creating supertype and subtype entities is the last of four analysis steps:

1. Discover entities, relationships, and attributes
2. Determine cardinality
3. Distinguish strong and weak entities
4. Create supertype and subtype entities

Logical design follows analysis. Logical design converts an entity-relationship model to tables, columns, and keys for a specific database system.

**4.6 ALTERNATIVE MODELING CONVENTIONS**

### Diagram conventions

ER diagram conventions vary widely. Ex: Some ER diagrams may:

* Depict relationship names inside a diamond.
* Depict weak entities and identifying relationships with double lines.
* Depict subtype entities with IsA relationships rather than inside of supertype entities.
* Use color, dashed lines, or double lines to convey additional information.

Variations in cardinality conventions are common. One popular convention, called ***crow's foot notation***, depicts cardinality as a circle (zero), a short line (one), or three short lines (many). The three short lines look like a bird's foot, hence the name "crow's foot notation".

### Model conventions

ER modeling concepts also vary. Ex: Some ER models may:

* Allow relationships between three or more entities.
* Decompose a complex model into a group of related entities, called a ***subject area***.
* Refer to strong entities as ***independent*** and weak entities as ***dependent***.

Several model conventions are standardized and widely used. Leading conventions include:

* ***Unified Modeling Language***, or ***UML***, is commonly used for software development. Software data structures are similar to database structures, so UML includes ER conventions.
* ***IDEF1X*** stands for Information DEFinition version 1X. IDEF1X became popular, in part, due to early adoption by the United States Department of Defense.
* ***Chen notation*** appeared in an early ER modeling paper by Peter Chen. Chen notation is not standardized but often appears in literature and tools.

By and large, differences between conventions are stylistic rather than substantial. The choice of convention does not usually affect the resulting database design.

An **intangible entity** is documented in the data model, but not tracked with data in the database. Ex: In the animations above, the Part entity is intangible. In an ER diagram, intangible entities are distinguished with special notation, such as a dashed rectangle or distinct color.

**4.7 IMPLEMENTING ENTITIES**

### Selecting primary keys

In the first step of the logical design phase, each entity becomes a table and each attribute becomes a column. Tables and columns are revised in subsequent steps.

As tables and columns are specified, primary keys are selected. Primary keys must be unique and not NULL, and thus correspond to unique and required attributes. Primary keys should also be:

* **Stable**. Primary key values should not change. When a primary key value changes, statements that specify the old value must also change. Furthermore, the new primary key value must cascade to matching foreign keys.
* **Simple**. Primary key values should be easy to type and store. Small values are easy to specify in an SQL WHERE clause and speed up query processing. Ex: A 2-byte integer is easier to type and faster to process than a 15-byte character string.
* **Meaningless**. Primary keys should not contain descriptive information. Descriptive information occasionally changes, so primary keys containing descriptive information are unstable.

Stable, simple, and meaningless primary keys are desirable but not required. Depending on database standards, these guidelines may be violated in some cases.

### Implementing strong entities

A strong entity becomes a ***strong table***. The primary key must be unique and non-NULL, and should be stable, simple, and meaningless. Single-column primary keys are best, but if no such column exists, a composite primary key may have the required properties.

An ***artificial key*** is a single-column primary key created by the database designer when no suitable single-column or composite primary key exists. Usually artificial key values are integers, generated automatically by the database as new rows are inserted to the table. Artificial keys are stable, simple, and meaningless.

### Implementing subtype entities

A subtype entity becomes a ***subtype table*** and is implemented as follows:

* The primary key is identical to the supertype primary key.
* The primary key is also a foreign key that references the supertype primary key.

The foreign key implements the **IsA** identifying relationship. Foreign keys that implement identifying relationships usually have the following referential integrity actions:

* Cascade on primary key update and delete
* Restrict on foreign key insert and update

**4.8 IMPLEMENTING RELATIONSHIPS**

### Implementing many-one relationships

The 'implement relationships' step converts relationships into keys or tables, depending on relationship cardinality.

A many-one or one-many relationship becomes a foreign key:

* The foreign key goes in the table on the 'many' side of the relationship.
* The foreign key refers to the primary key on the 'one' side.
* The foreign key name is the primary key name with an optional prefix. The prefix is derived from the relationship name and clarifies the meaning of the foreign key.

### Implementing one-one relationships

A one-one relationship becomes a foreign key:

* The foreign key can go in the table on either side of the relationship. Usually, the foreign key is placed in the table with fewer rows, to minimize the number of NULL values.
* The foreign key refers to the primary key on the opposite side of the relationship.
* The foreign key name is the primary key name with an optional prefix. The prefix is derived from the relationship name and clarifies the meaning of the foreign key.

### Implementing many-many relationships

A many-many relationship becomes a new weak table:

* The new table contains two foreign keys, referring to the primary keys of the related tables.
* The primary key of the new table is the composite of the two foreign keys.
* The new table is identified by the related tables, so primary key cascade and foreign key restrict rules are specified.
* The new table name consists of the related table names with an optional qualifier in between. The qualifier is derived from the relationship name and clarifies the meaning of the table.

### Database design

The 'implement relationships' step adds foreign keys to the initial table design. Each many-one and one-one relationship becomes a new foreign key. Each many-many relationship becomes a new dependent table containing two foreign keys.

Foreign keys that implement dependency relationships usually have the following referential integrity actions:

* Cascade on primary key update and delete
* Restrict on foreign key insert and update

**4.9 IMPLEMENTING ATTRIBUTES**

### Implementing plural attributes

In the 'implement entities' step, entities become tables and attributes become columns. Singular attributes remain in the initial table, but plural attributes move to a new weak table:

* The new table contains the plural attribute and a foreign key referencing the initial table.
* The primary key of the new table is the composite of the plural attribute and the foreign key.
* The new table is identified by the initial table, so primary key cascade and foreign key restrict rules are specified.
* The new table name consists of the initial table name followed by the attribute name.

If a plural attribute has a small, fixed maximum, the plural attribute can be implemented as multiple columns in the initial table. However, implementing plural attributes in a new table simplifies queries and is usually a better solution.

**4.10 FIRST, SECOND, AND THIRD NORMAL FORM**

### Functional dependence

Column A ***depends on*** column B means each B value is related to at most one A value. Columns A and B may be simple or composite. 'A depends on B' is denoted B → A.

Dependence of one column on another is called ***functional dependence***. Functional dependence reflects business rules.

### Normal forms

***Redundancy*** is the repetition of related values in a table. Redundancy causes database management problems. When related values are updated, all copies must be changed, which makes queries slow and complex. If copies are not updated uniformly, the copies become inconsistent and the correct version is uncertain.

**Normal forms** are rules for designing tables with less redundancy. Normal forms are numbered, first through fifth. An additional normal form, Boyce-Codd, is an improved version of third normal form. The six normal forms comprise a sequence, with each successive normal form allowing less redundancy.

### First normal form

Every cell of a table contains exactly one value. A table is in ***first normal form*** when, in addition, the table has a primary key. This definition has two corollaries:

* **In a first normal form table, every non-key column depends on the primary key**. Each primary key value appears in exactly one row, and each non-key cell contains exactly one value. So each primary key value is related to exactly one non-key value.
* **A first normal form table has no duplicate rows**. Every row contains a different primary key value and therefore every row is different.

### Second normal form

A table is in ***second normal form*** when all non-key columns depend on the whole primary key. In other words, a non-key column cannot depend on part of a composite primary key. A table with a simple primary key is automatically in second normal form.

### Third normal form

Redundancy can occur in a second normal form table when a non-key column depends on another non-key column. Informally, a table is in ***third normal form*** when all non-key columns depend on the key, the whole key, and nothing but the key.

**4.11 BOYCE-CODD NORMAL FORM**

In a Boyce-Codd normal form table, all dependencies are on unique columns. Dependence on a unique column never creates redundancy, so Boyce-Codd normal form eliminates all redundancy arising from functional dependence.

A ***candidate key*** is a simple or composite column that is unique and minimal. ***Minimal*** means all columns are necessary for uniqueness. A table may have several candidate keys. The database designer designates one candidate key as the primary key.

A ***non-key*** column is a column that is not contained in a candidate key.

A table is in ***third normal form*** if, whenever a non-key column A depends on column B, then B is unique. Columns A and B may be simple or composite. Although B is unique, B is not necessarily minimal and therefore is not necessarily a candidate key.

### Boyce-Codd normal form

The definition of third normal form applies to *non-key* columns only, which allows for occasional redundancy. Boyce-Codd normal form applies to *all* columns and eliminates this redundancy.

A table is in ***Boyce-Codd normal form*** if, whenever column A depends on column B, then B is unique. Columns A and B may be simple or composite. This definition is identical to the definition of third normal form with the term 'non-key' removed.

Boyce-Codd normal form is considered the gold standard of table design.

Trivial dependencies

*When the columns of A are a subset of the columns of B, A always depends on B. Ex: FareClass depends on (FlightCode, FareClass). These dependencies are called* ***trivial****.*

**4.12 APPLYING NORMAL FORM**

### Normalization

Implementing entities, relationships, and attributes usually generates tables with no redundancy. Occasionally, however, implementation results in redundant tables. This redundancy is eliminated with normalization, the last step of logical design.

***Normalization*** eliminates redundancy by decomposing a table into two or more tables in higher normal form. Ex: A table in first normal form might be replaced by two tables in third normal form. In principle, normalization decomposes tables to any higher normal form.

Column A ***depends on*** column B when each B value is related to at most one A value. A and B may be simple or composite columns. In a ***Boyce-Codd normal form*** table, if column A depends on column B, then B must be unique. Normalizing a table to Boyce-Codd normal form involves three steps:

1. **List all unique columns**. Unique columns may be simple or composite. In composite columns, remove any columns that are not necessary for uniqueness. The primary key is unique and therefore always on this list.
2. **Identify dependencies on non-unique columns**. Non-unique columns are either **external** to all unique columns or **contained within** a composite unique column.
3. **Eliminate dependencies on non-unique columns**. If column A depends on a non-unique column B, A is removed from the original table. A new table is created containing A and B. B is a primary key in the new table and a foreign key in the original table.

Normalization eliminates redundancy by removing A from the original table. Since the data relating A and B is recorded in a new table, no information is lost.

### Denormalization

Boyce-Codd normal form is ideal for tables with frequent inserts, updates, and deletes. In a database used primarily for reporting, changes are infrequent and redundancy is acceptable. In fact, redundancy can be desirable in reporting databases, as processing is faster and queries are simpler. Therefore, reporting databases may contain tables that, by design, are not in third normal form.

***Denormalization*** means intentionally introducing redundancy by merging tables. Denormalization eliminates join queries and therefore improves query performance. Denormalization results in first and second normal form tables and should be applied selectively and cautiously.

### Database design

As tables and keys are specified, the database designer reviews each table for Boyce-Codd normal form. Dependencies and unique columns are identified. If any dependencies are not on unique columns, the table is decomposed into smaller tables in Boyce-Codd normal form. Tables that experience infrequent inserts, updates, and deletes may be denormalized to simplify and accelerate SELECT queries.

**5.1 STORAGE MEDIA**

Databases and file systems use a uniform size, called a ***block***, when transferring data between main memory and storage media. Block size is independent of storage media. Storage media are managed by controllers, which convert between blocks and sectors or pages. This conversion is internal to the storage device, so the database is unaware of page and sector sizes.

### Row-oriented storage

Most relational databases are optimized for transactional applications, which often read and write individual rows. To minimize block transfers, relational databases usually store an entire row within one block, which is called ***row-oriented storage***.

Row-oriented storage performs best when row size is small relative to block size, for two reasons:

* **Improved query performance.** When row size is small relative to block size, each block contains many rows. Queries that read and write multiple rows transfer fewer blocks, resulting in better performance.
* **Less wasted storage.** Row-oriented storage wastes a few bytes per block, since rows do not usually fit evenly into the available space. The wasted space is less than the row size. If row size is small relative to block size, this wasted space is insignificant.

### Column-oriented storage

Some newer relational databases are optimized for analytic applications rather than transactional applications. Analytic applications often read just a few columns from many rows. In this case, column-oriented storage is optimal. In ***column-oriented*** storage, also called ***columnar storage***, each block stores values for a single column only.

Column-oriented storage benefits analytic applications in several ways:

* **Faster data access**. More column values are transferred per block, reducing time to access storage media.
* **Better data compression**. Databases often apply data compression algorithms when storing data. Data compression is usually more effective when all values have the same data type. As a result, more values are stored per block, which reduces storage and access time.

With column-oriented storage, reading or writing an entire row requires accessing multiple blocks. Consequently, column-oriented storage is a poor design for most transactional applications.

**5.2 TABLE STRUCTURES**

### Heap table

Row-oriented storage performs better than column-oriented storage for most transactional databases. Consequently, relational databases commonly use row-oriented storage. A ***table structure*** is a scheme for organizing rows in blocks on storage media.

Databases commonly support four alternative table structures:

* Heap table
* Sorted table
* Hash table
* Table cluster

Each table in a database can have a different structure. Databases assign a default structure to all tables. Database administrators can override the default structure to optimize performance for specific queries.

In a ***heap table***, no order is imposed on rows. The database maintains a list of blocks assigned to the table, along with the address of the first available space for inserts. If all blocks are full, the database allocates a new block and inserts rows in the new block.

Heap tables optimize insert operations. Heap tables are particularly fast for bulk load of many rows, since rows are stored in load order. Heap tables are not optimal for queries that read rows in a specific order, such as a range of primary key values, since rows are scattered randomly across storage media.

### Sorted table

In a ***sorted table***, the database designer identifies a ***sort column*** that determines physical row order. The sort column is usually the primary key but can be a non-key column or group of columns.

Rows are assigned to blocks according to the value of the sort column. Each block contains all rows with values in a given range. Within each block, rows are located in order of sort column values.

Sorted tables are optimal for queries that read data in order of the sort column, such as:

* JOIN on the sort column
* SELECT with range of sort column values in the WHERE clause
* SELECT with ORDER BY the sort column

In summary, sorted tables are optimized for read queries at the expense of insert and update operations. Since reads are more frequent than updates and inserts in many databases, sorted tables are often used, usually with the primary key as the sort column.

### Hash table

In a ***hash table***, rows are assigned to buckets. A ***bucket*** is a block or group of blocks containing rows. Initially, each bucket has one block. As a table grows, some buckets eventually fill up with rows, and the database allocates additional blocks. New blocks are linked to the initial block, and the bucket becomes a chain of linked blocks.

The bucket containing each row is determined by a hash function and a hash key. The ***hash key*** is a column or group of columns, usually the primary key. The ***hash function*** computes the bucket containing the row from the hash key.

Hash functions are designed to scramble row locations and evenly distribute rows across blocks. The ***modulo function*** is a simple hash function with four steps:

1. Convert the hash key by interpreting the key's bits as an integer value.
2. Divide the integer by the number of buckets.
3. Interpret the division remainder as the bucket number.
4. Convert the bucket number to the physical address of the block containing the row.

A ***dynamic hash function*** automatically allocates more blocks to the table, creates additional buckets, and distributes rows across all buckets. With more buckets, fewer rows are assigned to each bucket and, on average, buckets contain fewer linked blocks.

### Table clusters

***Table clusters***, also called ***multi-tables***, interleave rows of two or more tables in the same storage area. Table clusters have a ***cluster key***, a column that is available in all interleaved tables. The cluster key determines the order in which rows are interleaved. Rows with the same cluster key value are stored together. Usually the cluster key is the primary key of one table and the corresponding foreign key of another,

Table clusters are not optimal for many queries and therefore are not commonly used.

**5.3 SINGLE LEVEL INDEXES**

A ***single-level index*** is a file containing column values, along with pointers to rows containing the column value. The pointer identifies the block containing the row. In some indexes, the pointer also identifies the exact location of the row within the block. Indexes are created by database designers with the CREATE INDEX command, described elsewhere in this material.

An index is usually defined on a single column, but an index can be defined on multiple columns. In a ***multi-column index***, each index entry is a composite of values from all indexed columns. In all other respects, multi-column indexes behave exactly like indexes on a single column.

### Query processing

To execute a SELECT query, the database can perform a table scan or an index scan:

* A ***table scan*** is a database operation that reads table blocks directly, without accessing an index.
* An ***index scan*** is a database operation that reads index blocks sequentially, in order to locate the needed table blocks.

***Hit ratio***, also called ***filter factor*** or ***selectivity***, is the percentage of table rows selected by a query. When a SELECT query is executed, the database examines the WHERE clause and estimates hit ratio. If hit ratio is high, the database performs a table scan. If hit ratio is low, the query needs only a few table blocks, so a table scan would be inefficient.

In a ***binary search***, the database repeatedly splits the index in two until it finds the entry containing the search value:

1. The database first compares the search value to an entry in the middle of the index.
2. If the search value is less than the entry value, the search value is in the first half of the index. If not, the search value is in the second half.
3. The database now compares the search value to the entry in the middle of the selected half, to narrow the search to one quarter of the index.
4. The database continues in this manner until it finds the index block containing the search value.

### Primary and secondary indexes

Indexes on a sorted table may be primary or secondary:

* A ***primary index***, also called a ***clustering index***, is an index on a sort column.
* A ***secondary index***, also called a ***nonclustering index***, is an index that is not on the sort column.

A sorted table can have only one sort column, and therefore only one primary index. Usually, the primary index is on the primary key column(s). In some database systems, the primary index may be created on any column. Tables can have many secondary indexes. All indexes of a heap or hash table are secondary, since heap and hash tables have no sort column.

Indexes may also be dense or sparse:

* A ***dense index*** contains an entry for every table row.
* A ***sparse index*** contains an entry for every table block.

**5.4 MULTI LEVEL INDEXES**

### Multi-level indexes

A ***multi-level index*** stores column values and row pointers in a hierarchy. The bottom level of the hierarchy is a sorted single-level index. The bottom level is sparse for primary indexes, or dense for secondary indexes.

Each level above the bottom is a sparse sorted index to the level below. Since all levels above the bottom are sparse, levels rapidly become smaller. The top level always fits in one block.

The number of index entries per block is called the ***fan-out*** of a multi-level index. The number of levels in a multi-level index can be computed from fan-out, number of rows, and rows per block

### Balanced indexes

Each path from the top-level block to a bottom-level block is called a ***branch***. Multi-level indexes are called ***balanced*** when all branches are the same length and ***imbalanced*** when branches are different lengths.

### B-tree and B+tree indexes

The balanced multi-level index described above is called a B+tree. B+tree structure is derived from an earlier approach called a B-tree. The two differ as follows:

* ***B+tree***. All indexed values appear in the bottom level. Pointers to table blocks appear only in the bottom level. Since some indexed values also appear in higher levels, values are occasionally repeated in the index.
* ***B-tree***. If an indexed value appears in a higher level, the value is not repeated at lower levels. Instead, a pointer to the corresponding table block appears in the higher level along with the value.

*Although most multi-level indexes are implemented as B+trees, the term* ***B-tree*** *is commonly used and often refers to a B+tree structure.* ***B+tree*** *is commonly written as B+-tree or B+-tree.*

**5.5 OTHER INDEXES**

# Hash indexes

The multi-level index is the most commonly used index type. Several additional index types are used less often but supported by many databases:

* Hash index
* Bitmap index
* Logical index
* Function index

In a ***hash index***, index entries are assigned to buckets. A ***bucket*** is a block or group of blocks containing index entries. Initially, each bucket has one block. As an index grows, some buckets eventually fill up, and additional blocks are allocated and linked to the initial block.

### Bitmap indexes

A ***bitmap index*** is a grid of bits:

Bitmap indexes contain ones and zeros. 'One' indicates that the table row corresponding to the index row number contains the table value corresponding to the index column number. 'Zero' indicates the row does not contain the value.

### Logical indexes

A single- or multi-level index normally contains pointers to table blocks and is called a ***physical index***.

A ***logical index*** is a single- or multi-level index in which pointers to table blocks are replaced with primary key values. Logical indexes are always secondary indexes and require a separate primary index on the same table.

Logical indexes change only when primary key values are updated, which occurs infrequently. Physical indexes change whenever a row moves to a new block, which occurs in several ways:

* **A row is inserted into a full block**. To create space for the new row, the block splits and some rows move to a new block.
* **The sort column is updated**. When the sort column is updated, the row may move to a new block to maintain sort order.
* **The table is reorganized**. Occasionally, a database administrator may physically reorganize a table to recover deleted space or order blocks contiguously on magnetic disk.

If a table has several indexes, the time required to update physical indexes is significant, and logical indexes are more efficient.

On read queries, a logical index requires an additional read of the primary index and is slower than a physical index. However, the primary index is often retained in memory, mitigating the cost of the additional read.

index entries do not match values in the WHERE clause, so the database cannot use the index to execute the query.

To address this problem, some databases support function indexes. In a ***function index***, the database designer specifies a function on the column value. Index entries contain the result of the function applied to column values, rather than the column values.

In principle, functions can be used with any index type, including single-level, multi-level, hash, bitmap, and logical indexes.

**5.6 TABLESPACES AND PARTITIONS**

### Tablespaces

Tablespaces and partitions are supported by most databases but are not specified in the SQL standard.

A ***tablespace*** is a database object that maps one or more tables to a single file. The CREATE TABLESPACE statement names a tablespace and assigns the tablespace to a file. The CREATE TABLE statement assigns a table to a tablespace. Indexes are stored in the same tablespace as the indexed table.

CREATE TABLESPACE TablespaceName  
[ ADD DATAFILE 'FileName' ];  
   
CREATE TABLE TableName  
( ColumnName ColumnDefintion, ... )  
[ TABLESPACE TablespaceName ];

By default, most databases automatically create one tablespace for each table, so each table is stored in a separate file. Database administrators can manually create tablespaces and assign one or multiple tables to each tablespace. Database administrators can improve query performance by assigning frequently accessed tables to tablespaces stored on fast storage media.

### Partitions

A ***partition*** is a subset of table data. One table has many partitions that do not overlap and, together, contain all table data. A ***horizontal partition*** is a subset of table rows. A ***vertical partition*** is a subset of table columns. MySQL and most relational databases partition tables horizontally, not vertically.

Partitions improve query performance by reducing the amount of data accessed by INSERT, UPDATE, DELETE, and SELECT statements.

*A shard is similar to a partition. Like a partition, a* ***shard*** *is a subset of table data, usually a subset of rows rather than columns. Unlike partitions, which are stored on different storage devices of a single computer, shards are stored on different computers of a distributed database.*

### Partition types

To partition a table, the database administrator specifies a ***partition expression*** based on one or more ***partition columns***. The partition expression may be simple, such as the value of a single partition column, or a complex expression based on several partition columns. Rows are assigned to partitions in one of the following ways:

* A ***range partition*** associates each partition with a range of partition expression values. The VALUES LESS THAN keywords specify the upper bound of each range. The MAXVALUE keyword represents the highest column value, and VALUES LESS THAN MAXVALUE specifies the highest range. Each partition is explicitly named by the database administrator.
* A ***list partition*** associates each partition with an explicit list of partition expression values using the VALUES IN keywords. Like a range partition, each partition is explicitly named.
* A ***hash partition*** requires a partition expression with positive integer values. The database administrator specifies the number of partitions, N, and partitions are automatically named p0 through p(*N*-1). The partition number for each row is computed as: (partition expression value) modulo N.

**5.7 PHYSICAL DESIGN**

### MySQL storage engines

***Logical design*** specifies tables, columns, and keys. ***Physical design*** specifies indexes, table structures, and partitions. Physical design affects query performance but never affects query results.

**A *storage engine* or *storage manager* translates instructions generated by a query processor into low-level commands that access data on storage media.** Storage engines support different index and table structures, so physical design is dependent on a specific storage engine.

|  |  |  |
| --- | --- | --- |
| Statement | Description | Syntax |
| CREATE INDEX | Create an index | CREATE INDEX IndexName ON TableName (Column1, Column2, ..., ColumnN); |
| DROP INDEX | Delete an index | DROP INDEX IndexName ON TableName; |
| SHOW INDEX | Show an index | SHOW INDEX FROM TableName; |

### EXPLAIN statement

The ***EXPLAIN*** statement generates a result table that describes how a statement is executed by the storage engine. EXPLAIN syntax is simple and uniform in most databases: EXPLAIN statement; The statement can be any SELECT, INSERT, UPDATE, or DELETE statement.

### Physical design process

A database administrator may take a simple approach to physical design for MySQL with InnoDB:

1. **Create initial physical design**. Create a primary index on primary keys and a secondary index on foreign keys. In MySQL with InnoDB, these indexes are created automatically for all tables. In other databases, this step is necessary for tables larger than roughly 100 kilobytes, but can be omitted for smaller tables.
2. **Identify slow queries**. The MySQL ***slow query log*** is a file that records all long-running queries submitted to the database. Identify slow queries by inspecting the log. Most other relational databases have similar query logs.
3. **EXPLAIN slow queries**. Run EXPLAIN on each slow query to assess the effectiveness of indexes. A high value for **rows** and a low value for **filtered** indicates either a table scan or an ineffective index.
4. **Create and drop indexes** based on the EXPLAIN result table. Consider creating an index when the **rows** value is high and the **filtered** value is low. Consider dropping indexes that are never used.
5. **Partition large tables**. If some queries are still slow after indexes are created, consider partitions. Partition when slow queries access a small subset of rows of a large table. The partition column should appear in the WHERE clause of slow queries. Often, a range partition is best.