The Power of SQL: Harnessing the Full Potential of Your Database

Nikolaos Sourtzinos - Apr. 2023

"The Power of SQL: Harnessing the Full Potential of Your Database"

Contents

Introduction

SQL is a powerful tool for managing databases and retrieving data efficiently. In this booklet, we will explore the basics of SQL, advanced techniques, database design, tips and tricks, and real-world examples of SQL in action.

Chapter 1: The Fundamentals of SQL

What is SQL?

SQL stands for Structured Query Language. It is a programming language that is used to manage and manipulate relational databases. SQL allows users to interact with databases by creating, updating, and querying data.

It was developed by IBM in the early 1970s and became commercially available in 1979. SQL is the standard relational database management system (RDBMS) used globally..

What is a database?

Nowadays, digital storage of information has become a necessity for almost every business and organization, and databases are a prevalent solution for this purpose. A database is a structured assortment of data that can be conveniently stored, categorized, retrieved, and searched.

There are a variety of database types, and which type you use will be dependent on the type of data you wish to store. Let's look at a few popular database types:

- **Relational databases**– are organized in a tabular format consisting of rows and columns, and they store data across multiple tables. This enables users to access and relate data from different tables. These databases are typically accessed using Structured Query Language (SQL). Microsoft SQL Server is a commonly used example of a relational database management system.
- **NoSQL databases**– refer to any type of database that does not rely on SQL as its primary querying language. These databases are ideal for situations where data does not need to be strictly structured or requires more flexibility. Later, we can delve more into the specifics of these databases. An example of a NoSQL database is CouchDB.
- **Cloud databases**– refer to databases that are hosted and operated in the cloud, and can be accessed via the internet as a service. These databases are generally low maintenance and provide greater flexibility and scalability, as the resources can be easily adjusted to match changing demands. Oracle Autonomous Database is an example of a cloud database.
- **Time series databases**– are designed to efficiently store and analyze time-stamped data, making it easier to derive meaningful insights from this data. These databases are optimized for handling data that changes over time, and can be used in a variety of applications, such as monitoring systems, financial analysis, and IoT devices. An example of a time series database is Druid.

What is SQL used for?

After learning about SQL, you may be curious about its applications. As previously mentioned, SQL is a programming language utilized for communicating with relational databases. SQL enables the user to query databases in various ways using English-like statements. It is frequently employed in websites for backend data storage and processing, as exemplified by Facebook's use of SQL. Other than Facebook, SQL is utilized in music apps such as Spotify, banking apps like Revolut, and social media platforms like Twitter and Instagram. As SQL is the most widely utilized database language, it can be applied to almost any business that requires relational data storage. SQL queries are utilized to retrieve data from databases, but their effectiveness can vary since several databases have their own exclusive, system-specific, proprietary extensions.

Basic Syntax

SQL has a simple syntax that consists of statements that are used to perform specific tasks. The most common statements are SELECT, INSERT, UPDATE, DELETE, and CREATE. These statements are used to retrieve data, insert new data, modify existing data, delete data, and create database objects such as tables, views, and indexes.

Here are some basic SQL syntax examples:

Creating a table:

```
CREATE TABLE my_table (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  age INT,
  email VARCHAR(100)
);
```

Inserting data into a table:

```
INSERT INTO my_table (id, name, age, email)
VALUES (1, 'John Doe', 25, 'john.doe@example.com');
```

Updating data in a table:

```
UPDATE my_table
SET age = 30
WHERE id = 1;
```

Retrieving data from a table:

```
SELECT * FROM my_table;
```

Filtering data using a WHERE clause:

```
SELECT * FROM my_table
WHERE age > 25;
```

Sorting data using an ORDER BY clause:

```
SELECT * FROM my_table
ORDER BY age DESC;
```

Joining tables using a JOIN clause:

```
SELECT *
FROM my_table
JOIN another_table
ON my_table.id = another_table.id;
```

Types of SQL commands

The language can be broken down into four types of SQL commands – DDL, DML, DQL and DCL. Let's look at each of these sections.

- **DDL** (data definition language) – this is used to create and modify database objects like tables, users, and indices.
- **DML** (data manipulation language) – this is used to delete, add, and modify data within databases.
- **DCL** (data control language) – this is used to control access to any data within a database.
- **DQL** (data query language) – this is used to perform queries on the data and find information, and is composed of COMMAND statements only.

There are tools available to help you write SQL, some of these tools include Microsoft's SQL Server Management Studio, DataGrip, Oracle's SQL developer, SQL Workbench and Toad.

Data Types

SQL has a variety of data types that are used to define the type of data that can be stored in a column. Some of the most common data types include INT, VARCHAR, DATE, and BOOLEAN. INT is used to store integer values, VARCHAR is used to store character strings, DATE is used to store dates, and BOOLEAN is used to store true/false values.

Here are some examples of SQL data types:

INTEGER: Used to store whole numbers. Example:

```
CREATE TABLE my_table (
  id INTEGER,
  age INTEGER
);
```

DECIMAL: Used to store decimal numbers with a specified precision and scale. Example:

```
CREATE TABLE my_table (
  id INTEGER,
  price DECIMAL(10, 2)
);
```

This creates a "price" column with a precision of 10 and a scale of 2, allowing it to store decimal numbers up to 8 digits before the decimal point and 2 digits after the decimal point.

VARCHAR: Used to store variable-length strings of text. Example:

```
CREATE TABLE my_table (
  id INTEGER,
  name VARCHAR(50)
);
```

This creates a "name" column with a maximum length of 50 characters.

DATE: Used to store dates. Example:

```
CREATE TABLE my_table (
  id INTEGER,
  birth_date DATE
);
```

BOOLEAN: Used to store true/false values. Example:

```
CREATE TABLE my_table (
  id INTEGER,
  is_active BOOLEAN
);
```

These are just a few examples of SQL data types. There are many more data types available in SQL, such as TEXT, TIMESTAMP, FLOAT, and more. The choice of data type depends on the type of data you need to store and the operations you need to perform on that data.

Tables

A table is a database object that is used to store data in rows and columns. Each table has a unique name and consists of columns that define the type of data that can be stored in that table. Tables can be created using the CREATE TABLE statement.

Here's an example of an SQL table:

Let's say we want to create a table to store information about books. We want the table to have columns for the book's title, author, publisher, publication date, and price. The SQL code to create this table would be:

```
CREATE TABLE books (
  id INTEGER PRIMARY KEY,
  title VARCHAR(100),
  author VARCHAR(50),
  publisher VARCHAR(50),
  publication_date DATE,
  price DECIMAL(10, 2)
);
```

This creates a table named **books** with six columns: **id**, **title**, **author**, **publisher**, **publication_date**, and **price**. The **id** column is set as the primary key to ensure each row has a unique identifier.

To insert data into this table, we can use an SQL INSERT statement, like this:
INSERT INTO books (title, author, publisher, publication_date, price)
VALUES ('To Kill a Mockingbird', 'Harper Lee', 'J. B. Lippincott & Co.', '1960-07-11', 12.99);

This inserts a new row into the **books** table with the specified values for each column.

To retrieve data from the table, we can use an SQL SELECT statement, like this:
SELECT * FROM books;

This retrieves all rows from the **books** table. We can also use the WHERE clause to filter the results based on specific criteria, like this:
SELECT * FROM books WHERE author = 'Harper Lee';

This retrieves all rows from the **books** table where the **author** column is equal to **'Harper Lee'**.

Queries

Queries are used to retrieve data from a database. The most common query is the SELECT statement, which is used to retrieve data from one or more tables based on specified conditions. Queries can also be used to sort data, group data, and perform calculations on data using aggregate functions such as SUM, AVG, and COUNT.

Here's an example of an SQL query:
Let's say we have a table named **employees** that has columns **id**, **name**, **age**, and **salary**. We want to retrieve the names and salaries of all employees who are over the age of 30. The SQL query for this would be:

SELECT name, salary FROM employees WHERE age > 30;

This query selects the **name** and **salary** columns from the **employees** table where the **age** column is greater than 30. The result will be a table with two columns: **name** and **salary** containing only the rows that meet the specified condition.

Joins

Joins are used to combine data from two or more tables into a single result set. There are several types of joins, including INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN. Joins are used to retrieve data from multiple tables based on matching values in specific columns.

Here's an example of SQL JOINs:

Let's say we have two tables: **employees** and **departments**. The **employees** table has columns **id**, **name**, **age**, **department_id**, and **salary**, while the **departments** table has columns **id** and **name**. We want to join these two tables to retrieve the name, age, and department name of each employee.

The SQL query for this would be:

```
SELECT employees.name, employees.age, departments.name
FROM employees
JOIN departments ON employees.department_id = departments.id;
```

This query selects the name and age columns from the employees table and the name column from the departments table. The two tables are joined using the JOIN clause with the condition that the department_id column in the employees table is equal to the id column in the departments table. The result will be a table with three columns: name, age, and name containing the name, age, and department name of each employee.

Indexes

Indexes are used to improve the performance of queries by allowing data to be retrieved more quickly. An index is a database object that is created on one or more columns of a table. When a query is executed, the database uses the index to locate the relevant data more quickly.

Here's an example of SQL indexes:

Let's say we have a large table named orders that has columns order_id, customer_id, order_date, and total_amount. We frequently need to retrieve orders by customer ID, so we want to create an index on the customer_id column to improve query performance.

The SQL query to create an index would be:

```
CREATE INDEX idx_customer_id ON orders(customer_id);
```

This query creates an index named idx_customer_id on the customer_id column of the orders table. This index will improve the performance of queries that filter or sort data based on the customer_id column.

To use the index in a query, we can simply include the customer_id column in the WHERE clause, like this:

```
SELECT * FROM orders WHERE customer_id = 12345;
```

These are just a few of the fundamentals of SQL. SQL is a versatile and powerful programming language that is used to manage and manipulate relational databases. Understanding the basics of SQL is essential for anyone who wants to work with databases.

This query will use the idx_customer_id index to quickly find all orders with a customer_id of 12345, rather than scanning the entire orders table.

SQL functions

SQL is a language used to interact with databases, and it provides a rich set of functions to manipulate and retrieve data from them.

Here's a list of some commonly used SQL functions:

Aggregate Functions:

- COUNT(): Returns the number of rows in a table or the number of rows that match a specified condition.
- SUM(): Returns the sum of values in a column or the sum of values that match a specified condition.
- AVG(): Returns the average of values in a column or the average of values that match a specified condition.
- MIN(): Returns the minimum value in a column or the minimum value that matches a specified condition.
- MAX(): Returns the maximum value in a column or the maximum value that matches a specified condition.

String Functions:

- CONCAT(): Concatenates two or more strings into a single string.
- SUBSTRING(): Extracts a portion of a string.
- LENGTH(): Returns the length of a string.
- TRIM(): Removes leading and trailing spaces from a string.
- UPPER(): Converts a string to uppercase.
- LOWER(): Converts a string to lowercase.

Date Functions:

- CURRENT_DATE: Returns the current date.
- CURRENT_TIME: Returns the current time.
- CURRENT_TIMESTAMP: Returns the current timestamp.
- EXTRACT(): Extracts a component (year, month, day, hour, etc.) from a date or timestamp.
- DATE_ADD(): Adds a specified interval to a date.
- DATE_DIFF(): Calculates the difference between two dates.

Conditional Functions:

- IF(): Returns one value if a condition is true and another value if it is false.
- CASE(): Evaluates a series of conditions and returns a result based on the first condition that is true.

Mathematical Functions:

- ROUND(): Rounds a number to a specified number of decimal places.
- CEILING(): Rounds a number up to the nearest integer.
- FLOOR(): Rounds a number down to the nearest integer.
- ABS(): Returns the absolute value of a number.

These are just a few of the many functions available in SQL. The specific functions available can vary depending on the database management system you are using.

Chapter 2: Advanced SQL Techniques

Subqueries

A subquery is a query that is embedded within another query. Subqueries can be used to retrieve data that is based on the results of another query. For example, a subquery can be used to retrieve the maximum or minimum value of a column from a table and then use that value in another query.

Here's an example of an SQL subquery:

Let's say we have two tables: employees and departments. The employees table has columns id, name, age, department_id, and salary, while the departments table has columns id and name. We want to find the names of all employees who work in a department with a name starting with the letter S.

We can use a subquery to accomplish this. The subquery will retrieve the department IDs of all departments with a name starting with S, and the outer query will retrieve the names of all employees whose department ID is in that list. Here's the SQL code to do this:

```
SELECT name
FROM employees
WHERE department_id IN (
  SELECT id
  FROM departments
  WHERE name LIKE 'S%'
);
```

his query uses a subquery in the WHERE clause to retrieve the department IDs of all departments with a name starting with S. The outer query then retrieves the names of all employees whose department ID is in that list. The '%' character is a wildcard that matches any number of characters, so the LIKE operator with 'S%' will match any department name that starts with S.

Note that the subquery is enclosed in parentheses and used as a value in the WHERE clause of the outer query. The subquery is executed first to retrieve the list of department IDs, and then the outer query uses that list to retrieve the names of the employees.

Common Table Expressions (CTEs)

A Common Table Expression is a temporary named result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. CTEs are used to simplify complex queries by allowing subqueries to be factored out into a separate query block that can be referenced multiple times within a larger query.

Here's an example of an SQL Common Table Expression (CTE):

Let's say we have a table named sales with columns id, date, amount, and employee_id. We want to find the total sales for each employee in the year 2022, but we also want to calculate the percentage of each employee's sales compared to the total sales for all employees. We can use a CTE to accomplish this.

```
WITH employee_sales AS (
  SELECT employee_id, SUM(amount) AS total_sales
  FROM sales
  WHERE date BETWEEN '2022-01-01' AND '2022-12-31'
  GROUP BY employee_id
), total_sales AS (
  SELECT SUM(amount) AS total_sales
  FROM sales
  WHERE date BETWEEN '2022-01-01' AND '2022-12-31'
)
SELECT employee_id, total_sales, total_sales / CAST((SELECT total_sales FROM total_sales) AS FLOAT) AS
percentage
FROM employee_sales;
```

This query uses two CTEs to calculate the total sales for each employee and the total sales for all employees. The first CTE named employee_sales retrieves the total sales for each employee in the year 2022, grouped by employee ID. The second CTE named total_sales retrieves the total sales for all employees in the year 2022.

The final query then uses the two CTEs to calculate the percentage of each employee's sales compared to the total sales for all employees. The percentage is calculated by dividing the total sales for each employee by the total sales for all employees, and then multiplying by 100 to get a percentage value. Note that we use the CAST function to convert the total_sales value to FLOAT before performing the division to avoid integer division.

The final result set includes the employee ID, the total sales for that employee in the year 2022, and the percentage of that employee's sales compared to the total sales for all employees in the year 2022.

Window Functions

Window functions are used to perform calculations across a set of rows that are related to the current row. Window functions can be used to calculate running totals, moving averages, and rank data within a partition. Window functions are used in conjunction with the OVER clause.

Here are some examples of SQL Window Functions:

Let's say we have a table named sales with columns id, date, amount, and region. We want to find the total sales for each region, as well as the rank of each region based on total sales.

Ranking Regions by Total Sales:

```
SELECT region, SUM(amount) AS total_sales, RANK() OVER (ORDER BY SUM(amount) DESC) AS rank
FROM sales
GROUP BY region;
```

This query uses the RANK function as a window function to rank regions based on total sales. The RANK function assigns a rank to each region based on the total sales for that region, ordered from highest to

lowest. The final result set includes the region, total sales for that region, and the rank of that region based on total sales.

Calculating Running Total of Sales for Each Region:
SELECT region, date, amount, SUM(amount) OVER (PARTITION BY region ORDER BY date) AS running_total FROM sales;

This query uses the SUM function as a window function to calculate a running total of sales for each region. The SUM function is applied over a partition of the data by region and ordered by date. The final result set includes the region, date, amount, and the running total of sales for that region up to that date.

Finding Percent of Total Sales by Region:
SELECT region, amount, SUM(amount) OVER (PARTITION BY region) / CAST(SUM(amount) OVER () AS FLOAT) * 100 AS percent_total_sales
FROM sales;

This query uses the SUM function as a window function to calculate the total sales for each region and the entire dataset. The percentage of total sales for each region is then calculated by dividing the total sales for each region by the total sales for the entire dataset and multiplying by 100. The result set includes the region, amount, and the percentage of total sales for that region.

Stored Procedures

A stored procedure is a precompiled set of SQL statements that are stored in a database. Stored procedures can be called by other programs or applications to execute a specific set of actions on a database. Stored procedures can improve performance and security by reducing the amount of data that is transferred between the database and the application.
Here's an example of an SQL Stored Procedure:

Let's say we have a table named employees with columns id, name, salary, and department. We want to create a stored procedure that takes a department name and a percentage increase as input and updates the salaries of all employees in that department by the given percentage.

```
CREATE PROCEDURE update_salaries(IN dept_name VARCHAR(50), IN increase DECIMAL(5,2))
BEGIN
  UPDATE employees
  SET salary = salary + (salary * increase / 100)
  WHERE department = dept_name;
END;
```

This stored procedure takes two input parameters: dept_name, which is the name of the department to update, and increase, which is the percentage increase to apply to the salaries of employees in that department. The UPDATE statement inside the stored procedure updates the salaries of all employees in the specified department by multiplying their current salary by the percentage increase and adding the result to their current salary.

To execute the stored procedure, we can call it like this:

```
CALL update_salaries('Marketing', 10.00);
```

This will update the salaries of all employees in the Marketing department by 10 percent.

Note that this is just a simple example and there are many more complex tasks that can be accomplished using SQL Stored Procedures.

Triggers

A trigger is a database object that is executed automatically in response to a specific event, such as an INSERT, UPDATE, or DELETE operation on a table. Triggers can be used to enforce data integrity constraints, audit data changes, and automate complex business logic.

Here's an example of an SQL Trigger:

Let's say we have a table named orders with columns id, customer_id, total_price, and order_date. We want to create a trigger that automatically updates a customer_summary table with the total amount spent by each customer whenever a new order is inserted into the orders table.

```
CREATE TRIGGER update_customer_summary
AFTER INSERT ON orders
FOR EACH ROW
BEGIN
  UPDATE customer_summary
  SET total_spent = total_spent + NEW.total_price
  WHERE customer_id = NEW.customer_id;
END;
```

This trigger fires after an INSERT operation is performed on the orders table. The UPDATE statement inside the trigger updates the customer_summary table by adding the total price of the new order to the total_spent column of the corresponding customer. The NEW keyword is used to refer to the newly inserted row in the orders table.

To create the customer_summary table, we can use the following SQL statement:

```
CREATE TABLE customer_summary (
  customer_id INT,
  total_spent DECIMAL(10, 2)
);
```

Whenever a new order is inserted into the orders table, the update_customer_summary trigger will automatically update the customer_summary table with the new total amount spent by the corresponding customer.

Note that this is just a simple example and there are many more complex tasks that can be accomplished using SQL Triggers.

Dynamic SQL

Dynamic SQL is a technique that allows SQL statements to be constructed and executed dynamically at runtime. Dynamic SQL is useful for creating flexible queries that can be customized based on user input or other dynamic conditions.

Here's an example of SQL Dynamic SQL:

Let's say we have a table named employees with columns id, name, salary, and department. We want to create a stored procedure that takes a department name and a minimum salary as input and returns all employees in that department whose salary is greater than or equal to the minimum salary.

```
CREATE PROCEDURE search_employees(IN dept_name VARCHAR(50), IN min_salary DECIMAL(10, 2))
BEGIN
  SET @sql = CONCAT('SELECT * FROM employees WHERE department = \'', dept_name, '\' AND salary >=
', min_salary);
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
  DEALLOCATE PREPARE stmt;
END;
```

This stored procedure takes two input parameters: dept_name, which is the name of the department to search, and min_salary, which is the minimum salary to filter by. The procedure then constructs a dynamic SQL statement using the CONCAT function to include the input parameters and stores it in a user-defined variable named @sql.

The PREPARE statement prepares the dynamic SQL statement for execution, and the EXECUTE statement executes it. The DEALLOCATE PREPARE statement frees the memory used by the prepared statement.

To call the stored procedure, we can use the following SQL statement:

```
CALL search_employees('Sales', 50000.00);
```

This will return all employees in the Sales department whose salary is greater than or equal to $50,000.

Note that this is just a simple example and there are many more complex tasks that can be accomplished using SQL Dynamic SQL. However, it is important to use dynamic SQL with caution, as it can introduce security vulnerabilities if not used properly.

These are just a few advanced SQL techniques that can be used to perform complex data manipulations and optimizations. Understanding these techniques can help you become a more effective database developer or administrator.

Chapter 3: Database Design with SQL

Some key considerations when designing a database with SQL are:

Identify the Purpose of the Database

Before designing a database, it's important to identify the purpose of the database and the type of data that will be stored in it. Will the database be used to store customer information, inventory data, or financial transactions? This will help determine the structure and organization of the database.

A well-designed database can help businesses and organizations manage their operations and make informed decisions based on data analysis.

Here is an example of identifying the purpose of a database:

Let's say we want to create a database for a retail store that sells clothing and accessories. The purpose of the database is to store information about the store's products, customers, and sales. By organizing this information in a database, the store can:

1. **Keep track of inventory**: The database can store information about the store's products, such as product name, description, price, and quantity in stock. This allows the store to keep track of inventory levels and ensure that popular items are always in stock.
2. **Manage customer information**: The database can store information about customers, such as their name, address, phone number, and email address. This allows the store to send targeted marketing messages and promotions to specific customers based on their preferences and purchase history.
3. **Analyze sales data**: The database can store information about each sale, such as the date, time, items purchased, and total price. This allows the store to analyze sales trends and identify popular products or categories. The store can also track sales by store location, employee, or time of day to identify areas for improvement.
4. **Improve customer experience**: By storing information about customer preferences and purchase history, the store can offer personalized recommendations and promotions to customers. This can help to improve the customer experience and increase customer loyalty.

Overall, the purpose of the retail store's database is to improve operational efficiency, provide insights into customer behavior, and improve the overall customer experience. By using a well-designed database, the store can optimize its operations and make data-driven decisions to stay competitive in the marketplace.

Identify the Entities and Relationships

The next step is to identify the entities (objects or concepts) that will be represented in the database and their relationships to each other. This can be done by creating an Entity-Relationship (ER) diagram, which shows the entities, attributes, and relationships between them.

Here is an example of identifying entities and relationships in a database:

Let's say we want to create a database for a university that stores information about students, courses, and professors. We can start by identifying the entities:

- **Student**: a person enrolled in a course at the university
- **Course**: a class offered by the university
- **Professor**: a person who teaches a course at the university

Next, we can identify the relationships between the entities:

- **One student can enroll in many courses, and one course can have many students.** This is a many-to-many relationship.
- **One professor can teach many courses, and one course can be taught by many professors.** This is also a many-to-many relationship.
- **One student can have many professors, and one professor can have many students**. This is a many-to-many relationship.

Based on these relationships, we can create the following tables in our database:

- Student table: **student_id** (primary key), **first_name**, **last_name**, **email**, **phone_number**
- Course table: **course_id** (primary key), **course_name**, **course_description**
- Professor table: **professor_id** (primary key), **first_name**, **last_name**, **email**, **phone_number**
- Enrollment table: **student_id** (foreign key to Student table), **course_id** (foreign key to Course table)
- Teaching table: **professor_id** (foreign key to Professor table), **course_id** (foreign key to Course table)
- Student_Professor table: **student_id** (foreign key to Student table), **professor_id** (foreign key to Professor table)

The **Enrollment** table represents the many-to-many relationship between **students** and **courses**, the **Teaching** table represents the many-to-many relationship between **professors** and **courses**, and the **Student_Professor** table represents the many-to-many relationship between **students** and **professors**.

By identifying the entities and relationships in our database, we can create a well-organized and efficient database structure that allows us to store and access information about the university's students, courses, and professors.

Normalize the Data

Normalization is the process of organizing data in a database so that it is consistent and efficient. This involves breaking down complex data structures into simpler, more manageable structures. There are several levels of normalization, with the most common being first, second, and third normal form.

Here's an example of normalizing a database:
Let's say we have a database for an online bookstore. The initial design has one table called Books with the following columns:

- book_id (int)
- book_title (varchar)
- author_name (varchar)
- publisher_name (varchar)
- publication_date (date)
- price (decimal)

- genre (varchar)
- ISBN (varchar)
- number_of_pages (int)

This table violates the first normal form (1NF) because some columns contain multiple values. For example, **author_name** and **publisher_name** can have multiple values for a single book. To normalize this table, we can break it down into multiple tables.

First, we can create a table for the authors:
- author_id (int)
- author_name (varchar)

Next, we can create a table for publishers:
- publisher_id (int)
- publisher_name (varchar)

Then, we can create a table for books:
- book_id (int)
- book_title (varchar)
- publication_date (date)
- price (decimal)
- genre (varchar)
- ISBN (varchar)
- number_of_pages (int)
- author_id (int) -> foreign key to the Authors table
- publisher_id (int) -> foreign key to the Publishers table

This new design follows the third normal form (3NF) because each table contains only one type of data and there is no data redundancy. We have also established relationships between the **Books** table and the **Authors** and **Publishers** tables using foreign keys.

By normalizing the database in this way, we can avoid data inconsistencies and make it easier to update and manage the data.

Define the Tables and Columns

Once the data has been normalized, the next step is to define the tables and columns that will be used to store the data. Each table should represent a single entity, and each column should represent a single attribute of that entity.

Here is an example of creating a table in SQL:

```
CREATE TABLE Customers (
   CustomerID int,
   Name varchar(50),
   Email varchar(50),
   Phone varchar(20)
);
```

In this example, the **Customers** table has four columns:

- **CustomerID**: An integer column used to uniquely identify each customer.
- **Name**: A varchar column used to store the name of each customer.
- **Email**: A varchar column used to store the email address of each customer.
- **Phone**: A varchar column used to store the phone number of each customer.

Each column is defined with a data type and a maximum length, as appropriate for the type of data being stored.

You can also add constraints to the columns to enforce additional rules or restrictions on the data being stored. For example, you could add a **NOT NULL** constraint to ensure that a column always has a value, or add a **UNIQUE** constraint to ensure that a column's values are always unique.

Here is an example of creating a table with constraints:

```
CREATE TABLE Orders (
   OrderID int,
   CustomerID int NOT NULL,
   OrderDate date,
   PRIMARY KEY (OrderID),
   FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

In this example, the **Orders** table has three columns and two constraints:

- **OrderID**: An integer column used to uniquely identify each order.
- **CustomerID**: An integer column used to identify the customer who placed the order, with a **NOT NULL** constraint to ensure that this column always has a value.
- **OrderDate**: A date column used to store the date the order was placed.
- **PRIMARY KEY**: A constraint that indicates that **OrderID** is the primary key for the table.
- **FOREIGN KEY**: A constraint that establishes a relationship between the **Orders** and **Customers** tables, based on the **CustomerID** column.

By creating tables and columns in SQL, you can organize and store your data in a structured and consistent way, making it easier to manage and query your data as needed.

Define the Primary Keys

Each table should have a primary key, which is a unique identifier for each row in the table. This is used to ensure that each row can be uniquely identified and accessed.

Here is an example of how to define a primary key in SQL:
Let's say we have a table named Customers with the following columns:

- customer_id (int)
- first_name (varchar)
- last_name (varchar)

- email (varchar)
- phone_number (varchar)
- address (varchar)

To define the primary key, we can use the following SQL statement:

```
ALTER TABLE Customers
ADD PRIMARY KEY (customer_id);
```

This statement adds a primary key constraint to the **Customers** table on the **customer_id** column. This means that each value in the **customer_id** column must be unique, and it cannot be null.
If we want to use multiple columns as the primary key, we can do it like this:

```
ALTER TABLE Customers
ADD PRIMARY KEY (customer_id, email);
```

This statement adds a primary key constraint to the **Customers** table on both the **customer_id** and **email** columns. This means that the combination of values in these columns must be unique, and neither column can be null.

Define the Foreign Keys

Foreign keys are used to establish relationships between tables. A foreign key is a column in one table that references the primary key of another table. This is used to ensure that data is consistent across multiple tables.

Here is an example of creating a foreign key in SQL:

```
CREATE TABLE Orders (
  OrderID int PRIMARY KEY,
  CustomerID int,
  OrderDate date,
  FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

In this example, the **Orders** table has a foreign key **CustomerID** that references the primary key **CustomerID** in the **Customers** table. This establishes a relationship between the **Orders** and **Customers** tables based on the **CustomerID** column. The foreign key constraint ensures that any value in the **CustomerID** column of the **Orders** table must exist in the **CustomerID** column of the **Customers** table, or the database engine will reject the operation.
This helps maintain data integrity and consistency between the two tables, preventing orphaned or invalid data from being entered into the system.

Define the Constraints

Constraints are used to enforce rules on the data in the database. This includes rules for data types, null values, and unique values. Constraints are used to ensure that the data is accurate, consistent, and valid.

Here's an example SQL code that defines constraints for a table:

```
CREATE TABLE products (
   id INT PRIMARY KEY,
   name VARCHAR(50) NOT NULL,
   price DECIMAL(10,2) NOT NULL CHECK (price > 0),
   stock INT DEFAULT 0 CHECK (stock >= 0),
   category_id INT REFERENCES categories(id)
);
```

In this example, we define a table named **products** with five columns: **id**, **name**, **price**, **stock**, and **category_id**. We also define several constraints on the table using the following keywords:

- **PRIMARY KEY**: specifies the primary key for the table, which uniquely identifies each row. In this case, the **id** column is the primary key.
- **NOT NULL**: specifies that the column cannot have a NULL value. In this case, the **name** and **price** columns cannot be NULL.
- **CHECK**: specifies a condition that must be true for the column. In this case, the **price** column must be greater than 0, and the **stock** column must be greater than or equal to 0.
- **DEFAULT**: specifies a default value for the column if no value is provided. In this case, the **stock** column defaults to 0 if no value is provided.
- **REFERENCES**: specifies a foreign key constraint that references another table. In this case, the **category_id** column references the **id** column of a table named **categories**.

By defining these constraints, we ensure that the data in the **products** table is consistent and accurate. For example, we ensure that every row has a unique ID, every product has a name and a price, the price and stock values are valid, and the **category_id** column refers to a valid category ID in the **categories** table.

Index the Tables

Indexes are used to speed up queries by providing faster access to data. Indexes are created on one or more columns of a table and are used to improve performance when searching, sorting, and filtering data. These are just a few key considerations when designing a database with SQL. A well-designed database can improve performance, accuracy, and reliability, and can make it easier to work with data in the long run.

Chapter 4: Tips and Tricks for Effective SQL

Some tips and tricks for writing effective SQL are:

Use Descriptive Names

Use descriptive names for tables, columns, and other database objects. This will make it easier to understand and maintain the database in the long run.

By using names that accurately reflect the meaning and purpose of each element, you can make it easier for other developers to understand your code and make changes or updates as needed.

Here's an example of using descriptive names for tables and columns:

```
CREATE TABLE Employees (
   EmployeeID int,
   FirstName varchar(50),
   LastName varchar(50),
   EmailAddress varchar(50),
   HireDate date,
   PRIMARY KEY (EmployeeID)
);
```

In this example, the **Employees** table has columns with names that clearly describe the type of data they store. For example:

- **EmployeeID**: An integer column used to uniquely identify each employee.
- **FirstName**: A varchar column used to store the first name of each employee.
- **LastName**: A varchar column used to store the last name of each employee.
- **EmailAddress**: A varchar column used to store the email address of each employee.
- **HireDate**: A date column used to store the date each employee was hired.

By using these descriptive names, it's easy to understand the purpose of each column and how it relates to the data being stored in the table. This can make it easier to write SQL queries, as well as to understand and maintain existing code.

Use Comments

Use comments to document your SQL code. This will make it easier to understand what the code is doing and why it was written that way.

SQL supports two types of comments:

Single-line comments: These comments start with two dashes (--) and continue until the end of the line. They are used to add explanatory notes or to temporarily disable a part of a query.

```
SELECT *
FROM employees
```

```
--WHERE department = 'Sales'
WHERE salary > 50000;
```

In this example, the comment is used to temporarily disable the WHERE clause that filters employees by department. The query will only return employees with a salary greater than $50,000.

Multi-line comments: These comments start with a slash-asterisk (/) and end with an asterisk-slash (/). They are used to add longer explanatory notes or to temporarily disable a large portion of a query.

```
/* This query calculates the total sales by region for the year 2022 */
SELECT region, SUM(sales) AS total_sales
FROM sales
WHERE YEAR(sale_date) = 2022
GROUP BY region;
*/

SELECT *
FROM employees
WHERE salary > 50000;
```

In this example, the multi-line comment is used to add an explanatory note about the purpose of the query. The query will only return employees with a salary greater than $50,000.

Note that comments in SQL are not executed and do not affect the outcome of a query. They are purely for human readability and documentation purposes.

Use Shortcuts

Use shortcuts to save time and make your code more efficient. For example, instead of writing SELECT * FROM, you can use SELECT t.* FROM table t to save typing and make the code more readable.

Use Aliases

Use aliases to make your code more readable. For example, instead of writing SELECT first_name, last_name FROM customers, you can use SELECT c.first_name, c.last_name FROM customers c to make it clear that c represents the customers table.

Use Joins

Use joins to combine data from multiple tables. Joins allow you to retrieve data from multiple tables in a single query, which can be more efficient than running multiple queries.

Use Indexes

Use indexes to improve performance when searching, sorting, and filtering data. Indexes provide faster access to data by creating a smaller subset of data that can be searched more quickly.

Use Stored Procedures

Use stored procedures to improve performance and security. Stored procedures can be compiled and executed faster than ad-hoc SQL code, and can also help prevent SQL injection attacks.

Use Transactions

Use transactions to ensure data consistency and reliability. Transactions allow you to group multiple SQL statements into a single unit of work that can be rolled back if there are errors or other issues.

Here's an example SQL code that demonstrates how to use transactions:

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 123;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 456;
COMMIT;
```

This code starts a transaction using the BEGIN TRANSACTION statement. It then performs two SQL UPDATE statements that deduct 100 from the balance of the account with ID 123 and add 100 to the balance of the account with ID 456. Finally, it commits the transaction using the COMMIT statement.

By using a transaction, we ensure that both updates either succeed or fail together as an atomic operation. For example, if the update for account 456 fails for any reason, the update for account 123 will also be rolled back, and both accounts will remain unchanged.

If an error occurs during the transaction, we can roll back the entire transaction using the ROLLBACK statement. This restores the database to its previous state before the transaction started.

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 123;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 999; -- Invalid account_id
ROLLBACK;
```

In this example, if the second UPDATE statement fails because of the invalid account_id, the entire transaction will be rolled back, and the balance of account 123 will not be deducted.

Note that not all database engines support transactions, or they may have different syntax or behavior. Be sure to consult your database engine's documentation for specific guidance.

Test Your Code

Test your SQL code to ensure that it produces the expected results and to catch errors before they cause problems in production.

These are just a few tips and tricks for writing effective SQL. By following these best practices, you can write cleaner, more efficient SQL code that is easier to understand and maintain.

Chapter 5: Real-World Examples of SQL in Action

Retail Industry - Inventory Management

In the retail industry, SQL is widely used to manage inventory. SQL can be used to track the quantity of each product in stock, its location, and its price. Using SQL, retailers can monitor inventory levels in real-time, determine which products are selling the most, and replenish stock when necessary.

Healthcare Industry - Patient Management

SQL is also widely used in the healthcare industry to manage patient information. Using SQL, hospitals can track patient records, including demographics, medical history, diagnoses, and treatment plans. This information can be used to generate reports on patient outcomes, improve the quality of care, and optimize resource allocation.

Finance Industry - Fraud Detection

SQL is used extensively in the finance industry to detect fraudulent activity. Using SQL, banks can analyze large volumes of data to identify patterns and anomalies that may indicate fraud. By flagging suspicious transactions, banks can prevent fraudulent activity and protect their customers from financial losses.

E-commerce Industry – Personalization

SQL is also used to provide personalized recommendations to online shoppers. Using SQL, e-commerce companies can analyze customer data, such as browsing history and purchase behavior, to make personalized product recommendations. This helps companies increase sales and customer satisfaction.

Manufacturing Industry - Production Planning

In the manufacturing industry, SQL is used to optimize production planning. Using SQL, manufacturers can analyze data on production rates, raw material usage, and equipment downtime to identify areas for improvement. By optimizing production planning, manufacturers can reduce costs and improve product quality.

These are just a few examples of how SQL is used in different industries. SQL is a versatile tool that can be used in many different applications to manage data efficiently and gain valuable insights.

Chapter 6: Types of SQL jobs

Learning SQL will open opportunities in a range of different careers. Let's look at some of the options available.

Data scientist

A data scientist is an analytical data expert – they extract, analyze and interpret big data from a range of A data scientist is an analytical expert who extracts, analyzes and interprets big data from various sources to solve problems. SQL is a critical tool for data scientists since databases are at the core of their work due to the data analytics they must perform.

Data analyst

A data analyst is a professional often use SQL to query databases, extract data, and perform operations such as sorting, filtering, and grouping. They use SQL to clean and transform data into a format that is suitable for analysis, and to create reports and visualizations that communicate their findings to decision-makers.

Business analyst

A business analyst analyzes data and documents market environments to provide advice for business decisions. This role is heavily reliant on SQL since it involves working with large amounts of data and relational databases.

SEO analyst

An SEO analyst is responsible for analyzing data and optimizing website content to increase organic search traffic. SQL is beneficial in this role since SEO analysts work with a lot of big data, and databases are more robust than Excel documents, which are often used.

Software engineer

A software engineer develops and builds computer systems and application software. As a software engineer, knowledge of programming languages is necessary to build software, and most programmers are required to have some understanding of SQL.

In addition to being useful for landing one of these roles, SQL can also be beneficial for entrepreneurs or those planning to start their own business. Relational databases can help individuals store, sort, and modify large amounts of data, which can be a valuable asset in running a business.

Conclusion

We hope you found this booklet informative and useful. SQL is a powerful tool for managing databases and retrieving data efficiently, and we hope that this book has helped you harness its full potential. Remember to keep practicing and exploring new ways to use SQL to improve your database management skills.