SQL Advanced

Joachim Walther

26.10.-27.10.2023

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Preface

In the digital realm, modern SQL Database Systems have evolved to become much more than just repositories of static data tables. They are now dynamic, multifaceted systems with capabilities that far exceed rudimentary data storage and retrieval. But with such vastness comes complexity, and truly harnessing the power of these systems is no trivial task—it demands more than just a cursory glance or a fleeting endeavor.

A significant cohort of developers, perhaps daunted by the abstract nature of relational models or the perceived rigidity of tabular structures, shy away from SQL. They yearn for the unbridled flight over data, craving the perceived freedom of non-relational paradigms. Yet, in sidestepping SQL, they inadvertently forsake a wealth of technological prowess and the nuanced understanding that comes with it.

This advanced course is not a mere exposition of arcane SQL scripts or an exercise in deciphering sprawling lines of database code. Rather, it aims to cultivate a foundational understanding—how to construct, think through, and design such scripts. And while we might explore the creation of extensive SQL scripts, it's imperative to recognize that length and complexity for their own sake can be counterproductive. Elegant SQL, much like in other programming languages, leans on principles of design, clarity, and purpose.

The intricacies of database programming can indeed be formidable. But are there methodologies, tools, and philosophies to navigate this labyrinth? Absolutely. By assimilating design principles, adhering to best practices, and internalizing the tenets of clean code, SQL can be as graceful and expressive as any other programming language.

While we will delve into various advanced facets of SQL, this course is not, and cannot be, an exhaustive compendium of all that SQL and its associated systems offer. They are mature, expansive, and ever-evolving. Thus, an integral part of this learning journey is the encouragement for you, the student, to seek, explore, and discover the nuances independently.

Welcome to a deep dive into SQL—not just as a query language but as an avenue for robust, efficient, and elegant solutions.

1 Introduction

- The „SQL Advanced“ course aims to enhance your understanding of using SQL as a productive language.
- You should gain an insight into the power and flexibility of this language.
- You will recognize its advantages and limitations.
- You will learn about the wide range of available databases and tools.
- We will look at the advanced functions of databases.
- We will get some glimps of design patterns.

1.1 Content of the course

- Deeper dive into the power and flexibility of SQL.
- Recognizing the limitations of SQL.
- The big picture, databases and their universe.
- How databases integrate into applications.
- Examination of development environments, databases, and tools.
- Insights into NoSQL databases for a comprehensive understanding.
- Review of SQL language elements (DDL, DML, etc.).
- Initiating database design and the role of UML diagrams.
- Design considerations and problematic patterns.
- Emphasis on the creative process of creating a database in the 3rd normal form.
- Introduction to complex data structures.
- Exploration of advanced programming SQL methods.
- Methods for testing databases.
- Deployment of databases.

1.2 Conclusion

Participants will share their impressions and experiences. They are providing feedback and insights into the course's contend and areas of further interest.

2 The Power and Advantages of SQL over Other Languages

SQL, or Structured Query Language, has been the backbone of relational database management systems (RDBMS) for over four decades. Its longevity itself speaks volumes about its relevance and efficiency. When comparing SQL to other languages or data querying methods, several distinct advantages highlight its unparalleled power in data management and analysis:

1. **Purpose-Specific Language**: SQL is domain-specific, designed exclusively for querying and managing data. This specialization makes it extremely efficient for its intended tasks, eliminating the unnecessary complexities found in general-purpose languages.

2. **Standardization**: With ANSI SQL standards in place, the core commands and queries work across almost all RDBMS platforms, ensuring consistent behavior and wide applicability. This means that once you learn SQL, you can work with a vast array of databases, from Oracle to MySQL to PostgreSQL and beyond.

3. **Comprehensive and Expressive**: SQL allows for both simple queries („retrieve all records from a table“) and complex operations involving multiple tables, aggregation, and filtering. Its expressive power enables users to conduct intricate data manipulations, computations, and transformations.

4. **Optimized Performance**: Database systems are built with SQL as a fundamental layer. As a result, these systems are optimized to interpret and run SQL queries at blazing speeds, making data retrieval and manipulation fast, even for large datasets.

5. **Data Integrity and ACID Compliance**: SQL databases follow ACID (Atomicity, Consistency, Isolation, Durability) principles, ensuring data reliability and robustness. Constraints, primary and foreign keys, and normalized design patterns inherent to relational databases powered by SQL guarantee data integrity and accuracy.

6. **Concurrent Data Access**: SQL databases are designed to handle multiple concurrent users, ensuring data consistency and preventing conflicts, crucial for businesses with multiple stakeholders accessing data simultaneously.

7. **Mature Ecosystem**: Being in use for so long, SQL benefits from a mature ecosystem. Tools, libraries, connectors, and extensions abound, enhancing its capabilities and integrating it with other technologies.

8. **Security**: SQL databases offer granular security features, allowing database administrators to define access levels, roles, and permissions. This ensures that sensitive data is safeguarded, and only authorized individuals can perform specific operations.

9. **Platform Agnostic**: SQL can be used across multiple operating systems, cloud platforms, and devices. It doesn't tie you down to a specific vendor or platform.
10. **Community and Resources**: A robust global community supports SQL. With countless tutorials, forums, courses, and experts available, new learners find it relatively easy to climb the learning curve, and seasoned professionals always have a platform for collaboration and problem-solving.

In contrast to NoSQL or newer data retrieval methods tailored for specific use-cases, SQL offers unmatched versatility. While NoSQL databases excel in flexibility, scalability, and handling unstructured data, SQL remains unbeaten in structured data operations, complex queries, and situations demanding data integrity and consistency.

In conclusion, SQL's power is derived from its specialization, standardization, and the breadth and depth of its capabilities. Its continued dominance in the realm of relational databases, even as newer languages and technologies emerge, is a testament to its robustness and the value it offers. Embracing SQL equips one with a versatile toolset, ideal for navigating the evolving landscape of data-driven decision-making in modern businesses.

3 The Flexibility of SQL

SQL is often considered flexible in its domain of data management and querying, and here's why:

1. **Expressiveness**: SQL can handle a wide variety of data querying and manipulation tasks. From simple data retrieval to complex analytical operations involving multiple tables, joins, subqueries, aggregation, and more, SQL provides the necessary constructs.

2. **Adaptability across RDBMS**: Core SQL commands and syntax are standardized. Even though there might be some differences in specific implementations across relational database systems (like Oracle, MySQL, SQL Server, etc.), the foundational concepts and commands remain consistent. This adaptability means that once you learn SQL, you can work with various databases with minor adjustments.

3. **Extensibility**: Many RDBMS platforms support proprietary extensions to standard SQL, offering specialized functions, data types, and procedures to meet specific needs. For example, PostgreSQL has a range of unique data types and functions, while Microsoft's T-SQL provides procedural extensions.

4. **Dynamic SQL**: In many applications and platforms, SQL can be dynamically constructed and executed, allowing for adaptable and changing queries based on context or user input.

5. **Integration with Other Languages**: SQL integrates well with numerous programming languages, from Python to Java to C#. This allows developers to embed SQL commands within other codes, creating hybrid solutions and enabling databases to work seamlessly with application logic.

6. **Handling Structured and Semi-Structured Data**: While SQL's primary strength lies in querying structured data, modern SQL databases also provide ways to query semi-structured data formats like JSON and XML.

However, it's essential to contextualize the term „flexibility". SQL is designed for specific tasks related to relational databases, and within that domain, it's highly flexible. When compared to more general-purpose languages or NoSQL querying methods, SQL might have limitations. For example, NoSQL databases can be more „flexible" in terms of schema design and scalability in certain scenarios.

In summary, SQL is flexible within its primary domain of structured data management and querying. Still, like all tools, it has its strengths and limitations.

4 Limitations and Challenges of SQL

While SQL has been a powerful tool for data management and querying for decades, it's not without its challenges:

1. **Complexity for Complex Queries**: As the complexity of a query increases, SQL commands can become long and challenging to decipher, especially for those unfamiliar with the specific query's intent or SQL in general.
2. **Performance Issues**: Inefficiencies in SQL queries or database design can lead to slow performance. Optimizing SQL queries and understanding execution plans is crucial but requires deep expertise.
3. **Scalability Concerns**: Traditional SQL-based databases can face scalability challenges, especially when dealing with vast amounts of data or high request rates. While there are solutions, like sharding or replication, they introduce additional complexity.
4. **Rigidity in Schema Design**: Once a database schema is set, making changes can be problematic, especially in large and populated databases. This can hinder the agility required in rapidly evolving applications.
5. **Vendor Lock-in**: Although SQL is standardized, different vendors can have proprietary extensions or nuances in their SQL implementations, making migrations between different systems non-trivial.
6. **Concurrency Issues**: Managing concurrent access to data can be challenging, especially when ensuring data consistency and integrity.
7. **Not Always the Best Fit**: For certain types of applications, like those needing high scalability or flexibility in data models (such as rapidly evolving applications), NoSQL databases might be a better fit.
8. **Overhead**: Relational databases bring an overhead in terms of resources and system maintenance, especially when compared to lightweight, schema-less NoSQL databases.

It's important to understand these challenges to make informed decisions when choosing technologies for a project. Nevertheless, SQL's longevity and continued use testify to its robust capabilities and its ability to evolve and adapt in the face of changing technological landscapes.

5 The big Picture

Out there exist a lot of different RDBMS. They all share a core of equal functionality. This is ANSI-SQL standard. Here is a small overview of RDBMSs and some samples of cloud hosted engines. Entity frameworks shall form a point of contact to the databases, so that the connection from the applikation to the database is facilitated. In this field of work the publishing is located as well. Its aim is to perform updates in data or strucutre to the database in one big action.

5.1 Prominent Relational SQL Database Engines

Several relational database management systems (RDBMS) have stood the test of time and continue to be widely used in various applications and industries. Here's a list of some of the most prominent ones:

- **MySQL**: An open-source database system owned by Oracle Corporation, known for its reliability and easy-to-use nature. It's often used in web applications.
- **PostgreSQL**: A powerful, open-source object-relational database system that offers extensibility and SQL compliance. It has a strong community and is known for its robustness.
- **Oracle Database**: A commercial RDBMS that offers advanced features, scalability, and security. It's often employed in large enterprises and critical applications.
- **Microsoft SQL Server**: Developed by Microsoft, this RDBMS is known for its performance, integration with .NET, and comprehensive tools for data management.
- **SQLite**: A C-library that provides a lightweight disk-based database. It doesn't require a server setup, making it ideal for mobile applications and desktop software.
- **IBM Db2**: A family of data management products by IBM, including database servers, that boasts strong data compression, security, and scalability features.
- **MariaDB**: A fork of MySQL created by the original developers, ensuring open-source availability and incorporating enhancements and new features.
- **SAP HANA**: An in-memory RDBMS and application platform by SAP. It's known for its high-performance analytics and real-time data processing capabilities.
- **Sybase (SAP ASE)**: Originally developed by Sybase Inc., and now owned by SAP, it's known for high performance and a strong focus on data consistency.

- **SQL Anywhere (SAP SQL Anywhere)**: A relational database management system (RDBMS) designed for embedded use in applications, offering robustness, synchronization, and mobile device deployment. It's especially known for its ease of use in decentralized or occasionally connected environments.

5.2 Cloud-Based Systems Hosting RDBMS

As cloud computing continues to dominate the IT landscape, several major cloud providers have introduced managed relational database services. These services simplify the operational burden and are preferred by organizations that want to focus on application development rather than database administration.

- **Amazon Web Services (AWS)**:
  - *Amazon RDS*: Supports various database engines including MySQL, PostgreSQL, MariaDB, Oracle, and SQL Server.
  - *Amazon Aurora*: A MySQL and PostgreSQL-compatible database with higher performance and availability.
- **Microsoft Azure**:
  - *Azure SQL Database*: Managed service based on SQL Server with built-in intelligence and robust security.
  - *Azure Database for PostgreSQL, MySQL, and MariaDB*: Managed services for these open-source databases.
- **Google Cloud Platform (GCP)**:
  - *Cloud SQL*: Managed relational database supporting SQL Server, PostgreSQL, and MySQL.
  - *Cloud Spanner*: Globally distributed, scalable, and consistent relational database.
- **Oracle Cloud**:
  - *Oracle Cloud Infrastructure Database Service*: Managed Oracle database with various configurations and optimizations.
- **IBM Cloud**:
  - *Db2 on Cloud*: Managed Db2 service with high availability, disaster recovery, and scalable storage.
- **Alibaba Cloud**:
  - *ApsaraDB for RDS*: Managed service supporting MySQL, SQL Server, and PostgreSQL.

- **DigitalOcean**:
  - *Managed Databases*: Services for PostgreSQL, MySQL, and Redis.

It's worth noting that while these managed services abstract much of the administrative overhead (like backups, patching, and high availability configurations), many organizations also choose to run their database instances directly on cloud VMs. This approach offers more granular control but requires more manual management compared to using managed services.

5.3 Entity Frameworks

Entity Frameworks (EF) are Object-Relational Mapping (ORM) tools that allow developers to work with relational data using domain-specific objects, without having to write SQL code for data access. They abstract the underlying database operations and provide a higher-level interface to work with data.

- **Purpose**: EF simplifies database operations, manages database connections, executes SQL queries, and maps the results to objects in the application.
- **Advantages**:
  - Simplified data access code.
  - Automatic generation of SQL queries.
  - Change tracking and easy updates.
  - Database schema migrations.

5.4 Publishing MS SQL Server

`sqlpackage.exe` is a command-line utility for SQL Server that allows developers to perform database deployment operations by using DACPAC and BACPAC packages.

- **Publishing**: The act of applying a DACPAC to a database, which may include creating a new database or updating an existing one to match the schema and objects defined in the DACPAC.
- **Usage**:https://learn.microsoft.com/en-us/sql/tools/sqlpackage/sqlpackage?view=sql-server-ver16.

5.5 Database Publishing in Other Systems

While SQL Server uses tools like `sqlpackage.exe`, other RDBMS have their own mechanisms:

- **MySQL**: Uses tools like `mysqldump` for exporting databases and `mysql` command-line for importing.
- **PostgreSQL**: Uses tools like `pg_dump` and `pg_restore` for exporting and importing respectively.
- **Oracle**: Employs tools such as Data Pump (`expdp` and `impdp`).

5.6 OData, Node.js Methods, and Similar Protocols

OData (Open Data Protocol) is a standardized REST-based protocol for building and consuming data APIs. It allows for the querying and updating of data over the web using standard HTTP protocols.

- **Features**: Supports CRUD operations, querying, filtering, and pagination.
- **Use Cases**: Building and consuming data services, mobile and web apps, integration with other systems.

In the realm of **Node.js**, several tools, libraries, and frameworks aid in database operations and API design:

- **Express.js**: A minimal web application framework that offers methods to build robust APIs quickly and efficiently.
- **Sequelize and TypeORM**: ORMs that support multiple database backends. They abstract database interactions and allow developers to work with data in a more JavaScript-friendly way.
- **Mongoose**: An ORM for MongoDB, letting developers interact with the database using schemas and models.
- **RESTful API design**: A design methodology for creating APIs that adhere to REST principles, leveraging standard HTTP methods.
- **GraphQL**: Originally developed by Facebook, it's a query language for APIs that allows clients to request only the data they need, and it has seen adoption in Node.js environments.

Other systems and protocols similar to OData:

- **GraphQL**: As mentioned, a query language for APIs, allowing clients to request specific data structures.
- **JSON:API**: A specification for building APIs in JSON. It defines how clients should request resources and how servers should respond.

- **Falcor**: A JavaScript library created by Netflix for efficient data fetching. It aims to minimize data transfer by allowing clients to request the exact data they need.

6 The Role and Relevance of AI Systems in Development

Artificial Intelligence (AI) tools, such as ChatGPT and Copilot, have transformed the landscape of software development. Their capabilities range from offering coding suggestions to providing answers on intricate development queries. However, their utility and their imperfections both shape the way developers interact with these tools.

6.1 Use and Benefits

- **Efficiency:** AI systems significantly reduce the amount of typing time. With autosuggestions and code completions, mundane tasks become streamlined.
- **Knowledge Repository:** These tools encapsulate vast amounts of information. Whether it's a rarely used SQL command or an intricate algorithm, developers have quick access to a wealth of knowledge.
- **Ideation:** AI not only provides answers but also inspires solutions. When a developer is stuck or in need of a fresh perspective, AI can provide alternative approaches or methods.
- **Language Agnostic:** Whether it's a mainstream programming language or a niche markup language, AI models like LLMs are designed to understand and assist across a broad spectrum.

6.2 The Imperfections and the Need for Vigilance

While the capabilities of AI are vast, they are not infallible:

- **Error Prone:** Like any tool, AI systems can and do make mistakes. They might not always grasp the nuances of a specific problem or might provide generic solutions when a specialized one is needed.
- **Lack of Context:** AI doesn't have a holistic understanding of a project's entirety. It operates based on patterns and information it's been trained on, potentially lacking the contextual depth that human developers possess.
- **Over-reliance Risk:** There's a temptation to accept AI suggestions without thorough review. This can lead to unoptimized or even erroneous code if developers aren't careful.

6.3 Conclusion

AI systems like ChatGPT and Copilot are undoubtedly valuable tools in the developer's arsenal, acting as assistants that enhance productivity and creativity. However, their role is to complement, not replace, the expertise and judgment of human developers. A balanced approach, leveraging AI's strengths while being cognizant of its limitations, will yield the best results in the software development process.

6.4 This book

This book was written with a lot of help from ChatGPT. Within the available time for preparation it would not have been possible to write it with this amount of hints and aspects of the SQL language. Taking this into consideration some work of the course will be done with the help of AI. Its already that common to use that we cannot ignore those benefits.

7 Types of SQL Statements

- **DDL (Data Definition Language)**: Commands that define, manage, and modify database structures. Examples include CREATE, ALTER, DROP, and TRUNCATE.
- **DML (Data Manipulation Language)**: Commands that handle data stored in the database. Examples include SELECT, INSERT, UPDATE, and DELETE.
- **DCL (Data Control Language)**: Commands that handle permissions and access rights. Examples are GRANT and REVOKE.
- **TCL (Transaction Control Language)**: Commands that manage changes by DML statements. Examples are COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION.
- **DQL (Data Query Language)**: Pertains mainly to the SELECT command, which queries the database to retrieve and filter data.
- **DSL (Data Search Language)**: Relates to full-text search capabilities.

8 Environment and Tools

8.1 Common Tools for SQL Developers

1. **Database Management Systems (DBMS)**:
    - *Relational DBMS*: PostgreSQL, MySQL, Oracle, SQL Server, SQLite, etc.
    - *NoSQL DBMS*: MongoDB, Cassandra, Redis, etc.
2. **SQL Clients**:
    - Azure Database Studio: A modern integrated environment for managing, developing, and administering Microsoft SQL Server (M1 capable).
    - SSMS: SQL Server Management Studio (SSMS): The classical integrated environment for managing, developing, and administering Microsoft SQL Server.
    - DBeaver: Universal database tool for developers and database administrators.
    - HeidiSQL: Useful for managing MySQL, Microsoft SQL databases.
    - SQL Workbench/J: A free, DBMS-independent, cross-platform SQL query tool.
    - pgAdmin (for PostgreSQL): The most popular administration tool for PostgreSQL.
    - phpMyAdmin: Specifically designed for managing MySQL databases.
3. **Database Version Control**:
    - Liquibase: Source control for your database schema changes.
    - Flyway: Version control for databases.
    - SQL Source Control: Links your SQL Server database to your version control system.
4. **Database Documentation**:
    - Schemaspy: Generates database diagrams and documentation.
    - Dataedo: Tool for creating, maintaining, and visualizing database documentation.
5. **Performance Profilers and Monitoring**:
    - SolarWinds Database Performance Analyzer: Monitors and analyzes database performance.
    - SQL Diagnostic Manager for SQL Server: Performance monitoring, alerting, and diagnostics.
    - pgBadger: A PostgreSQL log analyzer.

6. **ETL Tools (Extract, Transform, Load)**:
    - Talend: Open source data integration platform.
    - Apache NiFi: An easy to use, powerful, and reliable system to process and distribute data.
    - Microsoft SSIS (SQL Server Integration Services): A tool for data extraction, transformation, and loading (ETL).
7. **Database Backup Solutions**:
    - Bacula: Open source, enterprise-level computer backup system.
    - Barman (for PostgreSQL): Backup and recovery manager.
8. **Data Modeling Tools**:
    - ER/Studio: Data architect tool for modeling and design.
    - Navicat Data Modeler: Tool for designing and managing database structures.
    - MySQL Workbench: Provides data modeling, SQL development, and more.
9. **SQL Formatters & Beautifiers**:
    - Poor SQL: An online SQL formatter.
    - SQLFormat: Another online SQL beautifier and formatter.
10. **Comparison & Sync Tools**:
    - SQL Compare: Compares and synchronizes SQL Server database schemas.
    - ApexSQL Diff: SQL Server database comparison and synchronization tool.
11. **Testing Tools**:
    - tSQLt: Unit testing framework for SQL Server.
    - SQLUnit: A unit testing framework for SQL stored procedures and functions.
12. **Data Visualization Tools**:
    - Tableau: Data visualization software that is used for converting raw data into an understandable format; graphs, visuals.
    - Power BI: Microsoft's data visualization tool.
13. **Database Deployment Automation**:
    - Octopus Deploy: Automated deployment for .NET, SQL Server, and more.
    - DBmaestro: Database DevOps solutions to keep database in sync with your development.
14. **API Integration Tools**:

- Postman: For testing API endpoints that interact with your database.
- DreamFactory: Turns any SQL DB into a comprehensive REST API platform.

8.2 Database Migration

- **AWS Database Migration Service (DMS)**:
  - Supports migration to and from most widely used commercial and open-source databases.
  - Enables live migration scenarios.
- **Oracle SQL Developer**:
  - Contains a feature called „Migration Workbench“.
  - Supports migration from several non-Oracle databases to Oracle.
- **Flyway**:
  - Mostly used for versioning, but can be leveraged for migrations.
  - Supports various DBMSs and allows custom migration scripts.
- **pgloader**:
  - Specifically designed to load data into PostgreSQL from various sources, including other databases.
- **Microsoft's SQL Server Migration Assistant (SSMA)**:
  - Supports migration from Oracle, MySQL, Sybase, DB2, and Access to SQL Server and Azure SQL.
- **SwissSQL**:
  - Offers data migration and SQL migration tools for several databases.
- **Full Convert**:
  - Directly migrates between many different databases.
  - Supports a wide array of source and target databases.
- **ESF Database Migration Toolkit**:
  - Supports migration between various database formats.
  - Allows direct reading from ODBC, MySQL, MariaDB, SQL Server, and more.
- **Ispirer MnMTK**:
  - Converts databases and transfers data between SQL Server, Oracle, MySQL, and more.
  - Provides customization capabilities for migrations.

8.3 ETL vs. Database Migration

The term ETL stands for *Extract, Transform, Load*.

The ETL process encompasses:

1. **Extracting** data from different source systems.
2. **Transforming** the extracted data, which can involve tasks like cleaning, filtering, enriching, and applying various business rules.
3. **Loading** the transformed data into a target system, often a data warehouse.

Database migration tools, like those previously mentioned, can have functionalities overlapping with ETL processes. These tools are primarily crafted for transferring data from one database flavor or version to another. During such migrations, some transformations might occur to ensure the data's compatibility with the destination system.

However, traditional ETL tools are often more comprehensive in their transformation capabilities. They're typically utilized to:

- Consolidate data from various sources.
- Conduct complex transformations.
- Load the final data into a target system.

Moreover, while ETL tools frequently handle multiple heterogeneous data sources (such as databases, flat files, APIs, etc.), migration tools are mainly geared towards database-to-database transitions.

Popular examples of traditional ETL tools include:

- **Talend**
- **Informatica PowerCenter**
- **Microsoft SQL Server Integration Services (SSIS)**
- **Apache Nifi**
- **Pentaho Data Integration**

In conclusion, even though there's an overlap, especially in the *Extract* and *Load* components, database migration is distinct from ETL. Database migration tools might perform some fundamental transformations, but they generally lack the extensive transformation capabilities found in dedicated ETL tools.

9 Introduction to NoSQL Databases

9.1 Technical Differences Between NoSQL Database Paradigms

Each type of NoSQL database is optimized for a specific set of use cases and offers unique technical attributes:

- **Document-Based Databases**:
  - *Data Structure*: Data is stored in document format, typically JSON or BSON.
  - *Schema Flexibility*: Dynamic schemas allow varied structures and fields within a single collection.
  - *Use Cases*: Best for content management systems, e-commerce platforms, and mobile applications where flexibility in data modeling is essential.
- **Column-Based Databases**:
  - *Data Structure*: Data is organized by columns rather than rows, allowing for efficient reads/writes of columnar data.
  - *Scalability*: Typically distributed and excel in large-scale data storage scenarios.
  - *Use Cases*: Suitable for big data analytics, time-series data, and any use case where aggregation queries are predominant.
- **Key-Value Stores**:
  - *Data Structure*: Uses a simple key-value pair mechanism.
  - *Performance*: Extremely fast read/writes since data retrieval is based on a unique key.
  - *Use Cases*: Ideal for caching, session management, and other scenarios where rapid, simple lookups are needed.
- **Graph Databases**:
  - *Data Structure*: Data is stored as nodes and edges, representing entities and their relationships.
  - *Traversal Efficiency*: Optimized for queries that traverse relationships and find patterns.
  - *Use Cases*: Perfect for social networks, recommendation engines, and any scenario with interconnected data.

9.2 Types of NoSQL Databases

- **Document-Based**: MongoDB, CouchDB.
- **Column-Based**: Cassandra, HBase.
- **Key-Value Stores**: Redis, Riak.
- **Graph Databases**: Neo4j, OrientDB.

9.3 Key Features and Considerations

- **Flexibility**: Dynamic Schema.
- **Scalability**: Horizontal scaling and distributed architecture.
- **Performance**: Optimized for specific use cases, with in-memory options.
- **CAP Theorem**: Trade-offs between Consistency, Availability, and Partition Tolerance.
- **JSON and APIs**: Many use JSON and offer RESTful APIs.
- **Consistency Models**: Eventual consistency.
- **Real-Time Processing**: Suited for real-time data processing.
- **Challenges**: Maturity, standardization, and complexity.
- **Use Cases**: Big data, content management, gaming, IoT, and social networks.

9.4 Inverted Indexes in NoSQL

Inverted indexes, key for rapid full-text search capabilities, are significant in the NoSQL world. They provide efficiency in text searches, space-saving, real-time search, term frequency and ranking, and are foundational in search engines.

10 The Definition of the Standard for SQL

After discussing significant parts of SQL-DML with SELECT queries and INSERT, UPDATE, and DELETE modification commands, an overview of all components of the standard will be presented in this section. It reveals that mostly parts of the so-called Part 2 were discussed, which simultaneously represent the core of the practically implemented part of the standard.

10.1 Structure

The SQL standard, as defined by the International Standard Organisation (ISO), is currently available in the 2016 version, following the previous 2011 version. The standard comprises 10 parts, which due to certain parts being withdrawn or never validated over time, aren't consistently numbered but carry numbers from 1 to 14. In addition, there are 7 multimedia and application packages and the yet-to-be-released Part 15 (multidimensional arrays). The individual parts are:

Part 1 Framework: Overview and overarching definitions

Part 2 Foundation: The core SQL language description

Part 3 Call-Level Interface: API descriptions for accessing SQL databases (C and COBOL)

Part 4 Persistent Stored Modules: Server-side programs (stored procedures)

Part 9 Management of External Data: Access to data outside the database

Part 10 Object Language Bindings: SQL embedded in Java

Part 11 Information and Definition Schemas: Definition of schemas

Part 13 Routines and Types Using the Java Programming Language: Java programs within the database itself.

Part 14 XML-related Specifications: XML data types and functions

An impression of the scope of the standard is evident when considering that Part 2 alone, of particular relevance, spans 1579 pages. In practice, not all parts have ever been implemented. On the contrary, only a few parts have been realized in existing database management systems, often based on older versions of the standard, sometimes even just based on SQL92. SQL defines various compliance levels (entry, intermediate, full), representing subsets of the standard and allowing database manufacturers to declare their systems compliant with these levels. The standard differentiates between mandatory and optional features.

10.2 Extensions

Individual versions of the standard contained extensions motivated by current developments, and some were already implemented in certain systems. For instance, the 1999 SQL standard (SQL 3) introduced widely used additions like control structures (IF, CASE, ...), triggers for action initiation, regular expressions, arrays, and structured data types. It also introduced embedding in Java programs and vice versa. The 2003 version particularly introduced the definition of XML elements and initial access functions. Moreover, it defined how column values can automatically be filled, especially interesting for technical primary keys. Window functions were also added. The 2008 version substantially expanded XML embedding. Import and export, storage and generation of XML documents were introduced, as well as the ability to use XQuery for combined queries of SQL and XML data. The 2008 version introduced the TRUNCATE command and the FETCH clause, along with improvements in ORDER BY. The 2011 version introduced the PERIOD FOR command to consider temporal references. Extensions were also made to window functions and the FETCH clause. The 2016 version now standardizes JSON support, which is already present in most databases, but not in a standardized form. Moreover, the new match_recognize clause allows pattern identification in rows using regular expressions. Finally, the formatting of date and time data has been standardized, the output of CSV files simplified, and several other details have been added. Overall, the situation reflects typical standardization and the supplementation of functions already available in practice. The standardization of JSON represents the adoption of a de-facto standard.

This text was translated by ChatGPT from[46].

11 ANSI SQL Reserved Words

The following list contains reserved words as per the ANSI SQL standard. These words should not be used as identifiers (such as table or column names) unless enclosed in double quotes or brackets:

- ABSOLUTE
- ACTION
- ADD
- ALL
- ALLOCATE
- ALTER
- AND
- ANY
- ARE
- AS
- ASC
- ASSERTION
- AT
- AUTHORIZATION
- AVG
- BEGIN
- BETWEEN
- BIT
- BOOLEAN
- BOTH
- BY
- CASCADE
- CASE
- CAST
- CATALOG
- CHAR
- CHARACTER
- CHECK
- CLOSE
- COALESCE
- COLLATE
- COLLATION
- COLUMN
- COMMIT
- CONNECT
- CONNECTION
- CONSTRAINT
- CONSTRAINTS
- CONTINUE
- CONVERT
- CORRESPONDING
- COUNT
- CREATE
- CROSS
- CURRENT
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURRENT_USER
- CURSOR
- DATE
- DAY
- DEALLOCATE
- DEC
- DECIMAL
- DECLARE
- DEFAULT
- DEFERRABLE
- DEFERRED
- DELETE
- DESC
- DESCRIBE
- DESCRIPTOR
- DIAGNOSTICS
- DISCONNECT
- DISTINCT
- DOMAIN
- DOUBLE
- DROP
- ELSE
- END
- END-EXEC
- ESCAPE
- EXCEPT
- EXCEPTION
- EXEC
- EXECUTE
- EXISTS
- EXTERNAL
- EXTRACT
- FALSE
- FETCH
- FIRST
- FLOAT
- FOR
- FOREIGN
- FOUND
- FROM
- FULL
- FUNCTION
- GET
- GLOBAL
- GO

- GOTO
- GRANT
- GROUP
- HAVING
- HOUR
- IDENTITY
- IMMEDIATE
- IN
- INDICATOR
- INITIALLY
- INNER
- INPUT
- INSENSITIVE
- INSERT
- INT
- INTEGER
- INTERSECT
- INTERVAL
- INTO
- IS
- ISOLATION
- JOIN
- KEY
- LANGUAGE
- LAST
- LEADING
- LEFT
- LEVEL
- LIKE
- LOCAL
- LOWER
- MATCH
- MAX
- MIN
- MINUTE
- MODULE
- MONTH
- NAMES
- NATIONAL
- NATURAL
- NCHAR
- NEXT
- NO
- NOT
- NULL
- NULLIF
- NUMERIC
- OCTET_LENGTH
- OF
- ON
- ONLY
- OPEN
- OPTION
- OR
- ORDER
- OUTER
- OUTPUT
- OVERLAPS
- PAD
- PARTIAL
- POSITION
- PRECISION
- PREPARE
- PRESERVE
- PRIMARY
- PRIOR
- PRIVILEGES
- PROCEDURE
- PUBLIC
- READ
- REAL
- REFERENCES
- RELATIVE
- RESTRICT
- REVOKE
- RIGHT
- ROLLBACK
- ROWS
- SCHEMA
- SCROLL
- SECOND
- SECTION
- SELECT
- SESSION
- SESSION_USER
- SET
- SIZE
- SMALLINT
- SOME
- SPACE
- SQL
- SQLCODE
- SQLERROR
- SQLSTATE
- SUBSTRING
- SUM
- SYSTEM_USER
- TABLE
- TEMPORARY
- THEN
- TIME
- TIMESTAMP
- TIMEZONE_HOUR
- TIMEZONE_MINUTE

- TO
- TRAILING
- TRANSACTION
- TRANSLATE
- TRANSLATION
- TRIM
- TRUE
- UNION
- UNIQUE
- UNKNOWN
- UPDATE
- UPPER
- USAGE
- USER
- USING
- VALUE
- VALUES
- VARCHAR
- VARYING
- VIEW
- WHEN
- WHENEVER
- WHERE
- WITH
- WORK
- WRITE
- YEAR
- ZONE

12 Common SQL Functions and Procedures

The following list contains commonly used functions and procedures in SQL. Their availability and exact usage might vary across different SQL implementations:

String Functions

- CHAR_LENGTH
- CONCAT
- LOWER
- LTRIM
- RTRIM
- POSITION
- REPLACE
- SUBSTRING
- TRIM
- UPPER

Numeric Functions

- ABS
- CEIL/CEILING
- FLOOR
- MOD
- POWER
- ROUND
- SIGN
- SQRT

Date and Time Functions

- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- DATE_TRUNC
- EXTRACT
- NOW
- AGE

Conversion Functions

- CAST
- CONVERT
- TO_CHAR

Aggregate Functions

- AVG
- COUNT
- MAX
- MIN
- SUM
- GROUP_CONCAT

System Functions

- COALESCE
- NULLIF
- CASE
- IIF (SQL Server specific)

Advanced Functions

- RANK
- DENSE_RANK
- ROW_NUMBER
- LAG
- LEAD
- FIRST_VALUE
- LAST_VALUE
- NTH_VALUE

Logical Functions

- ALL
- ANY
- EXISTS
- IN
- SOME

13 Imperative Aspects of SQL

While SQL is primarily known for its declarative nature in querying data, it is enriched with imperative constructs, especially when one dives into the procedural extensions of SQL provided by various relational database management systems (RDBMS). These procedural extensions, such as PL/SQL (Oracle) and T-SQL (SQL Server), enable more complex operations and encapsulate imperative logic within the database system.

1. Variables

Procedural SQL allows for the declaration and utilization of variables within stored procedures, triggers, and functions, facilitating dynamic data manipulation and control flow.

2. Control Flow Statements

Conditional Statements

- IF ... THEN ... ELSE
- CASE

Looping Constructs

- FOR loops
- WHILE loops
- REPEAT ... UNTIL (in some RDBMS)

3. Cursors

Cursors provide a mechanism to navigate through a set of rows, processing each row individually. This is especially useful when an operation needs to be performed row-by-row rather than on an entire set.

4. Exception Handling

Exception handling in procedural SQL allows developers to capture and handle errors gracefully. By defining specific handlers for different exceptions, one can ensure that errors are addressed appropriately, be it through logging, user notifications, or other corrective measures.

5. Procedures and Functions

These are blocks of reusable code that can be invoked as needed. While both can encapsulate a series of SQL and procedural statements, functions return a value, whereas procedures perform actions without necessarily returning a value.

6. Triggers

Triggers are special types of stored procedures that run automatically when specific events occur in the database, such as data insertion, update, or deletion.

14 Functional Aspects of SQL

SQL, while inherently a declarative language, shares some intriguing similarities with functional programming paradigms. Below we delve into these functional characteristics inherent in SQL:

1. Declarative Nature

SQL's core principle is its declarative nature, mirroring functional programming's emphasis on declaring *what* rather than *how*.

2. Stateless Computation

Each SQL query operates in a stateless manner, evaluating data without any lingering memory of previous executions.

3. Higher-order Functions

Aggregate functions in SQL operate on sets of data, reminiscent of higher-order functions in functional programming.

4. Immutable Data Principle

SQL often produces new result sets without altering the original data, resonating with FP's immutability ideal.

5. Functional Transformations

- 'SELECT' as a map operation.
- 'WHERE' as a filter operation.

6. Expression Evaluation

SQL relies on transforming data based on expressions, aligning with functional programming's evaluative nature.

7. Recursion

Certain SQL dialects embrace recursion, especially through constructs like Common Table Expressions (CTEs).

8. Composability

The ability to nest SQL queries and use subqueries as modular blocks underpins the concept of composability.

15 Normalization in SQL

Normalization in SQL refers to the design process that involves organizing the attributes and tables of a relational database. The primary aim is to eliminate redundancy and ensure data integrity. It divides the larger tables into smaller tables and links them with relationships. The process is done through a series of rules known as „normal forms“ that a database must adhere to.

15.1 Why Normalize?

- **Eliminate Redundant Data:** Inconsistent data can cause a variety of problems, including wasting storage space and leading to inconsistencies.
- **Data Integrity:** When data is divided into different tables, it ensures that the data remains consistent across the database.
- **Efficient Queries:** Normalization can lead to simpler, more efficient queries.

15.2 Introduction to Normal Forms

There are several normal forms, but we'll briefly touch on the first three as they're the most commonly used:

15.2.1 First Normal Form (1NF)

A table is in 1NF if:

- It only contains atomic (indivisible) values; there are no repeating groups or arrays.
- Each column contains values of a single type.
- Each column has a unique name.

15.2.2 Second Normal Form (2NF)

A table is in 2NF if:

- It is in 1NF.
- All non-key columns are fully functionally dependent on the primary key.

15.2.3 Third Normal Form (3NF)

A table is in 3NF if:

- It is in 2NF.
- All the attributes in the table are functionally dependent only on the primary key.

15.3 Examples

Preface: Let's consider a university database with a table that stores information about students and the courses they're enrolled in.

```
CREATE TABLE Students_Courses (
    StudentID INT,
    StudentName VARCHAR(50),
    CourseID INT,
    CourseName VARCHAR(50),
    Instructor VARCHAR(50)
);
```

Listing 1: Students_Courses Table

This table is not normalized. Let's look at why and how to normalize it:
1NF: Divide the table so that each column contains atomic values.

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    StudentName VARCHAR(50)
);
```

Listing 2: Students Table for 1NF

```
CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    CourseName VARCHAR(50),
    Instructor VARCHAR(50)
);
```

Listing 3: Courses Table for 1NF

2NF: Ensure all non-key columns are fully functionally dependent on the primary key. The tables created above already satisfy this rule.
3NF: Remove columns that are not dependent on the primary key. The 'Instructor' column depends more on the 'CourseID' than the combination of 'StudentID' and 'CourseID', so it should only be in the 'Courses' table.

The tables above are already in 3NF based on our breakdown.

15.4 Conclusion

Normalization is essential for maintaining data integrity and optimizing the performance of relational databases. It helps in designing the schema of the database in such a way that duplication is minimized, and data retrieval and maintenance become more efficient.

16 Database Design: From Object-Oriented to Relational Model

16.1 Problem Definition

Customers buy articles from stock. Articles are stored in stock until they are bought. We need to track which customer bought which articles and how much stock remains.

16.2 Object-Oriented Approach

Entities:

1. Customer
2. Article
3. Stock
4. Transaction (to record each purchase)

Relationships:

1. A Customer can have many Transactions.
2. A Transaction is associated with one Customer.
3. A Transaction can have many Articles.
4. An Article can have many Transactions.
5. Stock is associated with each Article.

16.3 Database Design in T-SQL

16.3.1 Creating the Customer Table

```
    CREATE TABLE Customer (
    CustomerID UNIQUEIDENTIFIER DEFAULT NEWID() PRIMARY KEY CLUSTERED
        ,
    Name NVARCHAR(255) NOT NULL,
    Email NVARCHAR(255) UNIQUE NOT NULL
);
```

Listing 4: Creating Customer Table

16.3.2 Creating the Article Table

```
    CREATE TABLE Article (
    ArticleID UNIQUEIDENTIFIER DEFAULT NEWID() PRIMARY KEY CLUSTERED,
    Name NVARCHAR(255) NOT NULL,
    Description NVARCHAR(1000),
    Price DECIMAL(10,2) NOT NULL
);
```

Listing 5: Creating Article Table

16.3.3 Creating the Stock Table

```
    CREATE TABLE Stock (
    StockID UNIQUEIDENTIFIER DEFAULT NEWID() PRIMARY KEY CLUSTERED,
    ArticleID UNIQUEIDENTIFIER FOREIGN KEY REFERENCES Article(
       ArticleID),
    Quantity INT NOT NULL
);
```

Listing 6: Creating Stock Table

16.3.4 Creating the Transaction Table

```
    CREATE TABLE Transaction (
    TransactionID UNIQUEIDENTIFIER DEFAULT NEWID() PRIMARY KEY
       CLUSTERED,
    CustomerID UNIQUEIDENTIFIER FOREIGN KEY REFERENCES Customer(
       CustomerID),
    TransactionDate DATE NOT NULL
);
```

Listing 7: Creating Transaction Table

16.3.5 Creating the TransactionDetail Table

```
    CREATE TABLE TransactionDetail (
    TransactionDetailID UNIQUEIDENTIFIER DEFAULT NEWID() PRIMARY KEY
       CLUSTERED,
    TransactionID UNIQUEIDENTIFIER FOREIGN KEY REFERENCES Transaction
       (TransactionID),
    ArticleID UNIQUEIDENTIFIER FOREIGN KEY REFERENCES Article(
       ArticleID),
    Quantity INT NOT NULL
```

```
);
```

Listing 8: Creating TransactionDetail Table

16.4 Differences in Other SQL Dialects

The above SQL is written in T-SQL, which is specific to Microsoft SQL Server. In other SQL dialects, there might be subtle differences, especially with the 'NEWID()' function which is used to generate a new GUID.

For example:

- In PostgreSQL, you might use the 'UUID' type with a default of the 'uuid_generate_v4()' function (requires the 'pgcrypto' module).
- In MySQL, the equivalent type is 'UUID' and you'd use the 'UUID()' function.

For the sake of this book, we'll continue with the T-SQL dialect, but readers should be aware of these potential differences.

16.5 Normalization: Deconstructing the Customer Object

Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like insertion, update, and deletion anomalies. The main idea is to have a table for each real-world entity or relationship.

Consider a 'Customer' object represented in JSON:

```
    {
    "CustomerID": "some-unique-id",
    "Name": "John Doe",
    "Addresses": [
        {"type": "home", "address": "123 Main St"},
        {"type": "work", "address": "456 Elm St"}
    ],
    "Emails": [
        "john.doe@example.com",
        "johnd@work.com"
    ],
    "PhoneNumbers": [
        "123-456-7890",
        "098-765-4321"
    ]
}
```

Listing 9: Customer JSON representation

From the JSON representation, we can identify multiple repeating groups - addresses, emails, and phone numbers. To achieve normalization, we will decompose this structure into multiple tables.

16.5.1 First Normal Form (1NF)

To achieve the first normal form:

- Each table should have a primary key.
- No repeating groups or arrays.
- Each column should contain atomic (indivisible) values.

Using 1NF, our JSON structure would be decomposed into the following tables:

1. **Customer** - with attributes ‘CustomerID‘ and ‘Name‘.
2. **Address** - with attributes ‘AddressID‘, ‘CustomerID‘, ‘Type‘, and ‘Address‘.
3. **Email** - with attributes ‘EmailID‘, ‘CustomerID‘, and ‘EmailAddress‘.
4. **PhoneNumber** - with attributes ‘PhoneID‘, ‘CustomerID‘, and ‘PhoneNumber‘.

However, this structure can still be improved upon. Consider the scenario where two customers share the same phone number (like a home phone). Storing this in the current ‘PhoneNumber‘ table would result in redundancy.

16.5.2 Second Normal Form (2NF)

To achieve the second normal form:

- The table should be in 1NF.
- All non-key attributes should be fully functionally dependent on the primary key.

In our context, the tables are already in 2NF since all non-key attributes are dependent on the primary keys. However, as we refine our understanding of the domain, changes may be needed.

16.5.3 Third Normal Form (3NF)

To achieve the third normal form:

- The table should be in 2NF.
- There should be no transitive functional dependencies.

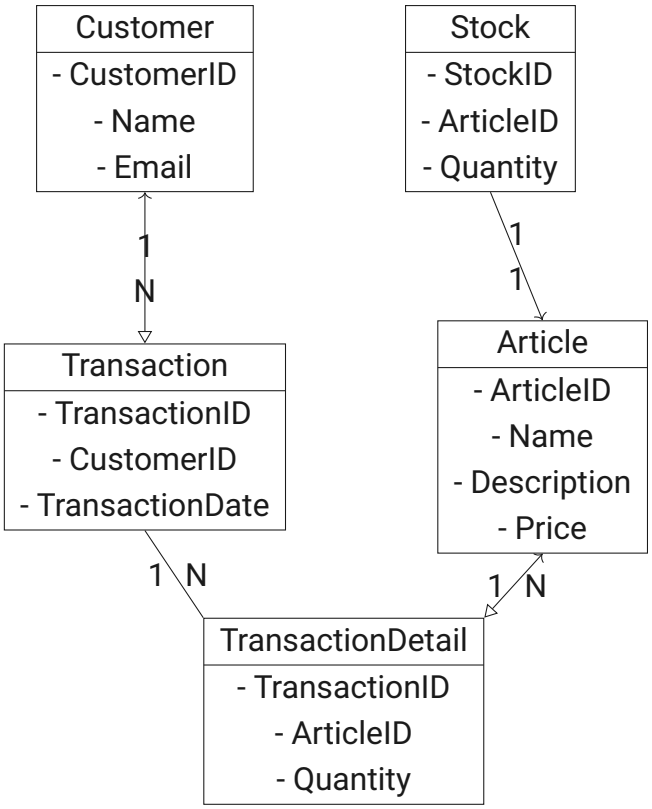

Abbildung 1: ER diagram representing the database design with referential integrity

Consider the case where the address type (home, work, etc.) has additional metadata like a description. This would introduce a transitive dependency: 'Customer' -> 'Address' -> 'AddressType'. To achieve 3NF, we'd need to move 'AddressType' to a separate table.

In this step-by-step approach, we've seen how a complex JSON object can be decomposed into multiple related tables in a normalized form. This process ensures data integrity and reduces redundancy.

16.6 Understanding Crow's Foot Notation

Crow's Foot Notation, also known as „IE notation" due to its origin with the Information Engineering methodology, is a graphical representation to depict relationships between tables or entities in a database. It's an intuitive notation that uses a combination of lines and symbols to express cardinality and optionality in relationships.

- **One (1):** A straight line indicates the „one" side of a relationship. In some cases, this straight line may be preceded by a '<' symbol to emphasize optionality, indicating that there can be zero or one occurrences in the relationship.
- **Many (N):** The „crow's foot", a three-pronged fork symbol, indicates the „many" side of a relationship.

- **Optional One (0 or 1):** The symbol ‘<‘ represents that the relationship is optional on the „one“ side, indicating that there can be zero or one instances in the relationship.
- **Optional Many (0 to N):** A combination of the ‘<‘ symbol with the crow’s foot denotes that there can be zero to many occurrences.

For example, consider the relationship between a ‘Customer‘ and a ‘Transaction‘. In our scenario, a ‘Customer‘ can have zero or more ‘Transactions‘, which is represented by a line starting with the ‘<‘ symbol (optional) at the ‘Customer‘ end and ending with the crow’s foot at the ‘Transaction‘ end. This represents an optional one-to-many relationship from the ‘Customer‘ to the ‘Transaction‘.

It’s worth noting that while Crow’s Foot Notation is widely adopted due to its clarity, there are other notations like Chen’s notation and UML that use different symbols and conventions to depict relationships. The choice of notation often depends on the context, tool, or personal preference.

17 Advanced SQL Features - Overview

SQL, being a powerful language for managing relational databases, has a wide range of advanced features. This grouped list is far from being complete. You are encouraged to find more details of the language in the specific documentations of the individual languages.

17.1 Database Features and Operations

- **Common Table Expressions (CTEs):** Allow the creation of temporary result sets that can be easily referenced within a ‘SELECT‘, ‘INSERT‘, ‘UPDATE‘, or ‘DELETE‘ statement. Often used for recursive queries.
- **Window Functions:** Provide a way to perform calculations across a set of rows related to the current row, useful for tasks like running totals or ranking.
- **Pivoting:** A way to transform data from rows into columns.
- **JSON Support:** Modern databases often offer functions to query and manipulate JSON formatted data.
- **Full-Text Search:** Efficiently search large text fields for specific words or phrases.
- **Transactions:** Ensure data integrity by treating a set of SQL operations as a single unit, which can be either fully completed or fully rolled back.
- **Partitioning:** Divides a table into smaller, more manageable pieces, and improves performance.
- **Indexes and Referential integrity:** Improve the speed of data retrieval operations on a database and keep things in good order.
- **Constraints:** You cannot do what you want. Obey the rules.

17.1.1 Common Table Expressions (CTEs) explained

Common Table Expressions (CTEs) offer a more readable and modular approach to writing SQL queries. Think of CTEs as temporary result sets or inline views that exist just for one query. They allow for the decomposition of complex queries into simpler parts, making the SQL more understandable. CTEs are particularly useful for recursive queries, enabling the traversal of hierarchical data structures with ease. By breaking down the query logic, CTEs not only enhance clarity but can also improve maintainability, as changes might only need to be made in one place.

Sample prerequisites

```
    -- Table definitions for purchase_history
CREATE TABLE purchase_history (
    name VARCHAR(255),
    purchase_amount DECIMAL(10,2)
);

-- Sample data for purchase_history
INSERT INTO purchase_history (name, purchase_amount)
VALUES
('John Doe', 500.50),
('Jane Smith', 600.75),
('Robert Brown', 900.00),
('Emily White', 1200.00);
```

Listing 10: Table definition and sample data for purchase_history

```
    -- Table definitions for products and sales
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(255),
    price DECIMAL(10,2)
);

CREATE TABLE sales (
    sale_id INT PRIMARY KEY,
    product_id INT,
    quantity_sold INT
);

-- Sample data for products
```

```sql
INSERT INTO products (product_id, product_name, price)
VALUES
(1, 'Laptop', 1000.00),
(2, 'Mouse', 25.00),
(3, 'Keyboard', 50.00);

-- Sample data for sales
INSERT INTO sales (sale_id, product_id, quantity_sold)
VALUES
(1, 1, 5),
(2, 2, 50),
(3, 3, 20);
```

Listing 11: Table definition and sample data for products and sales

```sql
    -- Table definitions for employees
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(255),
    manager_id INT
);

-- Sample data for employees
INSERT INTO employees (employee_id, employee_name, manager_id)
VALUES
(1, 'CEO', NULL),
(2, 'Director A', 1),
(3, 'Director B', 1),
(4, 'Manager A', 2),
(5, 'Employee A', 4),
(6, 'Employee B', 4);
```

Listing 12: Table definition and sample data for employees

Using Common Table Expressions (CTEs) to Simplify Subqueries

In SQL, there are often situations where we want to structure our queries in such a way that we first create a subset of our data and then perform further operations on this subset. One common approach to do this is by using subqueries. However, subqueries can sometimes make your SQL code lengthy and difficult to read, especially when multiple subqueries are nested. This is where Common Table Expressions, commonly referred to as CTEs, come into play.

A CTE is a temporary result set that you can reference within a 'SELECT', 'INSERT', 'UPDATE', or 'DELETE' statement. They can be thought of as alternatives to derived tables

and subqueries. The primary advantage of using CTEs is improving the readability and maintenance of your SQL code.

Here's a simple example to illustrate the concept:

```
    SELECT name, total_purchase
    FROM (
        SELECT name, SUM(purchase_amount) as total_purchase
        FROM purchase_history
        GROUP BY name
    ) AS subquery
    WHERE total_purchase > 1000;
```

Listing 13: Using subquery

The above code calculates the total purchase for each customer and then filters out the customers who have a total purchase greater than 1000. It uses a subquery to first calculate the total purchase for each customer.

Now, let's replace the subquery with a CTE:

```
    WITH CustomerPurchase AS (
        SELECT name, SUM(purchase_amount) as total_purchase
        FROM purchase_history
        GROUP BY name
    )

    SELECT name, total_purchase
    FROM CustomerPurchase
    WHERE total_purchase > 1000;
```

Listing 14: Using CTE

The CTE version of the query separates the creation of the temporary result set (using the 'WITH' clause) from the main query, which can make the SQL code cleaner and easier to understand.

In summary, while subqueries are powerful and allow for complex data manipulations, CTEs provide a more readable and structured way to break down complex queries, making your SQL code easier to maintain and understand.

The Power of Multiple CTEs

When working with intricate SQL queries that involve multiple stages of data transformation or aggregation, it might be necessary to use more than one Common Table Expression (CTE). The beauty of CTEs is that you're not limited to using just one. You can chain multiple CTEs together to create intermediate, temporary result sets that can be referenced in

subsequent CTEs or in the main query.

Consider two tables: ‘products‘ (containing product details) and ‘sales‘ (recording product sales). To determine the total revenue generated from each product and then pinpoint products surpassing a certain revenue threshold, you can leverage multiple CTEs.

```
WITH ProductRevenue AS (
    SELECT
        p.product_id,
        p.product_name,
        p.price,
        s.quantity_sold,
        p.price * s.quantity_sold AS revenue
    FROM products AS p
    JOIN sales AS s ON p.product_id = s.product_id
),

TopProducts AS (
    SELECT
        product_id,
        product_name,
        SUM(revenue) as total_revenue
    FROM ProductRevenue
    GROUP BY product_id, product_name
)

SELECT product_name, total_revenue
FROM TopProducts;
```

Listing 15: Harnessing multiple CTEs for complex querying

Here’s the breakdown of the example:

1. The ‘ProductRevenue‘ CTE determines the revenue for each product sale by combining the ‘products‘ and ‘sales‘ tables.
2. The ‘TopProducts‘ CTE aggregates this data, summing up the total revenue for each distinct product.
3. The main query then filters out products exceeding a revenue threshold (e.g., 5000).

This layered approach not only maintains clarity in your SQL code but also ensures that each stage of data processing is isolated and understandable. It’s a testament to the adaptability and strength of CTEs in SQL.

Generating Large Number Sequences Using Cartesian Products in Standard CTEs

Generating large sequences of numbers can be efficiently achieved using standard CTEs (non-recursive) combined with Cartesian products. By repeatedly joining a small set with itself, we experience exponential growth in the number of rows.

Consider the following SQL code:

```
CREATE TABLE #numbers (Number INT NOT NULL PRIMARY KEY);

WITH
    L0 AS (SELECT 1 AS C UNION ALL SELECT 1),
    L1 AS (SELECT 1 AS C FROM L0 AS A, L0 AS B),
    L2 AS (SELECT 1 AS C FROM L1 AS A, L1 AS B),
    L3 AS (SELECT 1 AS C FROM L2 AS A, L2 AS B),
    L4 AS (SELECT 1 AS C FROM L3 AS A, L3 AS B),
    Nums AS (SELECT ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS
       N FROM L4)
INSERT INTO #numbers SELECT N FROM Nums WHERE N <= 2000000000;

SELECT TOP 50 Number FROM #numbers ORDER BY Number DESC;
```

Listing 16: Generating large sequences using Cartesian products in standard CTEs

The breakdown of this technique is as follows:

1. Starting with the CTE ‘L0‘, we have a set containing two rows.
2. Each subsequent CTE (‘L1‘ through ‘L5‘) doubles its result set by performing a Cartesian join on the previous set with itself.
3. The final CTE, ‘Nums‘, uses the ‘ROW_NUMBER()‘ function to assign a unique sequential number to each row.
4. The result is then filtered to only include numbers up to a defined maximum (‘@MaxDatabaseSize‘) and inserted into the ‘#numbers‘ temporary table.

This approach is highly efficient for generating vast sequences rapidly. However, it’s essential to be mindful of the exponential growth when deciding on the number of Cartesian joins, to prevent unintended strains on system resources.

Introduction to Recursive CTEs

Common Table Expressions (CTEs) can be more than just a way to create temporary result sets; they can also be recursive. A recursive CTE is a CTE that references itself. This self-referencing capability makes recursive CTEs particularly useful for querying hierarchical data or breaking down complex problems into iterative steps.

Recursive CTEs also allow us to iterate through data sets, making them useful for generating sequences. Let's examine a straightforward example that generates numbers from 1 to 10 using a recursive CTE:

```
    WITH RecursiveNumbers AS (
    -- Anchor member (base case)
    SELECT 1 AS number

    UNION ALL

    -- Recursive member
    SELECT number + 1
    FROM RecursiveNumbers
    WHERE number < 10
)

SELECT number FROM RecursiveNumbers;
```

Listing 17: Simple recursive CTE for generating numbers

Here's a step-by-step explanation:

1. The anchor member initializes the sequence with the number '1'.
2. The recursive member then increments the current number by '1' and continues doing so until the number reaches '10'.
3. The 'WHERE' clause in the recursive member ensures termination once the number exceeds or equals 10.

This recursive CTE offers a clear example of how iterative processes can be implemented in SQL, providing an ordered sequence of numbers as a result.

A Basic Recursive CTE

A classic use case for recursive CTEs is navigating through hierarchical data like organization charts or family trees. For instance, consider a table 'employees' where each employee has an ID and a manager's ID, pointing to another employee in the same table. A recursive CTE can help trace the chain of command from a given employee upwards to the CEO.

Here's a simple example of a recursive CTE that lists an employee's hierarchy based on the 'employees' table:

```
    WITH RecursiveHierarchy AS (
    -- Anchor member (base case)
    SELECT employee_id, employee_name, manager_id
```

```
    FROM employees
    WHERE employee_id = 5  -- starting with a specific employee

    UNION ALL

    -- Recursive member
    SELECT e.employee_id, e.employee_name, e.manager_id
    FROM employees e
    JOIN RecursiveHierarchy rh ON e.employee_id = rh.manager_id
)

SELECT employee_name FROM RecursiveHierarchy;
```

Listing 18: Simple recursive CTE for employee hierarchy

In the example above:

1. The CTE starts with an anchor member, which is the base case and specifies a starting point. In this instance, we're beginning with the employee with 'employee_id = 5'.
2. The 'UNION ALL' operator is used to combine the results of the anchor member with the recursive member.
3. The recursive member references the CTE itself ('RecursiveHierarchy') to fetch the manager of the current employee and continues to do so until no more managers are found.

The result of this query is a list of employee names, starting from the specified employee and tracing back through their managerial hierarchy.

It's essential to be cautious when using recursive CTEs, as incorrectly constructed recursive CTEs might result in infinite loops. Always ensure there's a terminating condition, and the recursion will eventually reach it.

Note on Recursion Limits: SQL Server, by default, limits the maximum recursion level to 100 to prevent infinite loops. If you anticipate that your recursive CTE might exceed this limit, you can adjust it using the 'OPTION (MAXRECURSION n)' clause at the end of your query, where 'n' is the desired limit. Setting 'n' to '0' allows for unlimited recursion, but this should be used cautiously to avoid overwhelming the system.

17.1.2 Window Functions explained

Window functions provide a means to perform calculations across a set of rows related to the current one. Unlike standard SQL aggregations that return a single value for each grouped set, window functions operate within a „window“ of rows, allowing calculations like running totals, moving averages, or rankings. These functions are invaluable for analytical tasks, providing insights into data trends, distributions, and patterns. Using window functions, data analysts can answer questions that involve relative comparisons or require calculations to be reset based on certain conditions.

Preface: Sample Table and Data

Let's consider a table named 'sales' to showcase the use of window functions. The table is defined as:

```
    CREATE TABLE sales (
    salesperson VARCHAR(100),
    date DATE,
    amount INT
);
```

Listing 19: Defining the sales table

Assuming the table contains the following sample data:

```
    INSERT INTO sales (salesperson, date, amount) VALUES
('John', '2023-01-01', 100),
('John', '2023-01-02', 150),
('Jane', '2023-01-01', 50),
('Jane', '2023-01-02', 200);
```

Listing 20: Sample data for the sales table

Examples of Window Functions

1. Using ROW_NUMBER()

The 'ROW_NUMBER()' function assigns a unique sequential integer to rows within a partition of a result set.

```
    SELECT salesperson, date, amount,
       ROW_NUMBER() OVER (PARTITION BY salesperson ORDER BY date) AS
          row_num
```

```
FROM sales;
```

Listing 21: Using ROW_NUMBER()

2. Using SUM() with OVER()

You can compute a running total for each salesperson using the ‘SUM()‘ function combined with the ‘OVER()‘ clause.

```
    SELECT salesperson, date, amount,
        SUM(amount) OVER (PARTITION BY salesperson ORDER BY date) AS
            running_total
FROM sales;
```

Listing 22: Using SUM() with OVER()

3. Using LAG() and LEAD()

The ‘LAG()‘ function fetches the value from a previous row, while ‘LEAD()‘ fetches from a subsequent row.

```
    SELECT salesperson, date, amount,
        LAG(amount) OVER (PARTITION BY salesperson ORDER BY date) AS
            prev_amount,
        LEAD(amount) OVER (PARTITION BY salesperson ORDER BY date) AS
            next_amount
FROM sales;
```

Listing 23: Using LAG() and LEAD()

17.1.3 Pivoting explained

Pivoting is a technique that transforms data from a row-wise representation to a columnar format and vice-versa. This is often used in data analytics and reporting where data might be stored in a long format but needs to be presented in a wide format for clarity. For instance, sales data might be stored day-wise in rows, but a monthly report may require each day to be a column. Pivoting makes such transformations seamless, allowing for flexible data representation tailored to specific needs. The opposite of pivoting, often termed „unpivoting“, is equally crucial in data normalization tasks.

In relational databases, data is often stored in normalized tables. At times, we need to transform this tabular data to present it in a more comprehensible format, especially for reporting purposes. One common transformation is pivoting, where we turn unique data values from one column into multiple columns in the output, and aggregate results where necessary.

Let's begin with a simple example to understand the concept.

Preface: Consider a table `sales_data` that captures the sales made by different employees in different months:

```
CREATE TABLE sales_data (
    emp_id INT,
    month VARCHAR(50),
    sales INT
);

INSERT INTO sales_data VALUES
    (1, 'January', 100),
    (1, 'February', 120),
    (2, 'January', 110),
    (2, 'February', 115);
```

Listing 24: Defining sales_data table

The objective is to pivot the data such that we can see sales of each employee for January and February in separate columns.

```
SELECT emp_id,
       MAX(CASE WHEN month = 'January' THEN sales ELSE NULL END)
          AS January_sales,
       MAX(CASE WHEN month = 'February' THEN sales ELSE NULL END)
           AS February_sales
FROM sales_data
GROUP BY emp_id;
```

Listing 25: Pivoting sales data

In this query, we use a combination of conditional aggregation to achieve the pivot. The 'CASE' statement checks the month, and then we use the 'MAX' aggregate function to ensure we get one row per employee.

The result of the above SQL will give:

```
emp_id | January_sales | February_sales
-------+---------------+---------------
    1  | 100           | 120
    2  | 110           | 115
```

Pivoting functions might differ slightly based on the specific SQL database system in use. For instance, SQL Server has a 'PIVOT' operator designed specifically for this purpose, while Oracle has a 'PIVOT' clause. Always refer to the specific database documentation when working with pivots.

17.1.4 JSON Support explained

In the modern web-centric world, JSON (JavaScript Object Notation) has emerged as a de facto standard for data interchange. Recognizing this, many contemporary database systems have integrated support for JSON data types and associated functions. This amalgamation enables databases to store, query, and manipulate JSON data with ease, negating the need for external serialization and parsing. Direct JSON support bridges the gap between structured relational data and the more flexible, schema-less nature of JSON, offering the best of both realms.

Preface: Sample Table and Data Definition

For our JSON function examples, let's consider a table named 'user_profiles' that holds user data with a JSON column called 'details'. The column contains information about each user in JSON format.

```
CREATE TABLE user_profiles (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    details JSON
);
```

Listing 26: Table Definition

Here's some sample data to populate the 'user_profiles' table:

```
INSERT INTO user_profiles (id, name, details)
VALUES (1, 'Alice', '{"age": 25, "email": "alice@email.com", "address": {"city": "Wonderland", "state": "Dream"}}'),
    (2, 'Bob', '{"age": 30, "email": "bob@email.com", "address": {"city": "Uptown", "state": "Reality"}}');
```

Listing 27: Inserting Sample Data

17.1.5 Exploring SQL JSON Functions

JSON functions allow you to query and manipulate JSON data directly from SQL queries. Here are some basic examples to get started:

1. **Extracting a Value from JSON**: To fetch the age of each user from the 'details' column.

```
    SELECT name, details->>'$.age' AS age
```

```
2         FROM user_profiles;
```

Listing 28: Fetching age from JSON data

2. **Searching within a JSON Column**: To find users who live in 'Wonderland'.

```
1         SELECT name
2         FROM user_profiles
3         WHERE details->'$.address.city' = 'Wonderland';
```

Listing 29: Searching within JSON data

3. **Modifying a JSON Value**: To update the age of 'Alice' to 26.

```
1         UPDATE user_profiles
2         SET details = JSON_SET(details, '$.age', 26)
3         WHERE name = 'Alice';
```

Listing 30: Updating JSON data

These are just a few examples of the many JSON functions available in SQL. Depending on the SQL database system you are using, there may be more specialized functions or variations in the syntax.

17.1.6 Full-Text Search explained

Full-text search provides a sophisticated method for querying large text datasets. Unlike traditional query methods that search exact string matches, full-text search looks for approximate matches, making it ideal for searching textual content like articles, books, or documents. By indexing words in the text and understanding linguistic nuances, it supports operations like phrase search, proximity search, or weighted search. This capability is essential in modern applications where efficient and accurate text searches can enhance user experiences.

SQL offers powerful full-text search capabilities that allow users to perform complex text-based queries. With the help of these functions, you can search large datasets for terms or phrases, even if they are not an exact match.

Preface: Necessary Tables and Data

To illustrate the usage of full-text search functions, consider a simple ‘articles‘ table which stores articles with their respective titles and content:

```
    CREATE TABLE articles (
    id INT PRIMARY KEY,
    title VARCHAR(255),
    content TEXT
);
```

Listing 31: Creating articles table

Now, let’s insert some sample data:

```
    INSERT INTO articles (id, title, content)
VALUES
(1, 'Introduction to SQL', 'SQL stands for Structured Query Language.
    It is used for managing relational databases.'),
(2, 'Advanced SQL Techniques', 'As you delve deeper into SQL, you
   encounter more advanced topics like triggers, views, and full-text
    search.'),
(3, 'Database Normalization', 'Normalization is the process of
   organizing data in a database to reduce redundancy and improve
   data integrity.');
```

Listing 32: Inserting sample data into articles table

To enable full-text search on the ‘articles‘ table, we create a full-text index:

```
    CREATE FULLTEXT INDEX idx_articles_content
```

```
ON articles (title, content);
```

Listing 33: Creating a full-text index on the articles table

Using the MATCH...AGAINST Function

The primary mechanism for full-text search in SQL is the 'MATCH...AGAINST' function:

```
    SELECT title, content
FROM articles
WHERE MATCH(title, content) AGAINST('SQL');
```

Listing 34: Using MATCH...AGAINST for full-text search

This query will return articles that mention the word 'SQL' in either the title or the content.

Boolean Mode in Full-text Search

You can also use boolean operators like '+', '-', and '*' to refine your search:

```
    SELECT title, content
FROM articles
WHERE MATCH(title, content) AGAINST('+SQL -Introduction' IN BOOLEAN
   MODE);
```

Listing 35: Using boolean mode in full-text search

This query will return articles that contain the word 'SQL' but do not contain the word 'Introduction'.

There's a lot more to explore in the world of SQL full-text search, such as using query expansion or understanding the significance of the relevance score. This introduction should give students a foundation to start with.

17.1.7 Transactions explained

Transactions are foundational to database integrity, ensuring that a series of operations either complete in their entirety or not at all. By grouping operations, transactions ensure data consistency even in cases of system failures or errors. The primary properties of transactions, often termed as ACID properties (Atomicity, Consistency, Isolation, Durability), guarantee that data remains reliable and accurate throughout concurrent operations and system crashes. Transactions are pivotal in financial, booking, or any scenario demanding absolute data accuracy.

Before we delve into the transaction functions, it is essential to understand the context in which they'll be demonstrated. Let's consider the following table structure and data for a hypothetical online bookstore:

```
    CREATE TABLE books (
        book_id INT PRIMARY KEY,
        title VARCHAR(100),
        stock INT
    );

    INSERT INTO books (book_id, title, stock) VALUES
    (1, 'SQL Basics', 10),
    (2, 'Advanced SQL', 8),
    (3, 'Database Design', 12);
```

Listing 36: Table Definition for 'books'

Transactions are a sequence of one or multiple SQL statements that are executed as a single unit. They ensure data consistency and integrity. The primary transaction functions in SQL are 'BEGIN TRANSACTION', 'COMMIT', and 'ROLLBACK'.

BEGIN TRANSACTION

The 'BEGIN TRANSACTION' statement marks the starting point of an explicit transaction. Following this statement, all the SQL commands will be executed in a transactional context.

```
    BEGIN TRANSACTION;
    UPDATE books SET stock = stock - 1 WHERE book_id = 1;
```

Listing 37: Starting a transaction

COMMIT

The ‘COMMIT‘ command is used to save all the transactions to the database since the last ‘BEGIN TRANSACTION‘ was issued.

```
    COMMIT;
```

Listing 38: Committing a transaction

After this command, the stock of the book with ‘book_id‘ 1 will be reduced by one in the database.

ROLLBACK

In case of any error or if you decide not to proceed with the transaction, the ‘ROLLBACK‘ command can be used. It undoes all the changes made since the last ‘BEGIN TRANSACTION‘.

```
    ROLLBACK;
```

Listing 39: Rolling back a transaction

After executing the above code, any changes made after the ‘BEGIN TRANSACTION‘ will be discarded, and the database will return to its state before the transaction started.

Note: It’s essential to ensure that a ‘COMMIT‘ or ‘ROLLBACK‘ is issued at the end of a transaction to either save or discard the changes, respectively. Leaving transactions open can lock resources and prevent other operations on the database.

17.1.8 Partitioning explained

Partitioning divides a database table into smaller, more manageable chunks, while logically still treating them as a single entity. This division can be based on criteria like date ranges, value ranges, or other attributes. Partitioning enhances performance, especially in large datasets, by allowing more efficient data access and maintenance. For instance, querying a specific date range on a partitioned table will only access the relevant partition, leading to faster response times. Partitioning also aids in backup, recovery, and archival processes.

Partitioning in SQL: Logical and Physical

Partitioning in SQL can be approached from two main perspectives: Logical Partitioning, where result sets are segmented using functions for analytical purposes; and Physical Partitioning, where a table is divided into smaller chunks to enhance performance.

Logical Partitioning

This approach utilizes functions to segment and analyze result sets.

Preface: Necessary Tables and Sample Data

Let's consider a table named `sales`, which contains data on sales transactions:

```
    CREATE TABLE sales (
    id INT PRIMARY KEY,
    sale_date DATE,
    product_name VARCHAR(50),
    quantity INT,
    sale_price DECIMAL(10,2)
);
\end{submit mycode}

Assume this table contains the following data:

\begin{mycode}[caption=Sample sales data]
INSERT INTO sales VALUES
(1, '2023-01-01', 'Widget A', 10, 5.50),
(2, '2023-01-01', 'Widget B', 5, 7.25),
(3, '2023-01-02', 'Widget A', 8, 5.50),
(4, '2023-01-03', 'Widget C', 2, 12.75);
```

Listing 40: Defining the sales table

Examples of Partitioning Functions:

1. Using the `ROW_NUMBER()` function to number rows in each partition:

```
        SELECT
            product_name,
            sale_date,
            quantity,
            ROW_NUMBER() OVER (PARTITION BY product_name ORDER
               BY sale_date) as row_num
        FROM sales;
```

Listing 41: Using ROW_NUMBER()

2. Calculating the running total within each product partition using the `SUM()` function:

```
        SELECT
            product_name,
            sale_date,
            quantity,
            sale_price,
            SUM(quantity * sale_price) OVER (PARTITION BY
               product_name ORDER BY sale_date) as running_total
        FROM sales;
```

Listing 42: Calculating running total with SUM()

3. Finding the average sale price of each product using the `AVG()` function:

```
        SELECT
            product_name,
            AVG(sale_price) OVER (PARTITION BY product_name) as
               avg_price
        FROM sales
        GROUP BY product_name, sale_price;
```

Listing 43: Calculating average with AVG()

In SQL, partitioning functions allow us to segment a result set into partitions based on the values of one or more columns. These functions are immensely helpful when analyzing large datasets, as they can provide aggregated data for each partition without the need to produce multiple queries.

17.1.9 Physical Partitioning

Partitioning, in this sense, divides a database table into smaller, more manageable chunks, while logically still treating them as a single entity. This division can be based on criteria

like date ranges, value ranges, or other attributes. The primary reasons to employ physical partitioning include performance enhancement, efficient data access, and ease of maintenance.

Benefits:

- Enhances performance, especially with large datasets.
- Facilitates more efficient data access and maintenance.
- Aids in backup, recovery, and archival processes.

Example: Suppose you have a large sales table and want to partition it based on the sale date, storing each year's data in a separate partition.

```
    CREATE TABLE sales (
    id INT PRIMARY KEY,
    sale_date DATE,
    product_name VARCHAR(50),
    quantity INT,
    sale_price DECIMAL(10,2)
)
PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1992),
    PARTITION p2 VALUES LESS THAN (1993),
    ...
);
```

Listing 44: Physical Partitioning of a sales table by year

With this setup, if a query is specifically searching for sales in the year 1992, it will only access the partition 'p1', leading to more efficient and faster data retrieval.

Remember that the actual SQL syntax for physical partitioning might vary based on the SQL database system you are using (e.g., Oracle, MySQL, PostgreSQL).

17.1.10 Indexes and Referential Integrity explained

Indexes act as lookup tables used by databases to speed up data retrieval. An index provides a direct pathway to the data, eliminating the need to scan every row, much like an index in a book helps readers find information quickly. While indexes significantly enhance query performance, they come at the cost of additional storage and can affect the speed of data-insertion operations. Striking the right balance in index creation—focusing on frequently queried columns and understanding query patterns—is pivotal to optimized database performance.

Databases are essential repositories of information, and as such, it is crucial that they not only store data efficiently but also retrieve it rapidly. This is where indexing comes into play. Furthermore, maintaining the relationship and validity of data between tables is critical, which is achieved through referential integrity.

Indexes

An index is a data structure that improves the speed of data retrieval operations on a database. Without indexes, the database server must scan the entire table to retrieve the desired data, which becomes inefficient as data grows.

```
CREATE INDEX idx_name
ON users (name);
```

Listing 45: Creating an index

With the above index, queries that filter by the ‘name‘ column will be more efficient.

Referential Integrity

Referential integrity is a property that ensures that relationships between tables remain consistent. This is primarily achieved through the use of primary and foreign keys.

Preface: Consider two tables - ‘users‘ and ‘orders‘. The ‘users‘ table has a primary key ‘user_id‘, and the ‘orders‘ table has a foreign key ‘user_id‘ that references the ‘users‘ table.

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    product VARCHAR(100),
    user_id INT,
```

```
        FOREIGN KEY (user_id) REFERENCES users(user_id)
    );
```

Listing 46: Defining tables for referential integrity example

With the foreign key in place, you cannot insert an order with a 'user_id' that doesn't exist in the 'users' table. Also, if you try to delete a user that has associated orders, the database will raise an error.

```
-- Inserting a user
INSERT INTO users (user_id, name, age) VALUES (1, 'Alice', 28);

-- Trying to insert an order with an invalid user_id
-- This will raise an error
INSERT INTO orders (order_id, product, user_id) VALUES (1, '
    Laptop', 2);
```

Listing 47: Inserting data with referential integrity

Types of Indexes

There are multiple types of indexes, each serving its unique purpose and use case. Some of the most common types are:

- **Clustered Index**: Determines the physical order of data in a table. Because it defines the arrangement of data blocks on disk, there can be only one clustered index per table.
- **Non-Clustered Index**: Unlike clustered indexes, these do not dictate the physical order of data. Instead, they hold pointers to the data. A table can have multiple non-clustered indexes.
- **Unique Index**: Ensures that the indexed column contains no duplicate values.
- **Full-text Index**: Used for full-text search operations, it helps in efficiently querying large amounts of unstructured textual data.
- **Spatial Index**: Optimized for operations on spatial objects, such as geometry or geography data types.

Clustered vs. Non-Clustered Indexes

Clustered Index:

- Represents the physical order of data in the table.

- When you query data from a table with a clustered index, the database can quickly locate the required data because it knows its physical location.
- Think of a clustered index like the table of contents of a book, where the contents are organized in the same sequence as the chapters.

Non-Clustered Index:

- Does not represent the physical order of data, but holds pointers or references to the data.
- Useful for columns which are frequently involved in search conditions but not necessarily in the order of the data.
- Consider a non-clustered index as an index at the back of a book, where topics are alphabetically listed, but the page numbers might not be in sequence.

```
-- Creating a clustered index on the 'user_id' column
CREATE CLUSTERED INDEX idx_clustered_user_id
ON users (user_id);

-- Creating a non-clustered index on the 'name' column
CREATE NONCLUSTERED INDEX idx_nonclustered_name
ON users (name);
```

Listing 48: Demonstration of creating clustered and non-clustered indexes

Understanding the differences and mechanics behind clustered and non-clustered indexes can significantly improve database performance, especially as data volumes grow. Using them judiciously based on the query patterns ensures efficient data retrieval.

Composite (Combined) Indexes

Composite indexes are made up of two or more columns in a table. Their primary use is to enhance performance for queries that filter or sort by the combined columns.

Key Concepts

- **Column Order**: The order of columns in a composite index is significant. Generally, the column used in the WHERE clause's most restrictive condition should be placed first.
- **Prefixing**: Even if you're querying only by the first column in a composite index, the database can still use the index. This is referred to as using the „leftmost prefix" of an index.

- **Covering Index**: An index that includes all the columns retrieved by a query. For instance, if a query selects three columns, and a composite index exists that consists of those three columns (irrespective of the order), the index can satisfy the query without accessing the table's actual data.

Benefits

- **Performance**: Composite indexes can greatly improve the performance of certain queries. If a query filters or sorts based on the exact combination of columns in the index, the database can retrieve the data more efficiently.
- **Reduced Redundancy**: Instead of having separate indexes on each column, a composite index can sometimes serve the purpose of multiple single-column indexes, thus saving storage space.

Considerations

- It's essential to understand the query patterns to design effective composite indexes. Unnecessary or poorly designed composite indexes can be a burden rather than a help.
- Over-indexing can lead to increased storage usage and can also slow down the speed of write operations (INSERT, UPDATE, DELETE).

```
1    -- Creating a composite index on 'name' and 'age' columns
2    CREATE NONCLUSTERED INDEX idx_name_age
3    ON users (name, age);
```

Listing 49: Creating a composite index

By effectively using composite indexes, one can harness the potential of multiple columns and optimize the database for frequent and complex query patterns.

17.1.11 Constraints in SQL

SQL constraints are rules applied to data columns on a table. They ensure the accuracy and reliability of the data within the database. There are several types of constraints in SQL:

- **PRIMARY KEY**: Ensures that each row in a table has a unique identifier. A table can only have one primary key.
- **FOREIGN KEY**: Ensures rows in one table correspond to rows in another. It establishes a link between two tables based on a primary key and foreign key relationship.
- **UNIQUE**: Ensures that all values in a column are distinct.
- **CHECK**: Ensures that values in a column satisfy a specific condition.
- **DEFAULT**: Sets a default value for a column when no value is specified.
- **NOT NULL**: Ensures that a column cannot contain a NULL value.
- **INDEX**: Used to create and retrieve data from the database more quickly. It is not a constraint to ensure the integrity of data but rather to enhance the performance of the database.

For instance, if you have a ‘users‘ table and want to ensure that the ‘email‘ column contains unique values, you would use the UNIQUE constraint:

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) NOT NULL UNIQUE,
    name VARCHAR(255)
);
```

Constraints are fundamental in maintaining the integrity and quality of the data within a relational database system.

17.2 Programming and Development in SQL

- **Procedures and Functions:** Allow encapsulation of regularly used or complex SQL queries for reuse and modularity.
- **Imperative Constructs in SQL:** SQL encompasses control-of-flow constructs like conditional statements, loops, error handling, procedures, functions, triggers, dynamic SQL, transactions, cursors, and arrays/collections.
- **Triggers:** Automatically perform a function when certain conditions in the database are met, like after an ‘INSERT‘ or ‘UPDATE‘.
- **Views:** Create a virtual table based on the result set of a ‘SELECT‘ statement, which can be used as a table.
- **Editable Views:** Treat a view like a table and allow for read, insert and update operations.
- **Dynamic SQL:** Programs writing programs.

17.2.1 Procedures and Functions explained

Stored procedures and functions are SQL codes saved in the database, providing a way to encapsulate repetitive or complex logic. While both serve to modularize and reuse code, functions return a value and can be used in SQL statements, whereas procedures perform an action and don't necessarily return a value. They boost performance by reducing network traffic, allow for centralized logic which eases maintenance, and enhance security by abstracting underlying data structures and providing controlled data access pathways.

Procedures and functions serve as integral components that allow encapsulation of SQL statements. While both can accept parameters, they differ in their primary objectives and return mechanisms.

Preface: Sample Data Definition

To demonstrate, we'll use a 'products' table:

```
    CREATE TABLE products (
        id INT PRIMARY KEY,
        name VARCHAR(255),
        price DECIMAL(10, 2)
    );

    INSERT INTO products VALUES
    (1, 'Laptop', 800.00),
    (2, 'Phone', 300.00),
    (3, 'Headphones', 50.00);
```

Listing 50: Definition of the products table

Scalar Functions

Scalar functions return a single value. They can be utilized wherever an expression is used.

```
    CREATE FUNCTION PriceAfterTax(@product_id INT)
    RETURNS DECIMAL(10,2)
    AS
    BEGIN
        DECLARE @product_price DECIMAL(10, 2);
        SELECT @product_price = price FROM products WHERE id =
           @product_id;
        RETURN @product_price * 1.10; -- assuming 10% tax
    END;

```

```
    -- Usage:
    SELECT name, dbo.PriceAfterTax(id) AS price_with_tax
    FROM products;
```

Listing 51: SQL Server scalar function

Table-valued Functions

Apart from returning singular values, functions can return tables. Such functions are known as table-valued functions.

```
    CREATE FUNCTION ProductsAbovePrice(@threshold_price DECIMAL(10,2)
       )
    RETURNS TABLE
    AS
    RETURN
    (
        SELECT id, name, price
        FROM products
        WHERE price > @threshold_price
    );

    -- Usage:
    SELECT * FROM dbo.ProductsAbovePrice(100.00);
```

Listing 52: SQL Server table-valued function

Determinism of Functions

In SQL dialects, functions should often be deterministic. It means that for a given set of input values, the function always returns the same result. Non-deterministic behaviors can pose challenges, especially in replication or indexing scenarios. SQL Server e.g. insists on determinism for functions indexed or persisted.

Procedures with Output Parameters

Unlike functions, procedures do not return values directly. In SQL Server, if you want to get a value back from a procedure, you need to use output parameters.

```
    CREATE PROCEDURE GetProductPrice
    @product_id INT,
    @price DECIMAL(10,2) OUTPUT
    AS
```

```
    BEGIN
        SELECT @price = price FROM products WHERE id = @product_id;
    END;

    -- Usage:
    DECLARE @result_price DECIMAL(10,2);
    EXEC GetProductPrice 1, @result_price OUTPUT;
    SELECT @result_price AS 'Price';
```

Listing 53: SQL Server procedure with an output parameter

It's essential to be familiar with these distinctions and specifics when working with different dialects to ensure smooth development and effective data retrieval.

17.2.2 Imperative Constructs in SQL explained

SQL, while primarily a declarative language, incorporates imperative constructs allowing developers to dictate control flows. These constructs encompass conditional statements, loops, error handling, dynamic SQL, transactions, and more. This fusion empowers developers to craft logic-intensive routines, control transactional behaviors, or dynamically construct and execute SQL statements. The blend of declarative data querying and imperative control provides versatility in database programming, catering to a broader spectrum of application needs.

Preface: Tables and Data Definition

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    EmployeeName VARCHAR(255),
    Salary INT,
    Department VARCHAR(50)
);

INSERT INTO Employees VALUES (1, 'Alice', 50000, 'Sales');
INSERT INTO Employees VALUES (2, 'Bob', 60000, 'HR');
INSERT INTO Employees VALUES (3, 'Charlie', 55000, 'IT');
```

Listing 54: Defining sample table and data

1. Conditional Statements: Using the 'IF...ELSE' construct, SQL allows for conditional execution of code.

```
DECLARE @EmployeeSalary INT;
SET @EmployeeSalary = (SELECT Salary FROM Employees WHERE
   EmployeeName = 'Alice');

IF @EmployeeSalary > 55000
BEGIN
    PRINT 'Employee has a high salary.';
END
ELSE
BEGIN
    PRINT 'Employee has a moderate salary.';
END
```

Listing 55: IF...ELSE statement in SQL

2. Loops: SQL supports loops like 'WHILE'.

```
DECLARE @Count INT = 0;
```

```
    WHILE @Count < 5
    BEGIN
        PRINT 'Iteration number: ' + CAST(@Count AS VARCHAR);
        SET @Count = @Count + 1;
    END
```

Listing 56: Using WHILE loop in SQL

3. Cursors: Cursors are used to retrieve data row-by-row.

```
    DECLARE @EmployeeName VARCHAR(255);
    DECLARE cur_Employee CURSOR FOR
    SELECT EmployeeName FROM Employees WHERE Department = 'Sales';

    OPEN cur_Employee;

    FETCH NEXT FROM cur_Employee INTO @EmployeeName;

    WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT @EmployeeName;
        FETCH NEXT FROM cur_Employee INTO @EmployeeName;
    END

    CLOSE cur_Employee;
    DEALLOCATE cur_Employee;
```

Listing 57: Using SQL Cursors

4. Exception Handling: SQL provides the 'TRY...CATCH' construct to handle exceptions.

```
    BEGIN TRY
        -- Generate a divide-by-zero error
        SELECT 1/0;
    END TRY
    BEGIN CATCH
        PRINT 'An error occurred!';
    END CATCH
```

Listing 58: Exception handling in SQL

These imperative constructs enhance the capabilities of SQL and allow for more sophisticated operations, especially when creating stored procedures or functions.

17.2.3 Triggers explained

Triggers are automated responses initiated by specific database operations. They are powerful tools in maintaining data integrity, automating auditing, or enforcing business rules. Defined at a table level, triggers can be set to activate before or after ‘INSERT‘, ‘UPDATE‘, or ‘DELETE‘ operations. For example, an ‘AFTER INSERT‘ trigger on a sales table might automatically update an inventory table. Triggers thus act as sentinels, ensuring consistent and synchronized database states without manual intervention.

17.2.4 Views explained

Views are virtual tables derived from one or more existing tables. They provide a level of abstraction, allowing users to interact with data without needing to understand the underlying table structures or relationships. Views can encapsulate complex SQL queries, presenting data in a more accessible or meaningful format. Additionally, they serve security purposes, exposing only certain data columns while concealing others. As virtual constructs, views don't store data but refresh their contents dynamically based on their defining queries.

17.2.5 Editable Views explained

Traditionally, database views are considered read-only. However, editable (or updatable) views break this norm, allowing users not only to view but also to insert, update, or delete data. This is achieved by mapping modifications on the view back to the underlying tables. Editable views simplify complex data structures, provide data abstraction, and can enforce business rules or data integrity at the view level. They bring together the benefits of data abstraction with the flexibility of data manipulation.

In SQL, a view is essentially a virtual table based on the result set of an SQL statement. By default, views are read-only. However, there are cases where you might want to modify data through views. Such views are called „editable views“ or „updatable views“.

The ability for a view to be editable depends on several conditions. For a view to be editable:

- The view must be based on a single table. Joins generally make a view non-editable.
- The view must include the primary key of the table.
- The view cannot contain any aggregate functions, DISTINCT keyword, GROUP BY or HAVING clauses.
- Any computed or derived columns in the view cannot be edited.

Preface: Let's consider a table definition and data:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    city VARCHAR(100)
);

INSERT INTO users (user_id, name, age, city) VALUES
    (1, 'John Doe', 25, 'New York'),
    (2, 'Jane Smith', 30, 'Los Angeles'),
    (3, 'Alice Brown', 19, 'Chicago');
```

Listing 59: Table definition and data for users

Given the table above, we can create an editable view that filters out minors:

```
CREATE VIEW adult_users AS
    SELECT user_id, name, age, city
    FROM users
    WHERE age >= 18;
```

Listing 60: Editable view for adults

Through the `adult_users` view, you can make changes to the `name`, `age`, and `city` columns. Any change you make to the view reflects directly on the underlying `users` table. However, the `user_id` column remains non-editable as it's a primary key.

Always exercise caution when dealing with editable views. Modifying data through views can sometimes lead to unintended consequences on the underlying table.

Editable Views with Joins

While the standard and straightforward editable views are based on a single table, it's possible, albeit with added complexity, to make views derived from joined tables editable. The ability to edit such views primarily hinges on the database system's capability to trace changes made on the view back to a specific row in a particular table.

The primary challenges with making joined views editable are:

- Determining which table a change should be applied to, especially if columns from multiple tables are being edited.
- Handling situations where changes might violate constraints, such as foreign key relationships.

Many relational database systems address this by allowing only certain columns from one of the tables in the join to be editable, or by using „INSTEAD OF" triggers to define custom update/delete/insert behavior for the view.

Example:

Let's consider two tables: `authors` and `books`.

```
CREATE TABLE authors (
    author_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE books (
    book_id INT PRIMARY KEY,
    author_id INT REFERENCES authors(author_id),
    title VARCHAR(200)
);
```

Listing 61: Table definition for authors and books

We can create a view that joins these two tables:

```
CREATE VIEW authors_books AS
    SELECT a.author_id, a.name, b.book_id, b.title
    FROM authors a
```

```
4         JOIN books b ON a.author_id = b.author_id;
```

Listing 62: Joined view of authors and their books

To make this view editable, you might:

- Limit edits to only the `books` table columns in the view.
- Use „INSTEAD OF“ triggers (if supported by the database system) to define how inserts, updates, or deletes on the view should be propagated to the underlying tables.

Remember, the exact details and feasibility will depend on the database system you're using. Always consult the official documentation of your RDBMS when working with complex editable views.

Instead of triggers

```
1     CREATE TRIGGER authors_books_upsert_trigger
2     INSTEAD OF INSERT OR UPDATE ON authors_books
3     FOR EACH ROW AS
4     BEGIN
5         -- Check if the author exists
6         DECLARE @existingAuthorId INT;
7         SELECT @existingAuthorId = author_id FROM authors WHERE
              author_id = NEW.author_id;
8
9         -- Update or insert into the authors table
10        IF @existingAuthorId IS NOT NULL THEN
11            UPDATE authors SET name = NEW.name WHERE author_id = NEW.
                 author_id;
12        ELSE
13            INSERT INTO authors (author_id, name) VALUES (NEW.
                 author_id, NEW.name);
14        END IF;
15
16        -- Check if the book exists
17        DECLARE @existingBookId INT;
18        SELECT @existingBookId = book_id FROM books WHERE book_id =
             NEW.book_id;
19
20        -- Update or insert into the books table
21        IF @existingBookId IS NOT NULL THEN
22            UPDATE books SET title = NEW.title WHERE book_id = NEW.
                 book_id;
```

```
        ELSE
            INSERT INTO books (book_id, author_id, title) VALUES (NEW
                .book_id, NEW.author_id, NEW.title);
        END IF;
    END;
```

Listing 63: Upsert trigger for authors_books view

```
    CREATE TRIGGER authors_books_delete_trigger
    INSTEAD OF DELETE ON authors_books
    FOR EACH ROW AS
    BEGIN
        -- Delete the book
        DELETE FROM books WHERE book_id = OLD.book_id;

        -- Check if the author has any other books
        DECLARE @remainingBooksCount INT;
        SELECT @remainingBooksCount = COUNT(*) FROM books WHERE
            author_id = OLD.author_id;

        -- If the author has no more books, delete the author as well
        IF @remainingBooksCount = 0 THEN
            DELETE FROM authors WHERE author_id = OLD.author_id;
        END IF;
    END;
```

Listing 64: Delete trigger for authors_books view

```
    CREATE TRIGGER authors_books_upsert_trigger
    INSTEAD OF INSERT OR UPDATE ON authors_books
    FOR EACH ROW AS
    BEGIN
        -- Merge into the authors table
        MERGE INTO authors AS target
        USING (SELECT NEW.author_id, NEW.name) AS source (author_id,
            name)
        ON target.author_id = source.author_id
        WHEN MATCHED THEN
            UPDATE SET name = source.name
        WHEN NOT MATCHED THEN
            INSERT (author_id, name) VALUES (source.author_id, source
                .name);

```

```
14          -- Merge into the books table
15          MERGE INTO books AS target
16          USING (SELECT NEW.book_id, NEW.author_id, NEW.title) AS
                source (book_id, author_id, title)
17          ON target.book_id = source.book_id
18          WHEN MATCHED THEN
19              UPDATE SET title = source.title
20          WHEN NOT MATCHED THEN
21              INSERT (book_id, author_id, title) VALUES (source.book_id
                    , source.author_id, source.title);
22      END;
```

Listing 65: Upsert trigger using MERGE for authors_books view

17.2.6 Dynamic SQL explained

Dynamic SQL refers to SQL statements that are constructed and executed on-the-fly, typically in response to user input or application needs. This contrasts with static SQL, where statements are predefined and unchanging. Dynamic SQL offers flexibility, allowing applications to construct queries based on varying criteria, user inputs, or data structures. However, it introduces risks, especially around SQL injection attacks, necessitating stringent validation and parameterized queries to ensure both functionality and security.

Dynamic SQL refers to SQL commands that are generated and executed at runtime, typically as a result of user input or some other variable data source. Instead of being hardcoded in an application, dynamic SQL is constructed on-the-fly. This provides a great deal of flexibility, but can also introduce risks if not properly sanitized (e.g., SQL injection attacks).

Preface Before diving into the examples, let's set the stage by defining a simple table:

```
    CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    price DECIMAL(10, 2)
);
```

Listing 66: Definition of 'products' table

Let's assume we've inserted some sample data into the 'products' table:

```
    INSERT INTO products (product_id, product_name, price)
VALUES (1, 'Laptop', 999.99),
       (2, 'Mouse', 19.99),
       (3, 'Keyboard', 49.99);
```

Listing 67: Sample data for 'products' table

Example 1: Building a Query String Suppose a user wants to filter products based on their name. Instead of creating separate queries for each possible product name, you can use Dynamic SQL:

```
    DECLARE @ProductName NVARCHAR(100) = 'Laptop'; -- user input
    DECLARE @SQL NVARCHAR(1000);

    SET @SQL = 'SELECT * FROM products WHERE product_name = ''' +
       @ProductName + '''';

```

```
    -- To execute the generated SQL
    -- EXEC sp_executesql @SQL;
```

Listing 68: Dynamic SQL based on product name

Caution The above method of constructing SQL strings directly from user input can expose the application to SQL injection attacks. Always sanitize and validate user inputs and, where possible, use parameterized queries. See this chapter17.3.3.

```
    CREATE TABLE sales (
    sale_id INT PRIMARY KEY,
    product_id INT FOREIGN KEY REFERENCES products(product_id),
    sale_date DATE,
    quantity_sold INT
);
```

Listing 69: Definition of ‘sales‘ table

We’ve inserted some sample data into the ‘sales‘ table:

```
    INSERT INTO sales (sale_id, product_id, sale_date, quantity_sold)
VALUES (1, 1, '2023-01-01', 5),
       (2, 2, '2023-01-02', 15),
       (3, 3, '2023-01-03', 10),
       (4, 1, '2023-01-04', 3);
```

Listing 70: Sample data for ‘sales‘ table

Example 3: Dynamic SQL with Join Operation Suppose a user wants to retrieve the sales data for a specific product based on its name. We can achieve this using Dynamic SQL that involves a JOIN operation between the ‘products‘ and ‘sales‘ tables.

```
    DECLARE @ProductName NVARCHAR(100) = 'Laptop'; -- user input
DECLARE @SQL NVARCHAR(2000);
DECLARE @ParamDefinition NVARCHAR(500);

SET @SQL = 'SELECT p.product_name, s.sale_date, s.quantity_sold
            FROM products p
            JOIN sales s ON p.product_id = s.product_id
            WHERE p.product_name = @pName';

SET @ParamDefinition = '@pName NVARCHAR(100)';

-- To execute the generated SQL with parameter
```

```
-- EXEC sp_executesql @SQL, @ParamDefinition, @pName = @ProductName;
```

Listing 71: Dynamic SQL with JOIN

This Dynamic SQL command fetches sales data for the specified product name, merging data from both the ‘products‘ and ‘sales‘ tables to provide a comprehensive overview.

Example 4: Dynamic SQL with a Cursor If we wish to process sales data for each product in the ‘products‘ table individually, we can use a cursor to achieve this.

```
    DECLARE @ProductName NVARCHAR(100);
DECLARE @SQL NVARCHAR(2000);
DECLARE @ParamDefinition NVARCHAR(500);

-- Declare the cursor
DECLARE ProductCursor CURSOR FOR
SELECT product_name
FROM products;

-- Open the cursor
OPEN ProductCursor;

-- Fetch the first product name into the @ProductName variable
FETCH NEXT FROM ProductCursor INTO @ProductName;

-- Loop through each product
WHILE @@FETCH_STATUS = 0
BEGIN
    SET @SQL = 'SELECT p.product_name, s.sale_date, s.quantity_sold
                FROM products p
                JOIN sales s ON p.product_id = s.product_id
                WHERE p.product_name = @pName';

    SET @ParamDefinition = '@pName NVARCHAR(100)';

    -- Execute the generated SQL with the parameter for the current
       product
    -- EXEC sp_executesql @SQL, @ParamDefinition, @pName =
       @ProductName;

    -- Fetch the next product name
    FETCH NEXT FROM ProductCursor INTO @ProductName;
END

```

```
-- Close and deallocate the cursor
CLOSE ProductCursor;
DEALLOCATE ProductCursor;
```

Listing 72: Dynamic SQL with a Cursor

Using a cursor combined with Dynamic SQL enables us to process or retrieve data for each product one by one. However, be cautious when using cursors as they can be resource-intensive, especially for large result sets. Always evaluate if a set-based solution is possible before opting for a cursor.

17.3 Database Security and Management

- **Error Handling:** Expect the worst, but don't stop the program.
- **Bulk Operations:** Internal and external reading of data into databases.
- **Code Injection:** The ghost to be afraid of.
- **Logging:** Listen to the database, it's talking to you.

17.3.1 Error Handling explained

Error handling is a critical aspect of programming and database operations. As its name suggests, it pertains to how a system responds to unexpected or erroneous scenarios. Without proper error handling, a minor issue might escalate, potentially compromising system stability. In databases, appropriate error handling ensures data integrity, especially during transactions. It aids in debugging by providing meaningful error messages, allowing developers to trace and rectify issues promptly. Additionally, error handling can safeguard sensitive information, preventing the unintentional exposure of system details that can be exploited maliciously.

SQL transactions play a vital role in ensuring data consistency and integrity in database operations. By grouping a set of tasks into a single unit of work, transactions ensure that either all the tasks are executed or none of them are, avoiding partial updates that might lead to data inconsistency. However, during the course of a transaction, errors can occur. Handling these errors effectively is crucial.

To demonstrate error handling within SQL transactions, consider a simplified banking scenario where funds are transferred between accounts. The following tables will be used:

```
    CREATE TABLE Account (
    AccountID INT PRIMARY KEY,
    Balance DECIMAL(10, 2)
);

INSERT INTO Account VALUES (1, 5000.00);
INSERT INTO Account VALUES (2, 3000.00);
```

Listing 73: Tables Definition

The following SQL code attempts to transfer 1000.00 units from Account 1 to Account 2:

```
    BEGIN TRANSACTION;
    BEGIN TRY
        -- Deduct from Account 1
        UPDATE Account
        SET Balance = Balance - 1000
        WHERE AccountID = 1;

        -- Simulate an error
        DECLARE @SimulateError INT;
        SET @SimulateError = 1 / 0;

        -- Add to Account 2
        UPDATE Account
```

```
        SET Balance = Balance + 1000
        WHERE AccountID = 2;

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        PRINT ERROR_MESSAGE();
    END CATCH;
```

Listing 74: Transfer Funds with Error Handling

In the above code:

1. A transaction is started using 'BEGIN TRANSACTION'.
2. The 'TRY' block contains the code that attempts to deduct funds from one account and add to another.
3. For demonstration purposes, a deliberate error is simulated by trying to divide by zero.
4. If any statement within the 'TRY' block fails, execution is passed to the 'CATCH' block.
5. In the 'CATCH' block, the transaction is rolled back using 'ROLLBACK TRANSACTION', ensuring that no partial updates are made.
6. The 'ERROR_MESSAGE()' function is used to display a descriptive error message, assisting in debugging.

Using error handling in this manner ensures data consistency even when errors occur during transactional operations. Always ensure to test error scenarios extensively to guarantee system resilience.

17.3.2 Bulk Operations explained

Databases are often tasked with handling large volumes of data in single operations, known as bulk operations. These operations optimize the insertion, updating, or deletion of massive datasets, ensuring that they are processed efficiently and swiftly as opposed to row-by-row approaches. By reducing the overhead associated with individual transaction logs, locks, and index maintenance, bulk operations can significantly improve performance. They are especially valuable during data migration, initial population of databases, or periodic batch updates.

Defining Tables and Data

Before we delve into bulk operations examples, let's define a sample table and its associated data. This will facilitate a clearer understanding of the operations.

```
    CREATE TABLE bulk_sample (
    id INT PRIMARY KEY,
    name VARCHAR(255),
    age INT,
    country VARCHAR(100)
);
```

Listing 75: Table Definition

Bulk Insertion

The most common bulk operation is inserting large amounts of data into a database. There are several methods to perform bulk inserts:

- Using SQL's BULK INSERT

```
        BULK INSERT bulk_sample
        FROM 'full-filepath\datafile.csv'
        WITH (
            FIELDTERMINATOR = ',',
            ROWTERMINATOR = '\n',
            FIRSTROW = 2
        );
```

Listing 76: BULK INSERT from CSV in SQL

Here, the 'FIELDTERMINATOR' and 'ROWTERMINATOR' denote the column and row delimiters, respectively, and 'FIRSTROW' specifies the row at which the data starts (assuming the first row contains column headers).

- Using the ‘bcp‘ tool

```
        -- use command line
        bcp YourDatabaseName.dbo.bulk_sample in datafile.txt -c
           -T -S YourServerName
```

Listing 77: Using bcp for bulk insertion

In this command:

- ‘YourDatabaseName.dbo.bulk_sample‘: Specifies the target table.
- ‘in datafile.txt‘: Specifies the source data file.
- ‘-c‘: Indicates character type data.
- ‘-T‘: Uses a trusted connection.
- ‘-S‘: Specifies the server name.

Bulk Update

Bulk updating involves modifying a large set of records based on specific criteria.

```
    UPDATE bulk_sample
    SET country = 'USA'
    WHERE country = 'United States of America';
```

Listing 78: Bulk Update in SQL

Bulk Deletion

Bulk deletion targets a large set of records for removal.

```
    DELETE FROM bulk_sample
    WHERE age < 18;
```

Listing 79: Bulk Deletion in SQL

Conclusion

Bulk operations are an indispensable part of efficient database management. By leveraging built-in SQL commands, external tools like ‘bcp‘, and structuring SQL operations properly, one can drastically reduce the time taken to manipulate large datasets, making processes more efficient and reliable.

17.3.3 Code Injection explained

Code injection is a malicious technique where attackers introduce or „inject“ code into a vulnerable application, typically with the intent of stealing data, bypassing access controls, or otherwise compromising the system. SQL injection, a subtype, targets databases, enabling attackers to run arbitrary SQL code. Such vulnerabilities arise from improperly validated or unescaped user input. To combat this, developers must adopt best practices like using parameterized queries, input validation, and escaping user inputs, ensuring robust application security.

This section discusses methods to secure SQL procedures against such threats.

Dynamic SQL and Vulnerability: Constructing SQL commands by directly concatenating user inputs, like this initial example, exposes the database to SQL injection attacks.

```
DECLARE @SQL NVARCHAR(max) = 'BULK INSERT zs_legacy.importData FROM '
   '' + @Filename +
''' WITH (
FIELDTERMINATOR = '';'',
ROWTERMINATOR = ''\n'',
FIRSTROW = 2,
CODEPAGE = 65001
);'
```

Listing 80: Unsafe Code Sample

Use of `QUOTENAME`**:** The function `QUOTENAME` wraps an input string in delimiters and escapes any special characters within the input string, which can provide some level of protection. However, it is not a foolproof method against all forms of SQL injection.

```
DECLARE @SQL NVARCHAR(max) = 'BULK INSERT zs_legacy.importData FROM '
    + QUOTENAME(@Filename, '''') +
' WITH (
FIELDTERMINATOR = '';'',
ROWTERMINATOR = ''\n'',
FIRSTROW = 2,
CODEPAGE = 65001
);'
```

Listing 81: Better Code Sample

Parameterized Queries: The recommended approach to prevent SQL injection is the use of parameterized queries. With this method, data is separated from the query, thereby eliminating the possibility of injection. Additionally, parameterized queries can improve query performance by allowing database systems to reuse query plans for similar commands.

```
    DECLARE @SQL NVARCHAR(MAX) = 'BULK INSERT zs_legacy.importData
```

```
                                    FROM @Filename WITH (
                                        FIELDTERMINATOR = '';'',
                                                                ROWTERMINATOR
                                                                    = ''\n'',
                                                                FIRSTROW = 2,
                                                                CODEPAGE =
                                                                    65001);
                                ';
    DECLARE @Params NVARCHAR(MAX) = N'@Filename NVARCHAR(MAX)';
    DECLARE @Filename NVARCHAR(MAX) = 'MyFile.csv';

    EXEC sp_executesql @SQL, @Params, @Filename;
```

Listing 82: Injection Save Code

This method ensures that the input data is always treated as data and never as executable SQL code, regardless of its content.

Conclusion: While methods like `QUOTENAME` offer some protection, the most secure and performant approach to prevent SQL injection is the use of parameterized queries.

17.3.4 Logging explained

Logging in databases captures a record of all or selected database operations. This serves multiple purposes: auditing changes, aiding in troubleshooting, monitoring performance, or even facilitating recovery in case of failures. Logs can capture transaction details, data modifications, or system events. However, while logging is invaluable for monitoring and security, it introduces overhead. Thus, configuring logging levels appropriately—balancing between detail and performance—is crucial in database administration.

Logging essentials

Every SQL database system, such as SQL Server, Oracle, or PostgreSQL, maintains a set of logs. These logs can be transaction logs that capture every data modification or system logs that capture significant events or errors.

For the purpose of illustration, let's consider a simplified 'users' table:

```
CREATE TABLE users (
id INT PRIMARY KEY,
name VARCHAR(50),
age INT,
last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Listing 83: Definition of users table

Suppose we wish to log changes made to the 'users' table.

Trigger-based Logging

One method to achieve this is by using triggers. A trigger is a set of instructions that are automatically executed (or „triggered") by the database when specific events associated with a table occur.

```
CREATE TRIGGER log_user_updates
AFTER UPDATE ON users
FOR EACH ROW
BEGIN
    INSERT INTO user_changes_log (user_id, old_name, new_name,
       changed_on)
    VALUES (OLD.id, OLD.name, NEW.name, CURRENT_TIMESTAMP);
END;
```

Listing 84: Trigger for logging updates on users table

Here, any update on the ‘users‘ table will result in a new entry in the ‘user_changes_log‘ table.

Using External Tools

While triggers are effective, they can add overhead to the database. Sometimes, it’s beneficial to use external tools to capture and process logs. One such tool for SQL Server is ‘bcp‘ (bulk copy program), which allows for the exporting and importing of data between an instance of Microsoft SQL Server and a data file.

```
bcp "SELECT * FROM user_changes_log" queryout C:\logs\changes.log
    -c -U username -P password -S servername
```

Listing 85: Using bcp to export logs

This command will export all logs from the ‘user_changes_log‘ table to a file named ‘changes.log‘ on the ‘C:‘ drive.

Log Rotation and Retention

Over time, logs can consume significant disk space. It’s essential to regularly rotate and purge old logs. This can be achieved through database configurations or scheduled jobs, depending on the SQL system in use.

For instance, in PostgreSQL, the ‘log_rotation_age‘ parameter determines how often log rotation should occur. In SQL Server, one might set up a maintenance plan to manage log backup and cleanup.

Conclusion

Logging is a powerful tool for database administrators, offering insights, audit capabilities, and aiding in troubleshooting. However, it is essential to manage logs effectively to ensure they provide value without overwhelming resources.

17.4 Database Concepts

- **Concurrency:** The art of letting a lot of people do their work simultaneously.
- **MultiTenancy:** Share the same database among different tenants.
- **Explain:** See how databases solve the problem.

17.4.1 Concurrency explained

Concurrency is the ability of a database system to handle multiple operations simultaneously. As databases often serve multiple users, concurrency ensures that they can read, insert, update, or delete data without conflicts. Mechanisms like locks, latches, and timestamps help in managing concurrent access, ensuring data consistency and integrity. However, concurrency introduces challenges like deadlocks or contention, demanding careful design and management strategies. Proper concurrency control is vital for system reliability and user experience.

The Importance of Concurrency Control

Concurrency control is crucial in a multi-user database environment. Without proper concurrency controls, simultaneous transactions could interfere with each other, leading to data inconsistencies. For example, two users might try to book the last seat on an airplane simultaneously. Without concurrency control, both bookings could potentially be confirmed, causing overbooking.

Basic Concurrency Challenges

Two primary issues arise due to concurrent access:

Dirty Reads This happens when one transaction reads data modified by another ongoing transaction, which is then rolled back. This means the first transaction reads incorrect data.

Lost Updates When two transactions modify the same data simultaneously, the last commit might overwrite changes made by the first transaction.

Concurrency Control Mechanisms

To manage concurrency, databases employ several mechanisms:

- **Locking**: This prevents multiple transactions from accessing the same resource simultaneously. Depending on the scenario, databases might use shared locks, exclusive locks, or other types of locks.
- **Timestamping**: Transactions are given timestamps, and older transactions can't modify data already accessed by a newer transaction.
- **Optimistic Concurrency Control (OCC)**: Assumes that conflicts are rare. Transactions are checked only at commit time to see if conflicts occurred.

Examples of Concurrency Control

Let's consider a sample scenario and see how locking would work:

Table Definition:

```
    CREATE TABLE seats (
    seat_id INT PRIMARY KEY,
    status VARCHAR(10) NOT NULL
);
```

Listing 86: Table Definition for Seats

Sample Data:

```
    INSERT INTO seats(seat_id, status) VALUES (1, 'available');
    INSERT INTO seats(seat_id, status) VALUES (2, 'available');
```

Listing 87: Sample Data for Seats

Now, imagine two users, A and B, trying to book seat 1 simultaneously. Using locks, the database would ensure that once user A starts the booking process, user B would have to wait.

User A Transaction:

```
    START TRANSACTION;
    UPDATE seats SET status = 'booked' WHERE seat_id = 1;
    COMMIT;
```

Listing 88: User A Booking

If User B tries to book the same seat while User A's transaction is ongoing, the system using locks would either make User B wait or return a conflict error, depending on the lock type and timeout settings.

Dealing with Deadlocks

One major challenge with locking is the possibility of deadlocks. A deadlock occurs when two or more transactions are waiting for each other to release resources. To handle deadlocks, databases often employ deadlock detection mechanisms and might choose to terminate one of the transactions to break the deadlock.

In conclusion, concurrency in SQL ensures that databases operate reliably in multi-user scenarios. While it introduces challenges, using mechanisms like locks or timestamps can effectively manage concurrent access, ensuring data consistency and enhancing user experience.

17.4.2 MultiTenancy explained

MultiTenancy refers to an architecture where a single instance of a software application serves multiple tenants (customers or users), each with its isolated data and configurations. In database terms, multi-tenancy can mean either a shared database with shared tables or a shared database with isolated schema/table sets for each tenant. This design optimizes resource usage, centralizes management, and reduces costs. However, it poses challenges in ensuring data isolation, scalability, and performance, making it a critical design consideration in SaaS applications.

Delving Deeper into MultiTenancy in SQL

MultiTenancy in SQL databases can be achieved in various ways. The main techniques include:

- Shared schema: Tenants share the same tables, and data is differentiated using a tenant identifier.
- Separate schema: Each tenant has a unique set of tables within the same database.
- Separate database: Each tenant has its own dedicated database.

Among these techniques, the shared schema approach is often preferred for its simplicity and efficiency in resource usage, but it requires careful design to ensure data isolation.

Shared Schema Approach

In the shared schema approach, all tenants share the same set of tables. The data of each tenant is differentiated using a specific column, often called the tenant identifier or tenant ID. For every operation, the tenant ID is used to filter the data ensuring that one tenant cannot access the data of another.

Preface on Tables and Data: Consider a table named `orders` that stores order details. In a multi-tenant environment using the shared schema approach, this table might look as follows:

```
1    CREATE TABLE orders (
2        order_id INT PRIMARY KEY,
3        tenant_id INT NOT NULL,
4        product_name VARCHAR(255),
5        order_date DATE
6    );
```

Listing 89: Orders table for shared schema approach

The `tenant_id` column ensures that each order is associated with a specific tenant. When querying this table, you would always filter by `tenant_id`:

```
SELECT order_id, product_name, order_date
FROM orders
WHERE tenant_id = 1; -- Assuming 1 is the tenant ID in question
```

Listing 90: Fetching orders for a specific tenant

Separate Schema Approach

In the separate schema approach, each tenant has its own set of tables within the database. This provides better data isolation but may become complex to manage as the number of tenants increases.

Preface on Tables and Data: For instance, tenant A might have a table named `A_orders`, while tenant B has `B_orders`. Both tables have similar structures but are completely isolated from one another.

```
CREATE TABLE A_orders (
    order_id INT PRIMARY KEY,
    product_name VARCHAR(255),
    order_date DATE
);
```

Listing 91: Order table for tenant A

```
CREATE TABLE B_orders (
    order_id INT PRIMARY KEY,
    product_name VARCHAR(255),
    order_date DATE
);
```

Listing 92: Order table for tenant B

The challenge here is the need to dynamically generate and manage SQL statements for different tenants, which can get complicated.

Challenges and Considerations

Regardless of the approach you choose, there are several challenges to consider:

- **Data Isolation:** It is critical to ensure that tenants cannot access each other's data.
- **Performance:** As the number of tenants grows, it's essential to maintain database performance. Regular monitoring, tuning, and possibly sharding are needed.

- **Maintenance:** Backups, updates, and schema changes can be more complex in a multi-tenant environment.
- **Scalability:** As tenants grow in number and data volume, the system must scale without significant rearchitecture.

In conclusion, while multi-tenancy offers many benefits in terms of resource optimization and centralized management, it requires meticulous planning and design to ensure data isolation, performance, and scalability.

Setting and Retrieving the Active Tenant

In a multi-tenant environment, it's crucial to identify the active tenant for any given user or session. This allows the system to retrieve, present, and modify only the data related to that specific tenant. There are several techniques to manage the active tenant:

- **Session Management:** When a user logs into the system, the associated tenant ID can be stored in the user's session. This tenant ID is then used in all subsequent database queries during that session.
- **Token-Based Authentication:** For systems using token-based authentication (like JWT), the tenant ID can be embedded within the token. This way, every request accompanied by the token can be associated with the correct tenant.
- **Subdomain or URL Path:** Some systems use the subdomain or a specific path in the URL to determine the active tenant. For instance, `tenantA.example.com` and `tenantB.example.c` or `example.com/tenantA` and `example.com/tenantB`.
- **Custom Headers:** In API-driven systems, a custom header can be used to send the tenant ID with every request.

Example using Session Management:
Upon successful authentication, the tenant ID associated with the user can be set in the session.

```
    // Assuming a pseudo-code for session management
    session.set("tenant_id", user.tenant_id);
```

Listing 93: Setting tenant ID in a session

When querying the database or performing operations, this session value can be retrieved and used:

```
    int tenant_id = session.get("tenant_id");
    // SQL query can be something like:
```

```
3     // SELECT * FROM orders WHERE tenant_id = tenant_id;
```

Listing 94: Using tenant ID from session for database queries

It's essential to ensure that session data or tokens are secure and cannot be tampered with by users, as this could compromise data isolation between tenants.

Remember, the goal is always to guarantee that each user or session interacts only with the data related to its associated tenant. This promotes both data security and a tailored user experience.

17.4.3 Explain explained

„Explain“ is a tool offered by many database management systems to provide insights into how the system plans to execute a SQL query. By analyzing the „explain“ output, developers can understand the steps the database takes, the indexes it uses, and the joins it performs. This transparency allows for optimization, helping developers refactor queries, add or modify indexes, and ensure efficient data retrieval. Interpreting „explain“ plans is a fundamental skill for anyone aiming to optimize database performance and resources.

The Essence of „Explain“

The „Explain“ command provides a window into the internal workings of the database query planner. It offers an abstracted plan of how the query will be executed, even before any data is actually retrieved or modified. By employing this command, we can observe the anticipated path the database plans to take in order to retrieve the necessary data, based on the current statistics and indexes available.

Preface: Tables and Data

To effectively illustrate the use of the „Explain“ command, it’s essential to set a context. Let’s consider a table named ‘orders‘. This table will store order data with attributes such as ‘order_id‘, ‘product_id‘, ‘customer_id‘, and ‘order_date‘.

Below is the SQL command to create such a table, which can be executed in SQL Server Management Studio (SSMS):

```
    CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    product_id INT,
    customer_id INT,
    order_date DATE
);
```

Listing 95: SQL for creating the orders table

Utilizing „Explain“

Once our table is set up and populated with data, we can start crafting queries and then use the „Explain“ command to understand their execution plans.

For instance, consider a query to retrieve all orders made after a certain date:

```
    SELECT * FROM orders WHERE order_date > '2023-01-01';
```

Listing 96: Sample SQL query

To understand how the database plans to retrieve this data, we can prefix our query with the „EXPLAIN“ keyword:

```
EXPLAIN SELECT * FROM orders WHERE order_date > '2023-01-01';
```

Listing 97: Using EXPLAIN with the query

The result of this command will be a detailed plan that highlights the anticipated steps, the potential use of indexes, the type of scan employed, and more. By examining this output, developers can gain insights into potential bottlenecks or inefficiencies, thereby allowing for necessary optimizations.

Conclusion

Mastering the art of interpreting „Explain“ output is crucial for anyone serious about database optimization. With its insights, one can significantly improve the efficiency and speed of queries, ensuring smoother and faster database operations.

Key Aspects of SSMS Execution Plans

SQL Server Management Studio (SSMS) provides execution plans as a valuable tool for database administrators and developers. These plans offer insights into the steps taken by the SQL Server engine to execute a particular SQL query. Here are some essential components and aspects of execution plans in SSMS:

1. **Graphical Plans:** These are typically the most interacted-with type of execution plans. They present a visual flowchart of the various operations carried out, from scans and seeks to joins and sorts.
2. **Operators:** Every action or step in the plan is denoted by an operator. Common operators you might encounter include *Clustered Index Scan*, *Clustered Index Seek*, *Hash Match*, and *Nested Loops*.
3. **Plan Cost:** Each operator carries a cost, and the entire query possesses a cumulative cost. This „cost“ serves as a relative metric, aiding in the comparison of different query versions or distinct plans.
4. **Arrows:** Arrows, connecting the operators, signify the data flow. The arrow's thickness is an indicator of the data volume being passed between operations.
5. **Properties:** By right-clicking on an operator, you can access detailed properties. This includes information such as the number of processed rows, the chosen execution mode, among others.
6. **Key Lookouts:**

- *Table Scans:* Comprehensive scans of tables can be highly inefficient, especially for substantial tables. This often suggests a potential missing index.
- *Missing Index:* Occasionally, the plan might recommend the creation of an index to enhance the query's performance.
- *Parallelism Operators:* These denote that a query is distributed and processed in parallel across multiple CPU cores or processors.

7. **Actual vs. Estimated Plans:** SSMS offers the ability to view both the estimated execution plan (the database's anticipated actions) and the actual execution plan (what it genuinely undertook). Notable differences between these two can hint at outdated statistics or other prevalent issues.

Recursive Common Table Expressions in SQL Server

SQL Server, among other database management systems, supports the use of Common Table Expressions (CTEs) - these are temporary result sets that can be referenced within a 'SELECT', 'INSERT', 'UPDATE', or 'DELETE' statement. Recursive CTEs, in particular, are immensely powerful, as they can be used to query hierarchical data. Essentially, a recursive CTE is one that references itself!

The query you provided is an excellent example of a recursive CTE in SQL Server:

```
    WITH RecursiveHierarchy AS (
    -- Anchor member (base case)
    SELECT employee_id, employee_name, manager_id
    FROM employees
    WHERE employee_id = 5  -- starting with a specific employee

    UNION ALL

    -- Recursive member
    SELECT e.employee_id, e.employee_name, e.manager_id
    FROM employees e
    JOIN RecursiveHierarchy rh ON e.employee_id = rh.manager_id
)

SELECT employee_name FROM RecursiveHierarchy;
```

Listing 98: Recursive CTE in SQL Server

In the above example, the CTE starts with a base case, the anchor member, where it selects an employee with a specific ID. The recursive member then extends this hierarchy by joining the employees table with the CTE itself to find all direct and indirect subordinates of the specified employee. This continues recursively until no more matches are found.

To truly understand the execution and efficiency of this query, especially for large datasets, you would want to examine the execution plan. SQL Server Management Studio (SSMS) provides visual execution plans that can be invaluable in understanding and optimizing queries.

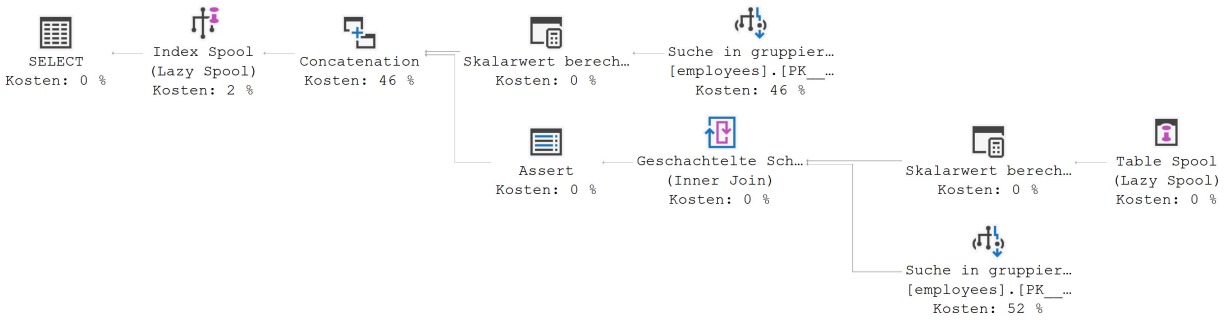

Abbildung 2: Execution plan for the Recursive CTE

18 Sources

- **MySQL**: https://dev.mysql.com/doc/
- **PostgreSQL**: https://www.postgresql.org/docs/
- **Oracle Database**: https://docs.oracle.com/en/database/
- **Microsoft SQL Server**: https://learn.microsoft.com/en-us/sql/?view=sql-server-ver16
- **SQLite**: https://sqlite.org/docs.html
- **IBM Db2**: https://www.ibm.com/docs/en/db2
- **MariaDB**: https://mariadb.com/kb/en/
- **SAP HANA**: https://help.sap.com/viewer/p/SAP_HANA_PLATFORM
- **Sybase (SAP ASE)**: https://help.sap.com/viewer/product/SAP_ASE/
- **SQL Anywhere (SAP SQL Anywhere)**: https://help.sap.com/docs/SAP_SQL_Anywhere?locale=en-US

19 Recommended Books

Literatur

[1] Scott J. Ambler und Pramodkumar J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[2] Scott W. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley, 2002.

[3] Scott W. Ambler und Pramod J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[4] Christian Antognini. *Troubleshooting Oracle Performance*. Apress, 2014.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[6] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[7] Philip A. Bernstein und Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 2009.

[8] Joe Celko. *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, 2005.

[9] Joe Celko. *SQL Programming Style*. Morgan Kaufmann, 2019.

[10] Joe Celko. *SQL Puzzles and Answers*. Morgan Kaufmann, 2009.

[11] Thomas M. Connolly und Carolyn E. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Pearson, 2017.

[12] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley Professional, 2003.

[13] David J. DeWitt und Samuel Madden. *Database Management Systems*. McGraw-Hill Education, 2008.

[14] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[15] Stephane Faroult und Peter Robson. *The Art of SQL*. O'Reilly Media, 2015.

[16] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[17] Kimberly Floss. *Oracle Performance Tuning 101*. McGraw-Hill Education, 2001.

[18] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[19] Martin Fowler, Pramod J. Sadalage und Mike Roberts. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.

[20] Martin Fowler u. a. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2019.

[21] Grant Fritchey. *SQL Server Execution Plans*. O'Reilly Media, 2017.

[22] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[23] Rowan Garnier und Jonathan Lewis. *Cost-Based Oracle Fundamentals*. Apress, 2005.

[24] J.B. Hecht. *Understanding DB2: Learning Visually with Examples*. IBM Press, 2013.

[25] Andrew Hunt und David Thomas. *The Pragmatic Programmer: Your Journey to Mastery*. Addison-Wesley Professional, 2018.

[26] Ron Jeffries, Ann Anderson und Chet Hendrickson. *Extreme Programming Installed*. Boston, MA, USA: Addison-Wesley Professional, 2000.

[27] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf, 2020.

[28] Ralph Kimball und Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Wiley, 2013.

[29] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2018.

[30] Kevin Kline. *Transact-SQL Programming: Covers Microsoft SQL Server 2008*. O'Reilly Media, 2008.

[31] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[32] Leslie Lamport. „The Temporal Logic of Actions". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), S. 872–923.

[33] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 2004.

[34] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[35] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[36] Jim Melton und Alan R. Simon. *SQL: Design Patterns: Expert Guide to SQL Programming*. Morgan Kaufmann, 2017.

[37] Jim Melton und Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 2008.

[38] Gavin JT Powell, Dr. Bert Scalzo und Kevin Meade. *Oracle 11g & 12c RAC Performance and Tuning*. McGraw-Hill Education, 2014.

[39] Pramod J. Sadalage und Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.

[40] Robert Schneider. *Database Testing for Dummies*. For Dummies, 2015.

[41] Len Silverston und Paul Agnew. *Data Model Patterns: A Metadata Map*. Technics Publications, 2016.

[42] Michael Stonebraker und Joe Hellerstein. *Readings in Database Systems*. MIT Press, 2015.

[43] Shaun M. Thomas. *Mastering PostgreSQL 11: Expert techniques to build scalable, reliable, and fault-tolerant database applications*. Packt Publishing, 2018.

[44] Dan Tow. *SQL Tuning*. O'Reilly Media, 2003.

[45] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.

[46] John-Harry Wieken. *Ernsthaft SQL verstehen: Den Standard verstehen und mit verschiedenen Datenbanken verwenden*. German Edition. Kindle-Version. ServiceValue Fachbücher Verlag, 2018, S. 223, 224.

[47] Markus Winand. *SQL Performance Explained*. Markus Winand, 2012.

[48] Markus Winand. *SQL Performance Explained*. Markus Winand, 2013.

[49] Markus Winand. *Use The Index, Luke!: A Guide to SQL Database Performance*. Markus Winand, 2018.