Deep Science Publishing, 2025
https://doi.org/10.70593/978-93-7185-191-6_2

# Chapter 2: SQL query design and optimization: A study of joins, window functions, and recursive constructs

Mohanraju Muppala

## 1. Introduction to Advanced SQL

SQL is a widely accepted language for interacting with databases. When databases become complex and the workload on the database becomes more, simple commands will not help [1-2]. Hence, there are numerous advance concepts in order to perform the SQL operation on complex projects. The concepts of SQL include Complex Join, Subqueries, Window functions, Common table expressions, Recursive queries, Error handling in SQL, Transactions in SQL, Isolation levels, Dynamic SQL, Stored procedures, etc. For topics such as Complex Join, Subqueries, Window Functions, CTE, and Recursive Queries, performance and optimization impact of these concepts are discussed together [2-4]. Topics related to Error handling in SQL, Transactions in SQL, Isolation levels, Dynamic SQL, and Stored procedures are also analyzed.

## 2. Complex Joins in SQL

Joins are one of the most important operations in a relational database management system and one of the major reasons to use a relational database. The join operation merges two tuple sets into one single table based on some common column (attribute) present in the two tuple sets. SQL joins are classified on the basis of how they operate and what tuples they include in the result [5-6]. They are: inner join, outer join, self join, natural join, and cross join.

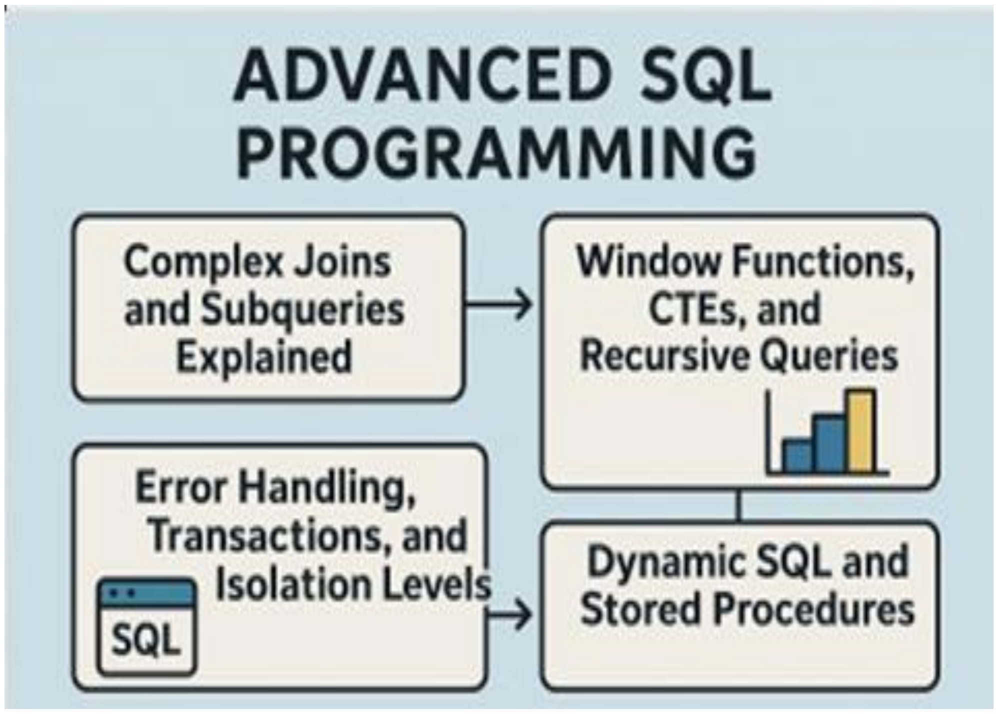
ADVANCED SQL PROGRAMMING
Complex Joins and Subqueries Explained
Window Functions, CTEs, and Recursive Queries
Error Handling, Transactions, and Isolation Levels
SQL
Dynamic SQL and Stored Procedures

Fig1. Advanced SQL programming

In an inner join, only a common subset from two or more relations is displayed. Outer joins include tuples not appearing in the join condition bases. A self-join is used to select tuples from the same relation for comparison. A Natural Join bases its join on attributes of the same name that share any overlapping domain(s). Cross Joins display the Cartesian product of the tuples in two or more relations.

## 2.1. Types of Joins

A join is a logical operator that allows data of multiple tables to be combined into a result table based on query join conditions. To do this, a join merges columns of at least two tables into a single row, which for a query involving three or more tables may therefore contain values from all those tables. The ordering of the columns in the result table depends on the join implementation.

Different types of semantic joins exist:

- Cartesian product joins multiply each row of the first table with each row of the second. The smallest tables on the inner part of the join minimise complexity and avoid a huge intermediate table. - Theta-joins return the field combination of the first table with the second table when the "joinCondition" holds true. - Equi-joins are Theta joins using equal conditions. - Non-equ-joins use operators other than equal operators. - Self-joins can be equi-joins or non-equ-joins but operate on only one table and use table aliasing to filter out columns. - Outer joins return the field combination of the first table with the second table when the "joinCondition" holds true and fill with missing values for the corresponding side if the condition is false. - Natural joins eliminate duplicate columns on already equi-joined columns. An equi-join generates a combined column automatically.

## 2.2. Performance Considerations

Optimizing SQL code demands intimate knowledge of its internal workings within the database engine [7,8]. Deep diving into the optimization techniques employed by the underlying RDBMS enables one to write improved SQL statements. The optimization process transforms a SQL statement into an efficient execution plan. To enhance implementation and maintenance, SQL optimizers can be integrated within a database application development environment. They then perform checks, evaluations, and rewrites of developing code.

Writing optimal SQL code implies knowing which internal database mechanisms affect performance, and thus determining which SQL constructs might degrade it. Understanding the internal workings of SQL statements is key to achieving this. Optimization translates SQL statements into execution plans. Embedding SQL optimizers into the development environment allows SQL code to be checked and evaluated as it is being developed, thereby enhancing overall application performance.

# 3. Subqueries Explained

A subquery is a query nested inside another query that evaluates to a result set (a table, a row or a scalar). Since a subquery is a query, it can be arbitrarily complex. A subquery can be used in the select-list or from-clause of an enclosing query, but the most common subquery usage is in the where-clause (using operators such as IN, EXISTS, ANY and ALL).

An operation that uses a list of constants can be reformulated using a subquery that references a table, a view or another query, which is often clearer and easier to maintain. An operation that compares a single value to a set of values can be reformulated using a subquery. Similarly operations that compare strings to other strings or strings to patterns can also be reformulated using a subquery.

## 3.1. Types of Subqueries

Subqueries, which are also called inner or nested queries, can be classified logically by the way they interact with the outer main query [9-12]. In a correlated subquery, the inner SELECT depends on the outer SELECT for its values. In other words, it references a column from a table or view in the outer statement. The result row from the outer statement is fed into the inner query as a column value rather than as a value up front. The inner query is then reexecuted, and the results are returned as a part of the inner statement. Then, these rows are reintegrated into the outer statement to finish the execution. Correlated outer subqueries nest a subquery within the FROM clause of a main query and differ from other types of correlated subqueries because the inner statement is evaluated first before being joined to the outer query.

A noncorrelated subquery executes independently and does not refer to a table or view of the outer statement. All required arguments are sent up front to the inner query and the results returned immediately. Once the subquery runs, the inner statement returns the rows to the outer query, which uses the values to complete its execution. The subquery can be nested in a SELECT, UPDATE, DELETE, or INSERT statement or another subquery clause. This type of subquery is categorized as an inner statement, and this key concept will be highlighted in subsequent discussions. The main query that nests the inner query is often deemed the "outer query."

## 3.2. Best Practices for Subqueries

Effective use of subqueries can result in compact and elegant SQL code. Subqueries are executed independently and are often more readable than equivalent table joins. Unlike joins, subqueries evaluate in isolation; this is why the correlated option requires frequent evaluation of the inner table for individual rows in the outer query. Null values and three-valued logic can complicate predicates, especially when the subquery contains nulls. In such cases, the predicate should employ NOT IN (subquery) rather than <> ALL (subquery).

Contemporary optimizers tend to produce similar execution plans for both correlated and uncorrelated subqueries. Nevertheless, some database engines handle correlated subqueries inefficiently.

Standard SQL forbids existing columns in the outer scope from being referenced in the inner subquery's WHERE clause. This restriction is lifted in correlated subqueries, which are evaluated once for every row procured from the outer table. Although such a predicate uses a subquery, it cannot be treated like the previous examples due to the test column's correlation to the inner table.

# 4. Window Functions

Window functions perform calculations across sets of rows related to the current query row. They vary in scope by defining a window of rows over which a function operates. Often, such functions help in ranking query results. Another useful category allows the analysis of data from a previous row to decide whether the current row matches the query filters. Window functions rely on an OVER() clause that defines their behavior. This clause consists of optional instructions that specify the sorting of rows and, in some cases, the boundaries of the window for the ranked function. For mutually ranked rows, the window boundary definition can be omitted.

Window functions are executed after the previous clauses are fully processed. Hence, window functions do not influence any other clause." Ranking Rows". When ranking a window of rows, it is necessary to ensure the returned query rows are sorted according to the ranking columns. Three frequently used window-ranking functions are RANK(), DENSE_RANK(), and ROW_NUMBER(). The RANK() function ranks the rows related to the row in question in ascending or descending order. Because of its ranking behavior, ROW_NUMBER() cannot be omitted from the ORDER BY clause, which arranges the query results according to the positions returned by ROW_NUMBER(). Unlike RANK(), which assigns the same rank number to multiple rows with the same value, ROW_NUMBER() assigns a unique row number to each entry. This means RANK()’s return values might not always be sequential, whereas ROW_NUMBER() returns consecutive numbers lacking such gaps.

## 4.1. Introduction to Window Functions

Since their introduction in SQL Server 2005, window functions need little excuse for their study. They form a distinctive category of functions that allows calculations across groups of rows and return a value for every row in the set. They solve many problems that otherwise required complex queries with self joins and nested SELECTs [7,13-15]. Three enthusiasts of window functions discuss their implementation and use in SQL Server 2012.

Window functions solve many problems that otherwise required complex queries with self-joins and nested SELECT queries. They have been known under the same name since the introduction of the SQL:2003 standard and are often called OLAP or Analytic Functions (both reflecting a business analysis background). Some analytic databases, like Oracle or DB2, supported such functions even before their standardisation.

## 4.2. Common Use Cases

Some of the many operations performed on a long string defined as a CLOB are listed here:

A specific portion of data is extracted from the CLOB, using the DBMS_LOB.SUBSTR function. This function takes as argument the starting position and length of the string portion to be extracted.

Some or all data from one LOB is copied to another LOB, using the DBMS_LOB.COPY procedure.

# 5. Common Table Expressions (CTEs)

A Common Table Expression (CTE) is a temporary named sub-result set introduced within the execution scope of a single SQL data manipulation command. In accessing systems modeled after SQL, such as Query by Example, a table expression is a valid subpart of a query that can stand in for a base relation or an output relation. It is bounded by parentheses. Most data-driven websites use CTEs to facilitate search functionality or place-search functionality. An example is the "/jobs/" page of LinkedIn, which supports the current page's search text and the nearby geographic location specified. Areas near the world location specified by the user can be toggled on and off.

The following example demonstrates a CTE implementation. The Common Table Expression "CurrentLocation" returns the base job search URL:

WITH CurrentLocation("href") AS ( SELECT '/jobs/'::TEXT )

The Common Table Expression "BaseURL" returns the BaseURL with an additional active parameter specifying the current job selection:

BaseURL("href") AS ( SELECT
'/jobs/?locationGeoId=103644278&geoRadius=0'::TEXT )

The CTE "NearByLocations" returns an array of areas near the current search area:

NearByLocations("href") AS ( SELECT ARRAY[
'/jobs/?geoId=2004&geoRadius=25', '/jobs/?geoId=2056042&geoRadius=25',
'/jobs/?geoId=103644278&geoRadius=25',
'/jobs/?geoId=102571732&geoRadius=25' ]::TEXT[] )

Finally, a SELECT statement fetches the data:

SELECT * FROM CurrentLocation, BaseURL, NearByLocations >;

## 5.1. Defining CTEs

A common Table Expression may be used in a bunch of interesting ways. It may be viewed as giving a temporary name to a subquery, allowing the subquery to be expressed as a separate item rather than being lumped into the main query [9,16-18]. This ability to name a subquery often makes the resulting code easier to read and understand—if only a code comment were needed when viewing the previous example! However, the power of CTEs doesn’t end there. The next example uses the CTE to express the relationship between employees and contracts that they have worked on. The main query then iterates over the CTE results, skipping a couple of rows each time via the ROW_NUMBER(). The employment periods for each employee contract instance are then sorted by start date, and a gap in each employee’s employment periods within the company is identified.

## 5.2. Using CTEs for Better Readability

You want to make your queries more readable. Using Common Table Expressions—CTE’s—can help. For example, say you want to find the parent of each employee in Employees. It’s not so easy using a simple SELECT statement. An inner join works, but it’s not so pretty:

SELECT e1.NAME "Employee", e2.NAME "Parent" FROM EMPLOYEES e1 JOIN EMPLOYEES e2 ON e1.PARENTID = e2.ID;

Using a CTE creates a more readable query:

WITH EmployeeParents AS (SELECT e1.ID, e1.NAME "Employee" , e2.NAME "Parent" FROM EMPLOYEES e1 JOIN EMPLOYEES e2 ON e1.PARENTID = e2.ID) SELECT * FROM EmployeeParents;

# 6. Recursive Queries

Recursive queries lie at the frontier of boolean logic and the comprehension of recursion. Although their practical utility remains doubtful—the implementation of recursive functions is often more straightforward—recursive queries exemplify the power of SQL. They can, for instance, assist in improving the display of objects' creation dates by including contemporary historical events, or be applied in the processing of bill-of-materials, by representing assemblies and parts, and all their subassemblies and component parts, as parts can themselves be assemblies.

The SQL2 standard proposed the WITH extension to conjunctions and disjunctions, which was subsequently adopted in the OLAP extension and certain DBMSs such as DB2 or Oracle. This extension enables the definition of local views within a query, and these views may be recursively defined. The syntax allows a list of local views to be declared before a query, with the designation of which views are recursive. A recursive view is a union of two queries (the first forming the base case and the second the recursive case), with the additional constraint that the recursive term can only refer to previously defined local views—that is, the recursive term cannot include the recursive view itself but only the other local views.

## 6.1. Understanding Recursive CTEs

Common Table Expressions (CTE) are often used to organize complex SELECT queries, simplify maintenance and enhance readability with code folding and syntax highlighting enabled by most available editors, and also enable query functionality that cannot be achieved using derived tables. Another major advantage of CTEs is that CTEs recursive in nature can be developed. Recursive CTEs help locate data patterns that are hierarchical in nature, such as a company hierarchy or a folder structure, or relationships based on recursion, such as friendships in social media.

A Recursive CTE contains an anchor member and a recursive member. The anchor member is a non-recursive CTE, and the recursive member consists of the query that references the recursive CTE. An additional recursive query between the anchor and recursive member provides the correct output. The Recursive CTE's final output consists of the recursive query's UNION operation between the anchor and recursive members. The Recursive query's recursion depth is indirectly raised by adding a UNION ALL clause in SQL.

## 6.2. Applications of Recursive Queries

Recursive queries have many practical applications. The directory hierarchy model created in the previous section applied recursive queries to a simple tree structure [2,19-20]. The next examples examine the model’s agility for other applications. In the pathfinder example, the recursive query explores nodes along many paths and a more complex query predicate governs the search order.

Another context of the hierarchy example describes organizations by their departments, employees, and the tree of employee managers. This example maintains three models and discovers that the tree overlay is just as good as the hierarchy of a department.

# 7. Error Handling in SQL

The intricate process of creating and executing stored procedures not only involves their logical design but also demands meticulous error handling strategies. Erroneous input can lead to service disruptions, making it imperative to capture and manage exceptions effectively. It is essential for variable declarations to precede procedural statements to maintain syntactic and semantic consistency.

SQL, recognized as a sequence-based and condition-based language, operates through a sequential execution of commands split via semicolons, with a "WHERE" clause as its primary conditional filter. Procedural languages like PL/pgSQL introduce explicit control flow through structures such as "IF" and "WHILE." The procedural nature of stored procedures allows them to function akin to independent programs, obviating the need for explicit connections during runtime. This autonomy, however, necessitates safeguards like conditional triggers or exception handlers to avert unregulated execution that might disrupt overarching campaigns.

## 7.1. Error Types

Sophisticated SQL programming cannot avoid dealing with errors. Despite the best intentions, mistakes creep in, whether typographical, logical or semantic.

Errors in a program are normally classified as syntax, semantic or logical. Syntax errors violate the language rules and include such errors as misspellings. Logical and semantic errors are much more subtle. Logical errors are errors in design and may give incorrect results or cause other errors. Semantic errors are incorrect operations at different stages in program execution.

3GL programs include input validation code to lessen semantic errors. Nevertheless, these errors do happen. Consider, for example, the calculation of a customer ID using the following statement: EXEC SQL SELECT nvilsid INTO :cust FROM nvils . ccust WHERE cclient = :client AND nc_id = :id;.

## 7.2. Using TRY...CATCH

A BEGIN TRY...END TRY;—BEGIN CATCH...END CATCH construct—a block of statements designed for handling run-time errors—is also supported. When an error occurs in the BEGIN TRY...END TRY block, control is passed to the BEGIN CATCH...END CATCH construct rather than to the statement after the END TRY keyword.

An error is assumed to have occurred if the program reaches the END TRY keyword with an error raised or an unhandled return code generated. The code within the BEGIN CATCH block acts similarly to a process that traps exceptions. If no error occurs within the TRY block, the CATCH block is bypassed.

# 8. Transactions in SQL

By allowing the execution of several commands as a block and the backing out of the changes done by one of these commands in case a problem occurs, transactions were a logical extension of the atomic operations of earlier SQL implementations. The concept made it easier to maintain data integrity when several operations had to be applied to the database, for instance when a balance transfer was performed.

Providing transactions also enabled the use of SQL in interactive environments such as the client/server model and remote terminal access to the database server. Whereas SQL commands in the command-line environment immediately changed the database, clients in a distributed environment needed to be able to buffer the changes locally until all operations were completed, and then apply all changes atomically on the server. Until transactions were supported by the server, the client had to perform manual rollback, issuing commands to compensate for previous calls.

Among a number of other details, transactions affected the level of autocommit behavior, the visibility of data changes done by one client to both itself and other clients, and the resolution of deadlocks. All SQL implementations provide a degree of transaction support that covers autocommit and visibility, including the most basic COMMIT and ROLLBACK statements. The extent of deadlock resolution varies, and is sometimes beyond the direct control of the user.

## 8.1. Understanding Transactions

A transaction is basically a connection to the engine, with two properties. First, every change to the content of the database made inside a transaction is "temporary" until the transaction is concluded [9,21-23]. Second, a transaction is an atomic operation: either all changes to the database are refused, and the database content is left as if the transaction had never started; or all changes are accepted and committed and pushed to the database engine. A COMMIT operation sets the transaction to the second stage, while a ROLLBACK sets it to the first one.The main properties ensured by the transaction mechanism are: Atomicity, Consistency, Isolation, Durability, usually abbreviated as ACID.

The start of a transaction can be expressed with the statement BEGIN [TRANSACTION], although you'll very probably want to use the AutoCommit property of the Connection object instead.

In fact, Microsoft Jet (and other drivers) is always in autocommit, so every statement affecting the content of the database is committed when execution ends. Each time you put that property to False, you start a new transaction. It will last until you re-enable AutoCommit or close the connection.

## 8.2. Transaction Control Commands

The SQL Language includes the following commands for transaction control. COMMIT ACTION Proceeds with the execution of the task being performed, and confirms the results of the statements up to COMMIT ACTION in the task, regardless of whether the conditions for normal termination of the task have been satisfied. A ROLLBACK ROLLBACK A ROLLBACK ACTION executes ROLLBACK WORK. ROLLBACK ROLLBACK Cancels the execution of the particular task with ROLLBACK WORK, which erases the changes made to the data by the statements in the task. SAVEPOINT a name of a previously created SAVEPOINT ROLLBACK ROLLBACK Combines the processing of ROLLBACK WORK and a ROLLBACK to a designated name from SAVE- POINT, and cancels the data changes. Panel-In-Panel-Update Consider Indexing for Table-3 Using the data inserted into the database, the indexing will now be done.

# 9. Isolation Levels

The transaction isolation level determines the degree to which the data within one transaction is isolated from modifications made by other transactions at the same time [24-26]. Too loose an implementation results in anomalies such as “dirty” reads. Too strict an implementation prevents simultaneous updates to the same data, effectively turning all concurrent transactions into a serialized, sequential process.

The isolation level controls when the effects of one transaction become visible to other transactions, and what kinds of changes other transactions can make to that data while the original transaction is in progress. Several standard, ANSI/ISO SQL levels are defined: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, and SERIALIZABLE. Many vendors offer custom variations.

## 9.1. Overview of Isolation Levels

Isolation levels restrict the degree to which concurrently executing transactions must be isolated from each other, in terms of visibility of intermediate changes made by other transactions. By default, in most DBMSs, no explicit isolation level is specified so the DBMS uses the highest possible level that conforms to the SQL dialect. Keeping data operations isolated from each other is important, as otherwise, unexpected, inconsistent, or unpredictable effects may occur. However, enforcing a high isolation level limits concurrency because it delays data read/write operations until other transactions are completed. This can eventually result in a performance bottleneck for the overall database system.

Table 9.6 lists the available ANSI SQL:92 isolation levels, their degrees of possible data access imitrogeny, and the corresponding READ/write locks used to implement them. Some DBMSs also support the SERIALIZABLE SERIALIZABLE_2 isolation level, which is conceptually even more strict than SERIALIZABLE. The intent of this isolation level is to avoid phantoms by implementing range locks so all SELECT queries must lock all data in a range before they read it, not just the rows they actually select.

## 9.2. Choosing the Right Isolation Level

Once the locks have been set, the transaction essentially locks the resources at that lock level and operates on them accordingly. The next important step is to choose the isolation level. After all, the very reason one needs locking and isolation is because of concurrency.

Isolation levels deal with the locks during the transaction. Note that choosing a sufficient level of isolation generally locks more and locks for a longer time, but it avoids exposure to potential transaction anomalies. In the complete isolation scenario, every transaction sees a consistent snapshot of the database, and any changes made during the transaction are invisible to other transactions until the first one completes.

# 10. Dynamic SQL

Static SQL statements in ProgSQL source code reflect program logic and accessed database structures at compile time. Their clearly defined syntax facilitates syntactical and semantical checking during compilation [8,27-30]. Yet, some applications demand greater runtime flexibility, especially when operation parameters or involved tables and columns aren't known until then. Consequently, ProgSQL provides the EXECUTE IMMEDIATE statement to dynamically execute a statement-string.

When the statement-string defines a query statement, the execution result is assigned to the target variables, specified after the INTO keyword. For other statement types, the statement-string is simply executed. The operation type—selecting, inserting, updating, or deleting—depends solely on the statement-string.

Several dynamic SQL programming elements emerge. First, constructing the statement-string must respect SQL syntax and consider ProgSQL variable contents, formatted to the corresponding SQL literal format. Second, special attention is necessary when the statement-string contains static text enclosed in single or double quotation marks. Third, the target list following INTO must precisely match the query-list computed by the statement-string; any mismatching in number, order, or data types of elements leads to execution errors.

## 10.1. Creating Dynamic SQL Statements

Dynamic SQL statements are created differently from static statements. Static SQL statements are prepared during compilation, whereas dynamic SQL statements can be constructed as part of program execution; they are built by combining program text with input and other data. Dynamic statement capabilities are provided by the EXECUTE IMMEDIATE statement and the DBMS_SQL package.

The EXECUTE IMMEDIATE statement—introduced in Oracle7 Server—can then be used to execute dynamic statements, either because the statement is not known until run-time or because it may change during program execution. It can execute a string at run-time, assign values to host variables, or return values from a query. With dynamic SQL, embedding SQL statements can be constructed within the program during execution and executed subsequently—although the names of tables, views, and number of rows cannot be dynamic. If a program requires include files for each table, this method is preferable, assuming the tables have similar layouts and data; such an approach may not be feasible with a large amount of data in thousands of tables.

## 10.2. Security Considerations

Privacy protection programmes implemented in many countries set strict security requirements for commercial database programmes that are suitable for conducting business on the Internet. In database management, the main confidentiality aspect involves ensuring that commerce details for all customers are kept private from other customers. Other customers wishing to view their own information, of course, must be allowed to do so quickly and easily, and without the encumbrance of additional security procedures.

By contrast, the support for concurrency and multi-user update within the transactions system of the database management programme is typically a negligible part of the overall security aspect for an Internet application, because transactions usually come from a large variety of clients who access different portions of the database and update the database independently. Because of the Internet's open nature, customers and potential customers can try to enter as many conflicting transactions as possible, but eventually the transactions system will serialize them in some order and update without damage to consistencies.

# 11. Stored Procedures

Stored Procedures are stored in executable form inside the database. In addition to SQL statements, they embed procedural elements such as loops, conditionals, handling of variables and parameters, and they do not return virtual tables. Instead, they accept input parameters and return output results as either indexed output parameters or cursors.

Stored Procedures are executed in two steps: first. the CREATE PROCEDURE command creates and stores the procedure inside the database, just as a table is defined once and stored inside the database; second, the EXECUTE command runs a stored procedure, accepting the actual parameters. When a stored procedure runs, it can collect information from the database by issuing queries, perform calculations, and return values to the client.

## 11.1. Defining Stored Procedures

A stored procedure is a group of procedural and data manipulation language statements stored together as a unit and executed when invoked by a program. In a stored procedure, parameters can be defined [9,31-33]. Procedures allow the definition and execution of various actions in the database: they can contain logic that modifies data, test and control execution flow with conditions and loops, perform calculations, and even return a result in the form of a result set or by means of output parameters.

The SQL procedure language permits the embedding of full SQL data manipulation statements within SQL PL source. In addition to embedding SQL statements, SQL PL supports standard control flow statements; declaration and use of indicators, host variables, cursors, and conditions; and declarations to control transaction analysis as well as the use of dynamic and nested compound statements.

## 11.2. Advantages of Using Stored Procedures

SQL supports the writing of stored procedures, which are SQL statements stored on the server. Stored procedures are compiled, cached, and stored in the database for repeated use. They accept parameters and can be called from other SQL statements. Because the server processes the statements, performance can be significantly better than sending SQL statements one by one from a client application via the network.

Stored procedures also help maintain consistency, since they live on the server and are processed once. Several applications can call the same set of procedures, ensuring consistent results. Furthermore, stored procedures allow separation of the logical database design from its physical implementation (data access). If there's any change to access, such as a change in the physical layout of the database, it can be reflected in the stored procedures without altering the application code that calls them.

# 12. Best Practices for Advanced SQL Programming

A number of standards, guidelines, and rules apply to advanced SQL programming. The degree to which these should be followed varies from one environment to another. The "rules" that are most important in a given environment — in particular, the rules for coding style — take precedence over more general rules, guidelines, and standards: what is right in one environment might be wrong in another. A few general rules for SQL programming include the following:

Every piece of meaningful content should be in the database only once (a rule of both database design and SQL programming). Naming conventions should be well established and strictly followed. Join conditions should always be spelled out explicitly [34-35]. The use of undisclosed/hidden conditions should be avoided at all times. Transient data should never be stored. Group By does not belong in Karnaugh maps, but in SQL.

SQL is a powerful and sophisticated language that should be used only for "things that are supposed to be done in SQL." The establishment of a definite prohibition on the use of Group By in Karnaugh maps of any kind is a good example of such a rule for SQL advanced.

## 12.1. Performance Optimization Techniques

The preceding chapters describe SQL features and techniques. These enable developers to build and maintain effective applications but do not guarantee that the SQL will perform well when executed against real production data. Two useful techniques that can increase performance are described here: tablespaces and query optimisation.

Creating tablespaces is a method of physically distributing the data of a database on to different places on the hard disks. When records are inserted into tables in the database, the records are stored physically in the tablespaces.

When the SQL engine requires data for execution, it searches the tablespaces for the data. Therefore, is can be faster to access the data because the search is distributed across different places on the disk.

## 12.2. Maintainability and Readability

Readability and maintainability are important aspects of database programming. In fact, they are among the primary intentions of stored procedure support in SQL, as described in the introduction. SQL contains several features aimed at improving readability. These include: automatic indentation of source code, formatting source code for automatic line breaks, activating or deactivating keyword case, keyword coloring, and enabling or disabling recursive source code listings. For instance, source code can be displayed with all keywords in uppercase and source code comments highlighted in a specific color. Additionally, any procedural SQL editor can automatically indent source SQL statements. For example, in the sample procedure in Chap. 10, the SQL inside the BEGIN-END block is indented relative to the block itself, while the cursor is positioned after the semi-colon indicates the start of the main SQL statement of the procedure.

Most database vendors also provide third-party products in the form of add-ons to existing products that can perform functions that improve the readability and maintainability of database code. Typically, for stored procedures, these third-party add-ons include functionalities such as formatting the source code and creating documentation about the code. In addition to the inherent features that enhance readability, database systems also use cost-based optimization engines that perform run-time tuning, assisting anyone who attempts to maintain and tune the system.

# 13. Case Studies

Advanced SQL Programming Techniques

The following case studies illustrate the considerable potential of SQL's various elements when combined. The ongoing availability of a large data set on SQL Sounds has been instrumental in demonstrating the thoroughness with which SQL can be pursued insofar as those elements are concerned.

The first study highlights the various easings that can be implemented, together with the special easings with integral deltas. In each case, the elements displayed in the key connections matrix are used, as are the functions oxidize (to represent a rising edge—relative change), diodize (to represent differentials) and equalize (to represent a felony). Fundamental SQL, as demonstrated in the Groupings and Period & Dates sections, plays an equally important role. The harmonized scales have been set at an E min., standard tone length at quaver and standard tonality at maj.

## 13.1. Real-World Applications

Using subprograms to explain complex SQL statements helps students program more easily. It requires much practice and experience to create good subprograms; at first, a programmer cannot realize that. Programming since the fifth grade, a programmer can feel easily the importance of subprograms in advanced SQL programming. A clarified SQL statement developed by the teacher can be shared by the whole class. It helps students recall the knowledge taught in the class later.

In addition to using subprograms, an Enterprise Resource Planning system can be used for practice and innovation of subprograms. An ERPS of an institute can supply a real-campus environment for developing SQL statements of practice and project. Using a concise English description for the statement built in the ERPS increases productivity. In a real environment, the end user can realize the needs for subprograms. An innovative English description can help programmers understand quickly and successfully decide the function of a subprogram. It also helps stadium users select the purpose of a call in a stadium. The introduction of subprograms is listed in innovative advanced SQL training.

## 13.2. Lessons Learned

A thorough testing of SQL procedures with the routine call mechanism implemented in the package DBMS_SQLDBA led to several interesting observations:

Use with caution. In most situations, it is recommended to use SQL and PL/SQL statements in the procedure code instead of the routine call mechanism. This approach minimizes context switches between the PL/SQL and SQL engines and allows Oracle to cache and share statement execution plans. Only those statements that cannot be executed directly in PL/SQL or SQL—such as DDL commands, DCL commands, and TRUNCATE TABLE—should be coded with the routine call mechanism.

Don't use unbound VARCHAR2 variables. Bound VARCHAR2 variables should be avoided as well. Both bound and unbound VARCHAR2 variables code compile-time constant values only; their content cannot be changed during execution of the SQL statement. Although it is not officially documented yet, it seems that when a bounded VARCHAR2 variable is used, Oracle executes the statement outside of a transaction, which explains the failure of certain statements such as CREATE TABLE, DROP TABLE, and so forth.

## 14. Future Trends in SQL Programming

As part of the Future Trends in SQL Programming, new SQL extensions are planned for several major database management systems. These extensions recognize the growing importance of XML and object–relational databases, introduce recursive queries to simplify the processing of hierarchical data structures (e.g., bill of materials, threaded message boards, and directory listings), and support updates to views. For example, the delay of evaluation facility presently implemented in Sybase Adaptive Server allows updates to be performed on views (supersets) on the source tables that support those views.

The increasing importance of data warehousing and decision support databases is also recognized: IBM has added multiple indexes to a single table, interleaved indexing, bitmap indexes, table parallelism, support for DUAL aggregate function, and cost-based optimizer hints to the DB2 V2 product. Furthermore, Oracle provides a Partitioning option for the Oracle V7 product that allows tables and indexes to be split into partitions based on range of values.

## 15. Conclusion

With advanced SQL programming, you can manipulate data in numerous ways. You can control the data that your program generates; you can express simple queries or complex business rules that reflect all possible outcome scenarios, thus gaining insights into your business or organization. SQL is essential for displaying and updating data correctly. When you have no control over the data you generate, you are susceptible to the smallest changes that can derail your queries or calculations. Therefore, as a programmer, you aspire to design and write code that is surf-tastically flexible. In conclusion, SQL is highly effective for handling large amounts of data. Extensive testing can prevent boredom during debugging.

Advanced SQL programming provides tools for working with data that simple queries cannot handle. SQL Visual, which integrates SQL with Visual Basic for Applications (VBA), gives you complete control over data, presentation, and updates. You can create any kind of program imaginable due to the depth of the SQL language. Effectively managing large volumes of data becomes manageable, and thorough testing ensures that the development process remains engaging.

## References:

[1] Ojuri S, Han TA, Chiong R, Di Stefano A. Optimizing text-to-SQL conversion techniques through the integration of intelligent agents and large language models. Information Processing & Management. 2025 Sep 1;62(5):104136.

[2] Chen X, Zhu R, Ding B, Wang S, Zhou J. Lero: applying learning-to-rank in query optimizer. The VLDB Journal. 2024 Sep;33(5):1307-31.

[3] Thalji N, Raza A, Islam MS, Samee NA, Jamjoom MM. Ae-net: Novel autoencoder-based deep features for sql injection attack detection. IEEE access. 2023 Nov 28;11:135507-16.

[4] Chakraborty S, Paul S, Hasan KA. Performance comparison for data retrieval from nosql and sql databases: a case study for covid-19 genome sequence dataset. In2021 2nd International Conference on Robotics, electrical and signal processing techniques (ICREST) 2021 Jan 5 (pp. 324-328). IEEE.

[5] Crespo-Martínez IS, Campazas-Vega A, Guerrero-Higueras ÁM, Riego-DelCastillo V, Álvarez-Aparicio C, Fernández-Llamas C. SQL injection attack detection in network flow data. Computers & Security. 2023 Apr 1;127:103093.

[6] Antas J, Rocha Silva R, Bernardino J. Assessment of SQL and NoSQL systems to store and mine COVID-19 data. Computers. 2022 Feb 21;11(2):29.

[7] Shivadekar S, Kataria DB, Hundekar S, Wanjale K, Balpande VP, Suryawanshi R. Deep learning based image classification of lungs radiography for detecting covid-19 using a deep cnn and resnet 50. International Journal of Intelligent Systems and Applications in Engineering. 2023;11:241-50.

[8] Ashlam AA, Badii A, Stahl F. Multi-phase algorithmic framework to prevent SQL injection attacks using improved machine learning and deep learning to enhance database security in real-time. In2022 15th International Conference on Security of Information and Networks (SIN) 2022 Nov 11 (pp. 01-04). IEEE.

[9] Tanimura C. SQL for Data Analysis: Advanced Techniques for Transforming Data Into Insights. " O'Reilly Media, Inc."; 2021 Sep 9.

[10] Brunner U, Stockinger K. Valuenet: A natural language-to-sql system that learns from database information. In2021 IEEE 37th International Conference on Data Engineering (ICDE) 2021 Apr 19 (pp. 2177-2182). IEEE.

[11] Panda SP. Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems. Deep Science Publishing; 2025 Jun 22.

[12] Lawson JG, Street DA. Detecting dirty data using SQL: Rigorous house insurance case. Journal of Accounting Education. 2021 Jun 1;55:100714.

[13] Zhang B, Ren R, Liu J, Jiang M, Ren J, Li J. SQLPsdem: A proxy-based mechanism towards detecting, locating and preventing second-order SQL injections. IEEE Transactions on Software Engineering. 2024 May 14;50(7):1807-26.

[14] Panda SP. Artificial Intelligence Across Borders: Transforming Industries Through Intelligent Innovation. Deep Science Publishing; 2025 Jun 6.

[15] Gandhi N, Patel J, Sisodiya R, Doshi N, Mishra S. A CNN-BiLSTM based approach for detection of SQL injection attacks. In2021 International conference on computational intelligence and knowledge economy (ICCIKE) 2021 Mar 17 (pp. 378-383). IEEE.

[16] Dhanaraj RK, Ramakrishnan V, Poongodi M, Krishnasamy L, Hamdi M, Kotecha K, Vijayakumar V. Random forest bagging and x-means clustered antipattern detection from SQL query log for accessing secure mobile data. Wireless communications and mobile computing. 2021;2021(1):2730246.

[17] Panda SP, Muppala M, Koneti SB. The Contribution of AI in Climate Modeling and Sustainable Decision-Making. Available at SSRN 5283619. 2025 Jun 1.

[18] Shivadekar S. Artificial Intelligence for Cognitive Systems: Deep Learning, Neuro-symbolic Integration, and Human-Centric Intelligence. Deep Science Publishing; 2025 Jun 30.

[19] Davidson L. Pro SQL Server Relational Database Design and Implementation: Best Practices for Scalability and Performance. Apress; 2021.

[20] Zhang W, Li Y, Li X, Shao M, Mi Y, Zhang H, Zhi G. Deep Neural Network-Based SQL Injection Detection Method. Security and Communication Networks. 2022;2022(1):4836289.

[21] Roy P, Kumar R, Rani P. SQL injection attack detection by machine learning classifier. In2022 International conference on applied artificial intelligence and computing (ICAAIC) 2022 May 9 (pp. 394-400). IEEE.

[22] Katsogiannis-Meimarakis G, Koutrika G. A survey on deep learning approaches for text-to-SQL. The VLDB Journal. 2023 Jul;32(4):905-36.

[23] Khan W, Kumar T, Zhang C, Raj K, Roy AM, Luo B. SQL and NoSQL database software architecture performance analysis and assessments—a systematic literature review. Big Data and Cognitive Computing. 2023 May 12;7(2):97.

[24] Hong Z, Yuan Z, Zhang Q, Chen H, Dong J, Huang F, Huang X. Next-generation database interfaces: A survey of llm-based text-to-sql. arXiv preprint arXiv:2406.08426. 2024 Jun 12.

[25] Islam S. Future trends in SQL databases and big data analytics: Impact of machine learning and artificial intelligence. Available at SSRN 5064781. 2024 Aug 6.

[26] de Oliveira VF, Pessoa MA, Junqueira F, Miyagi PE. SQL and NoSQL Databases in the Context of Industry 4.0. Machines. 2021 Dec 27;10(1):20.

[27] Rockoff L. The language of SQL. Addison-Wesley Professional; 2021 Nov 4.

[28] Shivadekar S, Halem M, Yeah Y, Vibhute S. Edge AI cosmos blockchain distributed network for precise ablh detection. Multimedia tools and applications. 2024 Aug;83(27):69083-109.

[29] Fotache M, Munteanu A, Strîmbei C, Hrubaru I. Framework for the assessment of data masking performance penalties in SQL database servers. Case Study: Oracle. IEEE Access. 2023 Feb 22;11:18520-41.

[30] Panda SP. Augmented and Virtual Reality in Intelligent Systems. Available at SSRN. 2021 Apr 16.

[31] Karwin B. SQL Antipatterns, Volume 1: Avoiding the Pitfalls of Database Programming. The Pragmatic Programmers LLC; 2022 Oct 24.

[32] Nasereddin M, ALKhamaiseh A, Qasaimeh M, Al-Qassas R. A systematic review of detection and prevention techniques of SQL injection attacks. Information Security Journal: A Global Perspective. 2023 Jul 4;32(4):252-65.

[33] Chakraborty S, Aithal PS. CRUD Operation on WordPress Database Using C# SQL Client. International Journal of Case Studies in Business, IT, and Education (IJCSBE). 2023 Nov 28;7(4):138-49.

[34] Panda SP. The Evolution and Defense Against Social Engineering and Phishing Attacks. International Journal of Science and Research (IJSR). 2025 Jan 1.

[35] Choi H, Lee S, Jeong D. Forensic recovery of SQL server database: Practical approach. IEEE Access. 2021 Jan 18;9:14564-75.