Impact of Using Column Aliases in Oracle Database 26ai

Michal Kvet
University of Žilina
Žilina, Slovakia
Michal.Kvet@uniza.sk

***Abstract*— Aggregate SQL queries are performance-critical in analytical and decision-support systems. SQL standard does not allow column aliases to be referenced in the GROUP BY and HAVING clauses, requiring repeated expressions that reduce query readability and maintainability. Oracle Database 26ai introduces support for using column aliases in these clauses, extending standard SQL semantics. This paper evaluates the performance impact of this enhancement by comparing four approaches: a standard-compliant reference solution, two database-independent solutions based on alias extraction and logical constructs, and an embedded optimizer-level solution provided by Oracle Database. The evaluation is conducted using a real-world aviation dataset across multiple grouping and aggregation scenarios. The results show that database-independent approaches introduce an overhead of 0.85% to 1.47%, while the Oracle embedded solution exhibits no measurable performance impact, with overhead remaining below 0.05%.**

I. Introduction

Relational databases form the backbone of modern data management systems, supporting applications across domains such as business intelligence, finance, healthcare, and scientific research. They also strongly help with the sustainability, green economics by focusing on the future of the planet. Their widespread adoption is largely due to the relational model's ability to represent data in a structured, consistent, and scalable manner. At the core of interacting with relational databases is *Structured Query Language (SQL)*, a declarative language designed to define, manipulate, and retrieve data efficiently. SQL provides users with a powerful abstraction that allows them to specify *what* data is required rather than *how* it should be accessed, leaving the optimization and execution strategies to the *database management system (RDBMS)* [2] [8].

The relational model organizes data into *relations* (tables) composed of *rows* (tuples) and *columns* (attributes), where each table represents a specific entity or relationship. Tables are connected through keys, such as primary and foreign keys, which enforce referential integrity and enable meaningful relationships between datasets. This model promotes data consistency, minimizes redundancy through normalization, and supports flexible querying. SQL operates directly on this relational structure, offering a standardized and expressive interface for database operations. The main benefit of the relational database concept is associated with the strict *transaction support*, delimited by the ACID properties – *atomicity*, *consistency*, *isolation* and *durability* [6].

Among SQL's capabilities, the *SELECT statement* is the most fundamental and frequently used construct. It enables users to retrieve data from database tables, apply conditions, perform computations, group results, and generate summaries. The expressive power of the SELECT statement lies in its combination of individual clauses and built-in functions, particularly *aggregate* and *analytical functions*, which are essential for analytical and decision-support queries. Understanding the principles behind these components and how queries are logically processed is critical for both correct query formulation and performance optimization.

This paper examines the performance implications of using aggregate functions within relational database theory. Rather than concentrating on low-level data retrieval mechanisms—such as physical infrastructure, data block loading, or indexing strategies—this study emphasizes the impact on query formulation and user experience. In particular, Oracle 26ai introduces support for using column aliases in the *GROUP BY* and *HAVING* clauses, a feature that simplifies query construction and improves code readability. The paper analyzes two fundamental approaches enabled by this capability and highlights their respective performance characteristics, including the additional processing stages involved in query evaluation.

The remainder of this paper is organized as follows. Section 2 provides an overview of relational database concepts. Section 3 presents the definition of the *SELECT* statement and its available clauses. It also describes the query execution plan and evaluation stages. Aggregate functions are introduced in Section 4. Finally, Section 5 presents the solution followed by the performance evaluation methodology and discussion of the experimental results (Section 6).

II. Principles of SQL and Relational Databases

SQL is founded on the principles of *relational algebra* and *relational calculus*, providing a rigorous mathematical basis for data querying. As a declarative language, SQL enables users to specify the desired result set without defining the procedural steps required to obtain it. This abstraction between query formulation and execution allows RDBMSs to apply automatic optimizations, including query rewriting, indexing strategies, and cost-based optimization techniques.

Relational database systems enforce constraints such as entity integrity, referential integrity, and domain constraints to ensure data accuracy and consistency. These mechanisms are especially critical in multi-user environments, where concurrent

access can introduce anomalies. SQL supports these foundational principles by providing facilities for *data definition* (*DDL*), *data manipulation* (*DML*), and *data control* (*DCL*). Nevertheless, in analytical and research-oriented applications, the SELECT statement remains the central mechanism for retrieving and aggregating relational data to support analysis and decision-making [10].

III. The SELECT Statement and Its Clauses

The *SELECT* statement is composed of multiple clauses, each serving a specific purpose in defining the structure and content of the query result.

The *SELECT* clause specifies the columns or expressions to be returned, which may include direct attributes, computed values, or aggregate functions. This clause defines the projection of the query, determining which data elements appear in the final output.

The *FROM* clause identifies the source tables or views from which data is retrieved. It may include JOIN operations that combine rows from multiple tables based on specified conditions. JOINs are fundamental to relational databases, as they allow users to reconstruct relationships between normalized tables and produce meaningful combined datasets.

The *WHERE* clause applies row-level filtering conditions, restricting the dataset to only those records that satisfy specified predicates. These conditions are evaluated before aggregation occurs, making the WHERE clause essential for controlling the scope of data considered in a query. Logical operators, comparison operators, and subqueries are commonly used within this clause to express complex filtering logic.

The *GROUP BY* clause organizes rows into groups based on one or more attributes, enabling aggregate functions to be applied at the group level rather than across the entire dataset. This clause is central to summary queries, such as calculating totals, averages, or counts for each category within the data.

The *HAVING* clause further filters groups created by the GROUP BY clause. Unlike the WHERE clause, which operates on individual rows, HAVING operates on aggregated results. It is commonly used to impose conditions on aggregate values, such as selecting only groups with totals above a certain threshold.

Finally, the *ORDER BY* clause controls the presentation of the result set by sorting rows based on specified attributes or expressions. While it does not affect the logical content of the result, it plays an important role in data interpretation and reporting.

According to the *SQL standard* (ISO/IEC 9075) [5], *column aliases* defined in the SELECT list are scoped to the query result and become visible only after the projection phase of query evaluation. The standard explicitly allows the use of column aliases in the ORDER BY clause, as defined in ISO/IEC 9075-2 (SQL/Foundation), because ordering is logically applied after the SELECT list has been evaluated. Conversely, the standard does not permit column aliases to be referenced in the WHERE, GROUP BY, or HAVING clauses, since these clauses are logically processed prior to the evaluation of the SELECT list (ISO/IEC 9075-2, Clause 7 — Query expressions). Consequently, expressions used in the WHERE, GROUP BY, and HAVING clauses must be explicitly restated rather than referenced through aliases. These scoping rules ensure deterministic query semantics and portability across SQL-compliant database management systems, although some vendors provide extensions that relax these constraints beyond the standard specification. Oracle 26ai brings the ability to use column aliases in the GROUP BY and HAVING clauses by overriding the SQL standardization [9]. Performance impacts are discussed in this paper.

A. Logical process of SELECT statement execution

Although SQL SELECT statements are written in a logical, human-readable order, their execution follows a well-defined internal sequence. Understanding this execution order is essential for accurately interpreting query behavior, optimizing performance, and predicting the impact of filtering and aggregation operations. The execution plan generated by the query optimizer determines how each clause is evaluated, but the logical order of operations remains consistent across relational systems.

The execution begins with the FROM clause, where the database engine identifies the source tables, resolves JOIN operations, and constructs the initial working dataset. During this stage, JOIN conditions are applied and intermediate result sets may be generated. Next, the WHERE clause is evaluated to filter individual rows based on the specified predicates, reducing the dataset before any aggregation takes place. This step is critical for performance, as early row elimination minimizes the volume of data processed in subsequent stages.

After row-level filtering, the GROUP BY clause is applied to organize the remaining rows into groups based on the set of attributes.

The HAVING clause is evaluated after aggregation, allowing conditions to be applied to grouped results rather than individual rows. This distinction between WHERE and HAVING is fundamental to correct query semantics and execution. HAVING clause i sused for filtering based on the aggregate function.

Once grouping and aggregation are complete, the SELECT clause is processed to produce the final projection of columns, expressions, and aggregate values. At this stage, column aliases and derived expressions are resolved. Finally, the ORDER BY clause is executed to sort the result set according to the specified criteria, followed by result-limiting clauses such as FETCH, LIMIT, or ROWNUM, depending on the database system. Used Oracle Database uses FETCH sub-command. This execution order explains many common SQL behaviors and serves as a foundation for understanding query optimization and performance analysis in relational databases [5] [6] [7].

IV. Aggregate Functions in SQL

Aggregate functions in SQL are used to compute summary values over sets of rows, thereby enabling analytical and decision-support operations within relational database systems. Defined by the SQL standard (ISO/IEC 9075), aggregate functions operate on a collection of input values and return a single scalar result per group or per query. Commonly used aggregate functions include *COUNT*, *SUM*, *AVG*, *MIN*, and *MAX*, each serving a distinct role in summarizing data

distributions. These functions are most frequently applied in conjunction with the GROUP BY clause, which partitions the result set into logical groups prior to aggregation, allowing summaries to be computed at varying levels of granularity.

The principles governing aggregate functions are rooted in relational algebra and the formal semantics of SQL query evaluation. Aggregation is performed after row-level filtering (WHERE) and grouping (GROUP BY) but before result projection and ordering. The SQL standard enforces strict rules to ensure semantic correctness, such as requiring all non-aggregated expressions in the SELECT list to appear in the GROUP BY clause. Additionally, aggregate functions ignore NULL values. Additionally, SQL supports the use of aggregate functions with DISTINCT to eliminate duplicate values, further enhancing analytical precision. These principles ensure consistent and deterministic behavior across SQL-compliant database systems, while also enabling query optimizers to apply efficient execution strategies. As a result, aggregate functions form a cornerstone of SQL-based analytical processing, supporting both transactional reporting and large-scale data analysis [4].

Aggregate functions often require scanning large volumes of data, grouping records, and performing calculations that can be computationally expensive. Therefore the proper definition is crucial for the performance.

Poorly optimized aggregate queries can lead to increased query execution time, higher CPU and memory usage, and reduced system responsiveness, especially in databases supporting high-transaction or real-time applications. As datasets continue to grow in size and complexity, ensuring efficient execution of SELECT statements with aggregate functions becomes essential for maintaining acceptable performance levels and meeting application service requirements.

Efficient performance of aggregate SELECT queries also directly impacts scalability and overall database reliability. Techniques such as proper indexing, query optimization, use of execution plans, and efficient grouping strategies can significantly reduce the cost of aggregate operations. Optimized aggregate queries enable databases to handle concurrent users more effectively and support analytical workloads without degrading transactional performance. Therefore, understanding and improving the performance characteristics of SELECT statements with aggregate functions is fundamental to relational database design and tuning, as it contributes to faster data retrieval, better resource utilization, and improved decision-making based on timely and accurate aggregated results.

Traditionally, according to the SQL standard (ISO/IEC 9075), column aliases defined in the SELECT list are not visible to clauses such as WHERE, GROUP BY, or HAVING, because these clauses are evaluated before the projection phase where aliases are assigned. This restriction often necessitated the repetition of complex expressions in grouping and filtering conditions, potentially reducing code clarity and increasing maintenance overhead. Recent advancements in Oracle's AI-enhanced database (*Oracle AI Database 26ai*), address this limitation by allowing column aliases to be directly referenced in both the GROUP BY and HAVING clauses.

V. EXISTING COMPETING SOLUTIONS VS. ORACLE 26AI ENHANCEMENTS

A. Reference solution - SOLref

The reference solution follows the traditional approach prescribed by the SQL standard, which requires full repetition of column expressions within the GROUP BY and HAVING clauses. This approach is fully compliant with SQL standardization and is therefore portable across database systems and versions. It necessitates restating all functional expressions, including data transformations and formatting logic, wherever grouping or post-aggregation filtering is applied. While this method ensures unambiguous query semantics, it can increase query complexity and reduce readability. Fig. 1 illustrates the corresponding data processing flow for this standard-compliant evaluation model.

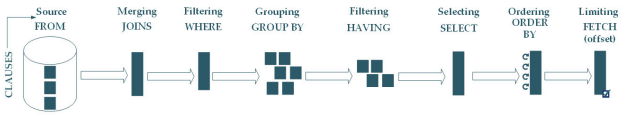


Fig. 1. Data flow – original SQL norm [5]

B. Extracting aliases – SOL1

In [5], an extended statement execution process is introduced that begins with a preliminary evaluation of the SELECT clause to identify column and expression aliases. This early extraction phase enables aliases to be visible throughout the entire statement, allowing them to be referenced in any clause. The proposed architecture, illustrated in Fig. 2, relies on direct mapping and systematic statement rewriting to preserve semantic correctness. As a result, query formulation is simplified, significantly improving code readability and overall user experience.

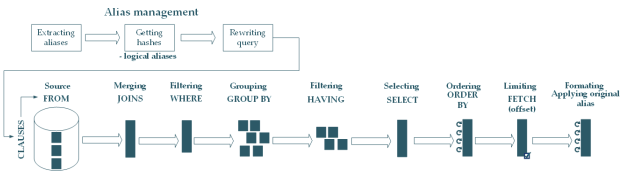


Fig. 2. Data flow – enhanced solution using alias extraction [5]

Despite the enhanced representation, the actual execution performed by the SQL engine remains unaffected. Before execution, the transformed statement is rewritten into an equivalent, standard-compliant form in which all column aliases are systematically replaced by their original expressions. As a result, the proposed enhancement operates purely as a representational and usability layer, improving query formulation from the user's perspective without altering internal execution semantics. The logical introduction of aliases has no impact on query optimization strategies or runtime behavior, thereby ensuring full compatibility with existing SQL execution engines, as illustrated in Fig. 3. The transformation of the statement is performed by the mapping layer. It is applied to the flight monitoring system within a *flight information region* (*FIR*) – Fig. 3.

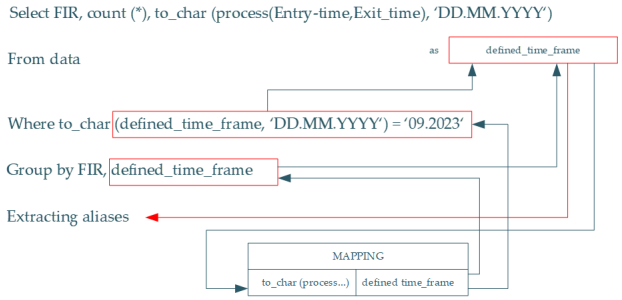


Fig. 3. Select statement transformation example [5]

A primary limitation of the described approach lies in its impact on usability and overall processing efficiency. Specifically, it introduces an additional execution phase in which column aliases must be resolved before data sources can be evaluated and merged. As a result, the database management system is required to delay further processing until alias extraction and reference resolution are completed. This additional step increases processing overhead and execution costs, even for statements that do not explicitly rely on aliases. Furthermore, the approach does not support alias nesting, and alias visibility is restricted to the scope of the individual query and its immediate hierarchical context.

The solution relies on direct pointers to definitions specified in the SELECT clause, which are physically interpreted through query rewriting prior to execution. This behavior may prevent the database engine from matching the transformed statement with an existing execution plan identifier. In cases where semantically equivalent queries differ only in alias expressions, the rewritten statements may vary syntactically, leading to failed plan reuse and the necessity for a hard parse. Consequently, this can negatively affect performance due to repeated optimization and plan generation.

C. *Logical constructs – SOL2*

To address these limitations, an enhanced approach extends the original concept by treating column aliases exclusively as logical constructs that define the result set, rather than as physical components of the execution plan. In this model, alias extraction still occurs during the initial phase; however, throughout execution, aliases are handled solely at a logical level. The physical structure of the output is determined only after the complete result set has been produced and, if specified, sorted. Logical aliases are represented using hash-based identifiers derived from the underlying column, expression, or function definitions. This mechanism ensures consistent execution plan identification, yielding identical hash values even when alias names or statement formulations differ. As a result, execution plan reuse is preserved, and performance stability is improved across semantically equivalent queries. Architecture and process flow is in Fig. 4.

The limitation of this approach is just the usability and efficiency of the whole process. Namely, it brings an additional step in the execution. Before evaluating and extracting data sources and merging them, the database manager must wait to proceed with aliases and referencing them. It increases costs and processing time demands, even in cases where no aliases are used in the statement. After all, such an approach does not allow nesting, and aliases are visible only for the particular query and its superior types.

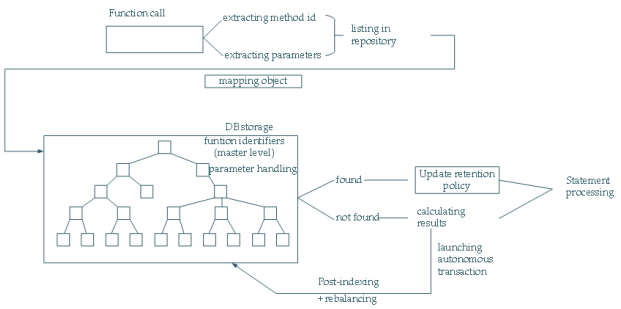


Fig. 4. Enhanced architecture and process flow

The above solution uses the pointers to the definition stated in the Select clause of the statement, which is physically interpreted as a rewrite of the query before it is processed. This may result in not finding a matching existing execution plan identifier. Additionally, if different alias expressions are used, the statement would be written differently, and again, the mapping association between the statement and existing execution plans would not be recognized, resulting in the necessity to perform a hard parse.

An enhanced approach extends the pure concepts, and aliases are treated only as a logical unit forming the result set. This brings a significant generalization of the statement, making it further applicable. Namely, extraction of the aliases is done in the first phase; however, during the execution, column aliases are treated only as logical units, and the physical shape of the output is defined at the end of the processing, just after the result is set is completely formed, even sorted, if specified. Logical aliases are obtained by the hash function, which gets the definition of the column, expression, or function. It consequences in getting the same hash identifier for the statement execution plan identification, even if the aliases (and therefore the form of the statement, as well) are different.

D. *Oracle 26ai – SOL3*

Beginning with Oracle Database 23ai and further enhanced in Oracle Database 26ai, column aliases defined in the SELECT clause may be directly referenced in the GROUP BY and HAVING clauses [1] [3]. This capability represents a significant evolution from traditional SQL semantics, where such aliases were not visible during grouping or post-aggregation filtering. The use of the column alias in the GROUP BY clause eliminates the need to restate the expression, thereby simplifying query structure and improving readability. Similarly, the column alias can be referenced in the HAVING clause instead of repeating the aggregate expression.

This enhancement reduces redundancy in query definitions and minimizes the risk of inconsistencies arising from duplicated expressions, particularly in complex analytical queries involving nested functions or multiple aggregations. By allowing aliases to be reused consistently across clauses, Oracle 23ai/26ai improves developer productivity and code maintainability without altering the underlying execution semantics. Internally, the database engine preserves standard-compliant behavior by resolving aliases logically and rewriting

the statement as necessary prior to execution. As a result, this feature provides a more expressive and user-friendly SQL syntax while maintaining compatibility with existing optimization and execution mechanisms.

Starting with Oracle Database 23ai, it is possible to reference columns by their ordinal position in the GROUP BY clause. This functionality is not enabled by default and must be explicitly activated at either the session or system level. In the present study, the feature is enabled for the current session to demonstrate its application and assess its impact on query formulation and readability:

```
alter session
 set group_by_position_enabled=true;
```

Query transformation

The *Shared Pool* is a centralized memory area that stores information required for efficient SQL execution and instance management. It is part of the *System Global Area* (*SGA*) in Oracle Database. Among its primary responsibilities, the Shared Pool maintains parsed SQL statements and their associated execution plans, allowing multiple sessions to reuse previously compiled queries without repeating the parsing and optimization process. By caching execution plans, the Shared Pool reduces CPU overhead, minimizes redundant parsing, and improves response time for frequently executed queries. In addition to execution plans, the Shared Pool contains other structures such as the *Library Cache*, which stores SQL and PL/SQL objects, and the *Dictionary Cache*, which holds metadata about database objects. To make the results consistent and independent of the workload, the whole Shared Pool is flushed before each operation:

```
alter system flush Shared_pool;
```

To analyze the evaluation of a SQL statement by the database optimizer, tracing was enabled for the statement under execution. The corresponding log file was identified using the following command:

```
select value
 from v$diag_info
  where name = 'Default Trace File';
```

Statement tracing itself was delimited by monitoring *10053 event*:

```
alter session
 set events '10053 trace name context forever';
```

Testing environment

Controlled experimental environment configured with the Oracle Database system. was used for the performance evaluation. The system was equipped with a Intel(R) Core(TM) Ultra 7 155U processor with Intel(R) graphics 128MB, operating at 1.70 GHz. It used 32 GB RAM, DDR4 type – 6400 MT/s. 4 TB NVMe solid-state drive providing read and write speeds of up to 7000 MB/s was used. The operating system was Windows 11 Pro, version 25H2. The database platform used in the experiments was Oracle Database 26ai Free (Release 26.0.0.0.0).

Internal used configuration parameters:

- SGA: 16 GB target, Fixed Size: 4 GB, Database Buffers: 8 GB, REDO buffers: 16 GB
- PGA: 2 GB target,
- Memory management: automatic
- STATISTICS_LEVEL: TYPICAL,
- OPTIMIZER_MODE: ALL_ROWS,
- CURSOR_SHARING: EXACT,
- PARALLEL_DEGREE_POLICY: MANUAL
- PARALLEL_MAX_SERVERS: 480
- DB_BLOCK_SIZE: 8 192 B
- PROCESSES / SESSIONS: 1 920 / 2 904
- UNDO_MANAGEMENT: AUTO
- MEMORY_TARGET, SGA_TARGET: 0 (automatic shared memory management approach)
- PDBS: 2

For the computational study, a real-world dataset from the aviation domain was employed. The dataset comprises three main components: (i) planned routes for aircraft operations, (ii) actual flight trajectories derived from positional data of aircraft and other relevant objects, along with the operational status of flight regions, and (iii) flight monitoring data obtained by assigning aircraft to *Flight Information Regions* (*FIRs*) based on their entry and exit times.

Fig. 5 illustrates an example of the input data used for FIR assignment. Each record includes the aircraft identifier (ECTRL_ID), the FIR assignment reference (AUA_ID), and the corresponding temporal boundaries (ENTRY_TIME and EXIT_TIME). The records are stored sequentially according to the SEQUENCE_NUMBER attribute.

```
"ECTRL ID","Sequence Number","AUA ID","Entry Time","Exit Time"
"186858226","1","EGGXOCA","01-06-2015 04:55:00","01-06-2015 05:57:51"
"186858226","2","EISNCTA","01-06-2015 05:57:51","01-06-2015 06:28:00"
"186858226","3","EGTTCTA","01-06-2015 06:28:00","01-06-2015 07:00:44"
"186858226","4","EGTTTCTA","01-06-2015 07:00:44","01-06-2015 07:11:45"
"186858226","5","EGTTICTA","01-06-2015 07:11:45","01-06-2015 07:15:55"
"186858227","1","EGGXOCA","01-06-2015 04:08:00","01-06-2015 05:01:00"
"186858227","2","EISNCTA","01-06-2015 05:01:00","01-06-2015 05:34:00"
```

Fig. 5. Data set

A *Flight Information Region* (*FIR*) is a defined area of airspace within which a designated national air traffic control (*ATC*) authority is responsible for the provision of *Flight Information Services* (*FIS*) and alerting services. FIRs are established by the *International Civil Aviation Organization* (*ICAO*) and collectively cover the entire global airspace, including both terrestrial and oceanic regions. The principal functions of FIRs include ensuring the safety and efficiency of air traffic through the dissemination of essential information and the coordination of emergency responses, managing and controlling airspace operations within their boundaries, and supporting flight route planning and optimization with the aim of reducing environmental impact and operational costs.

The boundaries of FIRs are time-dependent and may evolve over time. The complete dataset used in this study consisted of 1,000 flights. On average, each flight was represented by

approximately 1,000 records for both the planned and actual flight trajectories.

Results

For the performance evaluation study, eight scenarios were used. For each scenario, 5 different Select statements were defined, each executed 10 times:

- S1 – GROUP BY clause based on the conventional attribute, no HAVING clause.
- S2 – GROUP BY clause based on the standard function, no HAVING clause.
- S3 – GROUP BY clause based on the user-defined function, no HAVING clause.
- S4 – HAVING clause based on 1 aggregate function.
- S5 - HAVING clause based on 2 aggregate functions.
- S6 – HAVING clause based on 1 aggregate function and one conventional attribute.
- S7 – HAVING clause based on 1 aggregate function and one standard function.
- S8 – HAVING clause based on 1 aggregate function and one user-defined function.

Table I shows the configuration for individual scenarios.

The evaluated scenarios cover all potential statement types that may be used in practical applications. The objective is to investigate whether the introduced column aliases can be utilized in the final solutions and how they affect query performance. Alternatively, the study examines whether their relevance is limited to simplifying the coding approach during experimentation and development.

Eight scenarios were applied to four solutions. *SOLref* serves as the reference solution, as it does not use column aliases.

TABLE I. SCENARIOS

Scenario	GROUP BY	ATR	FUNC STANDARD	FUNC USER	HAVING	AGG	ATR	FUNC STANDARD	FUNC USER
S1	✓	✓							
S2	✓		✓						
S3	✓			✓					
S4	✓	✓			✓	✓			
S5	✓	✓			✓	✓ (2x)			
S6	✓	✓			✓		✓		
S7	✓	✓			✓			✓	
S8	✓	✓			✓				✓

Table II summarizes the results. Solutions *Sol 1* and *Sol 2* are based on explicit column alias enhancements and affect both the processing-step scenario and existing pre-calculated, pre-stored execution plans. The additional overhead introduced by these solutions ranges from 0.85% to 1.47%. Overall, the performance difference between Solutions *Sol 1* and *Sol 2* is approximately 0.45%. Solution *Sol 3* represents the embedded approach provided directly by the Oracle Database. As indicated by the results, this solution does not have a measurable impact on performance. The observed overhead remains within the margin of statistical error (0.05%). Results is form of chart are depicted in Fig. 6.

TABLE II. RESULTS

Scenario	SOLref	SOL 1	SOL 2	SOL 3
	Performace - total costs [%]			
S1	100.00	100.00	100.00	100.00
S2	100.00	101.01	100.85	100.03
S3	100.00	101.27	100.87	100.04
S4	100.00	101.44	100.85	100.04
S5	100.00	101.49	101.02	100.04
S6	100.00	101.39	100.89	100.05
S7	100.00	101.41	100.92	100.04
S8	100.00	101.47	100.93	100.05

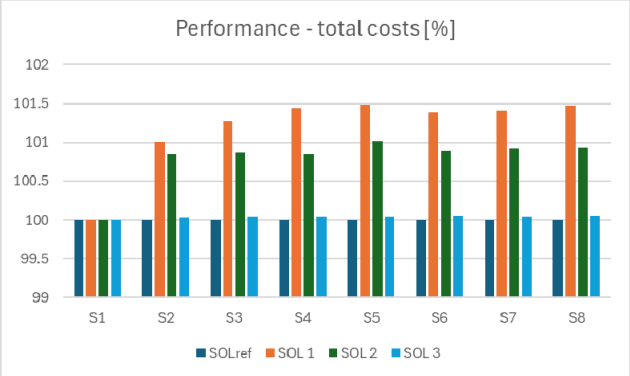


Fig. 6. Results

The evaluation study demonstrates that the embedded solution (*Sol 3*) represents purely syntactic sugar and does not affect performance. Statement tracing shows that queries are automatically rewritten into their original form without aliases, allowing existing execution plans stored in instance memory to be reused directly. However, *Sol 3* is limited to the Oracle Database. To ensure broader applicability across different database systems and releases, *Sol 2*—based on logical constructs—appears promising. The statement enhancement is done by the database optimizer prior to parsing and introduces less than 1% additional overhead.

VI. CONCLUSIONS

This paper addresses an important topic related to database query performance. Data retrieval is a critical operation not only in transactional systems but, more importantly, in data warehouses and analytical environments. Because the SQL standard does not permit the use of column aliases in the Group By and Having clauses, developers are required to repeat identical expressions or function calls multiple times.

This study highlights and evaluates four different approaches. The reference solution follows the SQL standard. The most efficient solution is provided by the Oracle Database,

which embeds query transformation into the optimizer and rewrites statements to their original form internally. However, to ensure broader applicability across database systems and versions, two additional database-independent solutions are also presented and evaluated.

Based on an evaluation study conducted in an aviation-related context, the additional overhead introduced by logical construct management ranges from 0.85% to 1.02%. When the processing is embedded directly within the optimizer, the additional overhead remains below 0.05%.

Future research in this field will focus on further investigating Oracle Database syntactic enhancements related to analytical and aggregate functions. The evaluation will be conducted across a wide range of relational database systems to assess the usability and general applicability of these enhancements. The findings will be supported by quantitative results obtained from the evaluation.

ACKNOWLEDGMENT

This paper was supported by the **VEGA 1/0192/24** project - *Developing and applying advanced techniques for efficient processing of large-scale data in the intelligent transport systems environment.*

REFERENCES

[1] P. Desmarets, B. Hammerschmidt, and S. Hoberman, Oracle Database 23ai Data Modeling and Schema Design for JSON-Relational Duality, Technics Publications, LLC, 2024, ISBN: 978-1634623636.

[2] R. Greenwald, R. Stackowiak, and J. Stern, "Oracle Essentials: Oracle Database 12c." Sebastopol, CA: O'Reilly Media, 2013, ISBN: 978-1-4493-4303-3.

[3] A. Chowdhury, Oracle 23AI & ADBS in Action: Exploring New Features with Hands-On Case Studies, Apress, 2026, ISBN: 979-8868820670.

[4] M. Kvet, Data Analysis - Aggregate Function Performance, SAMI 2025.

[5] M. Kvet and J. Papan, Enhancing Analytical Select Statements Using Reference Aliases, IEEE Access, vol. 12, pp. 27311 - 27330, 2024, doi: 10.1109/ACCESS.2024.3366455.

[6] M. Malcher and D. Kuhn, Pro Oracle Database 23ai Administration: Manage and Safeguard Your Organization's Data, 5th ed., Apress, 2024, ISBN: 978-9798868810374.

[7] B. Rosenzweig and K. L. Ritchie, Oracle SQL by Example, 5th ed., Pearson, 2011, ISBN: 978-0138062838.

[8] I. Suchý, I., M. Turčaník, Review of Large YOLOv8 and RT-DETR energy efficiency on edge devices for real-time detection. Sci. Rep. 16(1), 10908 (2026). https://doi.org/10.1038/s41598-026-46453-6

[9] M. Turčaník, Hash function generation based on neural networks and chaotic maps. In: 2017 Communication and Information Technologies (KIT), Vysoke Tatry, Slovakia, pp. 1–5 (2017). https://doi.org/10.23919/KIT.2017.8109467

[10] Erasmus+ project EverGreen dealing with the complex data analytics, Aug. 2025. [Online]. Available: https://evergreen.uniza.sk/